Technical Report

UCAM-CL-TR-714
ISSN 1476-2986

Number 714

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

A wide-area file system for migrating virtual machines

Tim Moreton

March 2008

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

Improvements in processing power and core bandwidth set against fundamental constraints on wide-area latency increasingly emphasise the position in the network at which services are deployed. The XenoServer project is building a platform for distributed computing that facilitates the migration of services between hosts to minimise client latency and balance load in response to changing patterns of demand. Applications run inside whole-system virtual machines, allowing the secure multiplexing of host resources.

Since services are specified in terms of a complete root file system and kernel image, a key component of this architecture is a substrate that provides an abstraction akin to local disks for these virtual machines, whether they are running, migrating or suspended. However, the same combination of wide-area latency, constrained bandwidth and global scale that motivates the XenoServer platform itself impedes the location, management and rapid transfer of storage between deployment sites. This dissertation describes *Xest*, a novel wide-area file system that aims to address these challenges.

I examine *Xest*'s design, centred on the abstraction of *virtual disks*, volumes that allow only a single writer yet are transparently available despite migration. Virtual disks support the creation of snapshots and may be rapidly forked into copies that can be modified independently. This encourages an architectural separation into node-local file system and global content distribution framework and reduces the dependence of local operations on wide-area interactions.

I then describe how *Xest* addresses the dual problem of latency and scale by managing, caching, advertising and retrieving storage on the basis of *groups*, sets of files that correspond to portions of inferred working sets of client applications. Coarsening the granularity of these interfaces further decouples local and global activity: fewer units can lead to fewer interactions and the maintenance of less addressing state. The precision of these interfaces is retained by clustering according to observed access patterns and, in response to evidence of poor clusterings, selectively degrading groups into their constituent elements.

I evaluate a real deployment of *Xest* over a wide-area testbed. Doing so entails developing new tools for capturing and replaying traces to simulate virtual machine workloads. My results demonstrate the practicality and high performance of my design and illustrate the trade-offs involved in modifying the granularity of established storage interfaces.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

This dissertation concerns the provision of storage for global service delivery platforms — distributed computing infrastructures organised to support latency-sensitive services on a global scale. In particular, it focuses on platforms in which services are deployed as whole-system virtual machines. In this setting, a virtual machine running at a node treats its storage as a single-writer local file system, but expects it to be transparently available after migration to another host. A disk image may need to be rapidly replicated for the deployment of a new service instance. Most importantly, by locating nodes at access networks throughout the Internet to allow services to reduce their latency to clients, this platform subjects its own storage infrastructure to an environment of high latency, limited bandwidth connections.

I describe the design and implementation of *Xest*, a file system built to fit these requirements and meet the challenge of obtaining good performance across the wide area. Central to this is an architectural organisation around **groups**, sets of objects with strong spatial locality that form the unit of storage management, location, fetching and caching. I describe how *Xest*'s design accommodates groups, and present new algorithms for recording reference streams and arranging storage into groups. By implementing and measuring *Xest* over a wide-area test bed that mimics as closely as possible the platform under construction, I demonstrate how coarser granularity operations can mitigate the effects of slow network connections and the difficulties of managing large volumes of storage.

In this chapter I introduce and motivate this work, describe its research contributions and outline the structure of this dissertation.

1.1 Motivation: storage for distributed service delivery platforms

Processing power, storage capacity and core network bandwidth continue to increase, so that the speed of light is rapidly becoming a practical limitation to improvements in wide-

area latency. No further order-of-magnitude reductions are possible: light passes between New York and London in 19 ms, but already an IP packet takes only 35 ms over the GÉANT transatlantic link. Worse, as online services take over rôles traditionally filled by desktop applications, this 'wide-area I/O gap' will become more prominent.

Addressing latency requires web applications, distributed services and content delivery systems to diversify the locations within the network from which they interact with clients. Platforms for distributed service delivery allow service instances to be deployed across a global overlay network according to patterns of demand, then migrate or replicate them as those patterns change. This allows services to reduce latency to clients, improve load balancing and reduce bandwidth costs.

The XenoServer project [Hand03, Kotsovinos05], which provides the motivating context for this dissertation, is building a platform for distributed computing that is both public and general-purpose. Arbitrary, untrusted services specified from the operating system up are deployed inside virtual machines that provide resource isolation, fine-grained billing and auditing. Virtual machines may be suspended and later resumed, or migrated between hosts while running. In contrast, established platforms such as Akamai's deploy trusted services written for a specific API under long-term contractual arrangements [Akamai03]. Section 2.1.1 discusses these projects in more detail.

A key component of the XenoServer platform is a distributed file system that aggregates the storage available at each site to provide persistent storage for running and suspended services. This dissertation describes the design and implementation of *Xest*, a file system that I have built for this purpose.

The platform makes novel demands of its storage substrate. First, each virtual machine expects to be provided with the abstraction of a 'local' file system at the host on which it is executing for access to its operating system image, applications and configuration data. It expects this **virtual disk** to be named independently of where the service is deployed and remain transparently available despite migration between hosts.

Second, the platform requires, and *Xest* permits, access to a virtual disk from only a single host at once. As I argue in Section 2.2.2, few distributed applications rely on write sharing for coordination or data exchange, and there is a large performance cost in enforcing strong consistency semantics over high latency links. At the same time, the XenoServer platform aims to support live migration of virtual machines, and rapidly migrating the virtual disk and write access to it is crucial to this process.

Finally, although *Xest* may manage a very large volume of storage, there will be significant incidental commonality of content between virtual disks. Each virtual disk is likely to incorporate one of a few operating system distributions and may share other infrastructure

software. Exploiting this commonality is important to minimise the platform's storage consumption, improve local cache hit rates and provide more sources from which to retrieve the same data.

1.2 Group-centric storage management

The primary challenge in designing *Xest* is its requirement to operate under the same network conditions that the XenoServers platform seeks to allow deployed services to avoid. In terms of access to local storage, the difference in throughput and latency between disk and memory has been a significant performance constraint for more than a decade [Ousterhout90]. Long-haul links exacerbate this: not only are latency and throughput orders of magnitude worse than for local devices, but they also suffer high variability. Unfortunately, the platform's global reach and bottom-up specification of services mean that it anticipates a very large volume of addressable data, and the two techniques most successfully applied to mitigating high latency and low throughput, cooperative caching and prefetching, tend to scale poorly.

Cooperative caching, employed by file systems including Oceanstore [Rhea03] and Pangaea [Saito02], uses a location-independent naming scheme to source content from nearby nodes. This allows transfers to be striped and conducted over higher bandwidth, lower latency paths. Such schemes require nodes to publish details of the contents of their own persistent caches, and maintain other nodes' addressing information to satisfy their own and others' requests.

However existing systems name and locate data at the granularity of individual files or blocks, which, this dissertation argues, will not scale to systems managing large amounts of storage. First, although the use of shared distributed data structures reduces update traffic, nodes expend a considerable proportion of their available bandwidth publishing their own cache state, thereby reducing their effective fetch bandwidth. Second, the latency introduced by frequently querying this addressing state can outweigh the benefit of retrieving data more rapidly.

Prefetching aims to anticipate cache misses and mask the I/O stalls that they cause, an important aspect to obtaining good performance from the cold cache conditions frequently induced by migrating services. Yet few schemes have been adopted successfully by wide-area file systems. Approaches that explicitly issue speculative fetches are hindered by a distant and highly variable prefetch horizon that complicates accurate estimation of delivery times. Other schemes carry out prefetching implicitly by enlarging the effective unit of

retrieval, as read-ahead schemes do. However the benefit of this approach is tempered by the ensuing loss of precision, which causes data to be fetched unnecessarily.

In this dissertation, I argue that the granularity of operations between infrastructure nodes should be coarsened, not along structural boundaries but rather into **groups**, variable-size sets of objects with strong spatial locality, selected on the basis of observed file system activity. Doing so lessens the impact of latency by reducing the effective scale of many storage management tasks. It also decouples wide-area interactions from local file system activity by performing operations on related data in aggregate. At the same time, the loss of precision is minimised by aligning groups with predicted sequences of accesses and varying their size according to the confidence of these predictions. Although users interact with the file system using a conventional file-centric API, interactions in the storage infrastructure specify whole groups.

Xest employs both cooperative caching and prefetching, but its use of grouping helps it to combine the two techniques while sidestepping their limitations. The fetch protocol can be considered to operate at the granularity of whole groups, so access to a single object results in other group members being retrieved, a form of implicit prefetching or 'just-in-time hoarding' that preloads the local persistent cache with data likely to be accessed in the near future.

The addressing protocol names and advertises data in terms of whole groups, rather than by individual files or blocks. Since groups are larger units, the volume of addressing data and the bandwidth required to maintain it is reduced. Since a group's members tend to be accessed together, the number of source retrieval operations is also reduced.

Although *Xest*'s architecture is structured around groups, I also demonstrate how its protocols degrade to operate on a per-object basis in the face of poor grouping arrangements, to prevent poor locality causing a loss of precision and a corresponding reduction in performance. Good clustering algorithms have high computational cost, only run periodically and cannot represent all access patterns.

The process of allocating objects to groups is as follows. While a virtual disk is mounted and writeable, the file system client observes the corresponding sequence of file and directory accesses, as in previous work on prefetching and stashing [Griffioen95, Kuenning97b]. When the virtual disk is unmounted, a clustering algorithm runs to adjust the assignment of files and directories to groups, to account for recent access patterns.

1.3 Contributions

It is my thesis that *Xest*'s design satisfies the requirements for a storage infrastructure for the XenoServer platform, and that the application of grouping to *Xest* can reduce the duration for which requests are stalled in cold cache conditions.

This dissertation makes a number of significant contributions.

A first contribution is a detailed presentation of the architecture of a wide-area file system designed to operate in the context of a platform of migrating virtual machines. I introduce a novel set of components whose design is influenced principally by assumptions of slow networks, large storage volume, commonality between volumes and single migrating writers. These components manage whole groups, and I describe the effect of this on *Xest*'s mechanisms for naming, advertising, locating and retrieving storage.

A second contribution is the dissertation's presentation of novel algorithms for recording statistics obtained from observing access patterns, and for incrementally forming groups on the basis of partial statistics. In addition, I discuss techniques for minimising the effect of poor clustering outcomes on storage management components.

Further, as part of a methodology for evaluating my implementation, this dissertation contributes a novel means of mimicking a distributed testbed of migrating services, and describes original methods to gather and faithfully replay traces so as to capture the effect of cold cache conditions on file system performance.

A final contribution is a detailed evaluation of the *Xest* prototype implementation. This comprises measurement of both the mechanics of the file system and of the group assignment process.

1.4 Outline

In this section I describe the organisation of the remainder of this dissertation.

Chapter 2 provides further background on the XenoServer platform and expands on the storage requirements of its services. It then discusses related work in two areas. First, I survey several architectural themes in the design of distributed file systems particularly relevant to *Xest*. Second, I explore how file systems have previously used access patterns to inform their operation.

Chapter 3 describes the architecture of the *Xest* prototype. It introduces the principal components of the file system and presents their interfaces. It presents *Xest*'s group-based storage substrate, over which is implemented a file system hierarchy. It shows how data is

cached, advertised, located and retrieved. The concept of the virtual disk is introduced and used to explain how the global storage namespace is organised and persistence of storage is maintained.

Chapter 4 describes the process of allocating objects to groups. It discusses how *Xest* observes file system accesses and accumulates statistics that are stored alongside regular data. A clustering process uses these statistics to reassign objects to groups. After exploring this algorithm's requirements and implementation in detail, the chapter concludes by describing how the new arrangement of groups is applied to an existing file system structure.

Chapter 5 describes my experimental methodology. After surveying previous work on file system tracing, it presents techniques for evaluating the file system in a distributed setting without access to a testbed of migrating services. I detail the implementation of a tracing toolkit developed for this purpose.

Chapter 6 evaluates the *Xest* prototype implementation and presents a measurement study of the grouping process. I evaluate the cost and effectiveness of *Xest*'s approach to cache advertisement, object location and retrieval, and virtual disk management. I am able to demonstrate that grouping can significantly improve the performance of remote file system access, and that *Xest*'s architecture makes the migration of volumes in the wide area practical in terms of downtime and impact on performance.

Chapter 7 concludes the dissertation by describing the key outcomes of this work and presenting areas for future research.

Chapter 2

Background and related work

This chapter begins by setting out the research context within which *Xest* has been developed: that of a file system for migrating services deployed over a global public computing platform. I distill a set of assumptions and properties desirable in such a storage substrate, and in Section 2.2 these requirements are used to assess the suitability of existing file system designs.

One important requirement is that of good cold cache performance under conditions of high latency, constrained bandwidth and very large storage volume. This thesis argues that a way to reduce the number of interactions over slow networks and mitigate the effects of scale is by coarsening the granularity at which object location and storage management are performed. To this end, Section 2.3 contrasts our approach with previous work on varying interface granularity, observing access patterns and using them to predict future file system activity.

2.1 Storage for distributed computing infrastructures

In this section I introduce the XenoServer project for global public computing, and compare it with related efforts. I then go on to consider how the properties of distributed computing frameworks in general, and the XenoServer platform in particular, influence the design of storage substrates that provide the 'local disks' of running and suspended services.

2.1.1 The XenoServer project

The XenoServer research project underway at the University of Cambridge Computer Laboratory's Systems Research Group aims to build a platform for public distributed computing [Hand03, Kotsovinos05], in which uncooperative clients can deploy arbitrary, untrusted services on managed servers at strategic points throughout the Internet. Services

are specified in terms of an operating system kernel image and root file system and execute within resource-managed virtual machines. The Xen hypervisor [Barham03] maintains isolation and fair sharing of local resources and enforces the platform's billing, auditing and security components.

The platform aspires to the 'utilification' [Wilkes04] of distributed computing, where services are 'configured once and deployed anywhere'. It aims to lower the barrier to entry for users of a service delivery platform by allowing server resources to be purchased at a fine granularity and on short timescales, without the need for contractual relationships with server owners.

Service images are prepared ahead of time and then deployed, perhaps in parallel to a number of locations; it is expected that in most deployment scenarios no user will ever need to login to the virtual machine, as the system will boot and start applications automatically. Service instances may then be migrated or replicated while live [Clark05], or suspended for later resumption, according to network conditions, request patterns or other external events. In this way clients can consolidate their network presence yet respond rapidly to changes in demand. Search tools already automate the server discovery and selection process [Spence03], and in future may assist in automatically load balancing distributed services.

The XenoServer platform differs from related efforts in terms of purpose, configurability, funding model, and barrier to entry. Akamai's EdgeComputing platform [Akamai03] is a global infrastructure for content distribution and service deployment, but client deployments require contractual agreement and are restricted to J2EE applications. VMatrix [Awadallah02] proposes to build an service delivery framework that migrates virtual machines between servers running VMWare Workstatation [VMWare02]. However it assumes a single administrative domain, and does not address authentication, auditing, or limiting resource consumption.

PlanetLab [Chun03, Bavier04] is a networking research testbed restricted to participating universities, which provide the computing resources. Servers are virtualised but resource isolation is weak and users have little control over operating system configuration. Whereas PlanetLab aims to provide a deployment platform for services sensitive to network position, Grid platforms focus on the parallelisation of computationally-intensive tasks. Most assume a cooperative user base and Globus [Foster97], the predominant Grid toolkit, requires services to be rewritten to support its API.

2.1.2 A storage substrate for the XenoServer project

The storage needs of virtual machines deployed on the XenoServer platform pose a set of assumptions and requirements which together constitute a significant challenge to the design of existing storage systems.

This section describes these requirements and outlines *Xest*'s approach to them. Section 2.2 contrasts related work.

- **Nomadic, isolated virtual disks.** The XenoServer platform offers resources in terms of node-local virtual machines, and any service attempting to coordinate multiple nodes must layer its own distributed functionality on top. The storage substrate is not distributing the execution of applications designed for a local context, as in [Veitch01], but is facilitating the migration and replication of services that are already distribution-aware.

  Each virtual machine views its virtual disk as a local file system, although in aggregate *Xest* acts as a distributed file system. While strong fine-grained concurrency control is expected between its applications, write sharing of virtual disks between different virtual machines is not necessary, and, I argue later in this section, is costly in both implementation and performance terms.

  In its place, *Xest* aims to facilitate storage administration with primitives to snapshot and fork virtual disks, as I describe below.

- **Slow networks.** Although the platform is comprised of well-provisioned server-class machines, they are necessarily dispersed throughout the world and operate from machines rooms positioned close to access networks. The long haul connections between them suffer high latency, constrained bandwidth and transient failures.

- **Very large storage volume.** The scale of the platform and the flexibility afforded by allowing complete disk images to be specified presents a considerable global storage burden.

  Scalability targets of several thousand machines each running tens of virtual machines have been made feasible by the continuing fall in cost per gigabyte of storage hardware [Gray00]. Existing content distribution networks such as Akamai's are of the same order of magnitude. In addition to storage for running virtual machines, capacity is required for offline and suspended services.

  It is possible to cheaply and reliably provision this volume of storage: each XenoServer may have commodity local or network-attached disk arrays several terabytes in size. However the challenge lies in the effective coordination of a very large volume of geographically dispersed storage to support service deployment.

- **Dominance of cold cache conditions.** Storage coordination is most important when a virtual machine is being deployed or migrated, since cold cache conditions at the target may have significant impact on its performance. Nodes' persistent caches are sufficiently large to eventually fit the virtual machine's entire file system, but having regard to current activity and managing the order in which material is transferred are important for mitigating the effect of cache misses. At the same time, if a virtual machine migrates again before accessing transferred portions of its image, bandwidth is wasted and other transfers were delayed unnecessarily.
- **Commonality between virtual disks.** While operating system installations are typically several gigabytes in size, significant commonality between many images may be exploited in order to limit transfers when migrating or deploying services. Nevertheless, services may require customised application suites, or the application of package or kernel updates, and some deployments' express purpose will be to deliver large volumes of content from near to users. Just as the XenoServer platform itself allows execution of arbitrary applications but charges for disk, network and CPU resources used, the storage system should allow each administrator to flexibly trade off the degree of customisation of disk images with the performance implications that follow from the reduction in effectiveness of caching.
- **Disk image management.** The automation of distributed service deployment must extend to the management of storage, so that administrative interfaces for preparing and maintaining file system images are separated from a virtual machine's use of an image. Disk images should be able to be 'forked' and modified to allow a service to be replicated into an additional virtual machine; this process should occur rapidly and exploit the fact that the file systems are nearly identical.

There are also further considerations that lie beyond the scope of this dissertation but are mentioned here for completeness.

As a public infrastructure itself, the storage system must securely multiplex service to untrustworthy and uncooperative clients. Not only must it perform access control, quota management and billing, but it must fairly partition its resources to prevent denial of service. The latter in particular is difficult and remains the subject of ongoing work: any sharing of resources necessarily compromises isolation.

A complementary component to a file system that orchestrates storage between XenoServers is a file system that multiplexes that service to the virtual machines running on a single computer. While existing network file system transports such as NFS [Pawlowski94] could be integrated, work on a higher performance Xen-specific shared memory transport is underway [Williamson05].

2.2 Distributed file system architectures

In this section I survey several architectural themes in the file system literature that are particularly consequential to the requirements drawn up above.

First, Section 2.2.1 considers the discovery and retrieval of stored objects, operations that are on the critical path of satisfying cache misses and that are particularly sensitive to the effects of slow networks. Then, Section 2.2.2 distinguishes existing approaches to concurrency control for shared storage from *Xest*'s approach of providing the abstraction of isolated, migrating virtual disks. Section 2.2.3 summarises previous work on detecting and unifying shared sections, techniques applicable to exploiting the significant similarity between disks. Finally, Section 2.2.4 considers existing systems that are tailored for migrating virtual machine state.

2.2.1 Object location and retrieval

How file systems locate and retrieve metadata and data is a strong determinant of their performance in cold cache conditions, a situation to which frequent migration of virtual machines makes *Xest* particularly susceptible.

Some file systems divide up logical regions of the namespace to distinct servers, simplifying object location at the expense of poor cold cache performance. NFS [Sandberg85] and Sprite [Nelson88] provide network access to a single, autonomous remote file system, rather than distributed storage. Since each server's namespace is disjoint, the location of its objects is implicit. However, the 'remote access' abstraction causes interfaces to dictate similar semantics and granularity of interactions for remote as for local clients. NFS requires synchronous writes and metadata operations mimic local ones, necessitating many round trips. Ignoring this inherent asymmetry results in high server load and poor response times, particularly over slow networks [Pawlowski94, Howard88]. Sprite delays write-backs and provides strong consistency guarantees but nevertheless operates poorly over slow networks.

AFS [Morris86, Howard88] is designed as a large-scale file system for sets of institutions spanned by slow networks. To reduce the number of server interactions, clients cache whole files persistently on disk and retain callbacks on cached data. However AFS still centralises access to metadata, and hence cached data, at a remote server. Its global namespace is partitioned along institutional boundaries into 'cells', sets of co-located servers, and read-write access to a volume is conducted through a single machine in that cell. This is a single point of failure and means that good cold cache performance relies on a client being local to the cell.

In xFS [Anderson95], in contrast, a local network of workstations act symmetrically as clients, storage servers and metadata managers. Nodes use 'cooperative caching' [Dahlin94] to serve each other blocks from their buffer caches. However, xFS is unsuitable for slow networks since its design assumes uniform latency between all nodes. It aims to balance load across all clients with cached copies, rather than locate the 'nearest' one.

Managers are divided by region of the file index space and hold a canonical list of the clients at which each block is cached, expensive to maintain at large scales. The granularity of caching in xFS is a single block, so all reads and writes are performed through the manager. Further, a manager failing requires all nodes to participate in an expensive agreement protocol to replace it; in a wide-area setting, however, network partition and node churn are frequent.

Farsite [Adya02] aims to share reliable storage across an organisational-scale set of machines connected by a high-speed network. It shares xFS' serverless design and latency assumptions, but focuses on ensuring the integrity, redundancy and privacy of data over an infrastructure comprised of untrusted workstations. The rôle of serving a directory tree is divided between two sets of hosts. A 'directory group' is presumed to reside at well-known locations and operates a Byzantine agreement protocol that agrees every file open or metadata operation. File data is then retrieved from a node in a separate 'file group', in order that high availability not also incur high storage replication costs.

Xest's managers (Section 3.6) in contrast unify storage and consistency management tasks, and may be located despite membership changes using the DHT. Adya *et al.* planned to allow directory groups to delegate the management of subtrees to new groups in order to distribute load. Since *Xest* assumes a single writer per virtual disk, consistency need only be enforced at the top level of the namespace. Performing agreement only on control-plane operations suits a high latency environment and results in very low manager load.

A number of systems, including *Xest*, use Distributed Hash Tables (DHTs) to locate file system data in the global Internet. Several use the DHT to partition the storage workload of the system by publishing the data itself to a set of nodes closest in the identifier space to the secure hash of the data. CFS [Dabek01] runs over Chord [Stoica01] and manages fixed-size blocks. PAST [Rowstron01b] and Pasta [Moreton02] run over Pastry [Rowstron01a] and manage whole files and variable-size chunks (see Section 2.2.3), respectively.

Pasta augments the read-only publishing and archival provided by CFS and PAST with mutability using hierarchies of self-certifying directories. However, all three suffer from the significant cost of having to transfer large volumes of data between peers to maintain replication invariants as their underlying DHTs suffer churn and network partition.

Oceanstore's use of 'promiscuous caching' [Kubiatowicz00] and Pangaea's 'pervasive replication' [Saito02] avoid relocating data but extend cooperative caching of rewritten content to a wide-area setting. Unfortunately, the fine granularity at which they address data challenges the scalability of their designs.

In Oceanstore's prototype, Pond [Rhea03], blocks of data and metadata are named independently of their location and replicated for availability and performance. The Tapestry DHT [Zhao04] is used to locate cached copies of blocks despite node churn. Each node advertises the contents of its persistent cache by periodically republishing soft-state pointers associating each block's identifier with the node's identifier. Pointers are cached along the path in the Tapestry overlay to the node with identifier closest to the block. By selecting neighbours that are close in the underlying network, a node looking up a block's identifier in the overlay with high probability first encounters pointers to a nearby copy.

In order to probabilistically reduce the latency of overlay lookups relative to IP latency for nearby objects, Oceanstore nodes maintain soft-state 'attenuated' Bloom filters [Rhea02] for each node in their routing table, an array of filters that at level i summarises the blocks stored by nodes within i hops of that neighbour. Before routing using Tapestry, neighbours whose filters indicate they may have the block are contacted.

However Oceanstore data blocks are 8KB in size, and most metadata blocks smaller still. First, as DHT lookup is on the critical path of servicing misses for which the locations of cached copies are not already known, smaller units of addressing mean that response times depend more on round trip time. Second, since locating nearby sources depends on pointer replication, fine granularity exacerbates the tension between the amount of storage consumed by addressing information and that used for caching. Pointers will consume 8.1GB at each node in a system of one thousand nodes storing 1GB of data each.[1]

Third, minimising response time by avoiding failed lookups depends on regularly expiring and reinserting pointers, according to the scheme described in [Rhea03]. With a lifetime of twenty-four hours, each node's reinsertion traffic requires 4 messages per second per GB stored in the global system.[2] Attenuated Bloom filters suffer similarly. To avoid saturation even at the first level, filters may consume 48MB per GB stored.[3]

Pangaea's 'pervasive replication' [Saito02] manages cached copies as replicas in a strongly-connected per-object graph. It uses a single, global namespace and locates nearby replicas of

[1] Pointers consist at minimum of a SHA1 hash and an IPv4 address and port, in total 26 bytes. Assuming a mean block size of 8KB [Rhea03], and that each pointer is cached at $log_{16}1000 = 2.5$ hops in the DHT, pointers consume 8.1MB globally per GB of data.

[2] By ((1GB/8KB)*2.5 hops) / 14400s.

[3] An acceptable false positive rate requires six bits per item per filter [Broder03]. Given a routing table size, $r = 16$, each unsaturated filter at the top level requires 1.5MB per GB assuming 8KB blocks. Ignoring saturation at the top level, filters to depth two total 48MB.

an object through 'gold replicas', pointers to nodes that are hard-coded in directory entries and updated only on node failure. Gold replicas in turn follow edges in the replica graph to locate nearby sources. Graph connectivity is monitored by pinging neighbours, in order to ensure that object location succeeds and updates can be received and disseminated. However, although Pangaea dispenses with block-level addressing and instead manages whole files, desktop workloads exhibit very large numbers of small files [Douceur99]. Managing hundreds of thousands of graph edges may lead to each node monitoring the health of every other node in the system, a significant impediment to scalability. Further, the main purpose of replica graphs, disseminating updates, is no longer necessary if write sharing is barred.

Other domain-specific transfer systems exist. Frisbee [Hibler03] distributes file system images within a LAN using minimal knowledge of file system structure to elide transmission of free blocks. Large-scale Internet content distribution networks such as CoDeploy [Park04] and Bittorrent [Cohen03] focus on rapid transfer of large, single files. In contrast, *Xest* aims to satisfy sub-file accesses, exploit commonality between images and offer global-scale deployment, per-node configuration and migration.

2.2.2 Write sharing

Recall that *Xest*'s aim is to make storage available transparently as execution moves or forks to new points in the network, and to simplify this task, it disallows write sharing of storage between virtual machines.

Applications that form distributed services, such as replicated databases, rarely rely on a file system for short-term data exchange or collaborative modification between remote peers. Rather, they use custom data transfer and concurrency control schemes over network-oriented protocols, exploiting their own knowledge of update semantics to better trade off availability and performance. Indeed, no consensus on concurrency control has emerged between distributed file systems: NFS v2 offers 'best effort' semantics [Pawlowski94], while AFS enforces close-to-open consistency but fails to honour the `lockf()` API.

In any case, enforcing strong consistency semantics in the wide area is difficult to achieve without a significant impact on warm cache client performance or file system availability [Petersen97]. Lease operations may suffer high latency, packet loss and manager failure, but may be necessary for each file open operation. On the other hand, optimistic concurrency control schemes, while well-suited to mobile file systems, may introduce write conflicts that must be resolved, requiring either user or application intervention [Kumar93]. Requiring administrator intervention for an automated service deployment or changing the file system interface to allow applications to perform conflict resolution are both impractical solutions.

2.2.3 Commonality in disk images

Although *Xest* disallows the sharing of writable virtual disks between nodes, extensive implicit read sharing of the contents of related disks is important for two reasons. First, the commonality between different images' operating system distributions, application binaries and shared service-specific data can be exploited in order to yield more fetch sources and thereby reduce transfer latency. Second, any efficient means of forking a new instance of a virtual disk, as required when a service is replicated, must afterwards share its contents with the other disk.

Previous work in which I collaborated [Kotsovinos04] uses a stacking file system to export a root file system for a virtual machine by overlaying an immutable *template* image with a sparse, custom image of modifications and additions. Overlaid images reside in shared remote storage, implemented in our prototype using AFS. Template images are cached at each XenoServer, eliminating the bulk of the transfer during deployment. Arbitrary overlays are possible, in order to allow per-service-instance customisation; per-file copy-on-write causes file modifications in the topmost writable mount, in a manner reminiscent of union directories in Plan 9 [Pike92].

Our evaluation shows that overlays for services such as an Apache web server and a Quake multiplayer game server are a small fraction of the size of a Linux distribution template image. While this demonstrates the significant sharing likely to be seen between services' disks, overlaying suffers keys limitations that stem from its 'engineering' of commonality. By basing images on fixed templates, it does not address the general case where commonality may be present between different overlays, rather than only between an overlay and a pre-defined template. This is important since overlaid images themselves may become large in size, due to additional content or significant modifications to the template's contents. Transfers of overlays are constrained by the distributed storage system they are stored in.

Managing modification to underlays is also problematic. If templates are fixed, service administrators must each manage their own operating system updates, causing overlays to increase in size relative to the template. If a template undergoes a central maintenance process, administrators are unable to perform acceptance testing on updates, and inconsistencies may arise due to updated versions of files in the template being overlaid by pre-existing files in the overlaid image. A similar argument applies to updates to different levels of overlay.

An alternative approach that *Xest* adopts uses snapshot and copy-on-write techniques, previously applied to maintaining file version history, simplifying crash recovery, and for write-once archival purposes [Santry99, Hitz94, Chutani92, Quinlan91, Quinlan02]. Whole disk fork operations can be implemented by snapshotting a disk and making the snap-

shot writable. In this way commonality is 'elicited': disks are forked from template images of operating system distributions, so that unmodified portions can be sourced from any node caching another disk sharing these.

Parallax [Warfield05], a file system for virtual machines running in a cluster environment, uses a radix tree to map between the contiguous logical block space of a 'cluster virtual disk' and corresponding physical blocks in the underlying storage substrate; snapshotting duplicates the tree root and marks its children read-only. Sapuntzakis *et al.* [Sapuntzakis02] store virtual machine state as a 'capsule', a chain of copy-on-write disks: each disk contains a set of sequential extents of modified blocks and a reference to its parent. WAFL [Hitz94] performs copy-on-write on inodes, since many write operations are in practice entire rewrites of a file [Vogels99].

The Low-bandwidth Network File System (LBFS) [Muthitacharoen01] aims to detect and exploit commonality in regions of data held by both sender and receiver in order to reduce wide-area transmission delays. It maintains a persistent client cache and extends the NFS protocol so that when a file is closed only regions not already held by the server are sent. An incremental Rabin fingerprint is computed [Rabin81] to split the file into variable-size chunks with boundaries determined by their content. The client initially sends chunk fingerprints, and the server uses a pre-computed index of its stored chunks to request any that it does not hold.

Pasta also employs Rabin fingerprints to support insertions into files efficiently and to improve global storage utilisation [Moreton02]. However a further study suggests that in typical workloads the sharing benefits of dividing files using Rabin fingerprints over fixed-width boundaries are insufficient to outweigh the cost of indexing [Policroniades04].

2.2.4 Migrating virtual machine state

Several projects have considered how to mask delays in transferring a virtual machine's memory and disk state between machines.

Xen allows virtual machines to be migrated while live by pre-copying their memory over a series of iterations [Clark05]. First, all of its resident pages are copied to the target host, then the process repeats to recopy any pages modified since the previous iteration. When the rate at which pages are copied falls below the rate at which they are dirtied, the virtual machine is suspended, the remainder of its pages is copied, and execution is resumed on the target host. While this approach results in very small downtimes, it assumes a LAN environment and uniform access to storage between both hosts. This dissertation lays the groundwork for migrating that storage in the wide area.

The Internet Suspend/Resume (ISR) project [Kozuch02a, Kozuch02b] uses Coda to transport the state of suspended virtual machines over the Internet so as to follow mobile users. A pseudo block device implements a virtual machine's disk, dividing its block space into large, fixed-size 'chunks' and writing modified ones into a Coda volume.

When a suspended virtual machine is resumed, the local Coda client at the target site attempts to hoard the whole volume. The virtual machine is able to proceed immediately, but misses in Coda's local persistent cache cause requests to block. Whereas Coda relies on the underlying file system's layout to achieve good spatial locality within its large chunks, *Xest* fetches groups of objects strongly related to current file system activity. Coda in effect attempts to prefetch the entire disk, while *Xest* only prefetches data likely to be accessed.

If ISR knows the resume site before suspension, the source writes chunks back to the Coda server, invalidating callbacks on it, so that the new chunk is hoarded when the target next polls the server. This server-centric interaction is necessitated by Coda's architecture, and may act as a bottleneck on the transfer. In *Xest* state transfers can be made directly, and specific prefetching hints sent to the target site before suspension.

2.3 Mitigating scale and slow networks using access patterns

A number of well-established techniques exist to mitigate the effect of slow networks. Cooperative caching, described in Section 2.2.1 aims to reduce the effective latency of operations, while prefetching, discussed in Section 2.3.3 aims to decouple network operations from blocking user interactions. However both techniques' effectiveness is constrained by scale: cooperative caching by the volume of addressing information to store and maintain, and prefetching by the volume of state required to make accurate predictions sufficiently far ahead that they can be acted on.

This dissertation argues that a solution to this problem lies in the granularity at which the file system conducts its interactions. Coarser addressing and prefetching units reduce the effective scale of the system; coarser operations themselves reduce the interactive dependency on remote servers over high latency links.

However, a protocol's granularity trades off aggregation with precision: although aggregation can amortise computation or communication overheads, a less precise interface can lead to 'aliasing' whereby an operation performs work on items unnecessarily. Retrieving unneeded data over a constrained bandwidth link, for example, can lead to a 'convoy effect' that increases fetch time. *Xest* aims to retain high precision by aggregating storage into **groups** on the basis of observed access patterns, rather than along logical or structural

lines, and allows operations on groups to degrade to processing their constituent pieces separately when predictions do not hold.

In this section I first consider how access patterns can be used to inform future file system activity, and describe several existing schemes for inferring behaviour by observing streams of file system operations. I then outline work that applies such statistics to prefetching and stashing. The remainder of the section concerns existing efforts that have considered the effect of granularity on operations over slow networks, and on laying out items into pages according to access predictions – a case where the granularity is fixed, but aliasing can be minimised.

2.3.1 Application hints *vs.* inferring future activity

Predictions of future access patterns must be either explicitly provided by applications or inferred by observing past behaviour. Since *Xest* is targeted at a platform running general-purpose commodity server software, this work adopts an observation-based method in order to avoid modifications to the file system interface.

However Patterson argues that programmers are best placed to expose both the concurrency of requests and likely future resource demands, but that the existing interface dictates a serial, operation-centric approach [Patterson93, Patterson95]. Steere advocates modifying the file system interface to add an operation that iterates over sets of files [Steere97]. Non-determinism in the order of an application's I/O requests would be exposed, and total latency could be reduced by allowing processing to proceed on the file whose contents are available first.

Griffioen and Appleton [Griffioen94] suggest that programmers should not be required to anticipate future accesses for the same reasons that modern compilers automate deallocation and code optimisation. The control flow of complex applications may not be well understood and code modularisation across narrow interfaces may hide information about the context of I/O activity and prevent accurate predictions being made.

Having compilers generate hints is also problematic. Scripting and other interpreted code is unamenable to this process, yet typically accounts for a significant proportion of I/O activity. In addition many programs perform work by invoking a series of other programs, but processes cannot make predictions or use knowledge from outside their own execution context. On the other hand, observation of accesses below the file system interface is necessarily system-wide and does not suffer these limitations.

Moreover there are significant practical considerations which hinder schemes that rely on application or compiler hints, particularly for file systems. System call interfaces tend to suffer significant inertia on account of the difficulty and expense of modifying and retesting

applications written for them. POSIX file system calls have remained virtually unchanged in thirty years, syntactically if not semantically.

Not all application and user hints need be explicit. Position within the directory hierarchy is a well-established indicator of spatial locality [McKusick84]. Other work has demonstrated how a file's name, mode and open flags are accurate predictors of its size, lifetime and access type [Ellard03b, Ellard03c].

Patterson proposes that hints 'disclose' operations that an application is likely to perform, rather than provide specific instructions to the prefetcher or buffer cache [Patterson93]. In this way the underlying implementation can weigh the demands of competing applications and take account of available resources, disk layout and other factors. However the distinction between disclosure and instruction is orthogonal to whether hints are provided by applications or inferred through their behaviour; rather it separates the mechanisms for obtaining information on process' probable future behaviour and for using that information in storage management decisions. Indeed, many of the techniques proposed in this dissertation are applicable to a system that uses disclosure hints.

2.3.2 Observing streams of file system operations

Most systems that do not use explicit application hints infer probable future file system activity by observing the profile of transitions between the files referenced in each consecutive system call.

Many schemes model reference sequences as a Markov chain, in which the probability of an item being referenced at any point depends only on the item referenced immediately previous to it [Tsangaris91]. Some record information only about the successor that is seen first, is seen most recently, or most regularly, and other such permutations [Amer01, Amer02b, Shah04]; others maintain conditional probabilities for more transitions [Geels01]. Others generalise the notion of successor to being within a fixed number of references, to account for sequentiality imposed by the interface [Griffioen94].

A number of schemes aim to provide greater look-ahead or model second- or higher-order transitions. Kroeger and Long [Kroeger96, Kroeger01] use a finite multi-order context model to store high probability series of transitions, and employ pruning and ageing techniques to limit storage requirements and reflect recent access patterns. Lei and Duchamp [Lei97] propose 'access trees' that record and detect repetitions of control-flow sequences and their file accesses.

SEER's 'semantic distance' measure [Kuenning97b] is the geometric mean of the number of intervening `open()` operations between pairs of files. While this effectively captures for a given file the extent to which other files are used in conjunction with it, when they are used

together, it does not account for frequency of access. In other words, a file A having lowest semantic distance to a file B demonstrates the existence of a working set containing both A and B, but there may also exist a more frequently occurring working set that excludes B.

2.3.3 Prefetching and hoarding

Prefetching aims to reduce read latency by retrieving data into a local cache before a process requests it. Each of the reference prediction schemes described in the previous section, with the exception of SEER, has an accompanying prefetch component: for most, if a sequence of references generates a prediction, then a request is issued for that object immediately. However, slow networks may require predictions to be made many references ahead in order to fetch an item before it is requested. Misprediction can lead to delays in transferring other objects for which cache misses do occur.

Hoarding selectively retrieves data from remote sites and stores it in a local persistent cache so that it is available during periods when network connectivity is unavailable. This process balances the same factors as prefetching but on a longer timescale — the cost of misses in terms of user progress against the cost of anticipatory fetches in terms of cache space and network bandwidth. Since the latency of locating and transferring objects in a wide area file system is orders of magnitude worse than from a local disk, and bandwidth is a scarcer resource than local disk space, the appropriate trade off for wide-area file systems lies between these two extremes.

SEER [Kuenning97b] uses its estimates of semantic distance to make hoarding decisions. Before disconnection, a clustering algorithm runs to gather files that, intuitively, are estimated to belong to the same 'project' or working set. Clusters are formed based on shared neighbours, and the contents of loosely related clusters are overlapped. The algorithm is described in more detail in Section 4.1.

The hoard is filled by fetching clusters that uniquely contain the most recently referenced files [Kuenning97a]. The size of hoards for which no misses occur during disconnections on traced machines represents only a small overhead on the working set size, and is significantly smaller than required by LRU hoarding as used by Coda [Kistler91].

2.3.4 Coarser granularity in transfer protocols

Aggregation can benefit fetch protocols operating over high latency connections. Day proposed that the Thor OODBMS perform transfers at the granularity of 'prefetch groups', fixed size sets of dynamically selected objects, rather than single objects or whole disk pages [Day93, Day95]. He argues that this reduces the transfer overhead compared to single object protocols, while avoiding the effects of poorly fitting static clusterings when

sending whole pages of objects. Unfortunately Day's evaluation only considers the 'intrinsic computational cost of the implementation' and does not take account of network characteristics, despite a wide-area context being a key motivation for his work.

Similarly, but in a file system setting, Amer, Long and Burns [Amer02a] propose that a server replies to a request for a file by sending in addition a group of files that is determined dynamically and consists of its observed successors, as per their earlier work described in Section 2.3.2. However, their simulations do not appear to model disk or network latency or bandwidth, and there is no discussion of the effect of additional transfers on the time taken to service client misses. As I discuss in Section 3.5.4, unnecessary transfers can have a significant impact on performance.

The notion of group membership being instantaneous and defined only at a single server has benefits and limitations. Access patterns may change with time and dynamic clustering can take account of the most recent reference data, unlike periodic static clustering. Moreover, dynamic clustering can return different groupings for different clients to reflect variations in their access patterns.

However if a group's membership is only transient, it cannot be used as a global name. This means that the technique cannot be used to reduce the number of addressable units, an important benefit of *Xest*'s approach. Further, group membership must be computed at every request, precluding the use of higher cost but more accurate clustering algorithms.

While this dissertation focuses on the effects of slow networks in the context of a global system, local file systems also operate on storage in aggregate to mitigate the effect of disk seek time. Most operating systems perform read-ahead for sequentially-accessed files.

Hummingbird [Shriver01] is a special-purpose file system designed for web proxy workloads in which the proxy tracks each client's request stream and issues hints that particular pairs of files are related. When reclaiming buffer cache space, it packs the least recently used file, together with all files related to it, into large, fixed-size 'clusters' that are the unit of disk access and whose contents are colocated on disk. Similarly, the Cheetah web server [Kaashoek96] colocates a page and its images on disk and reads them in together.

Hummingbird significantly reduces both the number of disk accesses and the mean time to complete a file read, since fetching each cluster implicitly prefetches related files. However there are no published measurements of its effect on precision: clustering could increase the volume of data that is fetched but never used and the disk space overhead resulting from replicating files in multiple clusters.

2.3.5 Minimising aliasing through layout

Good locality between aggregated items is important for minimising aliasing across coarser interfaces. Analogously, if caching is performed across the interface, *e.g.* for a fetch protocol, the items' locality determines the efficacy of the group as a unit of 'implicit prefetching'.

Laying out items within groups for good locality, however, can be complementary to coarsening granularity. Object-oriented databases, which must map fine-grain objects on to fixed-size disk pages, have scope to reduce the number of disk reads or the memory footprint required for traversals. Given the assumption that pages accesses are independent and identically distributed, Yue and Wong [Yue73] proved the optimality of the probability ranking partition, a scheme that assigns objects to pages in the order of their absolute probability of access.

Tsangaris and Naughton [Tsangaris91, Tsangaris92] evaluate partitioning schemes assuming a stochastic access model. SMC.KL is based on Kernighan and Lin's heuristic for solving weighted graph partitioning problems [Kernighan70], and aims to find a clustering that satisfies page size constraints and minimises the probability of transitions that cross partition boundaries. SMC.WISC, a stochastic variant of the probability rank partitioning with lower complexity and similar performance, starts a page by selecting the remaining object with greatest stationary probability, then fills the rest of the space with objects with greatest probability of transition from the contents of the group so far [Tsangaris92].

Allowing clustering processes to determine the size of each group, *i.e.* the degree of aggregation, introduces a further complication. In *Xest*, although reducing mean group size reduces the potential loss of precision and so volume of data that needs to be cached or fetched for a given working set, it also increases the volume of addressing state that must be managed, and increases the number of fetch requests that must be issued for a given working set. Existing systems that perform aggregation in terms of logical structures cannot easily make this trade off. Determining under what conditions to accept an incremental reduction in group size for a certain improvement in clustering must be estimated by parameter setting or feedback.

Disk layout algorithms aim to minimise the number of seeks by improving intra-cylinder locality [McKusick84], and each seek's length by permuting cylinders [Vongsathorn90, Ruemmler91]. Notably, Eaton *et al.* investigated allocating files to 'clumps' according to SEER's clustering algorithm and colocating their contents on disk [Eaton99].

2.3.6 Adapting granularity for clustering quality

Operations made coarser on the basis of predictions can suffer aliasing if those predictions are incorrect. Good clustering algorithms are computationally expensive and no clustering

arrangement can accommodate all patterns of requests [Tsangaris92]. Further, if groupings are static, access patterns may change with time and a periodic clustering process cannot take account of the most recent reference data.

One method of minimising the potential costs of aliasing is to respond dynamically by selectively reducing the granularity of operations. Castro *et al.* studied Thor's client cache and observed that poor clustering of objects to pages meant that some pages contained both hot and cold objects [Castro97], reducing its effective capacity. They developed a scheme called Hybrid Adaptive Caching (HAC) which evicts pages whose coldest objects consume most space. In so doing, hot objects in those pages are compacted, so that that portion of the cache degenerates to an object store.

I describe how *Xest* uses an analogous technique to manage its persistent cache in Section 3.4.3, and how it uses prioritisation information to selectively reduce the granularity of its fetch protocol in Section 3.5.4.

2.4 Summary

In this chapter I have described the research context in which *Xest* is motivated.

I began by introducing the XenoServer project and arguing that existing file system designs are poorly suited for satisfying the storage needs of services deployed on the platform. Current distributed object location protocols are challenged by expectations of large scale and cold cache conditions. Virtual disk images will exhibit significant commonality but few wide-area file systems can exploit this. Requiring only a single writer per volume presents an opportunity to avoid the complexity and performance impact of wide-area concurrency protocols. In addition, the file system must support migration of this writer between hosts.

I then described previous work in which access patterns were studied in order to make inferences about future system behaviour, to set in context *Xest*'s grouping proposals. In the next chapter, I describe the design and implementation of *Xest*, and show how its architecture and the interfaces that it exposes employ grouping in order to satisfy the requirements set out here.

Chapter 3

Architecture

In this chapter I describe the architecture of the *Xest* file system. I focus in particular on how its structural organisation around groups and virtual disks affects storage management tasks, including the advertising, locating and caching of data, as well as the maintenance of persistent storage. In doing so I detail techniques whereby *Xest* adapts to poor quality groupings in order to minimise their impact on the system's performance. This discussion sets in context the presentation in Chapter 4 of the components that observe access patterns and use these statistics to determine group membership.

3.1 Terminology

I begin by clarifying some terminology. I employed the terms below imprecisely in describing related work because usage of them is overloaded in the literature. Henceforth they will refer to specific operations and logical units within *Xest*'s architecture.

An **object** is an unstructured variable length byte extent managed by the storage system. *Xest* objects correspond to complete files and directories, but are also used to store access statistics as described in Section 4.2.1. A **group** has a globally-unique identifier and is a logical set of one or more objects. The set of all objects forms a surjection on to the set of all groups. A group carries its own metadata together with that of its constituent objects.

A **volume** delimits the set of groups (and hence objects) on to which a single, conventional, hierarchical file system is imposed. Volumes also delimit the scope of the regrouping process, in that no relationships are inferred between objects in different volumes.

An **active volume** is a volume that may be currently read and written to at a single node. A **snapshot** operation on an active volume creates an identical read-only copy termed a **snapshot volume**, or simply a **snapshot**. Similarly, an active volume becomes a snapshot when it is unmounted. This process is called **fixation**, and allows the volume and its constituent groups to be accessible in a read-only fashion to other nodes. A **fork** operation on

a snapshot creates a new, identical active volume using copy-on-write techniques.[1] Both fork and snapshot operations are implemented by **cloning**, creating identical copies of a volume's metadata. Likewise, the copy-on-write cycle of fixation and cloning applies also to group metadata, object metadata and data.

A **virtual disk** is a unit of administration that consists of one linear, contiguous sequence of snapshots, possibly concluded by a single active volume. Intuitively, a virtual disk captures the historical state of a file system since it was created or forked from its parent. Each volume is associated with exactly one virtual disk. Unlike volumes, groups and objects, virtual disks are globally writeable and are named using the same identifier despite modification.

Recall that virtual disks were described as 'isolated' and nomadic', in that they are only accessible to a single writer at a time yet remain transparently available despite migration. In fact, these properties do not pertain to a virtual disk itself, but rather to the sequence of active volumes that constitute the most recent representation of that virtual disk's associated file system.

3.2 An overview of the *Xest* prototype

This section elaborates on the relationships between *Xest*'s components. The structure of the *Xest* prototype is illustrated in Figure 3.1. It comprises both a network-facing service and a local file system, disjoint except at a few interfaces: this is a consequence of loosening the relationship between file system operations and network interaction.

Although this work concerns the storage requirements of distributed service delivery platforms, and in particular those of the XenoServer project, it focuses on the challenges of managing storage between nodes rather than between virtual machines at a node. Integrating file system level extensions into the XenoServer model is the subject of ongoing work [Kotsovinos04, Williamson05]. Since the prototype is not constrained to any particular client interface, it has been developed entirely as a user-level process to enable 'pluggable' clients for different settings, an approach that has also simplified development and debugging. Currently, it may be accessed as a VFS file system using FUSE [Still04], and for wide-area evaluation using the trace replay framework that I describe in Chapter 5; this evaluation is conducted over PlanetLab [Chun03], which prohibits use of kernel modules or modification.

Xest uses a Distributed Hash Table (DHT) to store addressing data with which to locate sources from which to transfer groups. The scope of this work necessitated that I use an es-

[1] Active volumes are comparable with Episode's 'read-write filesets', WAFL's 'active file systems' and Parallax's 'cluster virtual disks'; snapshot volumes with Episode's 'fileset clones' and WAFL and Parallax's snapshots [Hitz94, Chutani92, Warfield05].

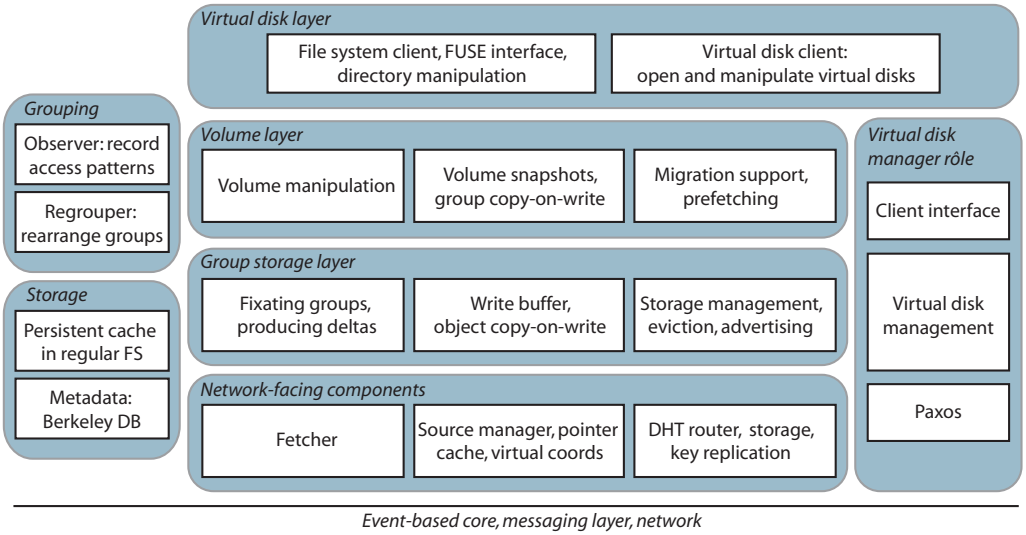


Figure 3.1: The structure of the *Xest* prototype.

tablished project as the basis for this functionality. The Bamboo DHT [Rhea04, Rhea05b] is itself the subject of a doctoral dissertation and has a mature, open-source implementation that has been deployed continuously on PlanetLab for over a year. *Xest* incorporates portions of Bamboo's DHT, event-based core and messaging components into its own code base.

Because Bamboo is developed in Java, *Xest* adopts it also. While user-level Java implementations of network services are widespread, it is an uncommon choice for a file system. I aim to justify this approach by showing that the associated costs are reasonable for a prototype implementation. To this end, Sections 3.2.3 and 3.2.4 describe the prototype's programming model and its network and disk subsystems.

3.2.1 File system components

Xest's file system components decompose access to a virtual disk through a conventional interface into operations on an object store organised into groups.

The **group storage layer**, described in Section 3.4, manages the persistent caching and storage of groups of objects and their metadata. It satisfies requests on objects and initiates fetches for their groups as needed, mediating the interaction of data retrieval with the fulfilment of ongoing reads, writes, and object clone operations. In addition, it manages eviction from the cache and uses the DHT to maintain pointers advertising its contents.

On top of this, the **volume layer** (Section 3.3) organises groups into volumes, manages the associated metadata and implements the copy-on-write logic that allows volumes to be created, snapshotted and forked. It exports an interface in which storage is addressed by

(volume identifier, index of group in volume, object identifier) triples, and addresses the group storage layer using (group identifier, object identifier) pairs. It conducts prefetching but passes operations on writeable object data through to the group storage layer.

The **grouping components** coordinate with the volume layer to implement the functionality described in Chapter 4. The **observer** is notified by the object file system of accesses to objects in an active volume, and uses it in turn to write back accumulated access statistics; it also determines the assignment of new objects to groups. The **regrouper** uses the observer's statistics to compute new groupings, applies them to new snapshots using the group file system and the client interface.

The **virtual disk layer** presents the abstraction of conventional, mountable file system hierarchies. Unlike lower layers, it logically spans the global storage system, and comprises the following two components.

Its **file system client** exposes a VFS-like interface for data-plane access to *Xest* volumes by managing the abstraction of a conventional hierarchy of directories, files and symbolic links over a volume's groups and their objects. It contains logic for interfacing with the FUSE library[2] and the trace replay framework described in Section 5.4.

In addition, a **virtual disk client** (Section 3.6) complements the file system client by providing a control-plane interface that clients use to locate, inspect, lease and mount volumes through their virtual disks. It issues requests to a virtual disk's managers, described below, which it locates by looking up the virtual disk's metadata in the DHT.

3.2.2 Network-facing components[3]

Xest's network-facing components maintain connectivity between nodes, facilitate data-plane transfers of file system content, and regulate control-plane manipulation of virtual disks.

The **DHT** (Section 3.5.1) is used to publish and retrieve addressing information for groups and volumes, and to cache the state of virtual disks. It comprises components from Bamboo that maintain a node's routing tables, forward lookup and insertion requests towards their key, and replicate them over the node's leaf set. *Xest* adds logic to store and deliver different types of pointer and reduces bandwidth usage by batching pointer insertions together.

The **source manager** (Section 3.5.2) uses the DHT to retrieve group location pointers and maintain a cache of available sources together with accrued liveness, latency and band-

[2] This component includes the FUSE-J bindings by Peter Levart and other contributors. Available at http://www.select-tech.si/fuse/.

[3] The Bamboo and Vivaldi components are derived from open source implementations by Sean Rhea, Steven Czerwinski and other contributors. Available at http://www.bamboo-dht.org/.

width information. It includes an implementation of the Vivaldi synthetic coordinates system [Dabek04a] to provide latency estimates in the absence of observed round trips.

The **fetcher** (Sections 3.5.3 and 3.5.4) schedules and implements the transfer of data and metadata from other nodes. To do this it interacts with a number of components. Operating as a client it takes requests for whole groups from the group storage layer, detailed prioritisation information from the volume layer, and locates peers using the source manager. As a server it retrieves data from the group storage layer.

The **virtual disk manager** (Section 3.6) receives requests, made by the virtual disk clients of this and other nodes, that modify the state of virtual disks that this node has agreed to manage. It uses an implementation of the Paxos protocol [Lamport98] to agree metadata updates with a virtual disk's other managers while maintaining a consistent global view of its state. Managers also ensure durability of storage of the contents of virtual disks, and publish their metadata directly in the DHT.

It is instructive to consider the points at which each node's autonomy is bridged to elicit a global, connected storage substrate. While quota management, billing and security aspects of a public *Xest* deployment will necessitate some degree of administrative centralisation, the architecture of the system's storage components only loosely couples its nodes.

Each *Xest* node maintains connectivity with its peers in two narrow senses. First, the DHT dynamically monitors and replaces nodes in its routing tables. Second, a virtual disk's managers monitor each other, and agree to replace failed members by recourse to the DHT.

File system activity at each node is contained locally except to manipulate a virtual disk or when data requested is not available in the cache. As such, all communication is notionally pairwise, except in three instances: first, to locate a source for a group, second, to locate a manager for a virtual disk, and third, to agree an operation on a virtual disk. The first two operations use the DHT; the third uses the Paxos protocol between the virtual disk's managers.

3.2.3 Programming model

The bulk of a node's work involves the marshaling of blocking operations between disk, its local clients and a set of network peers. Because of this *Xest* uses asynchronous I/O and a single-threaded event-driven design in order to eschew the cost of thread creation and to allow performance to degrade gracefully under high load. Components that need to interface with blocking libraries or present a blocking interface themselves run in separate threads.

Xest inherits Bamboo's function-callback model [Rhea05a] which enforces type safety and supports function currying, itself reminiscent of libasync [Mazières01]. Two departures from this implementation are outlined here and evaluated in Section 6.1.2.

First, rather than use a single priority queue for all callbacks, those that are eligible to be issued immediately, usually the majority, are maintained in a lock-free queue. Appending to this has lower cost, reduces contention with interfaces to blocking libraries, and preserves the relative ordering of events inserted by the same thread.

Second, timer precision is particularly important to the trace replay framework described in Section 5.4.3, which requires it in order to reproduce bursty traffic accurately [Anderson04]. *Xest* improves event issue time by waking early and pre-spinning, counting CPU cycles until the allotted issue time. This sidesteps use of coarse granularity system timers and reduces the effect of process scheduling quantisation.

3.2.4 Messaging and local disk access

The **messaging subsystem** comprises two components. Bamboo's userspace message transport is built over UDP and provides reliability, congestion awareness and fine-grain control over timeouts and retransmission. *Xest* uses a separate TCP transport for transfers bound by throughput, not latency, which provides connection throttling, bandwidth estimation and efficient transfer from disk as described below.

The **disk subsystem** provides an asynchronous, stateless interface to data in a node's persistent cache, which is located on a conventional local file system and laid out into a hierarchy of directories by group identifier. The data of each *Xest* file system object is stored as a separate file. A pool of threads uses the usual, blocking file system interface to provide concurrent, safe access to the cache contents. Files are opened when a read or write request is received and are closed only after a period of disuse to reduce per-operation costs.

Berkeley DB [Olson99] is used to store metadata items conveniently and consistently with each other. A durable database table is used to record a journal of operations to provide consistency between metadata and the data store, with its regular file system semantics; however most operations do not yet have journalling support in the prototype described in this dissertation.

Memory copies are the largest performance overhead observed by the developers of Pond, also a Java userspace file system [Rhea03]. *Xest* minimises this overhead by using native buffers and conducting some reads and writes through memory mapped regions of cached files. It also uses the zero-copy sendfile() primitive to satisfy remote transfer requests, to derive deltas from files and vice versa. Side-stepping large allocations on the heap for most data-flow operations also reduces the load on the garbage collector, another performance

limitation cited by Rhea *et al.*. The delay between agreeing to transfer a region using sendfile() and the eventual dispatch of the message requires additional locking to serialize writes and cache eviction for these regions.

3.3 Volumes

The abstraction of the **volume** demarcates the set of groups whose objects constitute a single logical file system instance. It is used to present to clients a conventional interface, a hierarchical structure, and an interface for performing whole file system snapshot and fork operations. Volumes construct this view from a storage substrate that partitions its namespace and management into groups, sets of objects transiently coassigned on account of their observed locality.

In this section I describe the role of the volume in *Xest*'s architecture and describe the structure of its metadata. I go on to demonstrate how volumes layer a hierarchical namespace over a set of objects organised by group, and how they implement copy-on-write.

3.3.1 The virtual disk layer and volume layer interface

Implementing a conventional VFS interface over a group store is a task that *Xest* separates into two components: the volume layer and, above it, the virtual disk layer.

The volume layer exports two principle interfaces whose operations are detailed below. The first provides operations to snapshot and fork whole volumes. The second exposes an object store interface over the volume which is intended to facilitate the construction of a hierarchical directory tree. The methods of the latter interface operate on a volume's objects directly, with minimal recourse to the groups by which they are organised.

The virtual disk layer builds on both of these interfaces. First, the virtual disk client logically extends the manipulation of whole volumes to their corresponding virtual disks, as described in Section 3.6. Second, its file system client provides the 'glue' to implement the structure of a volume's directory hierarchy. It exports a VFS interface over a volume's object store by maintaining directory entries and metadata, using locking to ensure the consistency of operations over multiple entries.[4] It also provides hooks to enable the volume layer to modify directory entries due to copy-on-write or group reallocation, as Section 3.3.4 explains.

[4] 'Locking' is needed when critical sections span callbacks. Locks prevent callbacks interleaving by admitting them on a multiple reader single writer basis; they do not block the event thread.

I proceed to describe the volume layer's interfaces before explaining their implementation in the remainder of this section. In the pseudocode below, the symbol ←-- denotes return values passed over a callback; error and status return values are omitted.

Operations on whole volumes:

$\mathsf{hnd}_{\mathsf{new\_vol}}$ ←-- createVolume ($\mathsf{id}_{\mathsf{new\_vol}}$, $\mathsf{id}_{\mathsf{new\_vd}}$)

Creates a new initially empty active volume, under the specified identifier and in the specified virtual disk, and return a handle on it.

$\mathsf{hnd}_{\mathsf{vol}}$ ←-- openVolumeRO ($\mathsf{id}_{\mathsf{vol}}$)

Opens a snapshot volume that exists under the specified identifier and returns a read-only handle for it.

$\mathsf{hnd}_{\mathsf{new\_vol}}$ ←-- forkVolume ($\mathsf{id}_{\mathsf{vol}}$, prefetch, $\mathsf{id}_{\mathsf{new\_vol}}$, $\mathsf{id}_{\mathsf{new\_vd}}$)

Forks the snapshot volume specified by $\mathsf{id}_{\mathsf{vol}}$ to create an identical clone under $\mathsf{id}_{\mathsf{new\_vol}}$ and returns a handle on the resulting active volume. The virtual disk parameter determines the location of the new active volume; it may be the same as the source, under the conditions described in Section 3.6.1.

() ←-- closeVolume ($\mathsf{hnd}_{\mathsf{vol}}$, $\mathsf{id}_{\mathsf{regrp\_vol}}$)

Closes the volume associated with the specified handle, fixating it as a snapshot if it is an active volume.

($\mathsf{id}_{\mathsf{snap\_vol}}$, $\mathsf{time}_{\mathsf{snap}}$, $\mathsf{hash}_{\mathsf{snap}}$) ←-- snapshot ($\mathsf{hnd}_{\mathsf{vol}}$, $\mathsf{id}_{\mathsf{snap\_vol}}$)

Creates a new snapshot of the specified active volume under the given identifier. When it completes, the call returns its identifier, creation time and the contents hash of its volume metadata. Only a single new snapshot volume can be created at once: overlapping calls wait and return the identifier of the single resulting new snapshot.

Operations on a volume's objects:

($\mathsf{index}_{\mathsf{grp}}$, obj) ←-- createObj ($\mathsf{hnd}_{\mathsf{vol}}$, isDir, backPtr, $\{(\mathsf{k}_1, \mathsf{v}_1), (\mathsf{k}_2, \mathsf{v}_2), \ldots\}$))

Creates a new object in the specified active volume. The object's metadata is placed in a group chosen as described in Section 4.2.4, but initially no data is attached to it. The pair in the callback identifies the object within the volume; its format is detailed in the subsequent section. The *backpointer* identifies the linking directory entry; if it is not specified, an object is created under a well-known label suitable for use as a root directory. The initial attributes of the object are also specified, as described below.

() ←-- setObjLinks ($\mathsf{hnd}_{\mathsf{vol}}$, $\mathsf{index}_{\mathsf{grp}}$, obj, $\mathsf{backPtr}_{\mathsf{link}}$, $\mathsf{backPtr}_{\mathsf{unlink}}$)

Adjusts the link profile of an object in the specified active volume. Any backpointer specified in $\mathsf{backPtr}_{\mathsf{link}}$ is added, and any in $\mathsf{backPtr}_{\mathsf{unlink}}$ is removed. By specifying

one or other or both, this method is used to implement VFS link, unlink and atomic rename operations.

$\{v_1, v_2, \ldots\}$ ←-- getObjAttrs (hnd_{vol}, $index_{grp}$, obj, $\{k_1, k_2, \ldots\}$)
() ←-- setObjAttrs (hnd_{vol}, $index_{grp}$, obj, $\{(k_1, v_1), (k_2, v_2), \ldots\}$)

Atomically retrieves or sets a group of metadata attributes for an object in the specified active volume. The volume layer supports extended attributes specified as (key, value) pairs; the virtual disk layer treats UNIX mode, uid and gid as special cases of these.

() ←-- setObjLength (hnd_{vol}, $index_{grp}$, obj, length)

Truncates or extends an object in the specified active volume. Although this is a metadata operation, the length attributed is treated differently from others as it is also adjusted by write operations.

hnd_{obj} ←-- openObj (hnd_{vol}, $index_{grp}$, obj, forWriting, noFetch)
() ←-- closeObj (hnd_{obj})

Opens an object in the specified active volume for data access and return a handle on it, performing prefetching or copy-on-write according to the flags specified. The open and close methods increment and decrement the specified object's 'open' reference count so as to ensure that it is not removed while still open if its link count falls to zero.

buffer ←-- readObj (hnd_{obj}, offset, length)
bytes_written ←-- writeObj (hnd_{obj}, offset, buffer)

Reads or writes an open object; operations are redirected to the group storage layer.

3.3.2 An overview of file system metadata

Each volume has an associated metadata object and is named by a globally unique identifier in the same identifier space as groups and virtual disks are.[5]

A volume's metadata details the set of groups that make up its file system hierarchy, and is described in detail in the following section. Volume metadata is opaque to the group storage layer, but is treated identically to group metadata for the purposes of storage, cache advertisement and retrieval.

Groups serve two distinct rôles at different logical levels. They are the granularity at which the group storage layer manages fetching, cache advertisement and eviction. These tasks are described later in this chapter. At the volume layer, group metadata then object data

[5] Identifiers are in the range $[0, 2^{160})$ in the *Xest* prototype.

form second and third tiers below volume metadata in a copy-on-write hierarchy used to implement whole file system snapshot and fork. I explain this functionality in Section 3.3.4.

Recall that a group's metadata also comprises the metadata of its objects. Storing, accessing and transferring it as a whole mitigates the difficulty that both types of metadata tend to be considerably smaller than a disk block. Since a group is intended to comprise only high locality objects, the loss in precision is likely to be limited.

In addition, arranging a group's objects' metadata together means that they can be addressed relative to the group. While groups have globally unique identifiers, relatively addressed objects need not. I describe an object's identifier, without at present extending its scope beyond its group, as a **label**. I elaborate on the management of object labels in the remainder of this section.

3.3.3 Addressing file system hierarchies

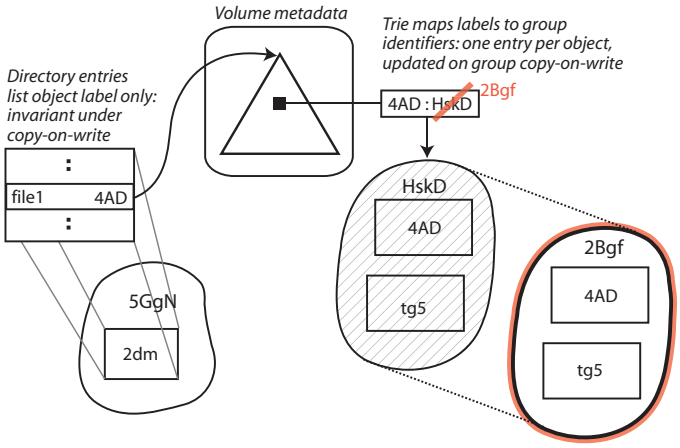
The primary purpose of volume metadata is to provide a means of organising a set of groups, and hence objects, into a hierarchical file system structure. In doing so, the dominant consideration is to ensure that a volume's metadata scales well regardless of file system size or locality of grouping. While metadata proportional to the number of groups may assist in associating a volume with its groups, those groups themselves identify objects, and each group contains many objects.

Scalability of metadata is important for several reasons. As will be seen, the metadata is involved in the control path of most operations on the file system that it encloses, and significant performance benefits are obtained from being able to cache it in memory. In addition, fork and snapshots on a volume's file system must block while the metadata is being cloned; opening the file system may also require the metadata to be fetched from another node.

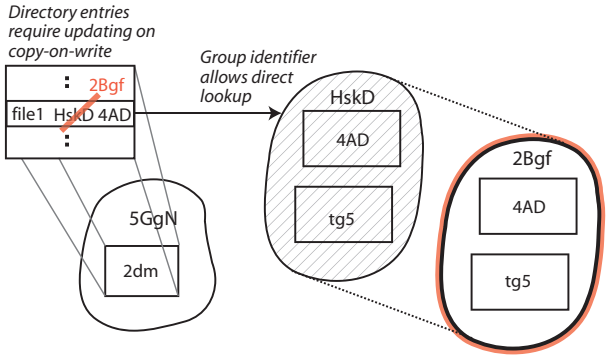
Xest allocates each file and directory as a separate object to create a POSIX-compliant hierarchical directory structure. The prototype implements directories as a list of entries that associate textual names with a link to a directory or file, whose format is discussed below.[6] The volume metadata maintains a reference to the root; this reference has a structure identical to a regular directory entry's link.

Within a volume, the set of directory entries form a surjective mapping on to the volume's set of objects, just as they do on to inodes in conventional file systems. Directory entries need to distinguish only those objects associated with the volume. However, the underlying storage substrate presents a group-centric interface that requires objects to be identified using their groups' identifiers. Moreover, as I explain in the next section, groups' and ob-

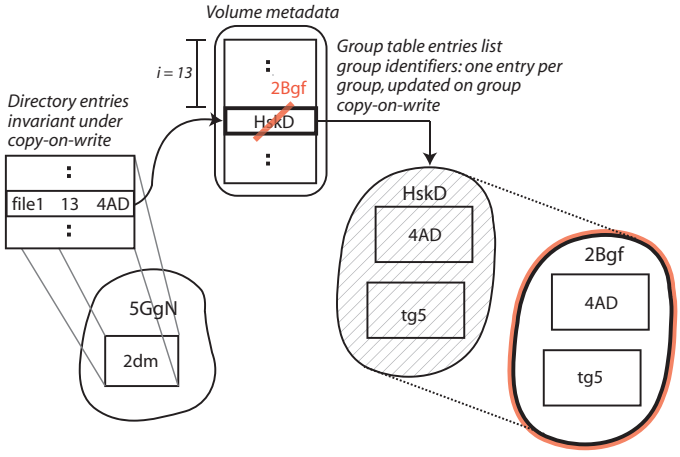
[6] Symbolic links maintain their targets inline, inside the directory entry.



(a) Using a trie in the volume metadata to lookup group identifiers



(b) Embedding group identifiers directly in directory entries



(c) Indexing into a table of group identifiers in the volume metadata

Figure 3.2: Approaches to addressing a hierarchical file system within a volume. Directory entries must identify their target object's group, but group identifiers change under copy-on-write. See the text for a comparative discussion. Groups and their objects are represented as rectangles inside ovals, with identifiers given. The exploded rectangles depict the contents of a directory object. Cross-hatched groups have undergone copy-on-write to new groups with new identifiers, as shown by the dashed lines.

jects' identifiers may need to change as they undergo copy-on-write. I proceed by exploring several schemes for performing this addressing, and examine their impact on the structure of metadata and storage management.

I consider first the case where directory entries contain only object labels, shown as Figure 3.2 (a). The volume metadata would need to contain a mapping to each object's corresponding group; this map would also be used when allocating new labels to maintain their uniqueness across the volume. An object's label may remain fixed for its lifetime: on copy-on-write or after group reassignment, the map entries would simply be updated to point to new groups.

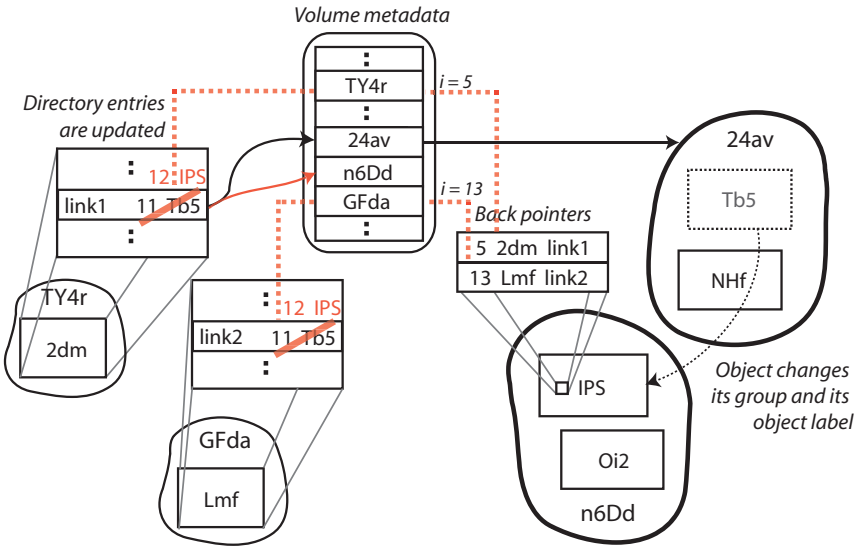
Unfortunately, volume metadata would scale poorly in this arrangement. The map or trie is proportional in size to the number of objects in the file system. Also, it would have to list a group identifier in each value, one per object, or else values would have to be indirected to a canonical table of identifiers, in addition to the map.

In the second case that I consider, illustrated in Figure 3.2 (b), both the object's label and its group's identifier are embedded in directory entries, and the volume metadata contains only a similar pair to identify the root directory. For now, assume that a separate mechanism ensures the uniqueness of object labels within a volume, as detailed below, so that labels may remain constant for the object's lifetime. In this arrangement, when a group is cloned to a new identifier during a copy-on-write operation, directory entries pointing to its objects must be repaired. This is a relatively frequent occurrence and since it affects all links to all of the group's objects, a considerable number of directories may be involved.

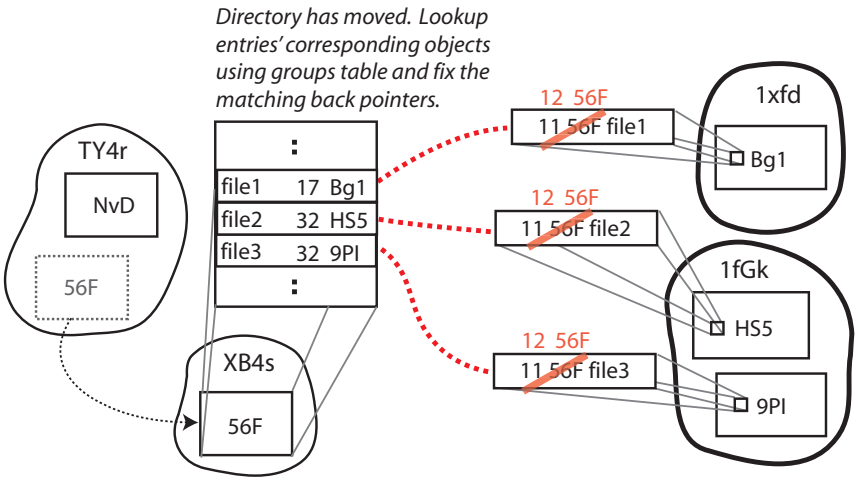
To avoid this cost, a third arrangement, Figure 3.2 (c), introduces indirection of group identifiers using a table in the volume metadata, similar to the canonical lookup table considered above. Directory entries are (i_g, o) pairs, where i_g denotes that the **group table** at index i lists the identifier of group g. This group contains object o. Only the volume metadata now needs updating when a group undergoes copy-on-write.

Nevertheless, group reallocation may still require two categories of repair to be made to the directory structure, as illustrated in Figure 3.3. First, an object moving into a different group will result in referring directory entries listing an incorrect group table index. In order to update these efficiently, the object's metadata lists its **back pointers**, which detail the directory entries that refers to the object. Each back pointer contains the group index and object label of the directory object, and the textual name of the entry in that directory.

Figure 3.3 (a) shows how these back pointers are enumerated to locate each directory entry to be updated. First the identifier of the group containing the directory object is looked up in the groups table, the object with matching label located inside that group, then the directory entry found with matching textual name.



(a) Repairing directory entries using a relocated object's back pointers



(b) Repairing back pointers using a relocated directory's entries. Objects are located, as usual, using the volume metadata's groups table, not shown in this figure.

Figure 3.3: Maintaining directory entries and back pointers

The second repair process arises when a directory is moved to a new group. *Xest* must then update the appropriate back pointer for each object that is referred to by an entry in that directory, as illustrated in Figure 3.3 (b). Each directory entry is examined and its target object's metadata located, looking up the group identifier in the groups table. The appropriate back pointer is located for each object using the directory's old identifier pair and is updated.

Since group reallocation is a relatively infrequent process that operates on considerable portions of a volume's groups at once, there is significant scope for batching updates to

minimise the cost of searching directories and repairing these structures. I discuss this further in Section 4.3.4.

When a group is to be removed, a situation that arises if all of its objects' link counts fall to zero and are closed, its entry needs removing from the groups table. The table is allowed to become fragmented and a free bitmap is maintained, rather than modify the index of remaining groups, which would require the repair of directory entries. Instead, group reallocation uses free entries first, and can compact the table in pathological cases of very high fragmentation with minimal additional disruption.

It is now instructive to expand on the enforcement of object label uniqueness across whole volumes. Without a map as described above, this guarantee requires the maintenance in the volume metadata of a bitmap or other structure detailing assigned labels, at best proportional in size to the number of objects in the file system. This markedly constrains the metadata's scalability. In addition, object labels must have a range sufficient to identify all objects in a volume. This extra storage itself contributes to the size of the volume metadata or of the objects that represent directories.

While object labels being unique within entire volumes yields some simplifications in storage management, these are small. Consider the case in which object labels are unique only within a group, trivially enforceable by the grouping components. Directory entries can still uniquely identify objects with a pair (i_g, o), but after group reassignment, some of the object labels in directory entries may need updating. However, these are the same entries whose group identifiers will also need updating, and which are located using the back pointers already maintained in the object metadata. Updating the object label in addition to the group identifier represents a negligible additional cost. This approach also means that labels need consume less space, since they are required only to address the maximum capacity of a group, not a volume.

Xest adopts this final model because of its particular trade-off between simplicity of storage management and scalability of volume metadata. To summarise, each volume's metadata contains a group table. Directory entries uniquely identify objects with a pair (i_g, o), where i_g denotes an index into the group table. Object labels are unique only within a group. Object metadata retains back pointers to identify directory entries whose group table index and object label may need updating after an object is assigned to a different group.

3.3.4 Copy-on-write mechanisms

Copy-on-write is conducted at three distinct logical levels: first, on volume metadata, to facilitate the snapshotting of complete client file systems; second, on the metadata of a whole group, and by implication the metadata of its objects; and third, on each individual object's

data.The metadata of a group and that of its constituent objects are not differentiated for copy-on-write purposes on the grounds of their good locality, small size, and their being stored together. On the other hand, the cost of cloning an object's data may be significantly higher.

Cloning object data in its entirety is appropriate because the mean file size is within an order of magnitude of the size of both a disk block and the metadata, and most write operations tend to be complete rewrites [Vogels99]. It simplifies the structure of the metadata and does not unnecessarily block reads or writes, as described in Section 3.4.1.

Before proceeding I clarify the relationship between volumes, groups and objects, and in doing so introduce some terminology. A group is said to be **local** to the single volume in which its metadata (or equivalently the metadata of a constituent object) was last modified. Similarly, an object is said to be **local** to the single group by which its data was last modified, and hence also to that group's local volume.

Note that each volume's set of local groups are disjoint. Consider that a tree of 'copy-on-write ancestors' may be created by fixating a volume and subsequently forking it, and repeating this process. A volume's groups are either local to it or to one of its copy-on-write ancestors, as group reallocation is never performed across volumes. An analogous pattern applies for objects within groups.

Figure 3.4 illustrates how these components and the 'local' relationship is used to determine an object's write status. Each entry in a volume's group table is augmented with a flag to indicate whether that group is local to the volume. Similarly, each object's metadata contains a flag set when the object is local to that group. This relationship is also used to determine storage ownership between snapshots, as described in Section 3.6.4.

Only **concrete references**, identifier pairs (g, o) for which each object o is local to some corresponding group g, are backed in any node's persistent cache with that object's data. Since the grouping layer's copy-on-write logic is isolated from the local storage layer's tasks, the interface between the two components names objects using only concrete references. As such, an object that is non-local to a group g must also be a pointer: it contains in addition to its flag a concrete reference to its data in its local group g_l. Further, consider that group reassignment may have necessitated a change of object label; the metadata also contains the previous label o_l, and hence the concrete reference is the pair (g_l, o_l). Even when an object is local to the group, this pair is sometimes retained to indicate its copy-on-write source to which to apply a delta, as described in Section 3.4.2.

In fact, an optimisation is possible that eliminates the duplication of storage within the metadata of group identifiers, which are 20 bytes in size. Observe that when a group is cloned, the same group identifier is installed in each previously local object's metadata.

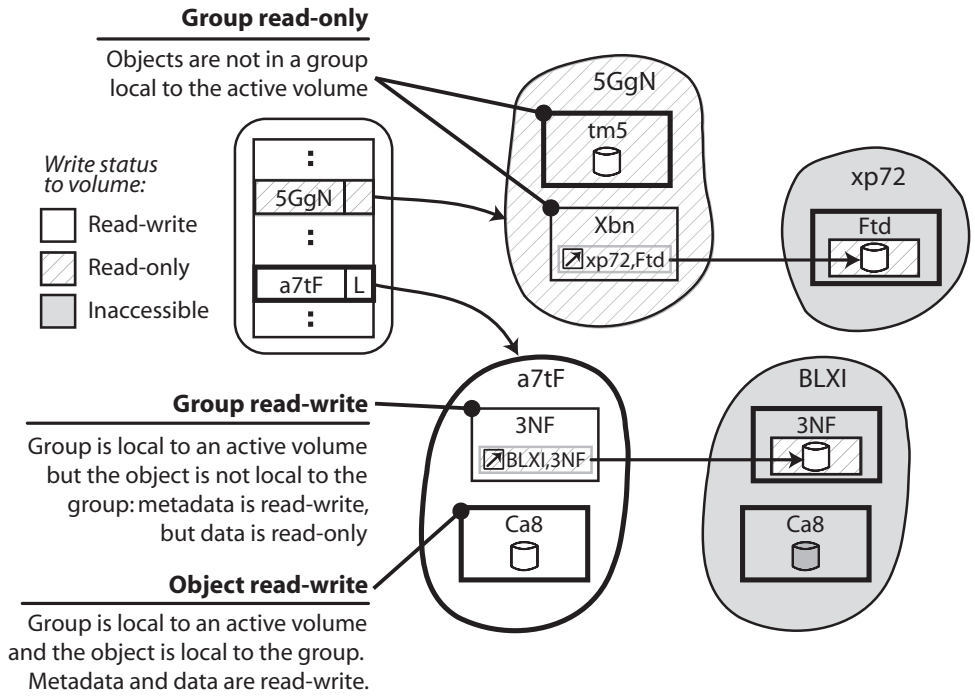


Figure 3.4: Determining the write status of an object for an active volume. An object local to its group has an emphasised border; a group local to the volume likewise. Cylinder symbols indicate the presence of associated object data. Objects (a7tF,3NF) and (5GgN,Xbn) are both pointers, and only their targets' data, not its metadata, is accessible. Observe that the metadata of (a7tF,3NF) is read-write because group a7tF is local to the volume, as marked in its group table entry.

To avoid this, the identifier is placed instead in a table that is maintained in the group metadata, and each object's metadata refers to the table entry's index. After a sequence of copy-on-writes and modification of objects' data, the set of groups for which its objects are local remains a subset of the group's clone history. A reference count is maintained with each entry and is decremented if copy-on-write subsequently makes a referring object's data local once more; the entry is removed when the count falls to zero.

Manipulating a volume's overlying file system involves determining and, if necessary, modifying the write status of one or more objects. With reference to Figure 3.5, this proceeds as follows:

- **Volume read-only.** If the volume is not active, then its contents cannot be modified; reads to its proceed as described in the subsequent bullets.

  In order to modify the hierarchical file system contained in a snapshot volume, a new active volume must be forked by cloning its volume metadata to a new identifier; the metadata is duplicated but all previously local entries in the group table are set

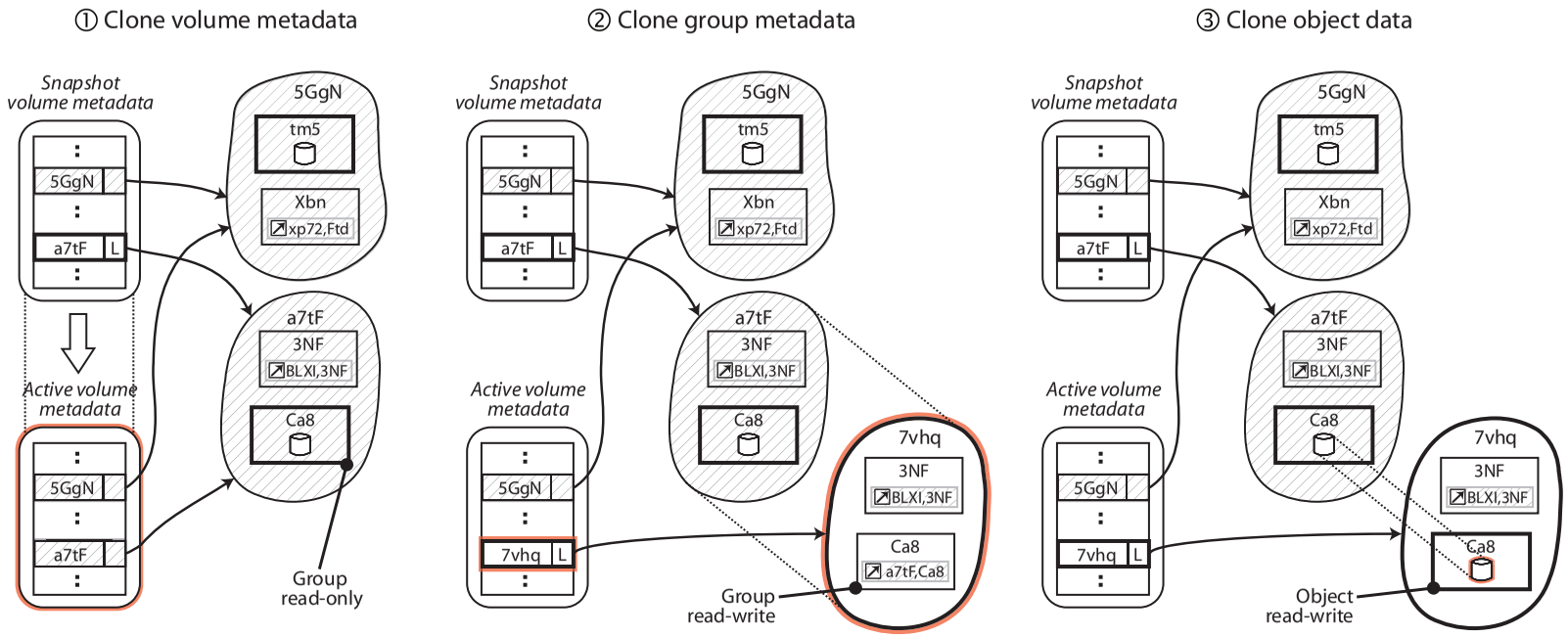


Figure 3.5: Copy-on-write of an object in a volume. An object local to its group is shown with an emphasised border; a group local to the active volume likewise. When the active volume is forked by cloning its metadata, shown in Step 1, its groups are marked non-local and so are read-only. In Step 2, a group's metadata undergoes copy-on-write, and the new identifier is inserted in the corresponding table entry and marked as local; the group's objects' metadata is read-write. In Step 3, the data of an object in the group is cloned and becomes read-write.

to non-local. In addition, the new active volume must be associated with a suitable virtual disk, as described in Section 3.6.3.

- **Group read-only.** The volume may be active, but if its group table marks an object's group as non-local, the group's metadata and its objects' data and metadata are read-only. Reads to the object's metadata proceed as normal; reads to its data proceed according to the two bullets below.

  Before data or metadata can be modified, the group metadata must be cloned to a new identifier. During the clone, all objects previously marked as local become non-local; since the data itself is not being cloned, access to them is redirected by installing the objects' former group's identifier in their metadata. Finally, the group table entry is marked as local and updated to point to the new group metadata.

- **Object read-only.** If an object's group g is read-write but its data has not yet been modified in this active volume, the object metadata is marked as non-local and contains a pointer to its local group, and the object's label in that group, the pair (g_l, o_l). Reads to the data proceed under that concrete reference, but a write necessitates duplicating the underlying data object; this third form of copy-on-write operation is detailed in Section 3.4.1.

  Note that, somewhat counter-intuitively, because the object's metadata is not differentiated from that of its group for the purposes of copy-on-write, metadata updates to the object, including truncation, can proceed without cloning the data, and hence without the object being local to the group, so long as the group is local to the volume.

- **Object read-write.** The object's group g is local and the object o is local; the data is accessible through the concrete reference (g, o) and is read-write for this active volume.

3.4 Group storage management

Each *Xest* node caches and maintains objects' data in a unified store organised by group within a regular local file system, and records group metadata separately in a database table. The group storage layer maintains this storage through three key activities. First, it mediates the accessing, fetching and cloning of data, as well as the management of group metadata. Second, it processes previously writeable groups whose local volumes have become snapshots to hash their contents and consider generating a delta with their previous copy-on-write source. Third, it performs background management of the store, including cache eviction and advertising its contents to other nodes.

Recall that the storage layer deals only with concrete references, objects whose data is associated with its metadata, or which are zero bytes in length. All such objects are read-only, unless they are local to a group that is itself local to an active volume, at the node at which that volume was created. With the exception of the cases described in Section 3.6.4, whereas read-only objects are cached, read-write objects are stored and so cannot be evicted.

To assist in the management of local storage, group metadata also has a 'node local' component, which is stripped whenever the metadata is transferred. It is used to distinguish the storage status of groups, indicate the presence in the cache of a group's objects, maintain usage data to assist in cache eviction and store details of deltas, as described below.

3.4.1 Reading and writing data

The group storage layer's interface exposes the following storage operations:

buffer ←-- readObj (id_{grp}, id_{obj}, offset, length)
bytes_written ←-- writeObj (id_{grp}, id_{obj}, offset, buffer)
() ←-- truncateObj (id_{grp}, id_{obj}, length)
: Reads from, writes to, or truncates the specified object. Reads will issue fetches as necessary and block until data to cover the region is available; write and truncate operations complete immediately, as detailed below.

() ←-- cloneObj (id_{grp}, id_{obj}, id_{dst_grp}, id_{dst_obj})
: Clones the data of the specified object to its new handle. Called by the volume layer to enact copy-on-write, which corrects object metadata and write status after this operation has completed.

(status, sendFileSrc) ←-- sendIfPresent (id_{grp}, id_{obj}, objOrDelta, offset, length)
: Returns a handle on a sendfile() operation whose source will be the specified region from the data or delta of the specified object, if it is available in the local store. If not, return details of the object's group's storage status and do not attempt to fetch it. This method is used only by the fetcher, whereas the methods above are used only by the volume layer.

Although the interface is straightforward, the implementations of its read and write operations are complicated by interactions with copy-on-writes and data retrieval, coupled with a necessity to minimise latency whereever possible.

Exploiting read parallelism is particularly important to mitigate the effect of slow networks on object transfers. When a read is performed on a read-only object, that object may not exist in the cache. If no fetch is in progress for its group, one is issued immediately and the read blocks. Otherwise, the read may proceed if the requested region is covered entirely

by the parts of the object already received. Each newly received portion of an object tests to see whether it allows any reads to complete. Since the object is read-only under that identifier, all operations commute.

Although a read-write object at a node is part of an active volume that was created at the same node and cannot be evicted, its contents may be unavailable to an operation issued on it. The copy-on-write that creates it may be in progress, or, as in most cases, may itself be blocked waiting for its read-only source object to be retrieved.

Observe that blocking a write or truncate operation until after the fetch and copy-on-write complete will cause some data just cloned to be immediately overwritten. Instead, *Xest* maintains a write buffer for the object, to which writes are made so they may complete immediately, and a corresponding map in node local metadata that records the regions written and condenses their description to a minimal disjoint set. When the fetch completes, the clone merges that object with the write buffer.

In addition, the written regions map can be used to determine whether reads can be satisfied from the write buffer at the time of their issue, rather than block waiting for a fetch preceding a copy-on-write to complete. However care must be taken to ensure that the relative ordering of reads and writes on the object is preserved. When a read must block, the portions of it that are satisfiable from the write buffer at that time are copied into the appropriate positions in the operation's buffer, since those portions of the write buffer may be subsequently overwritten. The remainder is filled in after the fetch completes from the copy-on-write's source, not its target.

Allowing write and truncate operations to complete while a fetch preceding a copy-on-write is in progress yields information about data it may no longer be necessary to retrieve; the benefit is particularly marked in the common case of an object being truncated to zero bytes immediately after it is opened. If the object was not already being retrieved when the copy-on-write was issued, then retrieval of regions covered by the write buffer are set to lowest priority.

3.4.2 Fixating groups and producing deltas

When an active volume is snapshotted or closed, each group that is local to it undergoes a fixation process in preparation for its metadata and its objects being advertised to and accessible from other nodes. This process computes hashes over object data, considers producing deltas for objects, adjusts the group's cache status, and inserts pointers into the DHT. Although fixation is typically a low-priority background activity, when a virtual disk is undergoing migration additional techniques are used to facilitate continuity of access, as described in Section 3.5.5.

In all cases, the secure hash over the data of each of the group's objects' is computed and stored in the metadata. If an object's length exceeds the size of the fetcher's transmission chunk, a digest is computed for each chunk's contents. These hashes are used to verify the integrity of data as it is received (Section 3.5.3).

If the group was created as the result of a copy-on-write, *Xest* considers creating a delta for each pre-existing object which has been modified that records the differences between the previous version and this version of the object. This aims to mitigate the effect of conducting copy-on-write at the granularity of whole objects; by trading off local disk space in which to store a delta, it reduces the retrieval time with which other nodes with the previous version can reproduce the new version.

Recall that the written regions map for an object is created then maintained while a fetch then copy-on-write progresses. In fact, it is also retained subsequently, while the number of bytes preserved from previous versions exceed a certain proportion of its size. When this property no longer holds, the map is discarded. If on fixation the map still exists, the regions that it lists are recorded in node local metadata are used to compile a delta cached locally on disk. In addition, the contents hash of the delta is added to the node local metadata to ensure its integrity during transfer.

Also, the pointer to the group to which the object is local before the copy-on-write is retained unless the map is discarded, now to identify the starting point of the delta. Before a node attempts to fetch a delta for the object in place of its data, it uses this pointer to determine whether it has the previous version to apply it to. Section 3.5.3 explains how the fetch process locates and uses deltas to speed the object's retrieval.

The final part of the fixation process sets the group's storage status to stored_sticky, as described in Section 3.6.4, and inserts a pointer into the DHT to advertise its availability, as described below.

3.4.3 Cache advertisement and eviction

A background process manages the contents of the cache by iterating over stored group and volume metadata, in effect maintaining a cyclic 'clock arm' on the store. It performs eviction if the store has reached a predetermined capacity, and reinserts pointers to cached contents into the DHT as necessary.

A **management epoch** is triggered probabilistically by every operation on a cached or stored group, or by a periodic timer while the system is idle. In each epoch, the arm advances a fixed number of times, except where this would cause the arm to reach a group it had already considered in that epoch, to avoid pathological cases for nearly empty caches.

Status	Objects present	Evictable	Advertisable	Read-write
cached_metadata*	None	✓		
cached_degraded*	Some	✓	(✓)	
cached_complete	All	✓	✓	
stored_sticky	All		✓	
stored_manager	All		✓	
stored_rw	All			✓

Table 3.1: A group or volume's storage status at a node. A volume never takes on those values marked with an asterisk (*).

Active volumes and groups local to them are read-write; once they are fixated as part of a snapshot, they become 'sticky' until stored persistently elsewhere, as described in Section 3.6.4. Groups for whose associated virtual disk this node is a manager are stored persistently. A cached group that is degraded has some objects that have not yet been fetched or have already been evicted; the conditions of their advertisement are described in the text. Volume metadata itself is always cached or stored in its entirety.

For each group or volume that the arm encounters, it first considers its metadata's node local storage status, the properties of whose values are summarised in Table 3.1. If an item is evictable, its details are added to a set of **eviction candidates**. If it is not, but it is advertisable, another node local field is checked to determine the time at which a pointer was last inserted into the DHT to advertise the group. A new pointer is inserted if the last is soon to expire or was never inserted.

If an item becomes an eviction candidate, reinsertion of its pointer is not considered until after it leaves the set, in case it is selected for eviction during that time. The cost to other nodes of following incorrect pointers outweighs the benefit of an additional, possibly available source.

The arm's idle advancement rate is adjusted dynamically with the aim that each group in the cache is considered several times during each period in which its pointer needs reinserting, although file system activity may cause it to iterate more rapidly. To ensure that pointer maintenance is a periodic, not reactive, process, insertions are rate limited (by time, as epochs have variable durations) to prevent the bandwidth that they consume interfering with time-sensitive fetches. A node with a full persistent cache restarted after some downtime may have had pointers expire for many thousands of groups. Pointer insertions for newly fixated groups and snapshot volumes join the same queue and are subject to the same rate limiting.

Node local usage information is maintained both for the group and volume metadata and, in the case of groups, for each object present in it. Each item of usage data comprises six bits. When an access to an object occurs, the top bit is set in both the group and object's usage fields. When the group leaves the eviction candidate set, its own usage field and those

of each of its objects are shifted right. Usage fields are interpreted as an integer to capture both frequency and recency of access. The absence of an object in the cache is denoted by setting the field to a value outside of its usual range.

Although the arm adds whole groups to be considered for cache eviction, *Xest* responds to the effect of aliasing within groups on cache hit rate by selectively reducing the granularity of cache management to individual objects. The eviction process aims to free space while retaining hot objects by **degrading** groups. It is inspired by Castro *et al.*'s HAC [Castro97], but takes account of the variable size of both groups and their objects. Whereas HAC has to compact the remaining hot objects into frames, in *Xest* the file system backing the cache manages fragmentation.

The scheme operates as follows. When a group is added to the set of candidates, a pair (t, c) is calculated for it, where t is the minimum usage value such that c, the number of bytes that are freed by evicting the group's objects whose usage is less than t, is greater than some proportion E of the total size of the group's objects currently cached. Intuitively, t determines the usage threshold that divides hot from cold objects in the group at the point that the space freed by evicting the cold objects justifies degrading it. Note that for a given group, c increases monotonically with t; the pair is found by a binary search over the range of usage values. To select an entry for eviction, the candidate from the set is chosen that has minimum t, and in the case of a tie, maximum c. It is removed from the set and, if it is a group, its cold objects are evicted along with any associated deltas.

The eviction of group metadata is considered on the basis of the value in its own usage field. Although none of a group's local objects may be cached, their metadata may be referred to and used as a read-only component of an active volume. To reflect this use, the top bit of a group's metadata's usage field is set whenever any of its object's metadata, local or not, is accessed. Recall that an object's metadata is treated as part of its group's metadata, so it is never removed separately.

Volumes and empty groups (ones with no local objects in the cache) enter the eviction set as usual, with t set to their metadata's usage value, and c set to zero or the size of their associated group tables, respectively. Further, consider a group's being selected for eviction with parameters (t, c) such that not of its objects are classed as hot and retained. In this case, the group metadata is retained only if its own usage is greater than or equal to t.

Candidates that are not selected for eviction are retained in the set for a fixed number of epochs, a trade off between the cost of arm advancement and the recency of candidates' access data. A new epoch is induced if eviction causes the set size to fall below a certain proportion of its steady state size to ensure that relatively hot groups are not degraded during bursty activity.

Although nodes advertise their caches in terms of groups, note that pointers are also inserted for degraded groups while the objects that they retain make up more than a certain proportion of their total bytes.

3.5 Group location and retrieval

In this section I describe how *Xest* nodes satisfy file system operations on groups that are not available locally, and how these components are organised for the purpose of minimising the completion delay on client requests.

Bamboo's DHT is adapted to manage the global storage substrate's addressing information, and a source manager uses that to determine a set of nearby, live peers from which required groups can be transferred. The fetch protocol schedules uploads and downloads with other peers and stripes transfers across multiple sources, taking account of priority information in order to mitigate the effects of poor groupings and unanticipated access patterns. As well as the prefetching implicit in the retrieval of whole groups at once, Section 3.5.5 describes how during migration explicit prefetches are chosen by the source node and used to warm the cache of the target node before resumption there.

3.5.1 Addressing using the DHT

Xest nodes operate a variant of the Bamboo DHT [Rhea04, Rhea05b] to provide a distributed repository of soft state addressing information. This section summarises how *Xest* uses the DHT and describes how I have adapted its implementation to fit *Xest*'s requirements.

As outlined in Section 3.2.2, the DHT stores two primary types of data:

- **Group pointers and volume metadata pointers.** (Section 3.4.3) A node advertises the contents of its local cache and store by periodically republishing soft-state pointers. Other nodes use these to locate nearby sources with which to perform transfers. Pointers contain the identity of the source and the storage status of the item at it.
- **Virtual disk metadata.** (Section 3.6.3) Each node that is a manager of a virtual disk maintains a soft-state copy of its metadata in the DHT, so that other nodes can use it to determine the virtual disk's set of managers and to examine its constituent volumes.

The rôle of this addressing data is further illustrated in Figure 3.6 in the context of a client's access to a file system object. Recall that groups, volumes and virtual disks all share the same identifier space as the DHT's key space, and addressing data is inserted directly using the appropriate identifier as the key. Inserted items have an associated expiry time

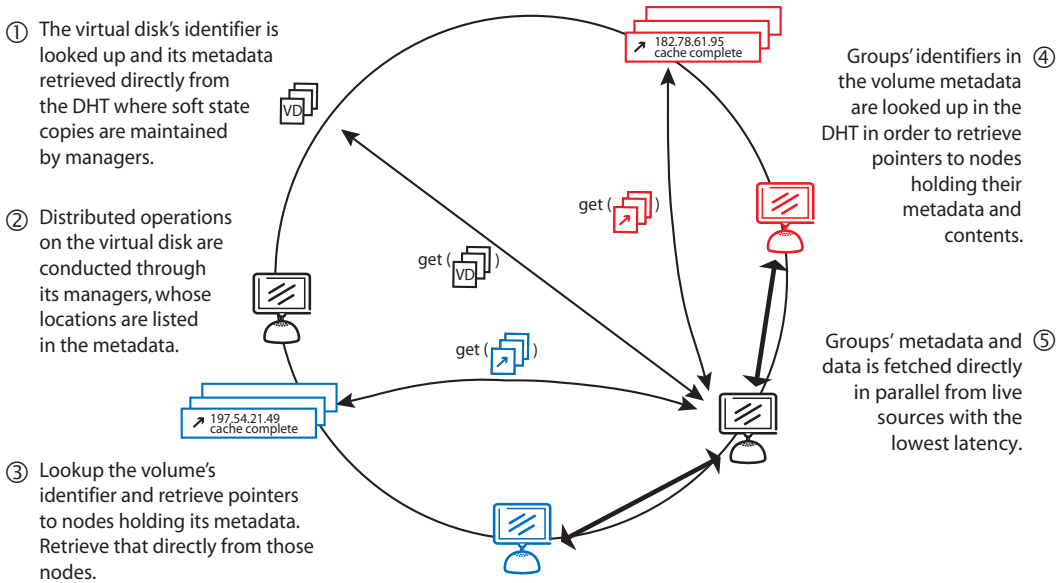


Figure 3.6: The rôle of the DHT in accessing file system data. The client, bottom right, first retrieves the virtual disk's metadata, then locates its managers, and looks up and fetches volume and group metadata.

and are unique with respect to their DHT key, their inserting node, and a type marker that distinguishes the DHT's various uses; this triple is termed the **item key**. Lookups specify the DHT key and the type marker and return all matching non-expired items. Unlike Bamboo, *Xest* does not record with each key a secure hash of its payload to distinguish different items under the same identifier. Instead, updated values can override previous versions. This also results in shorter keys.

For DHT lookups, *Xest*'s dominant concern is latency. Obtaining sources for a group and identifying a virtual disk's managers are on the critical paths to satisfying a local cache miss and mounting a volume, respectively. Items are replicated across the leaf set of the node whose identifier is numerically closest to that of the item. The pseudo-random distribution of node identifiers means that these nodes are likely to be diverse in location, and Bamboo's preferential selection of low latency forwarding paths tends to direct requests to the nearest such node.

In addition, each DHT lookup is duplicated at its source and sent to two distinct nodes to encourage it to be forwarded along two distinct paths, helping to reduce the dependence on nodes that may be heavily loaded or inaccessible, and reducing reply latency at the cost of an increase in messages [Rhea05c]. Further, *Xest* caches returned pointers and metadata, and issues new fetches to replace expired pointers for in-use groups as described in Section 3.5.2.

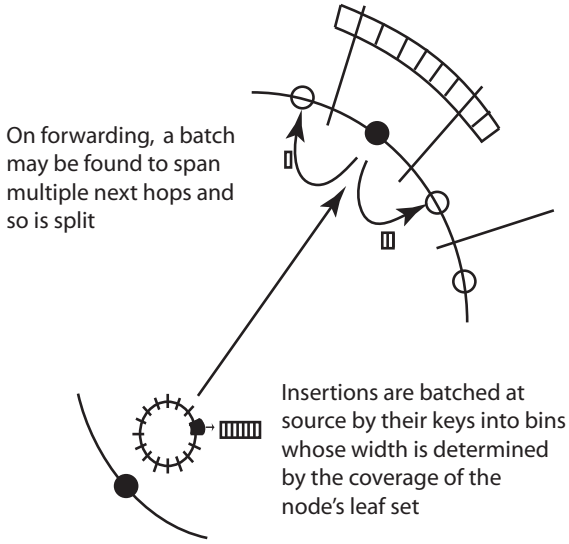


Figure 3.7: Routing batched requests through the DHT.

For pointer insertions, on the other hand, latency is unimportant, since addressing information has long expiry periods and is maintained by periodic processes. However constraining the impact on the network of pointer maintenance is a significant hurdle to the practicality of using DHTs for addressing, an issue that I introduced in Section 2.2.1.

Since most items are pointers and hence very few bytes in size, *Xest* uses batching, a technique widely applied in trading off messages sent for increased latency.[7] This amortises the overhead of IP and UDP headers and reduces the number of routing operations that must be made both by the underlying and overlay networks.

Batching is applied at each stage of the insertion process. First, DHT requests are gathered together by key and routed together as far as possible, as explained below. Then, the node nearest each inserted item's key forwards pointers to the members of their leaf sets as whole batches; the replies from each are also batched. Finally, having gathered sufficient replies from the leaf set, replies to the original inserter are also batched together.

Batching of insertions into the DHT operates as illustrated in Figure 3.7. Individual insertions are not routed immediately but placed into bins that cover the key space in regions likely to be associated with a single node; their width is estimated periodically from the region covered by a single node in the leaf set. The items in a bin are forwarded as a batch, sorted by key, before they exceed the maximal batch size or after a timeout.

Recall from Section 3.4.3 that pointers for groups and volumes are inserted either as the arm moves over their identifier or as they leave the set of eviction candidates a fixed number of management epochs later. Since this induces significant spatial locality in keys being

[7] For example, the solution to TCP's 'small-packet problem' [Nagle84].

inserted, nearly all insertions can be batched with others and some reach their maximum size, the network MTU, without incurring a significant latency penalty, as Section 6.3.1 demonstrates.

At the source and each subsequent hop, a batch is forwarded as follows. Observe that if the next hop calculated for two keys is the same, then all keys within the range bounded by those two keys share that destination.[8] The node determines the next hop for each of the minimum and maximum keys in a batch, and either the batch is forwarded in its entirety, or it is split recursively into sets of keys that share a common destination by calculating the next hop for the median key in the region at each step. Items are not recombined into larger batches as the spatial locality of keys being forwarded through en route nodes is likely to be poor. Nevertheless, even if batches are partially separated at later hops in the route, DHT batching reduces messages transmitted and the number of routing table lookups.

Xest does not support removal of items from the DHT, which Bamboo implements using 'tombstone' markers, since each such removal uses bandwidth and storage commensurate with the item that it masks. Instead, stale items are overridden on the next insertion or left to expire, and cached pointers are updated when found to be inaccurate as described below. Insertions also do not use multipath routing as lookups do.

3.5.2 Managing fetch sources

Although the structure of the DHT itself is maintained despite node and network failure, the pointers that it contains are soft state. The advertised groups may be no longer stored, or their storage status changed, or the node itself may have failed. Moreover, the addressing information not only suffers limited recency, but its value is also constrained by its lack of subjectivity. Details of nodes' liveness and performance are inherently pairwise. Non-transitive connectivity, frequently observed in wide-area distributed applications [Rhea05c], may mean that although a node can retrieve pointers published by a live node, it cannot fetch data from it. Synthetic coordinates provide useful latency estimates, but are derived from simple models and may differ considerably from observations.

To improve both recency and subjectivity, the *Xest* source manager provides local caches of both addressing information for groups and source information for the nodes that constitute those addresses. This amortises the latency of pointer lookups over fetches concerning the same group, allows addressing information to reflect recent information, and shares accurate and timely data on sources' liveness, latency and bandwidth between all of a node's components.

[8] As with other DHT routing logic, calculations are performed modulo 2^{160} to account for the circular key space.

An addressing cache retains group pointers that have been retrieved from the DHT. Sources for a whole group are evicted on an LRU basis, and individual pointers expire as they do in the DHT. If a group's recency of access exceeds a threshold, new lookups are issued to replace expired pointers if their number fall below a proportion of the original available. Pointers in the cache may also be overridden when the storage status that they list is found to be inaccurate, or when they are absent, in which case a 'tombstone' status is used. Updating a pointer resets its timestamp; when a replacement for the same group and source is retrieved subsequently, if its timestamp does not exceed the local observation, it is discarded.

A node state cache maintains entries for all nodes with which the local node has recently interacted, including all of those listed in the addressing cache's pointers; in fact, cached pointers do not list target information as they do in the DHT but rather reference the appropriate node's state. Exponential averages of latency and bandwidth are maintained using data from the messaging layer. In their absence, latency is estimated from synthetic coordinates; bandwidth is estimated from that latency using a fitted approximation [Oppenheimer05].

Xest responds to message delivery failure in three stages that reflect Bamboo's congestion-aware recovery [Rhea04]. First, if a delivery goes unacknowledged despite retransmission or an anticipated reply is not received before a timeout, it assumes transient link failure and avoids using the node in question. Different sources are selected for fetches, and DHT lookups forwarded along other paths. If the node does not respond to pings within a further timeout, *Xest* assumes node failure, and components remove the node from their data structures. In the final stage, components locate suitable replacement nodes using periodic, rather than reactive, maintenance processes.

3.5.3 Fetch protocol

A node's fetcher manages the scheduling and enacting of uploads and downloads between nodes for data and metadata. It exposes an interface by which the group storage layer makes requests for whole groups, but also takes account of priority data as described in the following section. It uses the source manager's cached node state to select peers with which to negotiate fetches for data or deltas, coordinates multiple concurrent transfers and deals with out-of-date pointer state and node failure during the fetch process.

A group's metadata is required before fetches can be issued for its constituent objects. Requests for metadata are treated independently of requests for data on account of their small size, and the process is aggressively optimised as it is on the critical path for many storage operations. Two requests are issued in parallel to separate sources, obtained from the DHT if not present in the addressing cache and chosen on the basis of their estimated

location and liveness. Nodes reply by sending metadata immediately, if they can, and also return the group's local storage status, or an absent 'tombstone' if they do not hold the metadata, to update the requester's addressing cache. Subsequent requests are issued to further sources after a timeout or a negative reply, so that two are in flight at any time until the metadata is retrieved or all sources exhausted.

Data transfer operates at the granularity of **chunks**, obtained by dividing an object at fixed intervals, the maximum chunk size. This allows portions of an object to be requested in parallel from different sources and individually prioritised, as described below. However the maximum chunk size informs a trade off, described further in Section 6.3.4. A smaller size provides more scope for striping and improves the precision with which a chunk's priority can represent that of a sub-object region. Consecutive transfers, though, may not completely overlap each other's latency, and the effect of that component in the duration of a typical transfer increases for smaller chunks.

The chunk transfer signalling protocol, conducted using the UDP messaging layer, comprises requests for downloads and replies indicating the chunks that the peer will upload. Transfers themselves use the TCP transport and, unlike the reply, are queued and later scheduled according to their priority.

Each request may list several items, so long as their total size does not exceed the maximum chunk size and they all share a single priority. This reduces latency in the common case where the maximum chunk size is larger than the mean size of files and directories. A request item refers to a chunk of a concrete object, and so lists its group identifier, object label and chunk number, as well as a *type*, described below. When a node receives a request, it prepares a reply that details its response to each item in the request. This is dispatched immediately so that the client can issue new requests for items that this node does not in fact hold and wait for the transfer of the remainder. Replies also list the storage status of any groups referred to in the request, whether or not the transfer can be performed, so that the client can update its addressing cache.

The priority of a fetch can change after it is issued, and fetches may also be cancelled: an active volume may be closed, an object may be truncated, or the chunk may be received from a different source, as described below. Unless a fetch increases to high priority, such notifications are batched by their relevant source and piggybacked on regular fetch requests, or sent together in an empty request after a timeout.

I now consider how the fetcher schedules transfers. The total size of anticipated downloads and in-progress uploads is maintained above a certain threshold, divided equally when uploads and downloads are contending, by issuing new fetch requests and commencing previously enqueued uploads. Issuing a request involves determining the next chunk

to be retrieved, selecting a destination source and, if space allows, adding other chunks retrievable from that source.

Ideally, the source for a fetch is chosen as the node most likely to have the object's data or delta and transfer it in least time. However, since these two factors are not comparable and the requisite data is soft-state, a heuristic approach is taken. Sources are ordered by completion time, estimated from the latency and bandwidth for a node and the number of downloads already expected from it, and adjusted by a factor to account for the recency of both the network measurements and the group's storage status. This formulation favours recently contacted peers, particularly when the storage status was obtained directly from them in a fetch reply. A further penalty is applied if the group is degraded, so that such sources are chosen only when others are heavily loaded.

As well as sources being selected on the basis of their ability to rapidly transfer the chunk in question, once a request has been issued the fetcher employs additional mechanisms to reduce the effect of a node's transient poor performance, heavy load or failure on its completion time. When a node replies to a regular request, if it has numerous transfers already enqueued of an equal or higher priority than the request, it includes a 'busy' hint but proceeds to fulfil the request as usual.

On receiving a reply with a 'busy' hint, a **secondary request** for the items is issued to another node by the usual procedure, but with an instruction 'unless busy' set, which means that the recipient sends the requested items only if it is not busy. A sequence of secondary requests are sent until one non-'busy' response is received or a counter is exceeded. Similarly, if the source manager decides that a node from whom a download is outstanding has suffered a transient link failure, or a timeout is exceeded, a secondary request is issued for those chunks. Of course, when a chunk is received, outstanding requests for it are cancelled. Note that while the fetcher never explicitly sheds uploads, a 'busy' reply will tend to cause load to be diverted to less busy sources.

Recall from Section 3.4.2 that if an object's metadata contains a concrete reference to the object from which it was derived by copy-on-write and the latter is stored locally, the fetcher can attempt instead to retrieve a delta to apply to it in order to generate the new object. However a node with an object's data will store its delta only if it generated the delta itself or retrieved then applied it, and has since retained it. Moreover this addressing information is per-object, and not published. So, in cases where a delta may be used, the fetcher should exploit this possibility without delaying the object's retrieval by forgoing transfers from nodes that hold only its data.

Before considering a fetch strategy for objects with deltas, I first describe how transfers of chunks are negotiated. As shown in Table 3.2, a request item's type is used by the recipient to determine whether the transfer will be satisfied by the specified chunk of: only data,

Request	Reply		
	Group or object absent	Object present, no delta	Object and delta present
tx_data, c	tx_absent	tx_data, c	tx_data, c
tx_delta_only, c	tx_absent	tx_delta_absent	tx_delta, c
tx_delta_pref, c	tx_absent	tx_data, c	tx_delta, 0

Table 3.2: The negotiation of chunk transfers. Values in the first column indicate the type and chunk index present in the request. Subsequent columns describe under different storage conditions the type of reply made and, if any chunk is to be transmitted, its index.

only delta, or either but with a preference for delta. The columns in the table correspond to these three cases; the item in the reply for that chunk indicates what will be sent.

One complication is elided from the table. The potential recipient of a delta also requires the number of chunks in it and their hashes and lengths. This is collectively referred to as the delta's metadata, although it is stored at the sender as an integral part of the group's node local metadata. When a request for a delta is made, that request also flags whether, if a delta chunk is sent in reply, the delta's metadata should be sent as well.

Note that even objects whose data comprises multiple chunks are unlikely to have a delta that does. So that a data transfer can be striped over several sources while offering to each the opportunity of instead sending the delta and shortcutting the retrieval, a request of type tx_delta_pref for a chunk c replies either with the data chunk of index c, or the *first* delta chunk. The delta metadata can then be used to retrieve further delta chunks if any exist.

The fetcher maintains per-object state that differentiates which of its data and delta chunk transfers are in progress, and from whom, versus those that have not yet been requested, and their priorities. To supplement the node's addressing cache, the object state maintains a 'blacklist' of sources that advertise the object's group as degraded but do not cache the object itself, and a 'greylist' of sources that have the object's data but not its delta. When a request of type tx_delta_only cannot be satisfied, the reply value explicitly differentiates these two cases; the source is added to the object's 'greylist' or 'blacklist', respectively. Similarly, sources that return data to tx_delta_pref requests are greylisted.

The fetcher uses the following strategy when fetching objects with deltas. First, the process described above is used to select sources to which a series of requests are sent of type tx_delta_pref specifying different chunk indices. In parallel, a tx_delta_only request is sent to a manager of the virtual disk with which the volume local to the group is associated, unless such a node has already been sent a tx_delta_pref request. Managers may be identified by locating pointers for the group with a storage status of stored_manager; as explained in Section 3.6.4, they are more likely than other nodes to have retained its objects' deltas.

The process thereafter depends on the number of chunks in the data and in the delta and the relative latencies of the nodes contacted. If a delta is available from any of them, that is transferred unless all of the data chunks are retrieved first. Any subsequent delta chunks are requested and the requests for data chunks are cancelled. If no delta is available, the fetcher reverts to making requests only for data chunks.

3.5.4 Demand-sensitive transfers

Slow networks are a performance bottleneck for any wide-area file system. In the period immediately following a cache miss, the time taken to retrieve the data necessary to satisfy the request directly impacts the stall time perceived by a user. While a number of round trips may be required to locate and request any item of data, performing transfers at the granularity of whole groups can lead to a 'convoy effect' whose severity is aggravated by constrained wide-area bandwidth. Poor grouping arrangements or unanticipated access patterns may result in all of a group's data being fetched ahead of data that is already the object of a client request.

By receiving priority information from the group storage layer and structuring the scheduling of transfers around individual chunks of objects, the fetcher can manipulate its behaviour so that the granularity of the protocol appears to be selectively degenerated in response to indications that locality is poor. Arranging for lower-priority transfers to take place only during periods in which bandwidth is not required for satisfying higher-priority fetches aims to reduce the aggregate stall time by reducing the miss rate without affecting the miss penalty. If clustering is very poor and no prefetched data is subsequently used, the fetcher aspires for its profile of retrievals to be no different from that of a conventional distributed file system without grouping.

The fetcher employs four discrete priority levels:

demand_rw: Applied to individual chunks that underlie regions that have already been specified in outstanding read or write operations and have not since been subject to truncation.

demand_open: Applied to individual objects that have already been opened for read-only or read-write access and not truncated to zero length.

group: Applied to whole groups one or more of whose objects has been accessed.

background: Applied to fetches for groups that contain access statistics (see Section 4.2.1) and for the retrieval groups to maintain persistent storage (Section 3.6.4).

Unless a group fetch is specifically low priority, it is assigned the priority group. Additional prioritisation may be applied first to all of the chunks that make up a single object, and

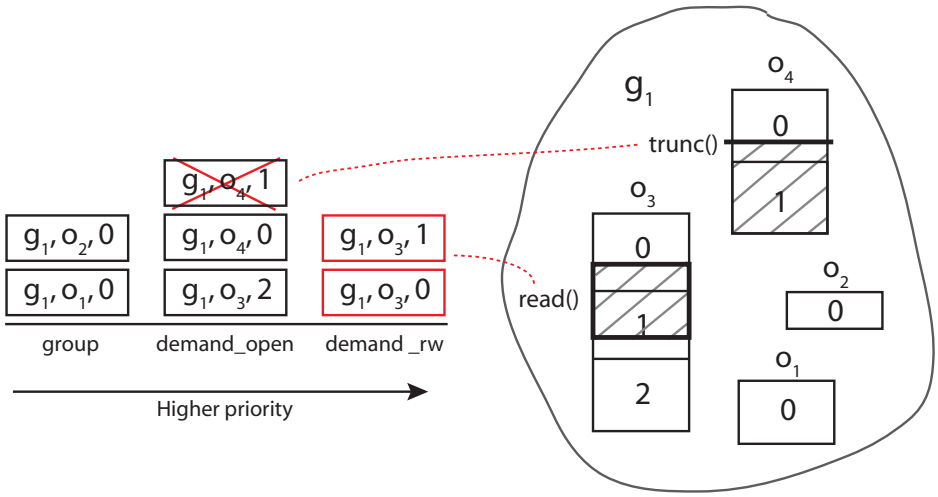


Figure 3.8: An illustration of fetch prioritisation. Objects o_3 and o_4 have been opened, causing their chunks to be fetched at priority demand_open and the chunks of other objects in the group at priority group. Next, a read on o_3 elevates the priority of the covered chunks to demand_rw. Finally, a truncate on o_4 causes ones of its fetches to be cancelled.

then also to individual chunks; each priority boost implies that the condition for the lower priority also holds. Hence the transfer of a group is not only 'critical object first', but 'critical region first', as Figure 3.8 shows. Prioritisation is dynamic: if for example an object that is enqueued as a background prefetch is opened, its priority is raised to demand_open and notified to nodes from whom a transfer for it is outstanding.

The fetcher enforces priority ordering by maintaining per-priority queues in order to delay new transfers until outstanding higher priority transfers are completed. This technique is applied equally to both uploads and downloads. Transfers of the same priority are allowed to proceed concurrently and compete with each other for available bandwidth.

Note that since dynamic priorities require the fetcher to operate reactively, lower-priority transfers may already be in progress. However, as described in Section 6.3.4, bandwidth throttling techniques were found to result in poor performance, and were not adopted.

An additional heuristic is used to minimise the likelihood of higher priority fetches competing with lower priority ones. Since file system activity tends to be bursty, the fetcher waits a certain delay after the final higher-priority fetch completes to commence the first background fetch.

A complementary but important technique is the cancellation of group prefetches that are long outstanding but no longer reflect recent access patterns. Each such prefetch increments a per-object reference count, which decrements after a timeout. An object with no higher-priority requests outstanding whose reference count falls to zero is cancelled unless its transfer is already in progress.

3.5.5 Migration preparation

Since writers to *Xest* volumes are anticipated to be Xen virtual machines that may themselves be explicitly migrated between hosts [Clark05], *Xest* includes additional support to expedite the process of migrating write access to their file systems.

Observe that active volumes themselves cannot be migrated, since access to them while they are writeable is isolated to a single node; only read-only data is ever addressed or transferred. Instead, support for migration is built atop a regular fork: the active volume earmarked for migration is closed and fixated at the source node then a new volume forked from the snapshot at the target. The additional functionality comprises two parts. First, as described in Section 3.6.3, the abstraction of the virtual disk and an operation on it specifically to assist migration allows a file system hierarchy to retain the same user-addressable name despite the change of node and the underlying fork. Second, *Xest* conducts performance optimisations to minimise the impact of cold cache conditions at the target; these are the subject of the remainder of this section.

A scheme for migrating execution state might be characterised in terms of three phases. First, a **push phase** may transfer state before execution is halted; any state subsequently modified must be re-sent. Second, during a **stop-and-copy phase**, state is sent while execution is halted, before execution resumes at the target. Finally, during a **pull phase**, state is retrieved from the source only on demand.

To migrate virtual machines' memory, Xen performs a number of pre-copy iterations of cold pages before pausing execution and copying the remainder. It does not use a pull phase as demand fetching would be conducted at the granularity of individual pages, causing a large number of high latency stalls, and the source would be required to retain pages until they were transferred or it had been notified that they had been overwritten.

Xest, on the other hand, pushes hot file system state to the destination, then pulls other requested state on demand. Unlike memory pages, groups are retained at the source and thereafter at the appropriate managers anyway, are accessible through *Xest*'s usual addressing and fetch procedures, and have a coarse granularity. By using a push phase as an optimisation and not requiring a stop-and-copy phase, *Xest* allows a memory transfer process such as Xen's to set the migration timetable.

Migration proceeds as detailed in Figure 3.9. During Xen's push phase, *Xest* aims to allow the target to prefetch hot objects that will not be overwritten and hence excluded from the final, forked file system by copy-on-write. It tracks the **read group working set** for the volume: the set of objects that belong to any group for which one or more objects have been recently accessed, less those objects that have been written to. This extension of the established notion of working set [Denning80] allows the migration target to benefit

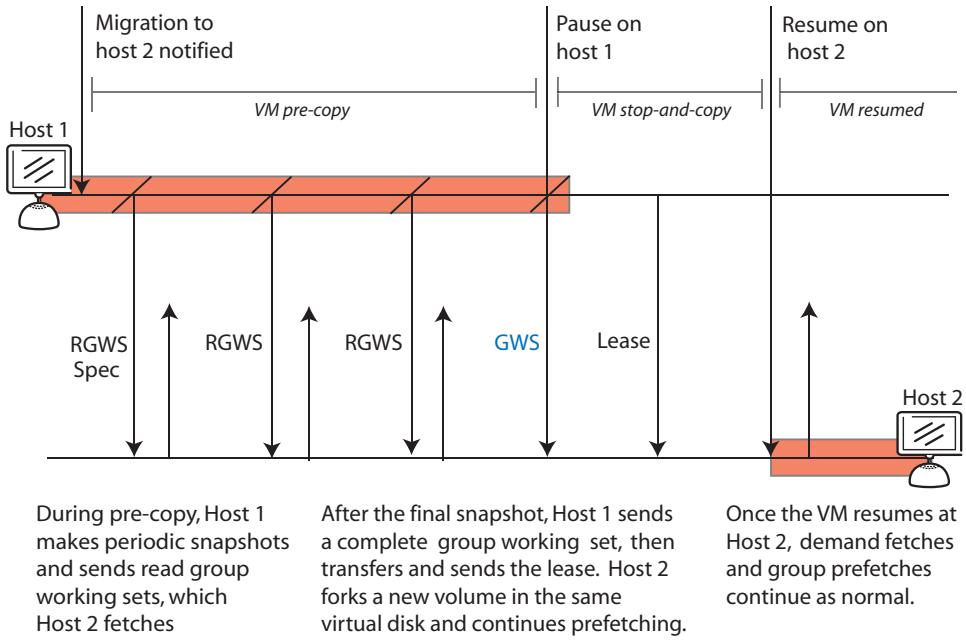


Figure 3.9: Preparing an active volume for migration.

from the prefetching implicit in retrieving whole groups at once while recognising that the grouping relationship does not distinguish read and write access. The working set tracks the indices of groups, which are invariant under copy-on-write.

At intervals the source volume is snapshotted and, as it is fixated, a list of the identifiers of the groups in the working set compiled, and sent to the target node, along with the identifiers of objects excluded for being recently written to. The target proceeds to prefetch the working set, first adding the source node into the pointer cache for each of its group to sidestep a DHT lookup. Then, any uncompleted prefetches for the previous working set sent for this volume are cancelled; by incrementing then afterwards decrementing their reference counts, the transfers of objects present in both working sets are unaffected.

As Xen's stop-and-copy phase commences, the source node closes the active volume, and sends a complete **group working set**, since all subsequent writes will be conducted at the target. After receiving the identifiers of the virtual disk and final snapshot, as well as the virtual disk's lease as described in Section 3.6.3, the target proceeds to fork the volume.

3.6 Virtual disks

As it appears to clients, the file system's global namespace is divided into virtual disks, each of which contains a file system hierarchy and its history as a series of snapshots. Internally, a virtual disk acts as a logical enclosure for a chain of snapshot volumes, and at most one

active volume. As a unit of administration, virtual disks enable storage management to be partitioned between different sets of nodes, a virtual disk's **managers**, which expose a number of distributed control-plane operations.

This section describes the structure of virtual disks, how *Xest* globally agrees updates to them, and their rôle in naming, locating, accessing, and durably storing the contents of their constituent volumes.

3.6.1 The structure of virtual disks

Figure 3.10 demonstrates the constraints on assigning volumes to virtual disks. These constraints serve two purposes. First, they allow logically-related storage to be managed together while separating the management of unrelated storage. Second, they provide a means of canonically naming a file system's state at a given point in time, as in previous copy-on-write file systems [Santry99].

Observe that the global set of volumes is a forest in which each volume that is created, rather than forked, is the root of a tree whose branches are subsequent snapshots and whose leaves may be either snapshots or active volumes. A virtual disk comprises a sequence of volumes that form a non-branching portion of such a tree: *i.e.* no two volumes in a virtual disk share a parent, and each volume's parent belongs to the same virtual disk, except for the first volume in the sequence. Also, every volume belongs to precisely one virtual disk.

This structure means that if the final volume in a virtual disk is a snapshot, it may be forked and the resulting new active volume may be appended. In addition, interim snapshots of that active volume can be added as they are produced. However forking a snapshot other than the last volume must cause a new virtual disk to be created. Note that although the sequence of snapshots in a virtual disk are not required to be maximal, the current implementation only splits a tree at a non-branching snapshot into two virtual disks when their storage requirement as a single virtual disk would otherwise exceed a certain threshold.

The primary component of the metadata of a virtual disk is a list of its constituent volumes, in the order in which they were created. Each entry also contains the snapshot's creation time and a list of the identifiers of virtual disks that have been forked from it.

The metadata also stores a list of user-specified textual labels, by which the virtual disk can be located. A list of managers' identifiers, IP addresses and synthetic coordinates is used by clients to locate their nearest managers. A version sequence number and details on the current holder of the disk's lease facilitate resynchronization or changing of a manager. As the unit of administration, the metadata would also contain quota and access control data in a production implementation.

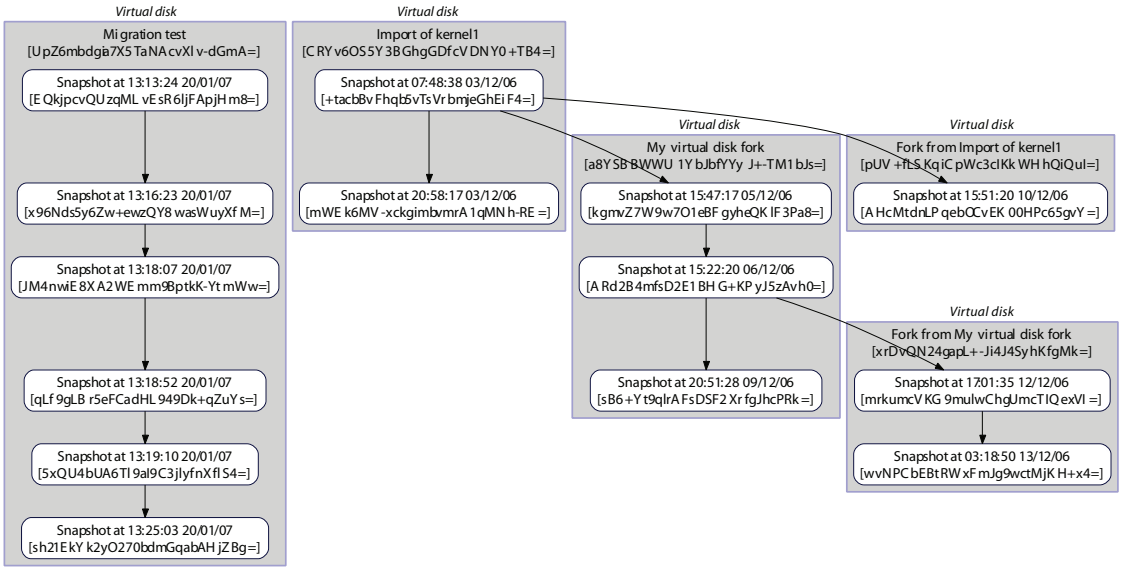


Figure 3.10: The assignment of volumes to virtual disks. A *Xest* utility generated this figure using the GraphViz package. The first fork from each snapshot volume may be appended inside the same virtual disk. Subsequent forks must instantiate a new virtual disk.

3.6.2 Virtual disk managers

Every virtual disk has a set of managers, which are regular nodes that assume a set of additional rôles for specific virtual disks. First, they mediate updates to the metadata by participating in a distributed agreement protocol. Second, they persistently store all of the contents of the virtual disk's volumes. Third, they detect each other's failure and agree on making replacements.

Each virtual disk specifies a target number of managers. When a client needs to create a new virtual disk, it generates its metadata and becomes its first manager. In the *Xest* prototype, the client determines the other managers by routing invitations to a set of random keys in the DHT. A node may refuse an invitation if its persistent storage burden exceeds a given portion of its available cache space. Managers periodically test each other for liveness; peers that together share the management of multiple virtual disks ping each other only once per period.

The strategy for selecting managers involves difficult trade-offs. Selecting managers by routing in the DHT as *Xest* does will tend to yield sets of nodes diverse in location and network connection, as would selecting neighbours from the node's leaf set. This minimises the probability of correlated failures between the managers of a given virtual disk. For this reason, neighbours are not selected from the DHT routing table, as Bamboo preferentially

populates it with nearby peers. Against that risk however is the potential to reduce the latency of agreement operations significantly.

Selecting neighbours as the *Xest* prototype does may lead to all nodes pinging each other as the number of virtual disks exceeds the number of nodes. On the other hand, rigidly constraining the selection of managers to a subset of the first manager's leaf set complicates the balancing of persistent storage across the system.

Xest will be revised in future to use a hybrid approach that preferentially selects nodes from its leaf set but resorts to routing in the DHT if insufficient nodes accept the invitation. It may also be desirable for nodes to reject invitations if they have too dispersed a set of management peers. Note, however, that this change would have little effect on the evaluation presented in this dissertation.

A manager is maintained until it has been unresponsive for a period that, heuristically, indicates it has become permanently disconnected. Any node that detects a failure routes in the DHT until it locates another node that accepts its tentative invitation; replacements are proposed to the remaining managers and the first to be agreed is accepted firmly.

After failure or network partition, a manager may be able to recover by replaying operations agreed subsequent to its failure in order over its version of the metadata. If it cannot, it refetches the latest metadata from the DHT, checks that it is still a manager, and uses that state and the version number recorded in it to rejoin the agreement protocol.

In normal operation, clients' requests on a virtual disk are translated into a set of operations on the global state. A Paxos protocol [Lamport98] is used to obtain, despite node failure, distributed consensus between the virtual disk's managers on the order in which operations are to be applied to the metadata. By the terminology of the original formulation, operations are 'decrees' and the metadata is the 'law book'. *Xest* implements an algorithm that is based on Boichat *et al.*'s decomposition of the protocol that requires fewer messages to be passed during stable periods [Boichat03a, Boichat03b].

When a manager receives a client request, it uses Paxos to propose it to the other managers, thereby eventually agreeing a total order on operations. Each manager enacts the request, if valid, on their identical copies of the metadata and returns the result to the client. This process is illustrated in Figure 3.11.

A virtual disk exposes the set of distributed operations listed below, whose use by clients is described in the following section:

(lease,metadata) ←-- getLease (client, last_snapshot)

> Obtains, renews or, if possible, recovers the exclusive lease on a virtual disk and its volumes for the given client, and returns a token identifying the lease as well as the up-to-date metadata.

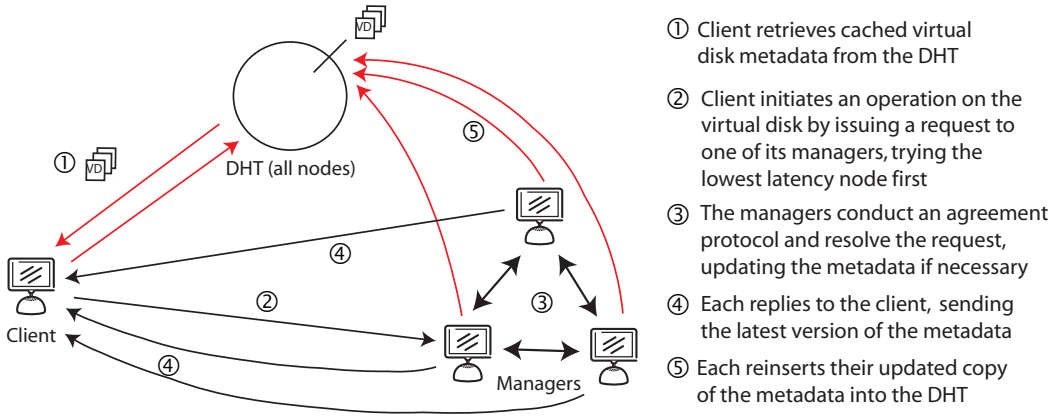


Figure 3.11: Operations on virtual disks.

() ←-- releaseLease(lease)
: Releases the specified lease on the virtual disk, if it is still in fact held.

() ←-- addSnapshot (lease, snapshot_info)
: Allows the client that holds the lease to add a snapshot with the given details, after checking that it is valid for this virtual disk and not already listed.

() ←-- addSnapshotAndTransferLease (lease, snapshot_info, new_client)
: Atomically add details of the given snapshot, as above, then transfer the existing lease to the specified new client, resetting its timeout. Assists in migrating the virtual disk between nodes.

() ←-- notifyFork (from_snapshot, dest_vd)
: Records that the first snapshot in a new virtual disk under the specified identifier was forked from the given snapshot in this virtual disk, if not already recorded.

() ←-- addLabel (label)
() ←-- removeLabel (label)
: Adds or removes a textual label under whose hash the metadata is inserted in the DHT.

3.6.3 Naming and accessing storage through virtual disks

Client virtual machines mount and access storage by naming virtual disks, not volumes. A virtual disk's name stays constant despite the the changing volumes that its metadata identifies as constituting its file system, both at the current point in time and throughout its snapshot history.

A virtual disk is named canonically by a single immutable opaque bit string, in the same identifier space as groups and volumes, and it can also be named with client-assigned non-

unique textual labels. Each manager of a virtual disk independently maintains a copy of its metadata published in the DHT under its unique identifier and also under the hashes of each of its textual labels. Note that the metadata itself is inserted, not a pointer to it; this reduces the number of round trips made by clients, latency being dominant on account of the metadata's small size.

To access storage, clients uniquely identify a virtual disk by specifying either its complete identifier or some subset of its textual labels and a prefix of its identifier. They also identify the volume within the virtual disk to open by specifying a time, which looks up the last snapshot with creation time previous to it, or using the special value 'latest'.

While a virtual disk's managers hold consistent up-to-date copies of its metadata, elsewhere it is soft-state; clients expire their cached copies only periodically, and in any case copies in the DHT may also be stale. In order to be able to guarantee to a node holding the lease for a virtual disk that it can identify its most recent volume, and hence record any volumes created back into the same virtual disk, whenever a lease is agreed the state of the metadata is obtained as an atomic operation and returned to the client. Since other nodes cannot add new snapshots while the lease is held, this is sufficient.

A primitive that atomically adds a final snapshot to a virtual disk and reassigns its lease to another node extends the guarantee of continuous lease ownership across multiple nodes, which in combination with the techniques described in Section 3.5.5 facilitates the appearance of virtual disk migration. At the point of migration, the source closes its active volume, invokes this operation, then notifies the target of the identifier of both the virtual disk and its final snapshot. The target node can immediately proceed to fork the volume and add snapshots to the virtual disk without waiting to retrieve up-to-date metadata for it from the DHT.

Although obtaining a consensus between managers involves a number of communication steps over high latency links, agreement is necessary to ensure the availability and consistency of virtual disk state in the face of node and network failure. Weaker consistency semantics may lead to multiple active volumes contending to belong to the same virtual disk, meaning that the top-level of the client namespace would become detached from the volume namespace.

Moreover, as I demonstrate in Section 6.4.1, the impact on performance of this arrangement is limited. Managers only require consensus on control-plane operations, most of which appear to the client as background tasks. The exceptions, obtaining an initial lease on a virtual disk and later transferring it, occur once only during a virtual machine's startup and migration, respectively. No operations are synchronous while a virtual disk is accessible. Nevertheless, while some of the operations require strong consistency semantics, it is

arguable that others commute and so do not. However their asynchronous nature means that the additional latency due to obtaining a consensus has little impact.

This asynchrony allows a client to continue accessing a volume and creating snapshots even if network partition or manager failure temporarily prevent operations on its virtual disk completing. In particular, the client persistently records details of any snapshots that it creates and ensures that they are eventually notified to a virtual disk in order to make them globally addressable.

If failure causes the client's lease to expire for a virtual disk on which a snapshot is outstanding, the lease on that virtual disk may be acquired by another client that appends new snapshots of its own. In this case, the first node's snapshot must be inserted in a new virtual disk as it would introduce a branch in the first virtual disk.

3.6.4 Maintaining durable storage

The autonomy of nodes' cache management means that separate global coordination is required to ensure that data is retained persistently, despite the failure of nodes and the activity of other clients. As the maintenance of durable storage is not the focus of this thesis, this section describes *Xest*'s approach in outline and discusses how it couples cache management with the administration of virtual disks.

Recall that each node's local store is unified between caching and persistent storage; management of the two differs only by eviction, determined using each group's storage status. Now, the set of managers of a virtual disk each commit to maintaining persistently its contents; this entails retaining a non-evictable copy of every group local to one of the virtual disk's snapshots. Maintaining this invariant requires interaction between nodes both when managers change and when snapshots are created.

When a snapshot is created, the groups that are fixated as a result are given a storage status of stored_sticky at that node, which prevents them being evicted before they can be retrieved by their virtual disk's managers. When the managers agree on the request to add the snapshot to the virtual disk, they each initiate a background process that issues background fetches first for the snapshot volume's metadata then, using that, for its local groups. As each group is retrieved, its storage status is set to stored_manager. On completion of the process, each manager marks the snapshot as 'locally complete' in its virtual disk metadata. This flag is local to each node and retained despite updates to or resyncs of the metadata.

Note that for all but the first snapshot in a virtual disk, ignoring manager churn, a manager will have the source to which to apply any deltas that are created, so will fetch and store them. Because of this, the fetch protocol preferentially locates deltas from managers, and hence these deltas are less likely to be evicted, reinforcing the pattern.

When a node becomes a manager for a virtual disk, it must perform a similar background retrieval of all of its snapshots. This process uses the regular fetch mechanism, so it takes advantage of copies cached on nearby nodes, and if required groups are already cached, they simply have their storage status adjusted.

The node that created the snapshot and retains its groups with status stored_sticky runs a periodic process so that, in an eventually consistent manner, it will revert to caching the data once each of the virtual disk managers stores it in its entirety. This process is conducted by the storage management arm. When it iterates over 'sticky' snapshot metadata, a reminder is sent to all managers of that virtual disk. Each returns a value that indicates whether it itself already stores a complete copy of the snapshot, determined trivially by consulting its 'locally complete' flag; any manager not doing so ensures that the retrieval process is underway. If an affirmative response is received from every manager within a timeout, the storage status of the snapshot volume's metadata is set back to cached_complete. Similarly, when a node notices that it is no longer a manager for a virtual disk, it reverts the storage status in each of its constituent volumes' metadata to cached_complete.

Thereafter, the storage status of the groups associated with a snapshot are updated lazily according to the storage status of the snapshot's volume metadata. Whenever the arm iterates to a group whose status is cached_sticky or stored_manager, that group's snapshot metadata is located and its storage status read. If it is absent, or is evictable, the group has its status fixed to cached_complete and is considered for eviction as normal. In fact, to reduce snapshot metadata lookups, a small cache of snapshots' identifiers and their corresponding storage status is maintained in memory.

3.7 Summary

This chapter presented in detail the architecture of the *Xest* prototype. It introduced and described a number of key abstractions, whose rôle and treatment are summarised below:

- **Objects.** An object corresponds to a single file or directory, and has a short label that identifies it relative to its group. Objects are advertised only as part of their group. The fetch protocol divides an object's variable-length data into fixed-width **chunks**, to allow it to prioritise and transfer sub-file regions. In all other aspects of storage management, objects are manipulated in their entirety.
- **Groups.** As a set of objects related by access pattern, groups provide a coarse-grain unit for storage management, cache advertisement and fetching purposes. Each group has a globally-unique identifier and is advertised by inserting pointers into the DHT. Groups facilitate copy-on-write of their objects' data and metadata.
- **Volumes.** A volume is a logical container for groups that constitute a single file system. It provides a hierarchical namespace and functionality to snapshot and fork whole file systems. Volumes are advertised and retrieved as groups are, though volumes have no associated objects. A volume, each of its groups, and each of those groups' objects, are mutable only between a volume being created or forked and it being fixated, and then only at that single node.
- **Virtual disks.** A virtual disk encapsulates a sequence of volumes and makes them globally accessible under a single name, thus facilitating migration between nodes. Each virtual disk maintains mutable metadata that is updated by agreement between a set of manager nodes. These nodes enforce a single global writer and maintain durability of its associated storage. Virtual disks have identifiers in the same space as volumes and groups but their metadata is inserted into the DHT directly, rather than use pointers.

In the next chapter I describe how objects, groups and volumes participate in the process of recording statistics on file system access patterns. I then show how these statistics are used to reorganise the objects associated with each volume into a set of groups.

Chapter 4

Grouping

In the previous chapter I described how *Xest* advertises, retrieves and manages storage in terms of variable-size groups of objects. This chapter describes the process by which objects are assigned to groups according to client access patterns, with the intention of these units being coarse-grained while retaining good locality between their objects.

In terms of abstraction, the grouping components operate within *Xest*'s volume layer (see Section 3.2.1). Since volumes enclose entire file system hierarchies and may be forked and migrated independently, they form the scope within which objects' inter-relationships are inferred, and within which the regrouping process operates. A volume's metadata is used to locate access statistics recorded about its groups, and its groups table allows references to objects to be invariant under copy-on-write. Nevertheless, since moving an object to a new group requires its referring directory entries to be corrected, the reclustering process is also tightly coupled to file system client routines.

The grouping process encompasses a number of stages. Client operations on the file system are observed and synthesised into statistics associated with each object by the **observer** component. The statistics are managed in order to meet with space constraints and to accumulate despite volume, group and object copy-on-write. Immediately after a volume has been closed, the **regrouper** component runs a clustering algorithm that uses these statistics to reallocate objects to groups.[1] Finally, the new arrangement is applied to the volume by adjusting references to relocated objects. In this chapter, the term **cluster** refers to a group and its constituent objects.

This chapter is structured as follows. In Section 4.1, I begin by explaining the requirements of *Xest*'s group allocation components, and outline my solution. In Section 4.2 I proceed by describing the collection of access statistics. In Section 4.3 I detail the group reallocation process.

[1] Although in the *Xest* prototype regrouping is performed while a volume is offline, the techniques described in this chapter are amenable to an online approach, as discussed in Section 7.2.

4.1 Approach

There are a number of requirements that distinguish *Xest*'s regrouping process from existing schemes. I first set out those characteristics before developing and outlining a corresponding solution.

- **Variable-sized groups**: Grouping is intended to increase the granularity at which interfaces operate, rather than rearrange the granules inside existing fixed-size pages or blocks. As such, accepting each additional member to a group must be informed by a trade-off between increased aliasing and reduced addressing state.
- **Canonical object names but secondary associations:** *Xest* uses groups to partition the object namespace for storage management purposes, so that they provide a canonical, primary location with which an object's metadata and data are associated. Nevertheless, objects tend to be accessed as part of a number of working sets or projects, and a clustering outcome that can reflect such secondary associations allows groups to better delineate their twin roles as units of both storage organisation and 'implicit prefetching'.
- **Scalable and partial clustering**: Even a modest volume may consist of several hundreds of thousands of objects. This scale requires the clustering algorithm to have low time and space complexity in the number of objects, but also means that portions of the file system may be accessed very rarely and lack reliable statistics. Performing partial clustering, over subsets of the objects in a volume, can reduce the problem size and eliminate sparse regions. Moreover, complete reclustering may not always be necessary under stable access patterns.
- **Incremental clustering**: Since moving an object between groups incurs an associated cost due to the updating of back pointers, directory entries and statistics, repeated clusterings should be stable, despite overlapping partial clusterings, and any modification made to an existing grouping arrangement should be commensurate with its perceived benefit.

4.1.1 Building variable-size groups

Xest builds groups and produces secondary associations using a method inspired by the second phase of Jarvis and Patrick's algorithm [Jarvis73] and SEER's implementation of it [Kuenning97a].

In the original formulation of the algorithm, for each object $o \in O$ a set of nearest neighbours $N_o \subseteq O$ is pre-computed with $|N_o| \leq n$ for some constant neighbour table size, n; the distance function $d : O \times O \mapsto \mathbb{R}$ that determines these neighbours need not define a

metric space. Clustering is performed in a single pass, with each object initially a singleton cluster. Each object is considered in turn; for each of its neighbours $m \in N_o$, if those two objects share at least $k \leq n$ of their neighbours, *i.e.* if $|N_o \cap N_m| \geq k$, their two clusters are merged.[2]

The variant employed by SEER determines each file's nearest neighbours as those closest according to Kuenning's 'semantic distance' measure, described in Section 2.3.2. Like SEER, *Xest* observes references between files using the sequence of open() operations. However, instead of using semantic distance, it defines a distance function $d(o_1, o_2) = 1 - w_r(o_1, o_2)/s(o_1)$, where the function $s(o_1)$ reflects the total number of occurrences of o_1 and $w_r(o_1, o_2)$ counts the number of times o_1 and o_2 are observed within a fixed number of references r of each other, the reference window size.

Since the reference stream is composed of all processes' interleaved reference streams, the window generalises the notion of a transition over the r closest accesses; equivalently, it admits only those references that would contribute to a semantic distance less than r. Thus an object's nearest neighbours are those with highest frequency of co-occurrence. I describe how file system activity is observed to acquire these statistics in Section 4.2.

Intuitively, SEER's use of semantic distance allows cluster boundaries to expand around every file that has been observed as being part of any associated working set, a property well-suited to a hoarding algorithm where single cache misses can halt use of the system. *Xest*, in contrast, operates in a context in which cache misses have a lower cost but unnecessary fetches a higher cost. Because of this it aims to constrain the membership of each group to those objects with a high conditional probability of access, comparable to 'basic blocks' in the reference stream.

Xest mediates the trade-off between a group's size and how closely related its objects are by performing the merge process over a fixed number of passes. In each successive pass, the strength of the relationship required to merge a pair of groups is relaxed, but the maximum size of the resulting group for which a merge can proceed is also progressively reduced. Each group's boundaries are formed around closely related objects first, and expanded to agglomerate more weakly related groups depending on the total size of their constituent objects.

To achieve this, *Xest* considers merging the groups of a pair of objects by counting their shared **significant neighbours**. For pass p, an object's set of significant neighbours $S_o(p)$ is the subset of its nearest neighbours whose distance is less than a threshold $t_p \leq t_{p+1}$, so $S_o(p) = \{m \in N_o \mid d(o, m) \leq t_p\}$. In addition, the threshold of shared neighbours above which the groups of two objects are merged is adapted to increase monotonically with the

[2] Although the original formulation also requires that $o \in N_m$, neither SEER nor *Xest* do as their distance measures are not symmetric.

pass, and is written $k_p \geq k_{p+1}$. So, given the function $g : O \mapsto \mathbb{N}_0$ that returns the total size of a group's objects and a group size limit l_p, then for an object o and its significant neighbour $s \in S_o(p)$ a merge proceeds if $|S_o(p) \cap S_s(p)| \geq k_p$ and $g(s) + g(o) \leq l_p$.

4.1.2 Incremental clustering

Xest's clustering algorithm does not begin with objects as singleton clusters, but rather introduces a divisive phase that runs before the agglomerative merge phase, in order to facilitate the incremental adjustment of existing grouping arrangements.

The divisive phase runs in a single pass, and operates on groups rather than individual objects. Each group is considered recursively, at the minimum pass p at which its size would have allowed it to be created. The algorithm walks its objects to ensure that on the basis of their current neighbour statistics the cluster would still have been formed. However, a reduction is made to both the neighbour significance threshold t_p and the shared neighbours threshold k_p in order to effect hysteresis and thereby minimise changes made due to small variations in object statistics. Hence, having determined p, $t'_p < t_p$ and $k'_p < k_p$ define a set of shared neighbours for split purposes $S'_o(p) = \{m \in N_o \mid d(o,m) \leq t'_p\}$ with $S_o \subseteq S'_o$. Then the set of walkable edges $E = \{(o,s) \mid |S'_o(p) \cap S'_s(p)| \geq k'_p\}$.

If necessary the group is split into its connected components and the procedure applied recursively. The algorithm is described in detail in Section 4.3.2.

4.1.3 Secondary associations

SEER's algorithm adds a second pass to the Jarvis-Patrick formulation in which overlapping clusters are created. In this, if two objects assigned to different clusters share at least $k' < k$ neighbours then they are each added to the other's cluster, but the clusters are not otherwise merged. *Xest* also performs a single pass to create overlaps after the merge phase. However it does not use it to detect access patterns with a weaker threshold but identical structure to complete merges, as SEER does; this is a role that *Xest* fulfils using multiple merge passes.

Instead, overlaps aim to compensate for asymmetric patterns of neighbours, typically observed between applications and shared libraries or source code and shared header files. An overlap of an object s is added to a group if its distance from any of its existing primary objects o is below a threshold l_n.[3] This indicates a high probability of transition from s to o, but not necessarily from o to s: a low-distance symmetric relationship would likely have resulted in their groups being merged. The clustering algorithm also incorporates checks

[3] Note that a reciprocal overlap of o is not necessarily added to the primary group of s, as it would be in SEER.

into its divisive phase, using a weaker threshold $l_e > l_n$ for hysteresis, to retain or remove overlaps from the new groups that result from splitting existing ones.

In addition, whereas SEER treats objects that are overlapped into a cluster as first-class members, *Xest* requires that groups canonically partition the object namespace. As such, a *Xest* group tags each of its members as either 'primary' or 'overlap', and records overlaps only as pointers that use a (group table index, object label) pair to identify an object in its primary group, with reference to that volume's metadata. Section 4.3.2 describes how the two classes of membership are treated during the clustering process.

Observe that *Xest*'s overlaps are analogous to symbolic links whose targets are hard links: their presence indicates a secondary semantic association with its contents, but they have no associated storage and instead must be resolved to a primary object. Overlaps' status as 'prefetch hints' means that *Xest* need not maintain any form of back pointer from a primary object to the groups in which it is overlapped. If an object's link count falls to zero, an overlap to it will only be removed when that group is next party to reclustering.

4.1.4 Scope of clustering

Since a *Xest* volume stores directories as regular objects, in order to partition its objects into groups the clustering process must consider directories as well as files. In contrast, SEER clusters only files; if defers decisions on hoarding directory blocks to its underlying replication substrate.

However associating operations on directories and their entries in the same context as file operations is problematic. Early prototyping suggested that it obfuscated the relationships between files; in-kernel path walking and dentry caching mean that access to a file may be preceded by accesses to none, some or all of its enclosing directories.

Xest takes the approach of completely segregating the processes of observing accesses and forming groups for files and directories. Colocating related directories in a group together prevents a sole directory access inducing fetches for large files and tends to allow the directory hierarchy to be walked with recourse to further groups only at the leaves.

4.1.5 Related approaches

Xest does not adopt a minimum-cut graph partitioning approach, as advanced by Tsangaris and Naughton. This represents accesses as a Markov process and seeks to minimise the cardinality of the working set of window size two [Tsangaris91].

First, this method cannot easily accommodate the trade-offs involved in variable-size groups: it models a cache of two fixed-size pages with uniform miss penalty, assumes a fixed num-

ber of partitions, and associates costs only with edges, not graph nodes (*i.e.* objects). Second, it is unclear how secondary associations could be incorporated into the heuristic algorithms most appropriate to this method, which either greedily allocate clusters or iteratively swap objects between them (for example SMC.WISC and SMC.KL, respectively, in [Tsangaris92]).

Third, graph partitioning tends to form uniform-radius clusters around their weighted centres, since new members are added to minimise the cut from all of a cluster's existing members. In contrast, by merging clusters that may share only a portion of their perimeters, methods derived from Jarvis-Patrick permit arbitrarily shaped clusters to form from a chain of related objects, without requiring higher order relationships to be recorded in object statistics. Tsangaris and Naughton note that employing a higher order Markov model leads to a hypergraph partitioning problem [Tsangaris90].

4.2 Recording access patterns

Xest's group reassignment process is conducted according to object access records, per-object nearest neighbour tables that embody the distance function described in Section 4.1.1. This section describes the processes by which these statistics are gathered and maintained.

In Section 4.2.1 I describe the structures that allow the access records for a complete volume to be stored, advertised and retrieved in groups alongside regular groups containing files or directories. In Section 4.2.2 I explain how file system events are recorded in object access records, and in Section 4.2.3 I discuss how *Xest* maintains accurate neighbour tables despite changes in access patterns and storage constraints.

4.2.1 Storing and maintaining statistics

Before discussing how a volume's statistics are retrieved and manipulated, I first describe the structures within which they are stored, and in doing so introduce some terminology.

Xest maintains each volume's statistics in **statistics groups** which differ in only a few ways from conventional groups. At the group storage layer, their group metadata is structurally identical: they can only be distinguished by a value in a 'type' field. At the volume layer, they are manipulated through the regular interfaces described in the previous chapter, except with the additions described below. They are listed in the volume metadata's groups table and undergo copy-on-write as regular groups do. However, statistics groups' objects are not accessible as part of the volume's file system hierarchy by virtue of never being referred to from its directories entries, and so do not maintain back pointers. In addi-

tion, they do not have statistics themselves and are never subject to the group assignment process.

An **object access record** encapsulates the statistics associated with a particular object and is used to determine its nearest neighbours. Its principal components are a field that records the total number of references for the object and a fixed-size table that lists neighbours as a (group table index, object label) pair together with a reference count. I describe this structure further in Section 4.2.3.

A **group access record** comprises the object access records for all of a regular group's objects, and is the unit of storage that corresponds to the contents of a **statistics object**, an object in a statistics group. Like their corresponding regular objects, the access records associated with a single group tend to be manipulated together. Since each is at most only a few hundred bytes in size, storing them together amortises the overhead of managing and retrieving them.

There are two key reasons why *Xest* stores multiple group access records in segregated statistics groups rather than attach each group's statistics to its own metadata. First, it allows the low-priority retrieval of group access records to be separated from high priority metadata fetches. Second, it means that groups whose contents are not modified need not undergo copy-on-write solely to update their statistics. This minimises the apparent changes each snapshot makes, so at nodes at which the volume is subsequently forked cache hit rates are improved and more sources are available for the group.

The extraction and updating of structured access records from statistics objects' byte extents is managed by the observer, which provides an interface to them through a write-behind cache. I now describe how it locates the group access record for a given group, a process illustrated in Figure 4.1.

Each statistics group manages the statistics of the regular groups within a contiguous region of the groups table with a fixed length s. Note that since group table entries may be empty or no accesses observed for other groups, statistics groups may contain fewer than s objects.

A **statistics table** in the volume metadata is managed by the observer and lists the group table indices of the volume's statistics groups such that the group at index i is responsible for the region $[si, s(i+1))$. By listing group table indices, the table remains invariant under group copy-on-write. Initially the table is empty, and statistics groups are created on demand; the indirection afforded by the statistics table allows them to occupy any group table entry.

Within each statistics group, a total order is defined over the statistics objects by interpreting their object labels as unsigned integers. The statistics object that stores the group access records for a group table index e has label $e \bmod s$. Despite this abstraction, the

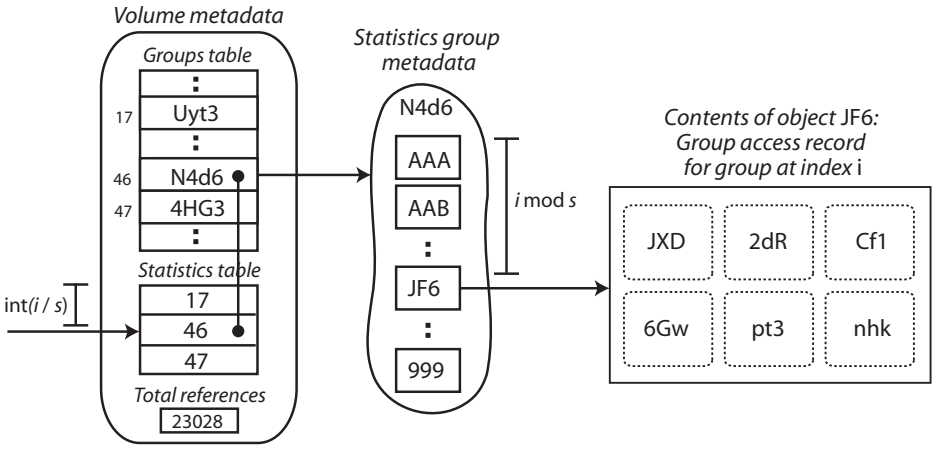


Figure 4.1: Storage of a volume's statistics. The volume metadata's statistics table is used to identify statistics groups, whose objects each contain a single group access record. The figure illustrates how the group access record is located for groups table index i. The constant s denotes the maximum number of regular groups on which a statistics group holds details.

volume layer continues to treat statistics groups as a set of unordered objects, as it does regular groups. Only one addition to the volume storage interface is necessary: when creating statistics objects, their label is specified as a parameter rather than generated randomly from those not already in use.

4.2.2 Observing accesses

Statistics are gathered cumulatively over volume snapshots and forks. This means that the group assignment process has a canonical source for the entirety of the volume's statistics, and does not have to reconcile and combine statistics from previous volumes. Disallowing the divergence of statistics, however, means that before a reference can be recorded, the statistics object containing the group access record in question must first be retrieved then undergo copy-on-write. This process occurs just as it would for regular groups and objects, with the exception that a statistics object which is awaiting updating is fetched at group priority, and other objects in the same statistics group are prefetched at background priority.

Queueing outstanding updates to statistics in memory presents no complication unless the volume is snapshotted or closed, in which case the queue is flushed to allow these operations to proceed immediately. The consequent loss of references is negligible except in pathological cases of a very high rate of access to the volume but very poor fetch performance of its statistics groups.

Reference trackers implement the effects of the reference window size r, introduced in Section 4.1.1. The observer maintains two reference trackers per volume, one for directory

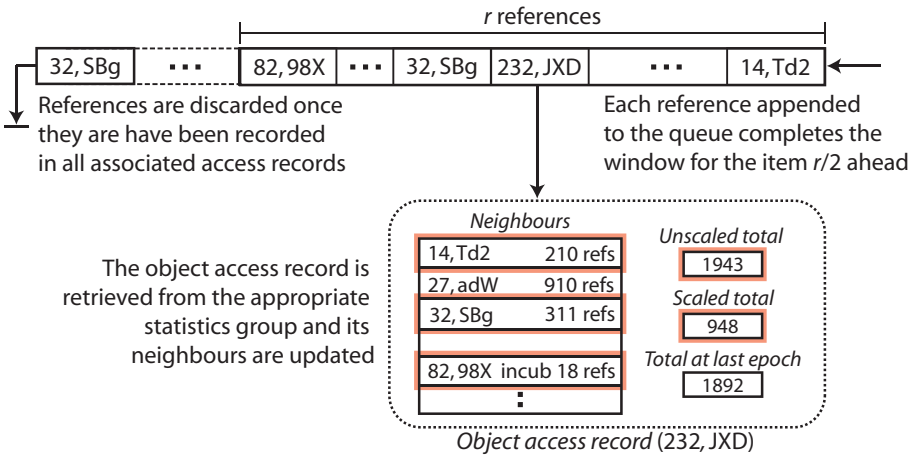


Figure 4.2: Updating an object access record

objects and one for files, and passes through to them notifications of file system accesses that it has received from the volume layer. Any metadata operation on an object or it being opened triggers an access event; since directories are overlaid on regular objects, listing or manipulating directory entries does too.

Figure 4.2 depicts the process of recording a reference. Each new reference is added to a circular queue, thereby completing the window of references for the item $r/2$ entries previously. That entry's object access record is then fetched and updated. This increments the total count once, and also increments once the counts associated with each unique member of the window, if neighbours for them are present. In addition, a field in the volume metadata records the total number of references observed for that volume, in order to allow the absolute frequency of objects to be estimated.

4.2.3 Maintaining object access records

A challenge for any scheme that records access statistics is to retain an object's most valuable neighbours in a fixed amount of space yet reflect long-term changes in access patterns. There are two aspects to this requirement.

First, neighbours should be aged, and obsolete entries must be removed. *Xest*'s approach to ageing neighbours is reminiscent of that employed in [Kroeger01]. Every r_e references to an object, both the total count and the counts of each neighbour are scaled by a factor $0 < f < 1$, and a new epoch begins. The frequency of each neighbour remains unchanged, notwithstanding rounding due to integer division, but references in each subsequent epoch are effectively weighted by a factor of f. Neighbours whose counts fall to zero as a result of rounding are removed. Since the volume's total reference count is never scaled, each object

access record also retains an unscaled total used for calculating its absolute frequency with respect to the volume.

Second, ageing should be complemented by a scheme to replace neighbours with more frequently referenced ones if all are equally recent. However, since an eviction decision must compare reference frequencies, which can only be determined over time, object access records must 'incubate' newly introduced neighbours until their count can be fairly compared with that of established neighbours.

To do this, a fixed number of additional table entries are reserved for incubating neighbours. Since integer division means that a scaled count is carried forward no more than $\lceil \log_{1/f} r_e \rceil$ epochs, incubation lasts for the same period. When the incubation period finishes, the neighbour is added to the table, or else is discarded if the table is full and no entry can be evicted because they all have higher counts. In addition, a neighbour may be discarded during incubation if it becomes clear it will not reach such a count, in order to free entries for other candidates more rapidly.

4.2.4 Interim placement of new objects

When a file or directory object is created, it cannot be assigned to a group by the clustering process. Instead, its group is chosen directly from the last observed references of the appropriate object type. Each reference is considered in turn beginning with the latest: if the group exists and is not full, the object is placed there. After a certain number of iterations, or if the reference stream is empty, a new group is inserted and the object is placed in that.

When a regular object's final back pointer is removed and its link count falls to zero, it is deleted. Neighbour references to it that may exist in statistics throughout the volume do not affect correctness, and so are removed lazily when the corresponding object's group is considered for reclustering. However until the neighbour becomes 'not recently used' and is evicted as described in the previous section, the effective capacity of the statistics is reduced.

This problem can be exacerbated by files being deleted then recreated within a very short period. Rather than assign such objects a new identifier and contribute to the degradation of their former neighbours' statistics, *Xest* attempts to reassociate such new objects with their previous identifiers and statistics using a small in-memory cache.

Observe that objects recreated with the same name and position in the directory hierarchy will have a back pointer identical to their predecessor. As an object is removed, a mapping is inserted from its final back pointer to its (group table index, object label) pair. When a new regular object is created, its initial back pointer is used to consult this cache. If a

mapping exists, the object adopts that identifier; otherwise, it is placed in a different group as described above.

4.3 Group assignment process

In this section I present the **regrouper** component, which uses a volume's statistics to arrange the objects in an entire volume or in a subset of its groups on to a new arrangement of groups, variable in number and size, as an incremental adjustment to the existing layout.

Section 4.3.1 describes how it builds an in-memory representation of the region to be clustered along with its statistics, and introduces the structures on which the clustering algorithm operates. Section 4.3.2 details the operation of the algorithm itself. With reference to pseudocode, I explain its divisive, agglomerative and overlap creation phases, and justify its scalability. Section 4.3.3 discusses how further consolidation of groups using hints from the directory structure can ameliorate the many small groups typical of weak access patterns. Finally, Section 4.3.4 describes how the volume's metadata, its directory structure and its statistics are adjusted to reflect the outcome of the clustering process.

Pseudocode notation. The pseudocode presented in the remainder of this chapter is written in an object-oriented high-level language resembling Java. I now clarify some of its notation. Compound statements are denoted by indentation and introduced by a colon, as in Python, and comments are denoted by //. Generic types are indicated using square brackets, as in Java. The language has a tuple datatype given as a round-bracketed comma-separated list; assignment to an expanded tuple ignores elements given on the left-hand side as underscores. Constants are written in SmallCaps. The assignment operator is written ←, and other operators use their obvious equivalents from mathematical notation.

4.3.1 Preparation

Each reclustering process has an associated maximum scope, specified either as an entire volume or as a subset of its groups.[4] Groups within this set that have an associated group access record are **included** and participate in the clustering process. All other groups are **excluded**, and no objects are moved into or out of them. In an included group, a **free** object has an associated object access record; objects without statistics are **fixed**. I describe the effect of this distinction in Section 4.3.2. Objects in excluded groups are effectively fixed.

The regrouper starts its preparations by first caching the bitmaps that indicate occupancy of the volume's groups table. It determines the set of included groups by checking that each

[4] I do not discuss policy for selecting the scope of partial clusterings in this dissertation.

exists and, by parsing the statistics groups, has an associated object access record. It then constructs representations of these groups, synthesising metadata and object access records into in-memory structures — instances of ClusGroup and ClusObject, outlined in Figure 4.3.

Storing clustering data in memory avoids unnecessary processing and simplifies the implementation. Although it limits the maximum scale of a single reclustering, this has not proved to be a practical limitation. Most metadata is unused in clustering, and some, *e.g.* object access records, can be processed once into a concise format for use directly by the algorithm. In any case, any whole volume may be rearranged by a series of partial reclusterings.

Next, the file system structure is built. The structure of every included groups is imported, as well as sparse representations of any excluded groups and objects that are the targets of included objects' back pointers or are overlapped in included groups. These are not considered for reclustering but are used when the new group arrangement is applied, as described in Section 4.3.4.

The metadata of each included group is considered in turn, and the ClusGroup and its ClusObjects constructed. Back pointers are direct references to their objects; the name in the directory entry is not stored. If a back pointer or overlap must reference an object in a group that has not yet been initialised, be it included or excluded, the regrouper creates the object as fixed and, if necessary, an enclosing, excluded group. If the group is included, its metadata will later be processed and its objects' details filled in.

This is necessary to avoid retrieving group metadata from the database several times while still allowing ClusGroups and ClusObjects to contain direct references to each other. However, since overlaps are removed lazily, ClusObjects added as the target of an overlap may in fact be found to no longer exist when their primary group's metadata is retrieved. If this is the case, the state is repaired by removing references to the object from that group and any others in which it was overlapped.

Finally, the regrouper processes object statistics. It retrieves the group access record of each included group in turn. It then organises all of the objects' neighbours according to their index in the groups table, and fetches each corresponding group access record one at a time.

For each pair of object and neighbour, the regrouper fills two vectors of type int[], mergeSS and splitSS, by calculating at each index $p < p_{\#}$ the sizes $|S_o(p)|$ and $|S'_o(p)|$, the number of shared neighbours whose significance meets the thresholds required for merges and splits, respectively. Note that these thresholds differ slightly from the distance functions presented in Section 4.1. Each object adds an entry to the other in its edgeWeights map specifying the pair of vectors; the calculation is made once for each pair of neighbours.

```
enum GrpType: FILE, DIRECTORY
enum MemType: PRIMARY, OVERLAP, DELETEDOVERLAP
enum NbrSgnf: ALLOWSNEWOVERLAP, KEEPEXISOVERLAP
abstract class ClusGroup:
    GrpType grpType
    int grpTableEntryIndex
    int priObjCount, fixedObjCount
    long priObjSize
    int totalRefs
    Map <ClusObject,MemType> objects
    abstract boolean isIncluded ()
    abstract boolean isPreexisting ()
    abstract void addObject (ClusObject o, MemType t)
    abstract void removeObject (ClusObject o)
abstract class ClusObject:
    long length
    int totalRefs
    ClusGroup priGroup
    Set <ClusGroup> overlappedIn
    List <ClusObject> backPtrs
    List <ClusObject> modifiedFwdPtrs
    Map <ClusObject, (int[], int[])> edgeWeights
    Map <ClusObject, NbrSgnf> eligibleOverlaps
    List <ClusObject> statsFwdPtrs
    abstract boolean isFixed ()
```

Figure 4.3: Pseudocode structures used in the clustering process.

Further, each object considers its distance from each of its neighbours so that it can record in its own eligibleOverlaps map whether the neighbour can be overlapped in the object's group. ALLOWSNEWOVERLAP and KEEPEXISOVERLAP correspond to $1-l_n$ and $1-l_e$ in Section 4.1.3, the thresholds for creating new overlaps or only retaining existing overlaps during the divisive phase, respectively. In addition, the object lists all of its neighbours, regardless of their significance, in its statsFwdPtrs field for use when the new grouping arrangement is applied.

Two maps, mapGroups and mapObjects, provide access to all of the groups and objects loaded for the reclustering. The ClusGroup methods addObject and removeObject manage the objects map in which both primary objects and overlaps are stored with appropriate ObjMem constants: primary entries override overlaps, and a new overlap is ignored where the object is already primary. The methods also track priObjCount and priObjSize, and maintain consistency with ClusObject's priGroup and overlappedIn fields. The isPreexisting method returns TRUE for groups loaded in for reclustering, and FALSE for groups allocated during the process.

Since a new group can be allocated by a split then immediately deallocated by a merge, the routines allocateGroup and deallocateGroup track free entries in the groups table using the cached bitmaps, and only commit modifications when applying the new arrangement.

4.3.2 Clustering algorithm

In this section I detail the operation of the clustering algorithm with reference to the pseudocode presented in Figures 4.4 and 4.5. Recall from Section 4.1 that directories and files are managed in segregated sets of groups. The procedure clusterGroupsOfType in Figure 4.4 is called with a parameter of type GrpType to perform the clustering process separately for each.

I first summarise this process. In lines 2–5, a divisive phase checks that existing groups remain connected at a sufficient threshold of significant shared neighbours to justify the group's size, splitting them as necessary. Then, in lines 6–17, groups are merged in $p_{\#}$ passes over which both the required threshold and the maximum resulting group size are progressively decreased. A single final pass in lines 18–30 creates secondary associations.

Divisive phase. The algorithm's divisive phase consists of one call per group from line 5 in Figure 4.4 to the splitGroup routine presented in Figure 4.5. The process commences in lines 6–8 by determining the thresholds necessary to retain the group as a whole, by finding the maximal pass in the agglomerative phase at which a merge could have proceeded given priObjSize, the sum size of the group's primary objects. The vector MinSplitGrpSize, whose elements correspond to a constant fraction of those in MaxMergeGrpSize, introduces hysteresis to account for variations in the size of objects between reclusterings.

In lines 13–14 a primary object is selected at random and passed to the routine splitGrpDFSStep which recursively computes a single connected component as a new group g'. Each primary object that it is invoked with is first moved to g' (lines 30–31), then it recurses to each object still in the unconnected component of the group that shares with that object at least as many significant neighbours as the threshold for the pass. Line 34 applies this test for each edge, incorporating hysteresis compared to an equivalent merge in two ways. First, splitSS[p] records the number of shared neighbours at pass p that are significant given a weaker frequency threshold; second, SplitSharedSignif[p] sets a lower threshold for the number of such neighbours.

Lines 38–40 consider overlaps in the unconnected component that are also eligible to be present in the connected component on account of the object being examined. Overlaps added to g' are not removed from g, so that they may be added to further components created as a result of splitting the group.

```
void clusterGroupsOfType (GrpType t):
    // Consider splits
    for ClusGroup g in mapGroups.values():
        if g.grpType = t ∧ g.isIncluded() ∧ g.isPreexisting():
            splitGroup (g)
    // Consider merges
    for int p in [0 ... MERGEPASSES):
        for ClusObject o in mapObjects.values():
            ClusGroup g ← o.priGroup
            if g.grpType ≠ t ∨ ¬g.isIncluded() ∨ o.isFixed() ∨ o.edgeWeights = NULL:
                continue
            for (ClusObject o', (int[] mergeSS, _)) in o.edgeWeights.entrySet():
                ClusGroup g' ← o'.priGroup
                if g ≠ g' ∧ mergeSS[p] > MERGESHAREDSIGNIF[p] ∧
                        g.priObjCount + g'.priObjCount < MAXOBJSPERGRP ∧
                        g.priObjSize + g'.priObjSize < MAXMERGEGRPSIZE[p]:
                    mergeGroups (g, g')
    // Consider overlaps
    for ClusObject o in mapObjects.values():
        if o.isFixed() ∨ o.eligibleOverlaps = NULL:
            continue
        g ← o.priGroup
        if g.grpType ≠ t ∨ ¬g.isIncluded():
            continue
        for (ClusObject o', NbrSgnf s) in o.eligibleOverlaps.entrySet():
            if g.objects.size() - g.priObjCount ≥ MAXOVERLAPS:
                break
            g' ← o'.priGroup
            if g ≠ g' ∧ s = ALLOWSNEWOVERLAP:
                g.addObject (o', OVERLAP)

// Merge a pair of groups and return the reference to the one not removed
ClusGroup mergeGroups (ClusGroup g, ClusGroup g'):
    // Consider swapping g and g' to prefer merges into larger, pre-existing groups
    if (g.isPreexisting() ∧ ¬g'.isPreexisting()) ∨ (g.priObjCount > g'.priObjCount):
        (g, g') ← (g', g)
    // Merge g into g'.
    for (ClusObject o, MemType mem) in g.objects.entrySet():
        if mem = PRIMARY ∨ g'.objects.size() - g'.priObjCount < MAXOVERLAPS:
            g.removeObject(o)
            g'.addObject(o', mem)
    deallocateGroup (g)
    return g'
```

Figure 4.4: Pseudocode for the clustering process

```
void splitGroup (ClusGroup g):
    // Partition the group into a connected and unconnected component until
    // no more splits are necessary or only fixed objects remain in the original.
    ClusGroup g' ← NULL
    while g.priObjCount > g.fixedObjCount:
        // Find the weakest step for which the group may need splitting
        int p ← 0
        while p < MERGEPASSES ∧ g.priObjSize < MINSPLITGRPSIZE[p]:
            p++
        // Allocate the group g' to become the connected component and
        // initiate the DFS from a random primary, non-fixed object.
        g' ← allocateGroup (g.grpType)
        ClusObject o ← g.pickPrimaryNonFixedObj()
        splitGrpDFSStep (g, g' o, p)
    if g.priObjCount > 0:
        // Fixed objects remain. Remove any left over overlaps
        for (ClusObject o, ObjMem mem) in g.objects.entrySet():
            if mem = OVERLAP:
                g.removeObject (o)
    else:
        // Only overlaps remain. Remove g and replace it with the last g'
        if g' ≠ NULL:
            for (ClusObject o, ObjMem mem) in g.objects.entrySet():
                g'.addObject (o, DELETEDOVERLAP)
            replaceGroupAtIndex (g, g')
        else:
            deallocateGroup (g)
void splitGrpDFSStep (ClusGroup g, ClusGroup g', ClusObject o, int p):
    // Object is connected: move it to g'
    g.removeObject (o)
    g'.addObject (o, PRIMARY)
    // Recurse to unconnected primary objects with reqd significant shared neighbours
    for (ClusObject o', (_ , int[] splitSS)) in o.edgeWeights.entrySet():
        if o'.priGroup = g ∧ ¬o'.isFixed() ∧ splitSS[p] ≥ SPLITSHAREDSIGNIF[p]:
            splitGrpDFSStep (g, g', o', p)
    // Ensure that any unconnected overlaps eligible to be connected to this object
    // are added if not already present: they are retained in the unconnected group.
    for (ClusObject o', _) in o.eligibleOverlaps.entrySet():
        if g in o'.overlappedIn:
            g'.addObject (o', OVERLAP)
```

Figure 4.5: Pseudocode for the clustering process' divisive phase

When the depth first search has returned, the group g' forms a connected component and does not need further splitting. If any free primary objects remain in the unconnected component, g, the loop continues. Otherwise, all free primary objects have been allocated to connected components in other groups: only overlaps and fixed primary objects remain in g.

If fixed objects remain, any overlaps are removed in lines 17–19 since they are not eligible to remain when the groups' only primary objects have no associated statistics. Invoking removeObject on an overlap replaces the entry's value with a tombstone marker, DeletedOverlap.

If the group is empty or only overlaps remain, the group can be deallocated. If the loop at line 5 was never entered and no connected component was ever created, the group is simply marked for deletion. However lines 22–25 deal with the common case whereby all primary objects have been assigned to connected components. The reference to the last such group g' is retained and, in order to reduce unnecessary allocation and deallocation of groups, is moved back to the index of g, which is in effect removed. Before that, to ensure that g' correctly assumes tombstone markers for overlaps present only in g, addObject is called specifying DeletedOverlap; overlaps already present in g' override this addition and are retained.

Agglomerative phase. The arrangement that results from any splitting of existing groups is taken as the starting point for the agglomerative phase. This is composed of a fixed number of passes, MergePasses, equal to $p_{\#}$ in Section 4.1.1. In each pass p, shown as lines 6–17 in Figure 4.4, every free object is considered in association with its neighbours, each of which corresponds to an entry in its edgeWeights map. If the two objects are in the same group, the sum size of their groups' primary objects exceeds MaxMergeGrpSize[p], or the new group would contain more objects than it could address, MaxObjsPerGroup, then the merge cannot proceed. Otherwise, the groups are merged at line 17 if mergeSS[p], the number of neighbours that they share whose frequencies are significant at this pass, is at least MergeSharedSignif[p].

The mergeGroups procedure implements the mechanics of combining pairs of groups. It begins by selecting which will be merged into the other. If a group is allocated during the divisive phase, its objects are already set to be relocated when the new clustering is applied. Hence if the other group exists as part of the current volume, moving the former's objects into the latter group incurs no additional work. Otherwise, the group to be retained, the target, is chosen to minimise the number of objects that will be moved. The merge itself is performed by removing in turn each of the source group's objects and adding them to the target group. This routine also ensures that the constraint on the number of overlaps holds for the merged group.

Overlap creation phase. A final pass creates new secondary associations. Free objects of the type being clustered are examined in turn at line 19, after checking their group has fewer overlaps than primary objects. Each of these object's neighbours in its eligibleOverlaps map is examined at line 25. Only entries with value AllowsNewOverlap exceed the greater of the thresholds and permit overlaps to be created: this is done using addObject, which takes account of the object already existing in the group.

Time complexity. I now proceed to sketch an analysis of the time complexity of the clustering algorithm in terms of the number of primary objects n. Bounds are not parametrised by the number of edges as this is a linear function in n, its constant being the maximum number of neighbours of each object, k. I assume that clusterGroupsOfType is called once for each type of group.

Consider the divisive phase in Figure 4.5. In each iteration of the loop at line 5, splitGrpDFSStep moves one or more free primary objects out of group g; when no more objects remain, the loop completes (or is never entered). Observe that splitGrpDFSStep is called at most once for each primary object, since it is invoked only on objects in g and then always removes that object. Each invocation inspects at most $2k$ edges, in $O(1)$ time each. clusterGroupsOfType calls splitGroup exactly once per included preexisting group. Since each primary object is present in exactly one group, splitGrpDFSStep is called at most n times. The remainder of the loop body in splitGrp requires $O(1)$ time.

The second half of splitGroup at lines 15–27 also runs in $O(n)$ time when taken over all calls to the procedure. To see this, observe that the body of the loop at lines 16–19 requires $O(1)$ time, and runs at most once for each primary object and once for each overlap. Since in each group there are at most MaxOverlaps, and the number of groups is at most n, it runs $O(n)$ times. Similarly, the loop at lines 23–24 iterates at most MaxOverlaps times and its body performs $O(1)$ work. Hence the divisive phase runs in $O(n)$ time.

The mergeGroups routine consumes $O(1)$ time. The sum number of primary objects and overlaps in each group is bounded by a constant, MaxObjsPerGroup + MaxOverlaps, and each call to addObject and removeObject requires $O(1)$ time to complete. In addition, deallocateGroup runs in $O(1)$ time. mergeGroups is called $O(n)$ times, at most once in each pass, of which there are a constant number, for each of the k neighbours of the n primary objects. Hence the agglomerative phase requires $O(n)$ time.

The final phase to create new overlaps examines each primary object precisely once. Since the number of entries in the eligibleOverlaps map is no greater than k and addObject runs in constant time, the phase runs in $O(n)$ time.

As such, the clustering process runs in $O(n)$ time, an asymptotic upper bound that makes it suitable for the large sets of objects typically contained in a file system or volume.

```
void clusterDirDist (GrpType t):
    // Populate tree of 'directories' within which to locate file groups
    GrpsAtDir root ← new GrpsAtDir()
    for ClusGroup g in mapGroups.values():
        if g.grpType ≠ t ∨ ¬g.isIncluded() ∨ g.priObjSize > MaxDirHeuMergeSize:
            continue
        Map <ClusObject, long> refsByDir ← new HashMap()
        (ClusObject, int) dirMostRefs ← (Null, 0)
        for (ClusObject o, ObjMem mem) in cg.objects.entrySet():
            if mem ≠ Primary ∨ o.isFixed() ∨ o.totalRefs = 0 ∨ o.backPtrs = Null:
                continue
            for ClusObject bp in o.backPtrs:
                int v ← o.totalRefs/o.backPtrs.size()
                if bp in refsByDir:
                    v += refsByDir.get (bp)
                refsByDir.put (bp, v)
                if v > dirMostRefs[1]:
                    dirMostRefs ← (bp, v)
        if dirMostRefs[0] ≠ Null ∧ dirMostRefs[1]/g.totalRefs > MinDirHeuRefs:
            root.getOrCreatePath (dirMostRefs[0]).groups.add (g)
    // Initiate merge from root
    root.mergeUnderSubtree ()
```

Figure 4.6: Preparing to merge using directory distance

4.3.3 Directory distance heuristics

In pathological cases, the algorithm presented above can generate grouping arrangements in which weak neighbour relationships lead to very small groups of rarely accessed objects. While allowing the arrangement to degenerate to singleton groups, if access patterns suggest it, may minimise the cost of aliasing, in practice these benefits are significantly outweighed by the overheads associated with managing, advertising and addressing storage at this granularity.

Xest addresses this by adopting an additional merge pass that runs after the main clustering process. It heuristically combines small groups on the basis of the position of their objects within the volume's directory structure, a well-established source of implicit hints for predicting access patterns [Tait91, Kuenning97a, Bolosky00].

Pseudocode for this process is given in Figures 4.6 and 4.7. In summary, the routine clusterDirDist arranges groups into a tree of GrpsAtDir instances using the directory structure disclosed by the pattern of their objects' back pointers. Then, the method mergeUnderSubtree is invoked on the root of the tree, causing groups to be merged first with others located under the same 'directories', then with others in immediate subdirectories, proceeding back up the tree.

The process operates separately for file and directory groups, and assesses every included group whose primary objects have a total size below the threshold MaxDirHeuMergeSize.

```
class GrpsAtDir:
    List <GrpsAtDir> dirEntries ← new List()
    List <ClusGroup> groups ← new List()

    void getOrCreatePath (ClusObject dir_o):
        if dir_o.backPtrs = NULL ∨ dir_o.backPtrs.isEmpty():
            return this
        else:
            return this.getOrCreatePath (o.backPtrs[0]).getOrCreateSubDir (dir_o)

    void getOrCreateSubDir (ClusObject dir_o):
        GrpsAtDir subdir ← this.dirEntries.get (dir_o)
        if subdir = NULL:
            this.direntries.put (dir_o, subdir ← new GrpsAtDir())
        return subdir

    void mergeUnderSubtree ():
        // Do depth first post order traversal
        for GrpsAtDir subdir in this.dirEntries:
            subdir.mergeUnderSubtree ()
        // Merge the groups in this directory with each other
        while TRUE:
            if ¬this.mergeGroupsLists (this):
                break
        // Now merge groups in subdirectories into here breadth first
        LinkedList <(GrpsAtDir,int)> q ← new LinkedList()
        this.addDirEntriesToQueue (q, 1)
        while ¬q.isEmpty():
            (GrpsAtDir subdir, int depth) ← q.removeFirst()
            subdir.mergeGroupsLists (this)
            if depth < MAXDIRHEUMERGEDEPTH:
                subdir.addDirEntriesToQueue (q, depth+1)

    void addDirEntriesToQueue (LinkedList<(GrpsAtDir,int)> q, int depth):
        for GrpsAtDir subdir in this.dirEntries:
            q.addLast ((subdir, depth))

    boolean mergeGroupsLists (GrpsAtDir ancestor):
        boolean anyMerges ← FALSE
        for i in [0 ... ancestor.groups.size()):
            ClusGroup g ← ancestor.groups[i]
            if g = NULL ∨ g.priObjSize ≥ MAXDIRHEUMERGESIZE:
                continue
            for j in [0 ... (ancestor = this ? i : this.groups.size())):
                ClusGroup g' ← this.groups[i]
                if g' ≠ NULL ∧ g.priObjCount + g'.priObjCount < MAXOBJSPERGRP ∧
                        g.priObjSize + g'.priObjSize < MAXDIRHEUMERGESIZE:
                    if mergeGroups (g, g') = g':
                        ancestor.groups.set (i, g')
                        g ← g'
                    this.groups.set (j, NULL)
                    anyMerges ← TRUE
        return anyMerges
```

Figure 4.7: Pseudocode for the directory distance merge process

Each of a group's objects contributes its number of total references in votes for the directory corresponding to its back pointer; multiply linked objects distribute their votes evenly over their back pointers. The group is assigned to the directory having most votes, so long as that exceeds a given proportion of the references, MinDirHeuRefs: otherwise, there is unlikely to be good locality between its scattered objects and groups at nearby directories.

A sparse directory structure is constructed on demand from GrpsAtDir instances whose groups field stores lists of associated groups. A root node is created, and its getOrCreatePath method is invoked with the ClusObject of each directory. It recurses through the directory's back pointers to the root directory (the only object without any) then builds subdirectories as the call returns.

The process of merging groups is initiated by calling mergeUnderSubtree on the root note. This conducts a depth first post-order traversal of the tree, shown as lines 16–17 in Figure 4.7. It first merges groups with others in the same directory, then performs a breadth-first traversal of its subdirectories, whose groups have already been merged with each other, and considers merging them with each group in the ancestor directory. Subdirectories are visited up to a maximum depth of MaxDirHeuMergeDepth. Merging a pair of groups removes the group from the descendant and replaces the one in the ancestor, so that a sequence of merges can process up the tree.

The merging of two vectors of groups is implemented by the mergeGroupsLists method: the range at line 39 ensures that it inspects every pair once, even if invoked on itself. Note that each list may contain Null elements. If a merge can proceed, but mergeGroups removes the group in the ancestor, its entry is overwritten with the extant group and the reference to g updated to allow further iterations of the inner loop to proceed.

4.3.4 Applying the new clustering

Once the clustering algorithm has completed and the in-memory structures reflect the new grouping arrangement, the updated configuration of groups and objects has to be applied back to the metadata of the active volume and its existing groups. This process comprises three stages: first, moving objects and creating and removing overlaps; second, updating directory entries and objects' back pointers to restore the volume's directory hierarchy; and finally, repairing the structures that store the volume's statistics.

Correct group membership. The regrouper performs most of the work required to rearrange objects to their new groups in a single pass. Iterating ClusGroups in ascending order of their group table indices, it retrieves in turn the metadata of each group that has had: (a) any primary objects moved *into* it; (b) any overlaps in it created or updated; (c) any pre-existing overlaps removed. Any groups allocated during the clustering process are first cre-

ated. To accommodate this, the volume storage interface is extended so that the regrouper can create groups at specific indices in the groups table, a requirement that parallels the observer creating objects with specific labels.

First, new overlaps are created and pre-existing overlaps that are no longer present in the group are removed. To assist this second task, explicit tombstone markers are maintained during the reclustering process, namely DELETEDOVERLAP values in ClusGroup's objects map. Without these, the metadata of otherwise unmodified groups would have to be retrieved and its list of overlaps compared in order to detect removals.

Next, objects being moved into the group are organised by their previous group, then the metadata for each source group is retrieved, and the necessary objects' metadata moved to the new group. In addition, the regrouper checks whether each object is local to its former group, in which case its associated data is relocated in the on-disk cache to reflect its new identifier.

Recall that since an object's label is only unique within its own group, an object that is moved to a new group may require a new label. Rather than undergo the costly detection of rare namespace collisions each time that an object is moved during the clustering process, the regrouper assigns new labels if a clash occurs while transferring an object's metadata. However, assigning a new label to an object part way through the pass means that overlaps to it in groups with a lower groups table index will now be incorrect (overlaps in groups at higher indices have yet to be updated). Rather than separate the passes for primary objects and overlaps, *Xest* optimises for the common case and goes back to repair any invalidated overlaps after the single pass completes.

Note that the maximum number of objects per group is set to be a fraction of the size of the object label space, which minimises both the occurrence of collisions and the expected number of probes required to find a free identifier if a collision does occur. In any case, the process may need to store double the usual maximum objects in a group while its membership is corrected, which can itself be a cause of false collisions and complicate attempts to pre-assign unique labels.

Repair directory structure. Once the arrangement of objects and overlaps in groups has been applied, two aspects of the volume's directory structure must be repaired. Recall from Section 3.3.3 that directory entries embed their target object's groups table index, and that objects maintain back pointers to identify their directory links. The repair process involves two tasks, shown in Figure 3.3. If an object is moved, its back pointers are used to update all of its referring directories. If a directory is moved, the back pointers for all of the objects referred to by each of its entries are updated. Of course, both procedures may need to be applied for some objects.

A directory determines which, if any, of its entries need correcting using ClusObjects' modifiedFwdPtrs lists, which the addObject and removeObject methods maintain during the clustering process using objects' back pointers. This avoids the repair process having to read and parse every directory's contents, then check each entry to see if its target object had been moved. Note that directories in need of updating may be in both free and fixed groups, but all have corresponding ClusGroup and ClusObject structures, since every back pointer from an included object is followed during construction.

If the list is non-empty, the regrouper loads the directory and corrects entries whose identifiers correspond to the former identifier of any object in the list. Since ClusObject back pointers do not store directory entry names, the process iterates the whole directory.

Since directory entries are not incorporated into the clustering structures when they are prepared, when a directory changes groups its contents must be parsed. Consequently its entries may refer not only to included and excluded ClusObjects, but also to objects not present in the partial clustering. In preparation for updating their back pointers, the entries' targets are placed in a structure organised by their groups, using their identifiers in the new clustering if applicable. In a subsequent pass, after having examined all groups that need moving, this structure is used to fetch the metadata of each required group in turn, and replace any of its objects' out-of-date back pointers. In this way, backpointer updates are batched together, so that each group's metadata need only be retrieved and modified once in order to reflect the movement of all directories in the file system.

Repair statistics. The structures used to store access statistics must also be updated to reflect the new clustering arrangement, so that immediately repeating the same partial clustering would result in the same arrangement, and so that new references can build on existing statistics.

First, the object access record of each relocated object is moved to the group access record corresponding to its new identifier. This proceeds in a similar manner to relocating objects' metadata, in a single pass in which each step considers object access records moving into that group. Second, each object's access record updates the identifiers of any of its neighbours that have been moved. Recall that a ClusObject lists all of its neighbours in its statsFwdPtrs field, regardless of their significance. This is used to determine whether the access record needs to be retrieved and its entries corrected.

4.4 Summary

In this chapter I presented *Xest*'s group allocation components. I described an approach that supports variable-sized groups, secondary associations and incremental and partial

clustering. I showed how each reference to an object incrementally built statistics for the clustering process, and how these statistics are built on structures introduced in Chapter 3.

I then described the process of reallocating a volume's objects between its groups. I presented algorithms for each stage of the clustering process, and set out how the new arrangement was applied to the volume and group membership, directory structure and statistics repaired.

Chapter 5

Experimental methodology

In this chapter I describe and motivate how the *Xest* prototype is evaluated. Demonstrating the practical viability of the system's design and implementation is a key aim of this work. Section 5.1 argues that measurement of global distributed services must be carried out 'in the wild', and that local and cluster-based simulation methods cannot recreate either the network or resource conditions that are imposed on real distributed services. As funding constraints and the maturity of the platform's control tools currently prohibit a wide-area deployment of XenoServers, the file system accesses of a set of migrating virtual machines are instead simulated using trace replay over a distributed deployment of *Xest* on PlanetLab [Chun03, Bavier04], an existing distributed test bed. Rather than simulating the underlying network or the resources available to the service, only the clients are simulated.

The remainder of this chapter develops this framework. Section 5.2 describes existing tracing and workload generation tools, and details a new VFS-level trace toolkit whose operations syntactically resemble the interface presented by in-kernel or FUSE file systems. Section 5.3 considers the selection of workloads and Section 5.4 describes how traces are replayed to measure read completion delays, inferring concurrency and blocking dependencies in the trace, and how the trace is frozen and moved between nodes to mimic virtual machine migration.

5.1 Evaluating global distributed services

Building bespoke tools for gathering and replaying traces is in itself a significant undertaking. I set out to justify the work involved in evaluating *Xest* over PlanetLab on the grounds that real distributed testbeds feature environmental conditions that are difficult to otherwise recreate. The intention is that the results are significantly more faithful to *Xest*'s target setting.

First, wide-area networks exhibit conditions and failure modes that are very difficult to recreate when modelling nodes' network layers using discrete-event simulators or emulation tools such as ModelNet [Vahdat02], especially for services spanning large numbers of nodes. These conditions include temporary non-transitivity of communication, network partition, message queueing and reordering, and large variations in cross-traffic. Some factors are dynamic and affected by feedback, further complicating accurate simulation: for example, a service's response to congestion affects future levels of congestion. Network conditions such as these have a significant performance impact on network-bound distributed services [Muir04, Rhea04].

Second, large variations in the performance across the members of a large set of nodes have been observed over short timescales, due not only to heavy resource sharing but also to machine misconfiguration and hardware failure. Such 'stragglers' can affect the performance of distributed operations by an order of magnitude [Rhea05c, Dean04]. Again, accurately recreating such performance effects in simulated or modelled environments is very difficult.

Third, lessons from the deployment of real distributed services are arguably an important contribution to systems research in their own right. Managing the challenges of wide-area distribution is prerequisite to a successful deployment over a set of XenoServers. In this setting, *Xest* would be subject both to real wide-area network conditions, and also to stragglers, despite Xen's stronger isolation properties: sharing still affects the price and availability of resources and provides no guarantee against misconfigured or defective hardware.

Unfortunately, deploying *Xest* over PlanetLab, rather than a XenoServers testbed, complicates my evaluation by requiring some components of the latter to be simulated, as Figure 5.1 illustrates.

First, *Xest* anticipates workloads that reflect whole system accesses to complete root file systems; in order to test the implementation's caching and migration components, these workloads must be able to be migrated mid-experiment. Without a virtual machine setup in which to run real clients, the file system's clients must be simulated.

In addition, access to the file system by the simulated clients must be simulated. In a regular deployment, the *Xest* prototype would run at user level, using FUSE [Still04] to allow access to it through the conventional file system interface. However on PlanetLab, which prohibits kernel recompilation and module insertion, operations need to be replayed directly to it at user level, should resemble the operations that would have been forwarded to it by FUSE, and should respect inter-process dependencies present in the workload.

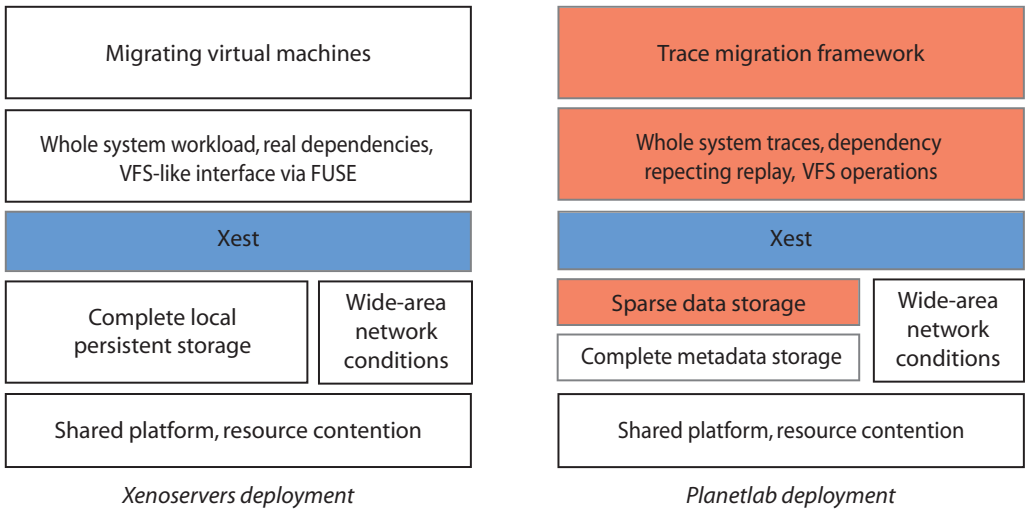


Figure 5.1: Approximating a XenoServer deployment over PlanetLab. Due to feature and resource constraints imposed by the PlanetLab environment, the workload of a set of migrating virtual machines must be simulated, and storage of data conducted sparsely.

Finally, since the storage capacity available to Planetlab slices is insufficient to accommodate large file systems, file data must be stored sparsely, but doing so should impact disk performance as little as possible.

Despite these deviations from *Xest*'s target environment, a PlanetLab deployment can still capture the interaction of file system nodes over a wide-area network, around which the evaluation centres, more effectively than emulated or simulated approaches. To complete this deployment framework, the remainder of this chapter presents methods for simulating the whole system workloads of a set of migrating virtual machines.

5.2 Simulating file system activity

Simulating file system activity involves selecting a balance between two contrasting approaches. In the first, *workload generation*, requests that result in file system operations are generated synthetically with the aim of replicating the overall balance, pattern and timing of operations seen previously in a reference workload. In the second, *trace replay*, the individual operations that comprise the reference workload are recorded, and that trace is later replayed in step with the intention of recreating an identical sequence of file system requests.

Application benchmarks generate file system activity as a side-effect of exercising a sample of applications or utilities. As such they typically require installation of test applications and their operating system dependencies. For example, SpecWeb [SPEC99] re-

quires web server software to be installed and configured, and the SYSmark benchmark suite [SYSmark02] requires an installation of Microsoft's Windows and Office software. This not only makes them unfeasible to run on PlanetLab, but it is also incompatible with measuring whole system disk activity, and complicates the migration of client workloads.

Synthetic workload generators, on the other hand, aim to specifically reproduce disk operations on the basis of the profile of a reference workload, and this simplicity of configuration makes them easily adapted to mimic client migration. However, since they are not accurate reproductions of a client's activity and aim only to be representative of a reference workload in aggregate, most model transitions between adjacent references as independent. However, file accesses in many workloads exhibit spatial locality, the detection of which forms the basis of the techniques discussed in Section 2.3, including those used by *Xest*'s observer. This limitation affects most existing workload generators, including PostMark [Katcher97] and Bonnie [Bonnie96].[1]

A recorded trace, in contrast, is a precise reproduction of a client's file system activity. Nevertheless, there are two notable challenges in replaying the recording as the source of a file system workload. First, traces need to be recorded at an appropriate logical level and with sufficient detail. Second, whereas synthetic workloads are generated interactively, recorded traces are static, yet the workload needs to adapt when the operation or timing of the file system being tested deviates from that of the recorded file system. The subsequent sections of this chapter address these challenges.

5.2.1 VFS trace toolkit[2]

In order to record traces suitable for replaying as accurate file system workloads, I have developed a new general-purpose VFS level trace toolkit that captures operations at the same detail and logical point that FUSE or an in-kernel file system would process them.

Existing tools take a variety of approaches to recording trace data. Many perform tracing at system call entry points, producing traces that are comprehensive with the notable exception of memory-mapped accesses, an increasingly prevalent class of operations [Roselli00]. While replaying such workloads from user space to a VFS file system is both precise and simple, replaying operations accurately to *Xest* from user space would necessitate recreating what is essentially the operation of the VFS, including inode and dentry caching, the processing of textual paths and the maintenance of process' file tables and current working directories. In contrast, recording events within the VFS allows accesses satisfied from the inode and dentry caches to be excluded, while including operations due to accesses of memory-mapped file regions.

[1] This problem is well documented in the literature; further references are given in [Kroeger01].
[2] The trace toolkit uses code contributed by Steven Smith and Henry Robinson.

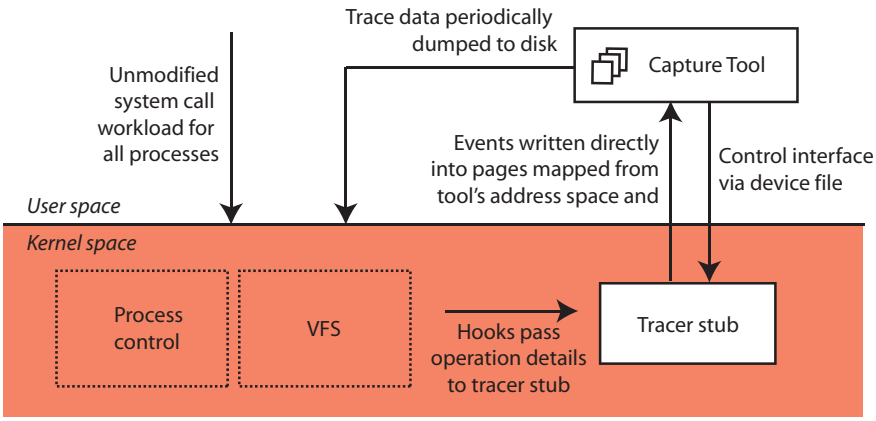


Figure 5.2: The structure of the VFS trace toolkit

Passive network tracing deduces operations from the protocol messages sent between clients and servers by file systems such as NFS [Blaze92, Ellard03a]. While the NFS protocol in some ways resembles VFS operations more closely than system calls do, it also elides details of many client operations, despite previous work on post-processing. NFS hides open and close calls, scheduling reorders messages and client caching suppresses many operations and transforms others into revalidations. Process operations such as fork and wait are also unavailable.

Two existing tracing tools are realistic candidates for my purposes. TraceFS [Aranya04] captures VFS operations in sufficient detail but does not trace process operations such as fork and wait. Unfortunately, its license prohibits modification. An open-source alternative, the Linux Trace Toolkit [Yaghmour00], captures a wide-range of in-kernel events but does record a sufficient range of file system calls with the necessary detail. Rather than extend this project and adapt it for the Xen kernel I was using, I instead built a lightweight alternative.

My trace toolkit captures events as follows. The Linux kernel is instrumented to enable the recording of VFS and process control operations. Trace events are recorded from 31 different points. For VFS operations, events are generated at the point that a file system implementation's method or the equivalent VFS default returns, having first recorded the time at which that routine was entered so that the duration of the operation can be recorded. Fork and clone calls are traced as the parent process is duplicated, and are recorded without a duration. Wait events are generated as the parent is removed from the wait queue, and record the time that it spent on the queue. Table 5.1 summarises the traced operations and the structure of their records.

Figure 5.2 illustrates the process of recording a trace event. In addition to modifications to existing kernel code, the tracer consists of an in-kernel stub and a user-level tool that

Timing information	$time_{stop}$ $time_{len}$
Process information	pid tgid fsuid fsgid
Inode operations	retval $device_{dir}$ $inode_{dir}$ $inode_{ent}$
mknod	$device_{node}$ mode name
mkdir, creat	mode name
setattr	mode uid gid len
rename	$inode_{dest}$ $name_{from}$ $name_{dest}$
symlink	target
lookup, link, unlink, rmdir, readlink, followlink, getattr	name
File operations	retval $device_{file}$ $inode_{file}$
mmap	offset len flags
ioctl	cmd
open	flags
read, write, sendpage, readv, writev	offset len
flush, release, readdir, poll	–
Process operations	pid_{child}
fork, clone	flags
wait, waitpid, wait4	flags retval

Table 5.1: The structure of operations recorded in the trace. All events carry timing and process information (whose fields correspond to Linux process, thread group, and effective user and group identifiers). Operations carry fields for both their operation type (lines listed in bold) and the operation itself.

communicate using a device node created by the stub. The stub is passed events; if tracing is enabled, it annotates each one with process and timing data common to all events, then writes it into a ring buffer of pages specified by the user-level tool. The tool polls for notifications for data in the buffer, and dumps it out to disk. The raw data is post-processed and compressed offline, in order to simplify the record structure and filter operations which refer to excluded parts of the file system. Note that events are ordered by their completion time, not by their start time, an issue returned to in Section 5.4.4.

5.2.2 Capturing initial file system state

Recording a file system's directory, link and inode state at the point at which tracing commences is an important part of ensuring replayed operations are faithfully reproduced at the destination site.

Without initial state, a file system's structure has to be inferred from the contents of the trace. There is no way to include files and directories present in the traced file system but not referenced in the trace. For *Xest*, this would affect storage consumption, directory lookup performance, and group membership.

Even for referenced files, some information cannot be derived from trace events alone. Without link count information, an unlink operation cannot determine whether the traced file system destroyed the inode and file data. Inferring link count means hard linked files may be misrepresented; similarly, for system call level traces such as SEER's, textual paths mean that symbolic links cannot be distinguished from separate entries. These problems result in false duplication of storage, metadata, caching, and observed access patterns. Inferring file sizes is also problematic. Derived distributions tend to be poor fits with observed distributions [Douceur99], and a file's type affects both its access patterns and its size [Vogels99].

Despite this, few trace studies record initial state. Joukov *et al.* [Joukov05] recognise the importance of initial state in accurate trace replay, but propose using the snapshot mechanisms supported by some file systems. This approach suffers a number of drawbacks. First, such a mechanism is rarely available. Second, it does not provide for the construction of traces that span multiple real file systems, or are formed from sparse regions thereof (both of which are requirements of the web workload described in Section 5.3). Third, it records the whole file system state in a manner that is difficult to transfer to remote test machines, such as PlanetLab nodes.

Instead, I adopt an approach that takes advantage of the fact that this evaluation does not need a replica of the original file system, only the metadata required to regenerate a sparse representation of it inside *Xest*'s cache structure, as described in Section 5.4.1. This saves storage and bandwidth on PlanetLab, and means that file sparseness information need not be measured. A set of scripts traverses the directory hierarchy breadth first, so that during regeneration every entry's parent will have been created previously. The name, type, device and inode of each entry is recorded; the inode's fields, including size, link count, ownership and permissions, are stored separately to eliminate duplication of storage for hard linked files.

Unlike with atomic snapshot mechanisms, it is possible that modifications will be made to the file system while the initial state is being captured. This does not affect the internal consistency of the captured state, only how accurately it represents the contents of the traced file system; in practice the effects are negligible in a quiescent system.

5.3 Workload selection

The trace toolkit was used to gather traces for three categories of workload. This section describes how these traces were recorded; their detailed characteristics are evaluated in Section 6.1.

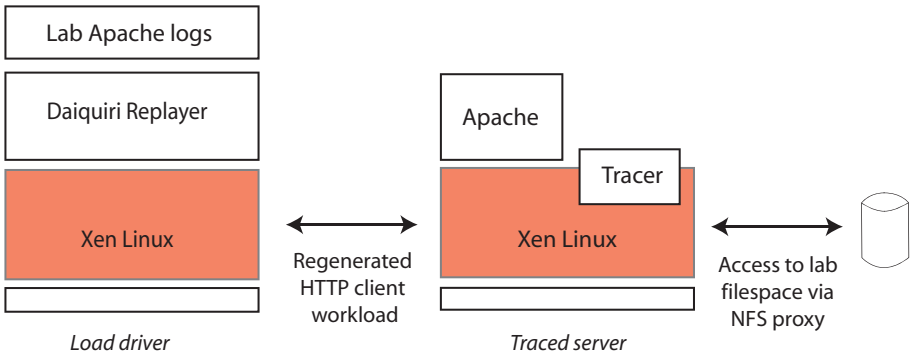


Figure 5.3: Capturing web server workloads

The first, long-running web server workloads, aims to be a candidate for the class of location-sensitive services that a distributed computing platform might host. The second and third are 'microbenchmarks' with more frequent I/O-bound periods: the booting of virtual machines is an important test case for *Xest*, and Linux kernel builds exercise write performance.

5.3.1 Web server workloads

The University of Cambridge Computer Laboratory uses Apache [Apache07] to host its web site[3] containing teaching and research information as well as home pages for staff and students. Its profile of client requests is a suitable server application workload from which to generate a file system trace, but since the web server is critical to the organisation, instrumenting it directly was not possible.

Instead, a test machine running Xen was used to host a Linux domain instrumented with the trace toolkit, running Apache and set up as to mimic the Laboratory's web server. On a separate machine, Daiquiri [Schlossnagle00], an Apache log replay tool, was used to replay logs recorded by the web server in order to reproduce the HTTP workload of all of the clients. This arrangement is shown in Figure 5.3. Although the server under test was quiescent other than by the load generated by requests to the web server, the traces deliberately capture all of the virtual machine's file system activity, including housekeeping operations performed by Apache and periodic operating system tasks.

The Laboratory's shared file system is several terabytes in size and is accessed through a number of separate NFS automounts. However, the data accessible through the web server is logically scattered throughout it, and represents only a small fraction of the total storage. Since a migrating virtual machine serving this web site would not store the organisation's private storage in its virtual disks, a set of scripts was developed to exclude these areas.

[3] Available at http://www.cl.cam.ac.uk/

First, the complete set of Apache logs are parsed, and the targets of each request are interpreted after applying the URL rewrite rules used by the web server. This generates a set of all the directories accessed, for inclusion in the virtual disk. This set is processed to generate a directory tree from which the initial state of the traced file system is built: 'included' directories are processed as usual, but 'sparse' directories list only the subdirectories necessary to connect an 'included' descendant into the tree. As such, the instrumented machine has an identical file system layout to the usual web server, but portions private to the Laboratory are absent from the corresponding *Xest* virtual disk.

5.3.2 Kernel build and virtual machine boot workloads

In addition to the web server workloads, two microbenchmark-style workloads were gathered from a Linux guest domain instrumented with the trace toolkit. Precise details of the set up are given in Section 6.1.

In the first, a Linux kernel source tree installed on the local disk was compiled, its modules built, then finally the source tree was cleaned. This process was repeated after rebooting the test machine, yielding two traces of separate builds. A second pair of traces recorded the file system activity during a domain's boot sequence. Tracing was initialised early in the domain's /etc/rc.d/rc.sysinit script, and stopped after login.

5.4 Reproducing traced workloads

This section describes how traces recorded using the VFS toolkit can be used to generate a workload of migrating clients for evaluating *Xest*.

First, Section 5.4.1 describes how a file system's initial state is used to generate a *Xest* virtual disk, and how the storage subsystem is modified to support sparse replay, so that large virtual disks can be represented on PlanetLab nodes with constrained disk space. Given this virtual disk, Section 5.4.2 addresses the translation of (device,inode) identifiers in trace lines into group and object identifiers.

The remainder of the chapter describes *Xest*'s trace replayer. In Section 5.4.3 I argue that modelling both the concurrency and sequentiality between trace operations is important for realistically capturing object transitions during replay. Section 5.4.4 presents the implementation of *Xest*'s replayer according to this structure. Finally, Section 5.4.5 details how trace replay can be migrated between hosts to order to model wide-area virtual machine migration.

5.4.1 Recreating the traced file system

Before a trace's replay can commence on PlanetLab, the file system whose operations it records must be regenerated as a sparse *Xest* virtual disk at the remote node using the metadata gathered by the initial state scripts.

A new virtual disk is created and the operation proceeds by adding in turn each directory entry listed in the initial state. Recall that the list of directory entries is a breadth first traversal of the hierarchy, so that any file or directory's parent appears earlier in the list than itself, excluding the root directory, which is first. The (device,inode) pair in the directory entry is used to locate the values of the inode's fields in the separate, sorted, inode file. The import routine then uses the usual FUSE client routines to create the object and set its permissions and ownership. In the case of a file, it is also truncated to set its length.

Recall that in order to simulate the operation of large virtual disks on a shared testbed that imposes disk usages quotas, *Xest* stores no file data. This means that the initial state of the file system, as well as write, writev and sendpage operations in the trace, need only detail metadata and operation arguments, respectively, and not the contents of files or buffers.

The virtual disk layer is modified to tag all storage operations on objects as either regular, for directories and object statistics, or sparse, for file data. The effect on disk performance is minimised by relying on Linux's support for sparse files. Writes cause buffers to be allocated but discarded in the disk subsystem; no content is written to sparse cache files, and only their length is modified. This allows reads to proceed as usual, albeit for zero-filled regions. The storage of group, volume and virtual disk metadata is managed using Berkeley DB and is unaffected.

5.4.2 Looking up objects from inodes

Recall that *Xest* embeds an object's identifier and group table index into each directory entry that is a link to it. While FUSE and VFS clients could obtain these values as they walk textual paths and store them in structures accessible from inodes' extension fields, when replaying a trace event only the (device,inode) of the object concerned is available, since the path lookup has already been carried out by the VFS on the instrumented machine.

Because of this, a mapping must be maintained between (device,inode) pairs and the corresponding object identifier and group table index; moreover, regrouping may update this mapping so traces cannot be statically rewritten to incorporate it. Instead, an 'inode translation map' is maintained in memory during replay to minimise the effect on performance of this translation. As files and directories are created (either when recreating a virtual disk's first volume or during replay) or destroyed, mappings are inserted or removed. When its

corresponding volume is forked or migrated (as described in Section 5.4.5), the evaluation copies the map between nodes.

In order to minimise its memory footprint, the map takes advantage of ext3's inode assignment patterns. Blocks of sequentially numbered inodes for a given device are stored in unstructured byte extents and unpacked into separate objects on the heap only when needed. Rather than maintaining a reverse mapping by group table index and object label throughout replay, so that the regrouper can lookup and update inodes whose identifiers have changed, the update process instead iterates every inode in the map and uses the clusterer to determine whether it requires updating.

However, one further complication arises from the assumption that a reverse mapping is only necessary during regrouping. When a rename operation overwrites a directory entry that is the sole link to its inode, that inode should be removed from the mapping, but its (device,inode) pair is not available from the trace operation or the directory entry. In response to this, the inode translation map maintains an additional list of removed inodes whose absence is reflected in lookups to the map but whose block entries are only removed lazily when the corresponding volume is closed.

5.4.3 Trace replay by dependencies

The blocking nature of the system call interface is an important factor in file system performance. Emulating it accurately during trace replay is a prerequisite to measuring, in particular, the effectiveness of prefetching and grouping techniques. This necessitates observing the concurrency and sequentiality of operations, not only between blocking calls issued by a single process, but also between parent and child processes.[4]

Figure 5.4 illustrates how trace replay is structured according to the partial order on operations imposed by the blocking dependencies and by the time taken for replayed operations to complete. Processes are modelled either as requiring fixed amounts of processing time between pairs of adjacent operations ('think time'), or as issuing operations in reaction to external events, such as a web server that reads files in response to HTTP requests. The replayer models the latter class of 'reactive processes' as special cases, which are specified specifically (for example, as the process tree that results from running specific executables).

The replayer treats a system call of the fork or wait families as a serial dependency of a child process on a parent process, or vice versa, respectively. The availability of process events means that the replayer is able to determine whether a process is runnable as a trace commences, and at what points a runnable process must be blocked to wait on a child.

[4] In this section, *process* refers in general to a thread of execution, and specifically to a Linux lightweight process.

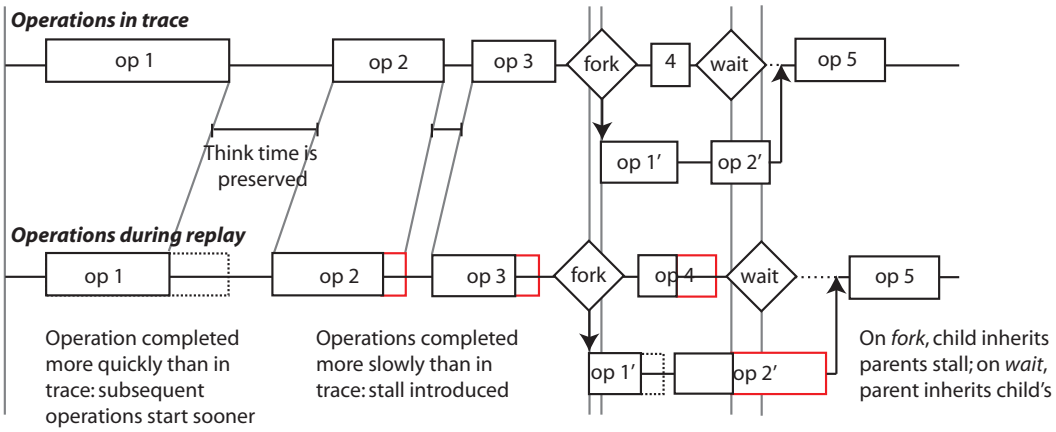


Figure 5.4: Dependencies between traced operations.

Since many Unix applications and scripts rely on executing child processes to do work, accurately replicating the control flow in these workloads is important.

The timing of events in the trace is used only to calculate the 'think time' between each pair of consecutive operations due to the same process. For non-reactive processes, this is reproduced as an idle period whose duration is fixed, regardless of whether operations complete at different rates from their equivalents in the trace; that is, if a replayed operation takes longer than it did when traced, the issuing of the next operation is delayed by that difference. For reactive processes, the 'think time' is treated as the interval until the next external event: subsequent operations are never brought forward, and are only delayed if the previous operation has not completed when the next would be issued.

Recreating a file system's behaviour from a trace is an inherently approximate task, due to the variations in timing between traced and replayed operations. This has two important effects. First, the inode and dentry locking enforced by the VFS, and respected if replay is performed in lock step, is insufficient when the relative timing of operations change. Second, since it is not possible or desirable to model precisely the behaviour of processes, nor their inter-dependencies, operations may be issued in an order that the replayer estimates to be valid but that in fact is not.

The replayer addresses these problems by implementing its own locking (contention for which can itself affect replay timing), and attempting all operations on a best-effort basis. It checks operations' parameters and return values, and records the frequency of inconsistencies, though no trace operations have been observed to fail because of this.

Previous file system trace replay tools differ in their modelling of think time and are unable to respect inter-process dependencies, since they assume that process control events are not recorded.

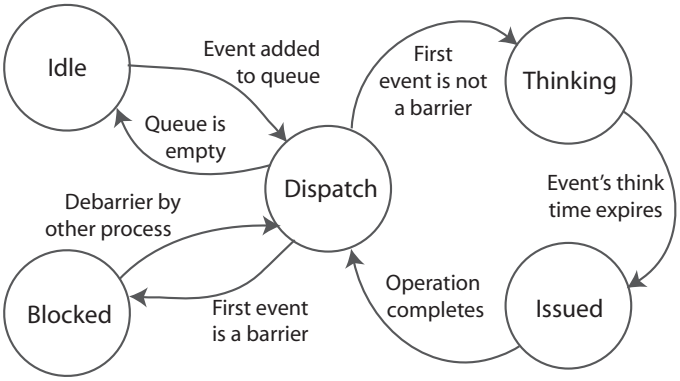


Figure 5.5: State transition diagram for simulating traced processes.

Replayfs [Joukov05] uses a separate thread to issue blocking VFS operations associated with each process, so a delay in completion of the replayed operation is able to delay its own process, but it cannot delay others: there is no means of determining when a process's first operation should be issued other than by its trace timestamp. In addition, an operation is issued no sooner than its trace time, and at the earliest point after that at which its process is not blocked. Hence all processes are modelled as reactive.

TBBT [Zhu05], an NFS trace replay tool, presents two ordering policies. 'Conservative' uses timestamps to form a total order on events and admits no concurrency between processes. 'Dependency' forms a partial order that only preserves serialization of writes and fails to capture many application-level dependencies. DFSTrace [Mummert96] records fork events only to determine user information for other operations during post-processing; few details of its replay tools are available.

Buttress [Anderson04] focuses on precision of event issue, and separates the logic for determining event ordering using a set of primitives that are sufficient to express the dependencies discussed in this section. However, no specific orderings are discussed. Recall that *Xest* adopts some of Buttress' techniques for improving timer precision, as described in Section 3.2.3.

5.4.4 Replayer implementation

The replayer issues operations according to the constraints set out above by reading events into, and managing dispatch from, per-process structures. Each process has associated with it a queue of outstanding operations; a list of the pids of its child processes; a **time offset**, positive or negative, relative to events in the trace; and the trace completion time of the process' previous event, used to calculate the 'think time' preceding the next event.

Each process dispatches events according to the transitions described in Figure 5.5. Except when about to migrate a trace (see Section 5.4.5), the operation at the head of every process' queue is in one of the following states:

Thinking: Due to become issued once a timer set to model its think time expires.

Issued: The operation has started but not yet completed.

Blocked: The operation is a **barrier**, the object of a cross-process dependency which is blocking the process.

When an operation has completed, it is removed from the head of its process' queue. The difference between the durations of the replayed event and its trace equivalent is calculated and added to a **global delay**. If the process is non-reactive, it is also added to the process' time offset; this is the sole implementation difference between the two types of process.

Cross-process dependencies are enforced as follows. A barrier in process A does not itself complete, but rather is removed by a **debarrier** operation called from process B when a dependency of process A on B is satisfied. This operation first locates the matching barrier in A's queue. If it exists, it removes it; if the barrier was at head of the queue, which will always be the case for fork operations, it sets A's time offset to B's to account for the time A spent blocked waiting for B, and A's first subsequent event is set in progress.

A dependency is introduced as a process control event is read from the trace. For a fork event, the child process is created, a barrier added to its empty queue and a debarrier operation appended to the parent's queue. For a wait event, a barrier is appended to the parent process' queue. Then, if the event specifies any child process, the parent's child list is used to add debarrier operations to all children, so that the first one issued unblocks the parent; if the event specifies a particular child, a debarrier operation is added only to that one's queue.[5]

Events are read from the trace sequentially, asynchronously and in batches, and a trace pointer is updated to record the line beyond which no operation has been read. Events must be fetched sufficiently far ahead that at no point does a process appear idle when in fact an operation could have been issued; this invariant is verified as each event is prepared. In addition, before replay commences, the replayer must have read the first events for all threads running concurrently at the point that tracing began, since those events are issued immediately.

Two factors complicate the read-ahead. First, recall that operations in the trace are ordered by their completion time on the instrumented system, not by their issue time, and wait

[5] Since wait events only inform blocking, calls with the W_NOHANG flag set are ignored. In addition, support for waiting by process group is not implemented, since such calls do not occur in the traces used for this work.

operations can be outstanding for many seconds. Second, processes with different time offsets read their next events from different offsets in the trace. During a buffering phase before replay commences, and once replay starts as every operation completes, the replayer ensures sufficient traces lines are enqueued using two heuristics: first, the number of such events is greater than a certain value; second, the most recently fetched event's timestamp is more than a constant period ahead of the effective current time of the process with greatest offset. In practice, values of 5000 events and 30 seconds, respectively, have proved sufficient.

5.4.5 Simulating virtual machine migration

The migration of the workload of a client virtual machine is simulated by pausing a trace being replayed on one machine, recording the state that describes the replay's progress, then transmitting that state and using it to resume replay on a different machine. This section describes the implementation details necessary to effect this process.

First, recall that *Xest* itself cannot migrate an active volume, but rather provides pre-copy techniques that aim to minimise the cache effects of forking a recently fixated snapshot on another, pre-determined host. So before a workload is migrated, outstanding operations are allowed to complete; then all open files are closed and the volume itself is fixated.

When pre-copy completes and replay is paused, each process that has a thinking operation in progress, *i.e.* one waiting on a timer that will issue it, cancels the timer using the token stored in its state and records the think time already elapsed for the operation in the migration state. Once the number of issued but uncompleted operations falls to zero, the trace migration commences. This records every active process' identifier, time offset and the events in its queue, along with the identity and reference count of all open files. Because traces are very large, and so are slow to transfer between nodes, they are distributed to all nodes before replay commences, so that the migration state need only refer to events' line numbers. During resume, events in process queues are fetched individually from the trace file, then buffering proceeds as usual from the line at the trace pointer, which is itself transferred in the migration state.

In order to minimise the outage between pausing and resuming replay due to transferring the volume's inode translation map to the destination node, the complete map is sent in the background as soon as migration is notified. Modifications to it during the pre-copy stage are recorded and a diff constructed and sent when replay is paused.

5.5 Summary

In this chapter I presented a methodology for evaluating *Xest* over the Planetlab distributed test bed, in order to more accurately capture the effects of wide area network conditions on its performance. I describe the implementation of a VFS-level trace toolkit and how I used it to capture workloads of web server activity, virtual machine startup and kernel compilation. Finally, I discussed how *Xest* replays these workloads while respecting inter-dependencies between the operations that they contain. In the following chapter, I go on to employ this methodology in the evaluation of the *Xest* prototype implementation.

Chapter 6

Evaluation

In this chapter I describe the traces that I proposed collecting in Section 5.3 and use them to evaluate the *Xest* prototype implementation.

I discuss the gathering of file system access statistics and measure how effectively the group reassignment algorithm is able to produce high-locality grouping arrangements. I then evaluate *Xest*'s mechanisms for cache advertisement and source selection in the context of a very large volume of storage. I demonstrate the feasibility of the pointer insertion process, and measure the latency with which sources for groups can be located and their metadata retrieved. I also test the effect of various optimisations on the fetch protocol. Finally, after evaluating the latency of virtual disk operations, I demonstrate virtual disk migration.

6.1 Traces

Before presenting evaluation results, I describe the characteristics of the traces used in the remainder of this chapter.

6.1.1 Traces gathered

All traces were recorded in an unprivileged virtual machine running the XenLinux 2.6.12 kernel patched with the VFS trace toolkit and a Fedora Core 2 distribution tailored for Xen.[1] The full system workload of the 'users' portion of the Computer Laboratory's web

[1] The performance results presented in this chapter make no absolute comparisons with the completion times of operations on the test machine. However, for completeness, the traces were obtained on a Dell 2650 dual processor 2.4GHz Xeon server with 2GB RAM, a Broadcom Tigon 3 Gigabit Ethernet NIC, and a single Hitachi DK32EJ 146GB 10k RPM SCSI disk. It ran Xen 3.0 changeset 7713. The traced domain was allocated 512MB RAM and had access to a 4GB raw disk partition formatted with an ext3 file system. During tracing, the machine ran only the traced domain and the management 'domain-0', which was quiescent. The Laboratory file-space was accessed using NFS through a server that re-exported it using a user-space NFS daemon. Care was taken to ensure that its own NFS mounts were maintained for the entire trace period so that inode numbers on the traced machine remained consistent.

Trace	Duration		File system size		Accessed portion	
	Ops $\times 10^3$	Period	Inodes $\times 10^3$	MB	Inodes $\times 10^3$	MB
lab-18-05	339	24 hrs	167	12,660	17	1,493
vmboot1	12	282 s	92	1,390	0.5	61
vmboot2	12	602 s			0.5	61
kernelbuild1	1,232	543 s	108	1,585	5	77
kernelbuild2	1,232	525 s			5	79

Table 6.1: Summary of traces used in the evaluation

Trace	Inode operations			File operations			
	creat	un/link	Other	read	write	open,release	Other
lab-18-05	<1%	<1%	22%	42%	1%	25%	9%
vmboot1	<1%	<1%	4%	38%	<1%	56%	<1%
vmboot2	<1%	<1%	4%	38%	<1%	56%	<1%
kernelbuild1	<1%	<1%	11%	21%	3%	63%	1%
kernelbuild2	<1%	<1%	11%	21%	3%	63%	1%

Table 6.2: The profile of operations in the traces

server was reproduced and recorded for a period of 24 hours. Two traces were taken of the virtual machine booting, and two of the virtual machine compiling the Linux 2.6.12 kernel. Table 6.1 summarises some properties of these traces.

The traces represent distinct forms of file system activity; their profiles of VFS operations are shown in Table 6.2.[2] The lab-18-05 web server trace consists of intermittent bursts of reads due to client requests and a small amount of write traffic due to system bookkeeping and log writing. 16% of its operations have an inter-arrival time greater than 1ms, and 0.5% greater than 100ms. On the other hand, the kernel build traces are a mixed read/write workload that is mostly I/O-bound. Only 0.5% of operations in the kernelbuild1 trace have an inter-arrival time greater than 50µs.

Note that only a portion of each file system is accessed during the traces. The effect of this on regrouping strategies is discussed further in Section 6.2.1.

6.1.2 Precision of trace event replay

Recall from Section 3.2.3 that *Xest* makes a number of changes to Bamboo's event-based core to improve the precision with which it issues timed events. While not critical to most network services, high precision is important for replaying file system traces in order to recreate accurately the fine-grain load effects of bursty periods. To measure the effect of these optimisations, 50 concurrent timers were set with uniform delay at microsecond gran-

[2] Recall that the raw trace files list all file system operations, not just those on inodes and objects within the regenerated file systems that are used to replay the traces. The number of operations quoted is the net number executed during replay; the file system sizes reflect the net size of the initial file system and of those pre-existing inodes that are accessed during replay.

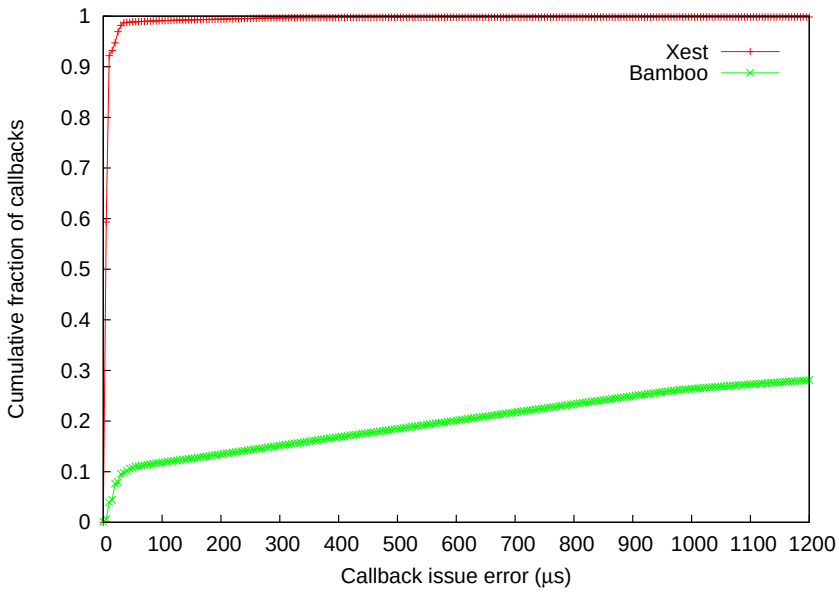


Figure 6.1: Cumulative distribution of callback issue error

ularity between 50ms and 750ms. Each callback's issue error, the difference between the expected and actual times that the callback was made, was measured by cycle counters, and each callback then repeated this process two thousand times.

Figure 6.1 compares the error in issue time for the *Xest* core and a Bamboo core deployed inside the *Xest* code base. The median errors are 4µs and 3,500µs, respectively. Although the Bamboo core only schedules events to the nearest millisecond and relies on coarse-grain OS timer system calls, the dominant effect is that of OS scheduler quantisation. To counter this, *Xest* sets its select() call timeouts 5ms earlier than its next event is due; if it is scheduled before the due time, it pre-spins. Further, if a subsequent timer is due within this interval, it will spin rather than yield. Without these modifications any trace event issued with a non-zero timer is delayed until the next scheduling quantum, so small timeouts have very large relative errors.

6.2 Grouping

Xest's grouping components observe file system accesses and synthesise them into statistics, which inform the process that reassigns objects to different groups. Before going on to evaluate *Xest*'s storage management components, I assess the statistics produced by the observer and test the effectiveness of the clustering algorithm.

Parameter	Studied range	Acceptable range	Chosen value
Ageing scale factor	0.5 – 1.0	0.7 – 0.9	0.9
Reference window size	1 – 50	20 – 50	20
Maximum neighbours	20 – 60	20 – 60	20
Incubating neighbours	1 – 5	3 – 5	5

Table 6.3: Settings for parameters for recording statistics.

6.2.1 Gathering and using access statistics

In this section I evaluate how *Xest* compiles statistics from the observation of file system access patterns, then go on to consider the suitability of this data for the clustering process.

I first measure the effect of a number of observer parameters by considering how the clusterings produced by these settings affect the performance of file system trace replay. To do this, I deployed *Xest* over a pair of nodes and a pristine lab-18-05 file system was initialised. In each step of the experiment, the volume was forked into a new virtual disk and the trace was replayed for a period of 12 hours, using a given set of observer settings. Once complete, the volume was closed and the group assignment process run, at each step using constant settings that I detail below. Then, from the second node in the pair, the resulting snapshot was forked and the trace replayed from its 12^{th} hour to conclusion. The persistent cache at the target node was cleared before each run.

Table 6.3 summarises my findings. Each parameter was studied at four discrete values in the range indicated, in combination with every other setting for the other parameters. After the trace replay had completed, the experiment recorded the proportion of objects retrieved as part of a group that are subsequently used, an indication of grouping quality. The acceptable range column indicates the settings combinations that are at most 10% less than the proportion used with the chosen settings combination.

The chosen settings reflect a good trade-off between the effects of various parameters. A very small reference window size can lead to genuine access relationships being missed, but too large a window size can lead to noise. As the maximum number of neighbours is increased, weaker neighbours are retained, and the size of object access records grows. These settings are used for the remainder of this evaluation.

Beyond parameter setting, an important constraint on this evaluation — or arguably an environmental factor with which *Xest* must deal — concerns the depth and breadth of statistics that the observer is able to accumulate. In terms of breadth, the number of objects accessed represents only a small fraction of the total file system for all of the traces considered. For the kernelbuild1 trace, of 109,317 objects in the volume at its completion, 96,031 reside in groups that have no associated statistics because no object in them has been accessed. Of the remaining objects, 8,390 have no neighbours; object access records

are stored for only 4,896 (4% of the total). For the lab-18-05 trace the pattern is similar: of 163,501 objects, object access records exist for 16,196 (10%).

In terms of depth, most objects are accessed infrequently, so few object access records are able to build up accurate neighbour transition probabilities. When a reference does occur, *Xest*'s use of a large reference window as described in Section 4.2.2 allows it build a neighbour profile rapidly. With the observer settings chosen above, 94% of objects with access records in the kernelbuild1 trace have the maximum 20 non-incubating neighbours; 69% for the lab-18-05 trace. However for the kernelbuild1 trace, only 559 objects are referenced 20 or more times; for the lab-18-05 trace, only 204 objects. A low total reference count for an object affects the accuracy with which neighbour transition frequencies can be calculated and hence provides low-quality information to the clustering process.

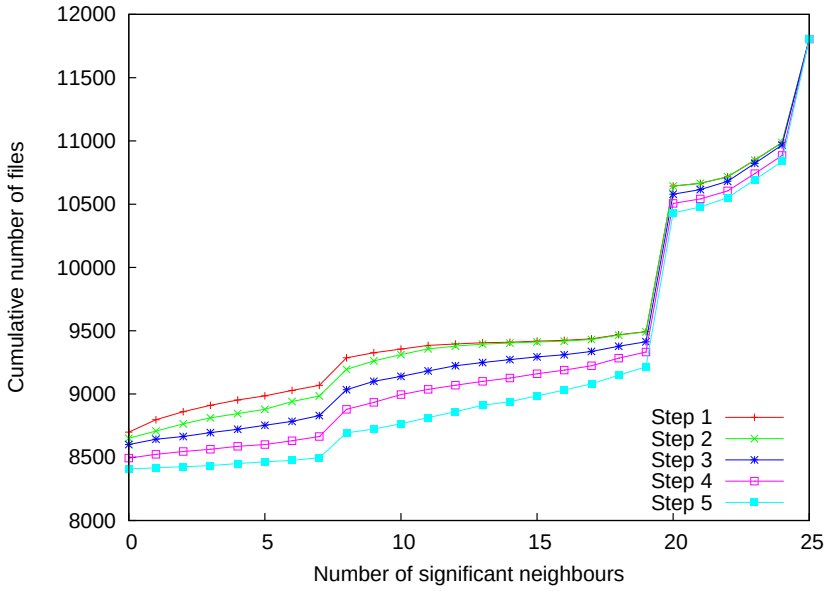
Considering how the clustering process prepares these statistics yields further details. Figure 6.2 shows, for both the kernelbuild1 and lab-18-05 traces, the distribution of the number of significant neighbours associated with each object for each of the merge passes. It includes all objects in non-fixed groups. Figure 6.3 plots the distribution of the number of significant neighbours shared between pairs of objects, the basis on which groups are merged or split.

These figures illustrate that some discrimination is possible between the objects considered at each merge step. However, as discussed above, most objects have no associated statistics, and objects accessed in the trace are typically accessed only a few times. More diverse trace data may yield a richer set of statistics with which to gain further experience tuning the grouping process.

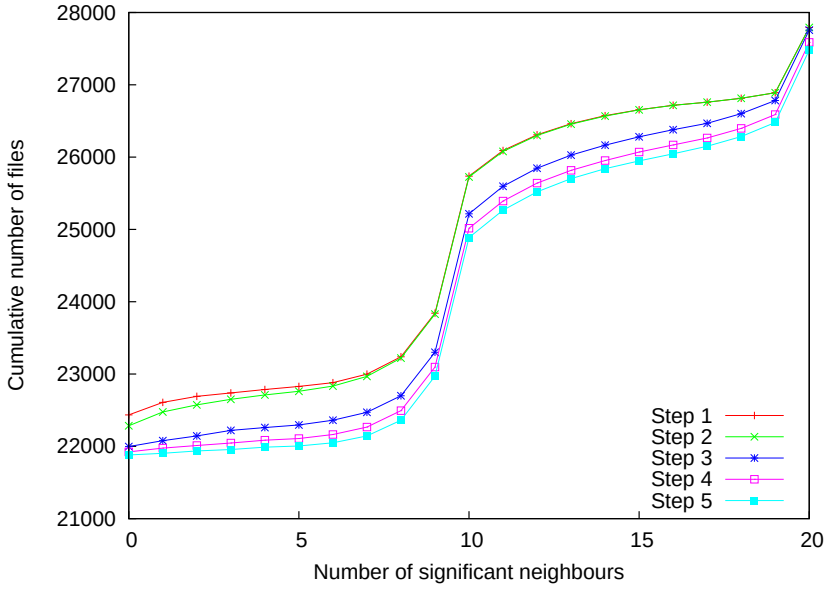
6.2.2 Clustering process

I now consider the effect of training time on clustering performance. Observing additional file system accesses both improves the accuracy of neighbour counts in existing access records and provides statistics for a broader range of objects. However it also relies on the observer to effectively age existing neighbours and incubate new ones.

Xest was deployed over two pairs of nodes in the global Internet, one to study the kernelbuild traces and the other the lab-18-05 trace. On one node of each pair a new virtual disk was created with that node as its sole manager. A copy of the appropriate file system was initialised inside it. The resulting snapshot volume was forked at the same node and a trace (kernelbuild1, or the start of lab-18-05) replayed. After a given period of time, the 'training time' referred to in the text below, replay was halted and the volume closed, allowing the group reallocation process to run. Then, from the second node in the pair, the resulting snapshot was forked and a second trace replayed: either kernelbuild2 in its entirety, or the

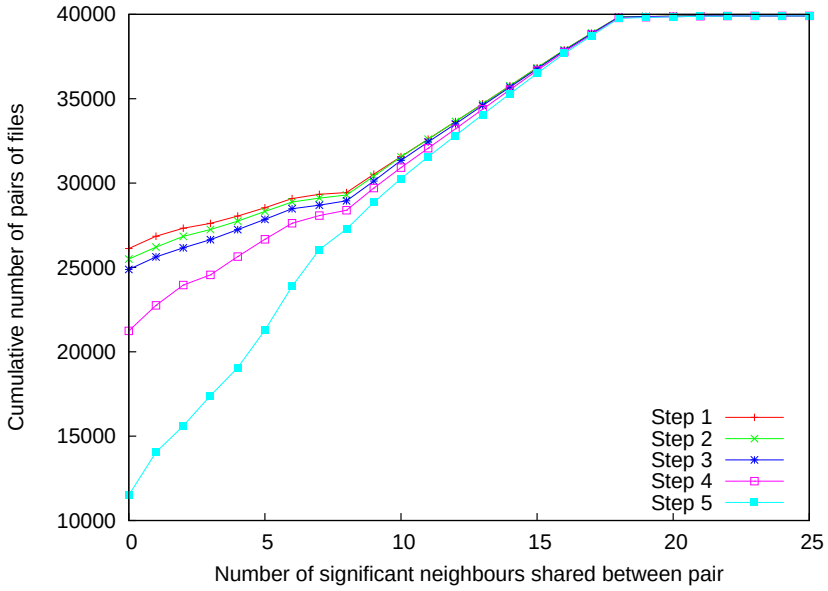


(a) For the trace kernelbuild1.

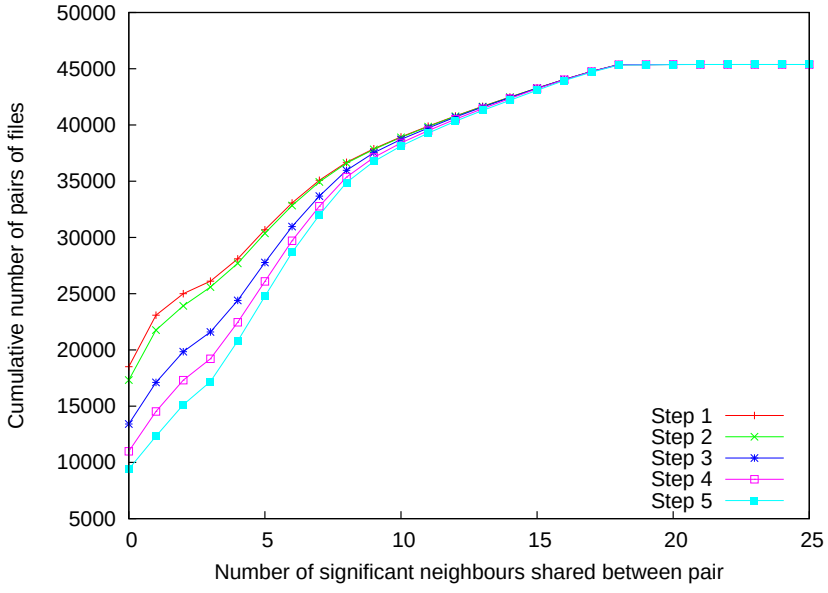


(b) For the trace lab-18-05.

Figure 6.2: Variation in the profile of neighbour relations due to the weakening of significance thresholds in successive passes of the clustering algorithm.



(a) For the trace kernelbuild1.



(b) For the trace lab-18-05.

Figure 6.3: Variation in the sharing of neighbours due to the weakening of significance thresholds in successive passes of the clustering algorithm.

2 hour period starting at the 12th hour of lab-18-05. This process was repeated for a range of training time periods. The target node's persistent cache was cleared between each step.

Afterwards, in order to measure the effect of grouping as a unit of implicit prefetching, each initial file system was forked and the second trace replayed with the group prefetch mechanism disabled. This process was repeated five times for each trace and the lowest total stall time recorded from these runs.

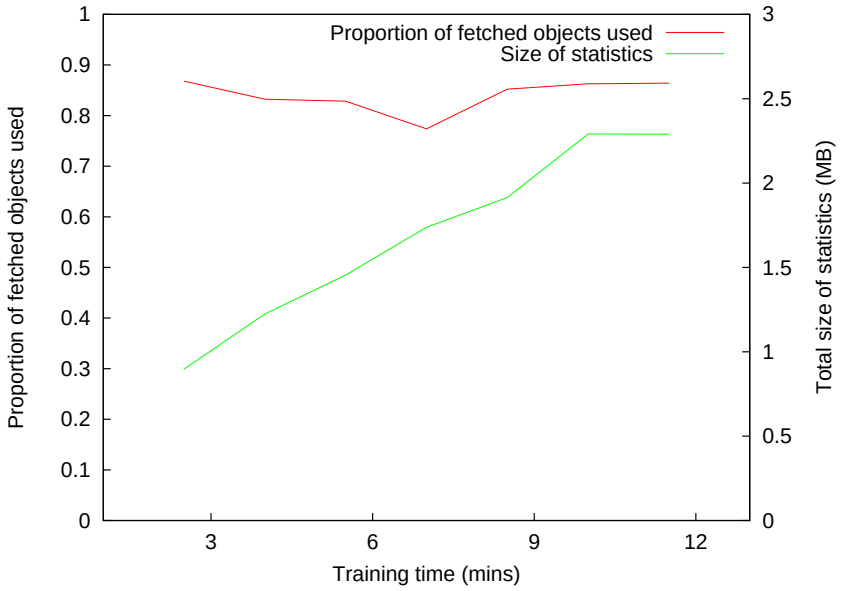
Figure 6.4 plots the proportion of objects that are used from all of those retrieved by the group fetch protocol, against the training time. This metric captures how effectively the groups produced by the clustering algorithm form units of prefetching. It corresponds to Tsangaris and Naughton's 'expansion factor' in the context of variable-size clustering units, which they describe as the most important metric in cold-cache conditions when only a fraction of the total addressable units need be accessed [Tsangaris92]. Figure 6.5 plots the total time spent stalled during trace replay, the user-perceived delay, as a fraction of the least time spent stalled by a run without prefetching, also against training time.

Note that the studied ranges of training time differ between these two experiments to account for the difference in the two trace's request rates. In each figure the graphs labelled (a) are obtained from the kernelbuild1 trace, and the graphs labelled (b) from the lab-18-05 trace. All graphs plot on secondary Y axes the total storage consumption of the statistics accumulated for that volume at the point of group reallocation.

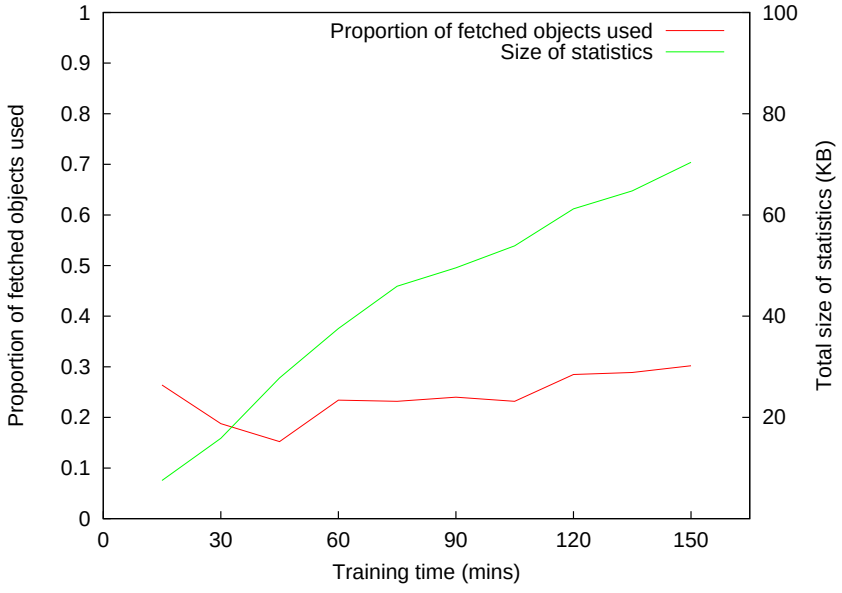
For the kernelbuild1 trace, *Xest* is able to gather 2.3MB of statistics before the end of the training trace and form groups that lead to a low proportion of additional objects being fetched. This leads to a net reduction in stall time for the majority of training durations, although it is not clear that training beyond the first four minutes of the trace is of benefit.

The lab-18-05 trace presents a considerably more challenging clustering task on account of only 70KB of access records being accrued over two and a half hours. This reduces the precision of constructed groups, leading to at most 30% of retrieved objects being used. However, the high inter-request period and *Xest*'s fetch prioritisation scheme mean that the large volume of additional fetches does not hinder demand transfers, at least for runs that receive training for an hour or more.

I next evaluate the rôle that overlaps and the directory distance heuristics play in the outcome of the group allocation process. A further experiment was set up using the kernelbuild traces in a similar fashion as those described above. For these four runs, each training phase was run for the trace's entirety and different settings were applied to the group assignment process at each step. In the first both the use of overlaps and the directory distance heuristics were disabled. In the second, only overlaps were enabled; in the third, only the directory distance heuristics. In the fourth, both were enabled.

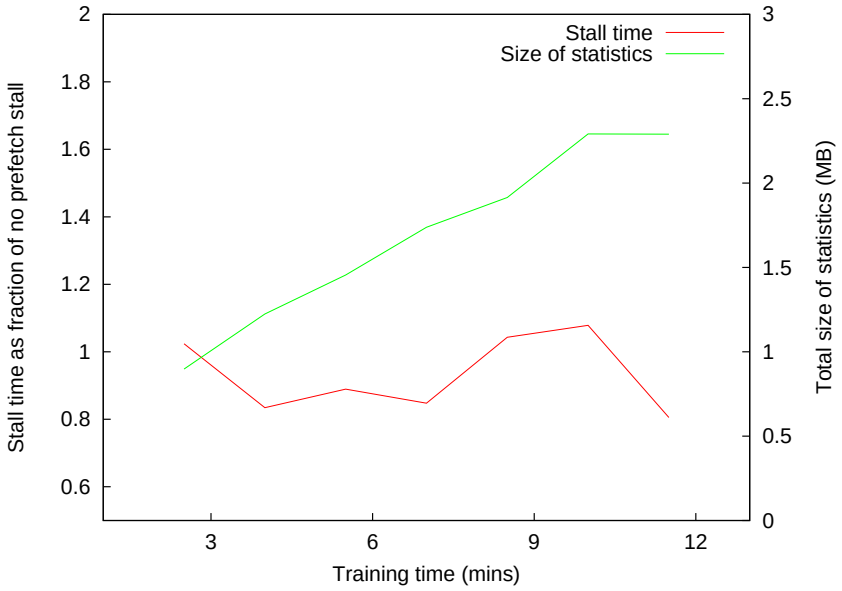


(a) For the trace kernelbuild1.

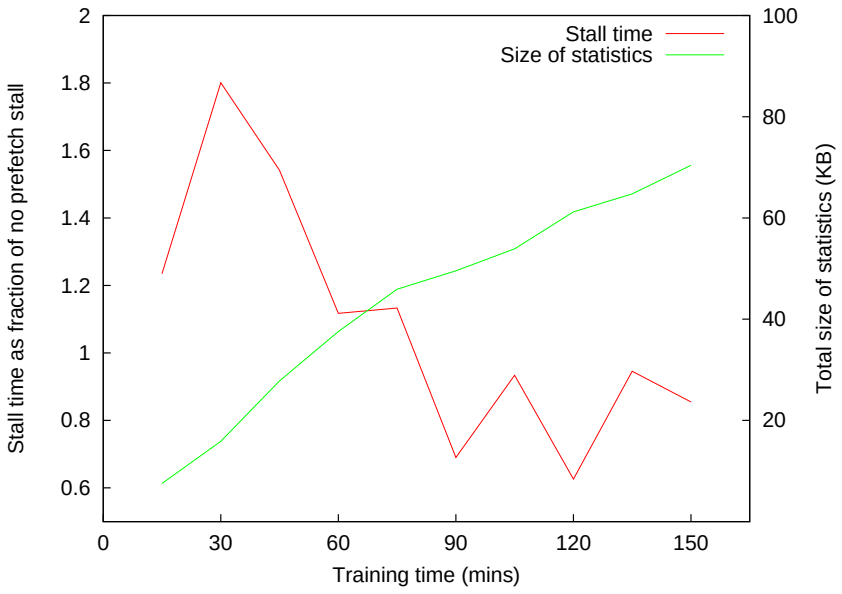


(b) For the trace lab-18-05.

Figure 6.4: The effect of training time on the precision of grouping, measured by the proportion of fetched objects used during trace replay.



(a) For the trace kernelbuild1.



(b) For the trace lab-18-05.

Figure 6.5: The effect of training time on total time spent stalled during trace replay, as a fraction of the stall time recorded without prefetching.

Clustering options		Objects (KB)		Overlaps (KB)		Stall	Proportion used
Overlaps	Dir dist	Median	90th	Median	90th		
		112	204	None		0.91	0.90
	✓	126	207			0.90	0.89
✓		112	204	0	278	0.87	0.89
✓	✓	126	207			0.87	0.87

Table 6.4: The effect of the directory distance heuristics and overlapping on the size and quality of grouping arrangements.

Table 6.4 sets out these results. The directory distance heuristics are intended to reduce the number of addressable units by merging sets of small groups that exhibit locality in the directory tree. This increases the median group size by 13% to 126KB, and the median number of objects in each group from 8 to 11. It has little impact on the total stall time, suggesting that the groups that it merges exhibit some locality.

Overlapping is intended to allow the clustering algorithm to accommodate asymmetric transition probabilities between objects, so 'links' to common shared files can be placed in multiple groups. Using the overlap settings described above causes a significant number of overlaps to be added to a minority of groups, with a 90th percentile of 10 objects, corresponding to 278KB. This results in a slight reduction in stall time, but also causes a slight reduction in the proportion of fetched objects that are subsequently used.

This difficulty stems from the fact that introducing an overlap requires only a single primary object in the group to have a sufficiently high number of shared neighbours, in the spirit of Jarvis-Patrick's algorithm [Jarvis73]. However poor quality access statistics can reduce the probability that access to that group will result in access to that object, and hence to the overlap, although it will be fetched on any access to the group. This problem is exacerbated by the median overlapped file being larger than the median primary object. Further study of this effect is warranted to maximise the benefit of overlaps while mitigating their potential costs.

6.2.3 Virtual machine startup

An important test of *Xest*'s grouping is its effect on file system performance during the start up of a virtual machine. Although these traces are short-running, virtual machines will be booted frequently: whenever a new service is forked or an existing service restarted.

In this section I measure in cold cache conditions the effectiveness of *Xest* groups as a unit of implicit prefetching. I also consider the case in which no prefetching is performed, and evaluate a conventional prefetcher that issues prefetches for objects in the precise order in which it expects requests to be made for them according to its training data.

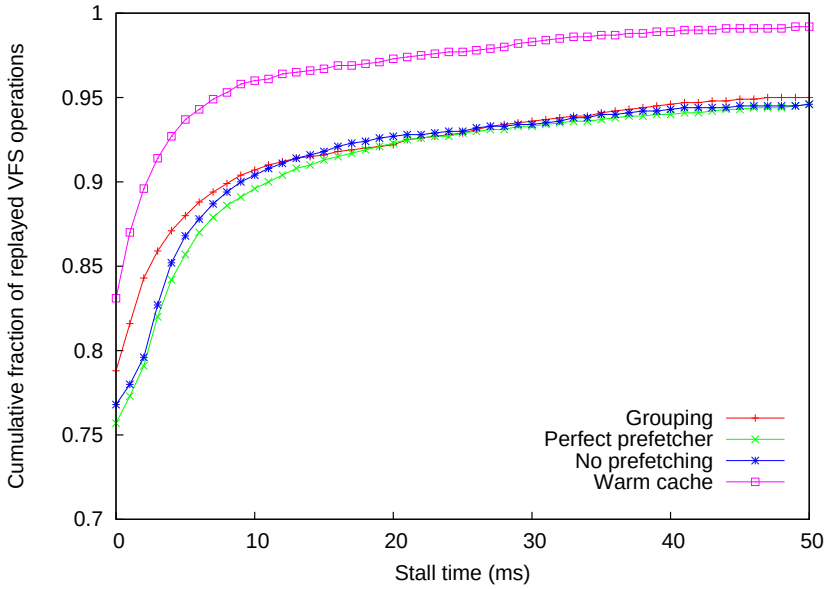


Figure 6.6: Cumulative distribution of the stall time of replayed VFS operations during virtual machine startup. Stall times are measured with respect to the time taken for the same operation as recorded in the trace, not to warm cache conditions. Note the range of the axes.

Method	Stall factor	Retrieved		Used	
		Objects	Size (MB)	Objects	Size (MB)
No prefetching	12.7	486	18	480	18
Conventional prefetcher	10.2	488	18	480	18
Xest grouping	10.5	570	58	480	18

Table 6.5: The performance of *Xest* during virtual machine startup. Values in the column labelled 'stall factor' record the total time spent blocked on I/O by all processes for the given set up divided by the time spent stalled for warm cache conditions.

Xest was deployed over a pair of Planetlab machines, between which the minimum observed RTT is 190ms. One of the pair was denoted the 'server' node. At this node, a new volume was initialised containing the vmboot template file system, and its virtual disk was assigned the same node as its single manager. Then, still on the server node, this volume was opened and the vmboot1 trace replayed, thereby generating training data for the group assignment process. The volume is then closed and regrouped using the clustering settings described in the previous section.

The resulting snapshot formed the basis of each run of the experiment. From the 'client' node, the second trace vmboot2 was replayed, and the results gathered. After each run the cache was cleared. This process was repeated five times for each category of result.

The conventional prefetcher operates as follows. Additional data is gathered during the training process at the server node. Whenever an object is opened for the first time, its

device and inode number are added to the tail of a prefetch queue. During replay, the prefetcher ensures that fetches are in progress for the first $c = 10$ objects not yet retrieved at the head of the prefetch queue, until the queue is empty.

Note that the conventional prefetcher is not an 'oracle' in that it does not have perfect information about the trace to be replayed. The two traces are recorded during different start-ups of the same virtual machine. In addition, a prefetch is enqueued for each open operation in the training trace, and not those that in fact cause *Xest* to stall. For example, it may unnecessarily prefetch objects that are opened and immediately closed, or immediately truncated.

Table 6.5 shows for each category the stall time relative to a warm cache run and the volume of data retrieved and used. Note that the figures include statistics objects. In addition, Figure 6.6 plots the cumulative distribution of the absolute stall time of replayed VFS operations for each category. The stall times plotted in the graph are relative to the time taken for operations to complete in the trace, not to warm cache conditions in *Xest*.

On these traces *Xest* performs well. Despite its grouping arrangement causing the retrieval of a large amount of data that is not accessed during the second trace, it nevertheless significantly outperforms the case in which prefetching is disabled, resulting in a stall time that is 83% of the non-prefetching run. Observe also that although the ratio of used bytes was poor, at 31%, the ratio of used objects was good, at 84%. Performance may be improved further by considering group and object size more carefully in the clustering algorithm.

6.3 Addressing

In this section I consider *Xest*'s cache advertisement and source selection mechanisms, which centre around the insertion of soft-state pointers into a DHT shared between all nodes, and the retrieval and synthesis of this data to locate sources from which to fetch groups. I first measure the mechanisms' costs, in terms of bandwidth and DHT operations, then proceed to evaluate their effectiveness in choosing low-latency sources.

For the experiments in this section, *Xest* was deployed over 97 Planetlab nodes and 42 logically distinct lab-18-05 template file systems each consisting of 14,178 groups were initialised in separate virtual disks, each on a different randomly selected node. The number of nodes over which DHT keys were replicated was fixed at 8 and retrieval of the virtual disks' contents by their managers was disabled, for a total of 4.8 million keys stored. The results vary under the absolute number of groups, rather than storage size, but the file systems inserted constitute a global storage burden of 7 million files in 532 GB.

6.3.1 Batching pointer insertion operations

Before considering the communication costs of refreshing soft-state addressing data, I evaluate the contribution of *Xest*'s scheme for batching DHT routing operations, described in Section 3.5.1, in which insertions for destinations close by in the key space are forwarded in a single packet.

Figure 6.7 (a) shows the distribution of the number of DHT operations sent in each batch, aggregated across all nodes. The sample size was 47,120 operations. Although the mechanism is a general-purpose one, in fact all batched items were group or volume pointer insertions, of which 17 can fit in a single unfragmented datagram.

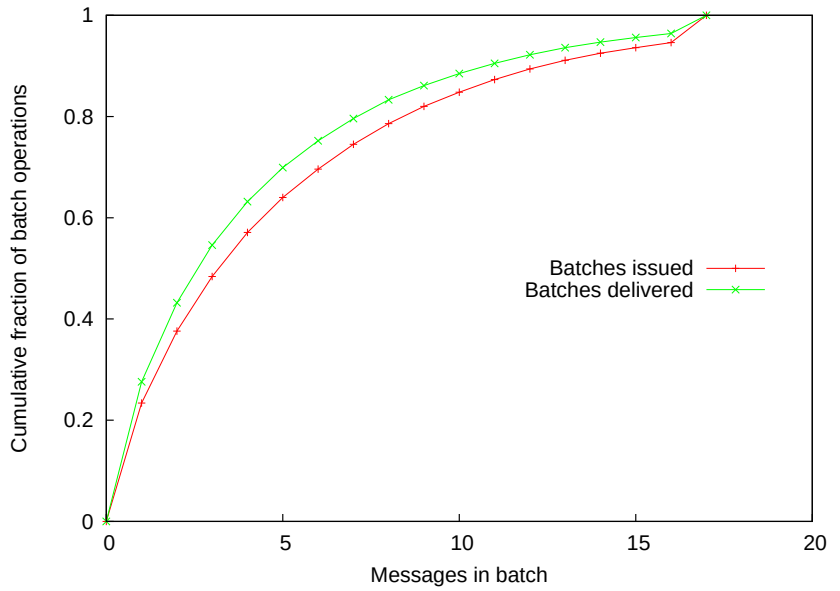
The two series compare the sizes of batches as they are routed from their source and at the node whose identifier is closest in the key space to all of the destinations in the batch. Observe that since the statistics cover all nodes, each fragment of a batch split in transit would appear as separate, smaller batches. The mean number of messages in transmitted and received batches was 5.3 and 4.6 respectively, indicating batches are rarely fragmented en route, and yielding a reduction in the number of packets sent by a commensurate factor.

Figure 6.7 (b) shows the distribution of the delays from inserting the first operation in a bucket to issuing a batch message for that bucket, taken over all nodes. The DHT sends a batch regardless of its size after a uniform timeout in the range of 4 to 6s. Note that this timeout only triggers the process that sends the batch, yet the delay shown in Figure 6.7 (b) measures the time until the batch is dispatched. A number of very heavily loaded nodes were observed to suffer significant delays before servicing timed requests, causing the measured delay to exceed 6s in some cases.

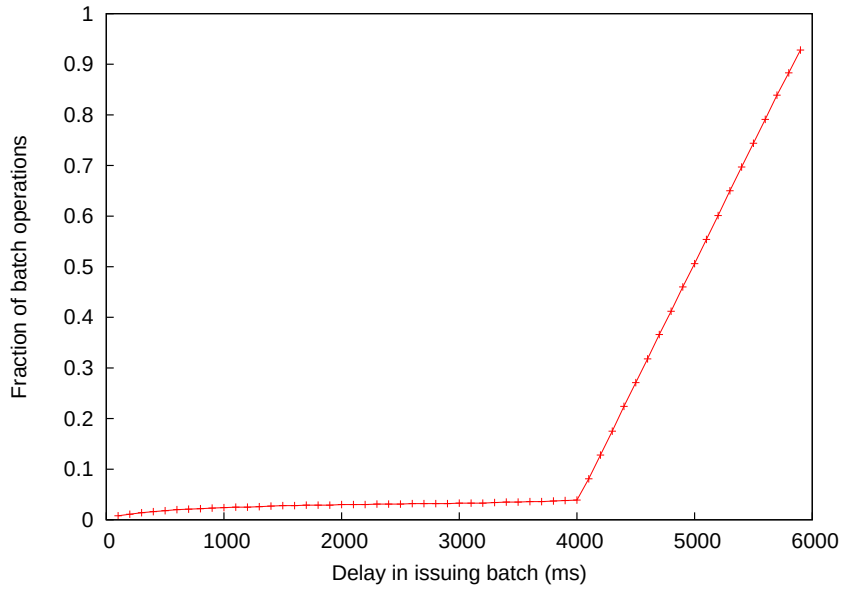
Obtaining large batch sizes with relatively small delays depends on spatial locality in the keys of destination requests. The cache arm's stride length is set to 18 groups for these experiments. Most strides however are likely to fall across batch bucket boundaries, and only renew pointers for some of the groups close to expiry and not added to the set of eviction candidates. Randomised timeouts also lead to insertions whose keys have poor spatial locality, and so tend to be sent as part of smaller batch sizes.

Recall that each insertion is forwarded to its replica set and a reply made only after a threshold of nodes have successfully stored the pointer. *Xest* also batches these messages according to their destination. The replica set centred on each node overlaps with its neighbours and messages concerning insertions made by different source nodes are aggregated together. These batches are sent within 1500ms containing a mean of 5.9 messages.

The cumulative distribution of completion times of batched pointer insertion operations is shown in Figure 6.8. Despite batching at three steps on the critical path and replica set nodes tending to be diverse in network location, the request completion time has a 90th



(a) The cumulative distribution of the number of pointer insertion requests in a batch, across all nodes, on forwarding from the batch's source node and on delivery to its destination.



(b) The distribution of the delay with which a batch is issued after its first constituent request is made. The peak at 6 seconds is explained in the text.

Figure 6.7: Batching DHT routing operations

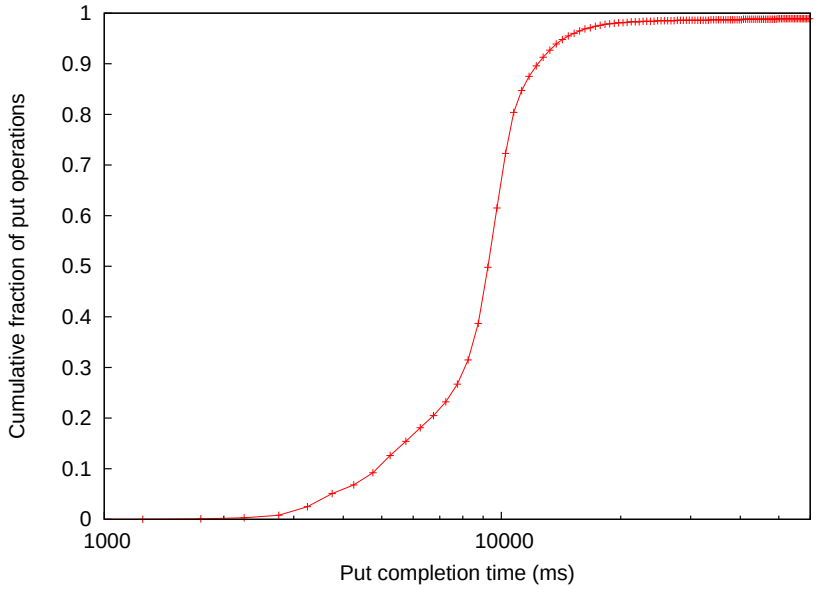


Figure 6.8: The cumulative distribution of completion times of pointers insertions. Note the logarithmic scale. Batching occurs before the insertion is routed through the DHT, before it is forwarded to each member of the destination node's leaf set, and before a reply is returned to the client.

percentile of 12,875ms. The long tail is due to large timeouts set for latency-insensitive operations. Table 6.6 summarises the latency profile of this and other addressing operations.

6.3.2 Addressing state maintenance

The cost of maintaining addressing state centres around the reinsertion period. This is constrained by the pointer expiry period, which trades off more frequent reinsertion against storage of more stale state and a higher probability of misdirecting latency-sensitive fetch operations.

However, the difference in periods between reinsertion and expiry also determines the minimum rate of iteration of the cache arm: if a group is not eligible for re-advertisement when the arm passes over it, it must be considered again before its existing pointer expires. Reducing the work done by the cache arm reduces the reinsertion period and hence increases the cost to the network and global DHT service. *Xest* also sets the next reinsertion time for each group by deducting a randomised time from the maximum period to mitigate the effect of synchronised insertions due to file system creation and snapshots.

However, rather than verifying these trade offs experimentally, I aim primarily to demonstrate that a real deployment of *Xest* can maintain large volumes of addressable storage. In this experiment the network is monitored for 15 hours subsequent to the initialisation

of the final virtual disk completing. An expiry period of 8 hours is used, with reinsertion occurring uniformly randomly in the period 6 to 7 hours. This anticipates 0.3 groups re-advertised, or 2.1 re-insertions across the replica set, per second per node.

Figure 6.9 (a) and (b) plot the profile of the mean outgoing bandwidth, as bytes and messages sent, respectively, of those nodes storing and advertising virtual disks. The nodes are otherwise quiescent; values include retransmissions but exclude the size of IP and UDP headers. The 'pointer maintenance' series incorporates all DHT routing and replica set operations from and through the node; the spikes due to synchronised reinsertion are clearly visible. The 'DHT maintenance' series compares the periodic costs of maintaining each node's leaf set and routing table. The 'pointer lookups' shows periodic virtual disk manager and Bamboo tasks that involve routing.

Figure 6.9 (c) details the mean number of pointers refreshed (93% of operations during the period) and newly inserted (7%). A further series estimates the mean accessible pointers at each node using record counts from Berkeley DB. A pointer may expire, and hence later be newly inserted, due to failure of the inserter or an inconsistency of the DHT. However an expiry does not necessarily indicate that the pointer is unavailable elsewhere in the DHT, as the very low lookup failure rates reported in the next section suggest. Recall that insertions do not wait to store a key on each member of the leaf set; the anti-entropy process may not yet have spread the refreshed key throughout the whole leaf set before it expires at some members.

These results indicate that *Xest*'s approach is practical for distributed addressing of large volumes of storage. However bandwidth usage is linear to the volume of storage advertised, and Section 7.2 proposes a number of enhancements to constrain this cost further at very large scales.

6.3.3 Pointer and metadata retrieval

The next experiment measures the latency of retrieving addressing information from the DHT. Three client nodes were selected at random for each of the 42 virtual disks, so that nodes were clients to a mean 1.3 virtual disks. For each virtual disk, a client first retrieved the metadata of its sole volume, then, at uniform random intervals of 3 to 9s, selected one of its groups at random and requested its metadata. For this experiment, any pointers retrieved from the DHT were flushed from the addressing cache after the request.

Figure 6.10 shows in the two series marked '1 source' the cumulative distributions of these operations taken over all nodes for a period of approximately 90 minutes, resulting in 122,831 pointer retrievals and 121,363 metadata retrievals. Each request causes a lookup in the DHT and, rarely, subsequent lookups if either no pointers are found within an end-to-

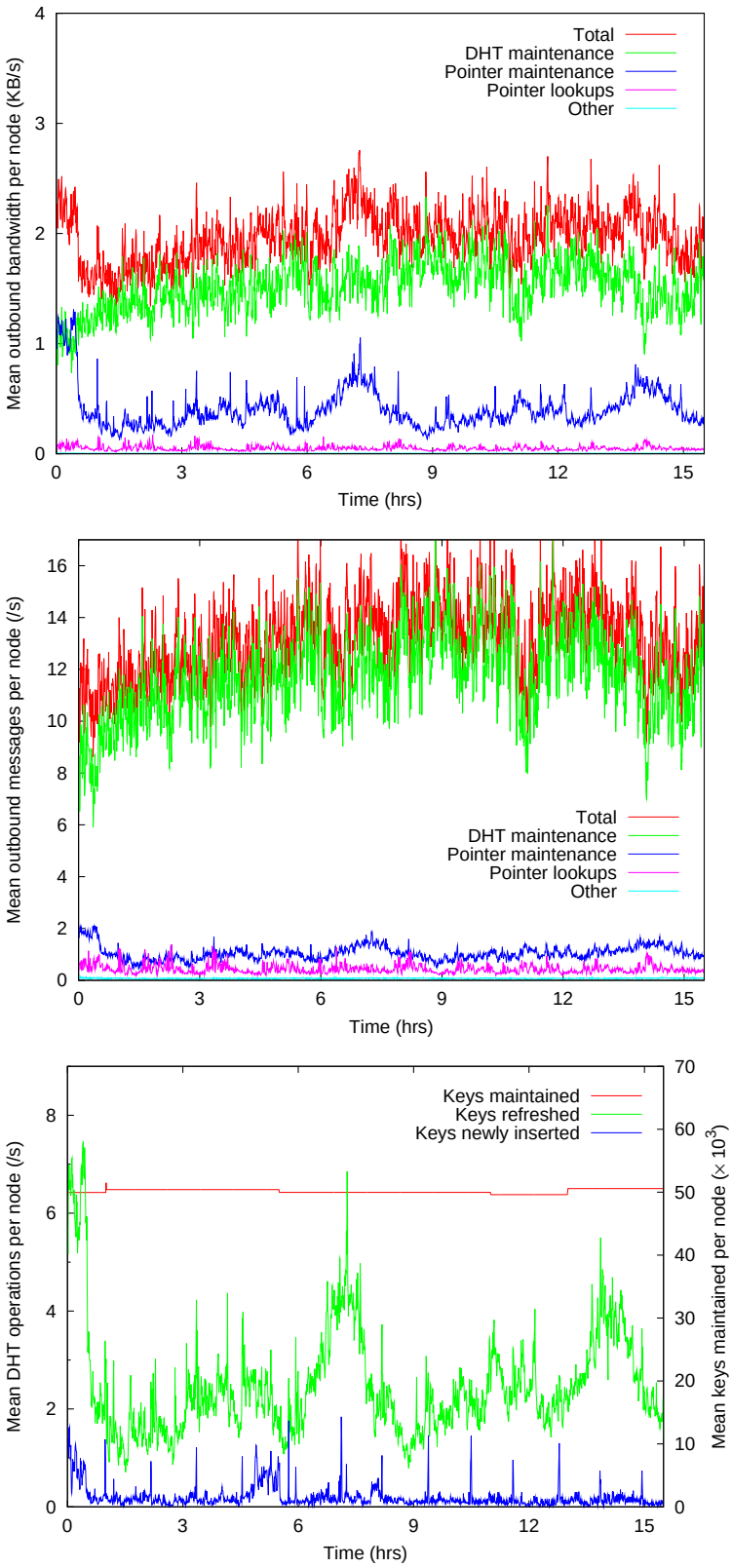


Figure 6.9: The cost of maintaining the DHT and its contents for each node in terms of: (a) outbound bandwidth; (b) messages sent; (c) DHT storage operations.

end timeout or if on requesting the metadata these sources all reply to indicate they do not hold it. The latency of each DHT lookup is recorded separately, though each incorporates a number of retries. In contrast, the latency of metadata retrieval incorporates all of the lookups necessary to locate a valid source for it.

Table 6.6 details the latency profile of these operations. In the experiment, DHT lookups have median latency 160ms and 90th percentile 580ms. This allows very rapid resolution of location-independent group names, especially considering their coarse granularity. Consider, in comparison, that DNS has a median latency of approximately 100ms and 90th percentile of approximately 500ms [Jung02]. Even so, there are two factors that are significant in understanding the context of these results and that suggest further improvement is possible.

First, the set of nodes have a median round-trip time (RTT) of 235ms, considerably higher than the Planetlab median latencies of 76ms, 82ms and 137ms previously reported [Dabek04b, Ramasubramanian04, Rhea05c][3]. In addition, 2.1% of all paths remain unreachable throughout. Note that most lookups attain a lower latency than the RTT by following paths through local peers to the nearest member of the destination's leaf set.

Second, access to the Berkeley DB pointer storage tables incurred significant delays in some cases. In order to reproduce a situation in which pointer storage exceeds memory capacity, each node's database cache size was set to 5MB, while the tables occupied around 45MB plus log files on disk, necessitating most accesses going to disk. In this configuration, median latency was found to be 2,240ms using Bamboo's original interface: gets tended to be queued behind large numbers of outstanding insertions and each operation retrieved only a single pointer.

By reforming the interface, using a separate request queue for latency-sensitive operations and reducing load by absorbing multiple requests and retries for the same key, as well as caching positive results, *Xest* reduces the median latency to 20ms. However, the 90th percentile remains high at 635ms, due mainly to periodic synchronous checkpoint operations. To allow nodes with heavy disk load but that are otherwise responsive to sidestep this long tail, *Xest* does not require gets to wait for a reply from the database before forwarding lookups to its leaf set, as Bamboo does, but instead forwards them after a delay of 100ms.

In the above arrangement, the process of looking up the pointers for a group then retrieving its metadata, which has a mean size of 1,011 bytes in these volumes, has a median latency of 425ms, with a 90th percentile of 1,450ms. However the latency of the fetch itself depends solely on the availability, load and network position of the single host.

---

[3] The RTT is calculated from reachable paths with network-layer latency estimates from both explicit pings and normal messaging activity during the experiment. This difference may in part be accounted to 48% of nodes having a /24 IP address prefix unique among those used in this experiment, in contrast to 6% of production Planetlab nodes.

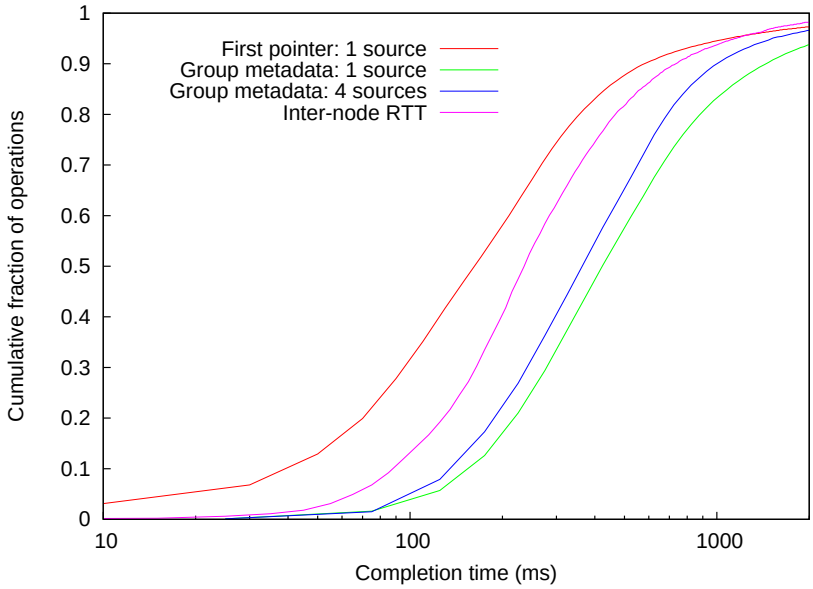


Figure 6.10: The cumulative distribution of completion times of addressing operations. Note the logarithmic scale.

Operation	Latency percentiles (ms)			
	Mean	50th	90th	99th
RTT of reachable paths	345	235	635	2,595
Publish pointer (batched)	10,600	9,250	12,350	66,750
Retrieve first pointer, 1 source	290	160	580	3,790
Retrieve group metadata, 1 source	670	425	1,445	5,575
Retrieve group metadata, 4 sources	525	365	1,000	4,175

Table 6.6: The latency profiles of addressing operations. The precision of quoted values is 5ms for all operations except 'publish pointer', for which it is 50ms. Note that the latency of retrieving group metadata incorporates the time taken to locate a valid source for it using the DHT.

In order to evaluate how effectively nodes can locate low-latency, live sources, retrieval of each virtual disks' contents by its managers was enabled and the process allowed to complete, providing three additional sources for all groups.

The previous experiment was then repeated. The latency of metadata retrieval recorded for 45,875 samples is also shown in Figure 6.10. The median latency is reduced to 365 ms, and the 90th percentile to 1,000 ms. Although the process requires the client to communicate with two other nodes in sequence, and may require additional hops within the DHT, the mean operation completes in 1.71 times the mean inter-node RTT, demonstrating the effectiveness of *Xest*'s selection of local peers.

Observe that the previous arrangement also depended on a single source to advertise its pointers. In fact, all of the 0.3% of DHT lookups that failed (after retrying for approxi-

mately 3 minutes, contributing to the 99thpercentile) were determined to be due to pointers expiring before their sources, suffering periods of disconnection, could renew them. Adding the additional sources reduces lookup failures, in which no pointers are returned, to less than 0.1%.

6.3.4 Fetch protocol performance

The *Xest* fetch protocol, described in Section 3.5.3, exposes an interface for the retrieval of whole groups at once, but also accepts detailed priority information so that it may selectively reduce its effective granularity to sub-file regions. In this section I measure the contribution of the techniques intended to achieve this, and in so doing demonstrate the protocol's good performance.

Xest was deployed over fifty Planetlab nodes, a subset of those used in the experiments of Section 6.3. A new virtual disk was created with five managers; a kernelbuild template file system was initialised at one of these manager nodes, then subsequently replicated to the others by the usual background process.

A node located at a site distinct from all of these managers was selected to act as a client. For context, the RTT to each manager was in the range of 150ms to 210ms. At each step, all of the nodes' configurations are altered to enable or disable an aspect of the fetch protocol, as described below. Before each run, all of the client node's stored state and persistent cache is cleared, then the deployment resumed and the DHT maintenance process allowed to stabilise. Then the client forks the initial volume and replays the kernelbuild1 workload. This causes the same 13,092 object fetches to be issued.

For these experiments and all others described in this chapter, a maximum chunk size of 256KB is used. This value aims to minimise the impact of latency on the protocol while mitigating the risk in pathological cases of engaging in a lengthy transfer that turns out to have been unnecessary. In this workload, only a single fetch was for a larger object: its 3MB content was fetched as thirteen separate chunks.

Figure 6.11 (a) and (b) plot for each of these configurations the cumulative distribution of objects' completion time, the time that elapses between an object being first requested and its final chunk being retrieved, and not cancelled in that interval. The distribution (a) includes only objects whose fetches, at the point of their completion, had priority demand_open. This includes any objects containing chunks further prioritised as demand_rw. The distribution (b) includes all fetches, including prefetches and retrieval of statistics objects.

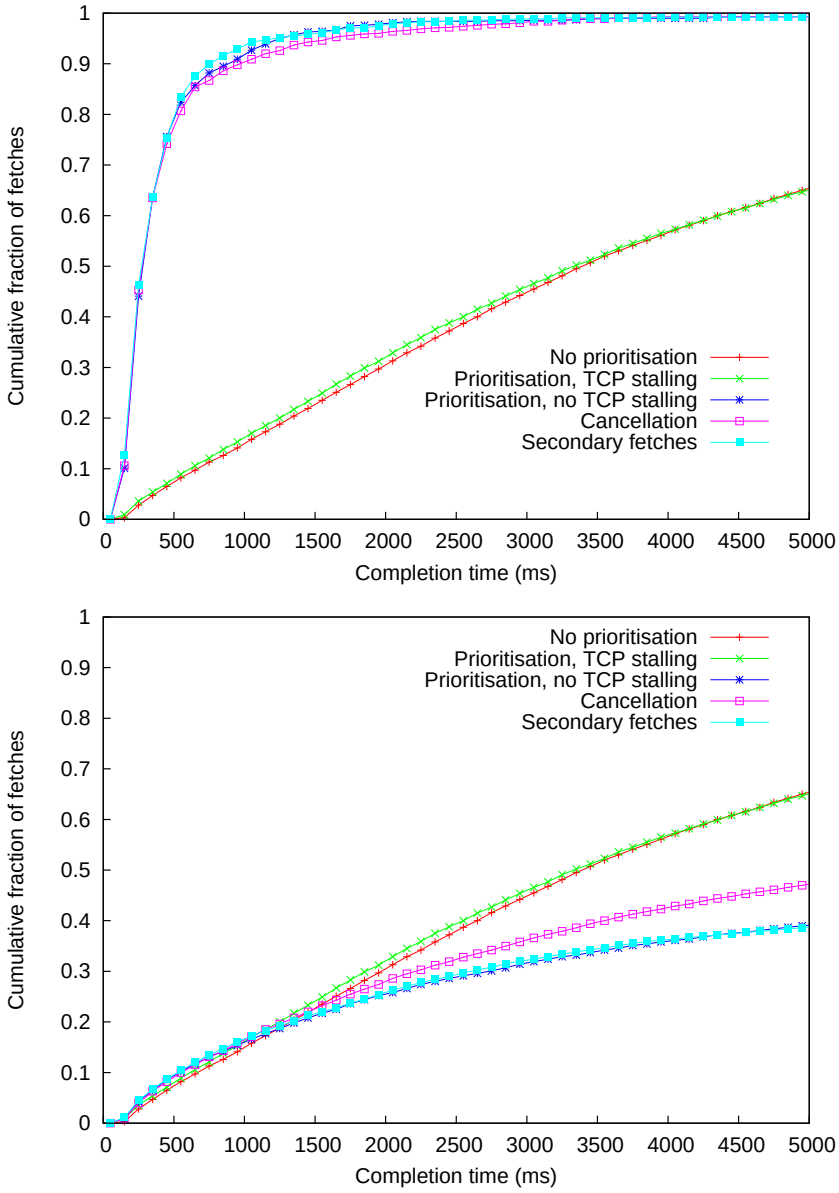


Figure 6.11: The effect of fetch protocol optimisations on the completion time of: (a) objects whose priority at the point of completion was demand; and (b) all object retrieval operations.

Each series evaluates a different configuration. The first, labelled 'no prioritisation', ignores all priority information. Fetches for objects that are the target of a cache miss are delayed by previously issued prefetches.

In the series 'prioritisation, no TCP stalling' per-object priority information is enabled and forwarded to the uploading peers, as described in Section 3.5.4. This allows higher priority fetches to be completed more rapidly, at the expense of prefetches completing more slowly.

The median completion time is reduced to 385ms, from 3,390ms. The 90th percentile is 2,450ms.

In addition, this series marks individual chunks whose data is required to satisfy outstanding read operations as demand_rw. This has two effects. First, it prioritises one chunk over other chunks in the same object. However, the workload under test means that this has a negligible effect on completion time. Second, it prioritises chunks required to satisfy blocking read operations from those whose priorities were elevated because of an open operation, which does not itself cause a process to block. This reduces total stall time by a further 11%, although the profile of completion time for all demand fetches cannot capture this effect.

Perhaps surprisingly, stalling outstanding prefetches to grant recent high priority fetches greater instantaneous bandwidth worsened the completion time of demand fetches. This may be due to poor interaction with *Xest*'s reuse of a finite set of long-lived TCP connections between each peer, a technique widely used to improve the accuracy of advertised congestion window sizes. *Xest* stalled existing prefetches by not servicing their connections, causing their buffers to become full and window sizes to fall to zero, hence slowing the transfer of subsequent chunks using that connection while the window grew. The remainder of the series described do not use this technique.

In the series labelled 'Cancellation', prefetches that are not either retrieved or re-requested within 90 seconds are cancelled, a technique designed to prevent the build up of a backlog of low priority transfers no longer relevant to the processes that generated them. In this series, fetches for 360 objects totalling 716KB were cancelled. A small improvement in the total prefetches completing may be observed.

The final series enables secondary requests, as described in Section 3.5.3. Unfortunately this experiment is unable to demonstrate any benefit from this feature, since it is designed to reduce stall when sources indicate that they are overloaded or fail to reply. Since node selection automatically balances requests between the five sources available and no source failures were observed, only 57 requests triggered secondary fetches. Further evaluation under conditions of high load is necessary to assess this technique further.

6.4 Virtual disks

This section evaluates aspects of *Xest* that involve the manipulation of virtual disks. First, I measure the latency introduced by the agreement protocol that runs between managers, and consider the latency perceived by clients in updating virtual disk metadata through managers. I then go on to demonstrate *Xest*'s support for virtual disk migration.

6.4.1 Latency of virtual disk operations

The metadata associated with a virtual disk is mutable for its entire lifetime and it may be modified from any node. However the managers of each virtual disk use the agreement protocol described in Section 3.6.2 in order to ensure that updates are applied to the metadata consistently and to provide high availability. In this section I set out to demonstrate that although this arrangement means that operations on virtual disks are susceptible to the effects of wide area latency, *Xest*'s design is nevertheless practicable.

I measure the latency of updates under the deployment arrangements described in the evaluation of the fetch protocol. A single virtual disk is initialised. The RTT between its five managers has a median of 65ms, with a 90th percentile of 215ms. During the experiment, each of the fifty deployed nodes (including the nodes that are also managers) issues operations on the virtual disk. Between each request, a node delays for a random period in the interval of sixty to ninety seconds. The experiment completes when each node has sent twenty-five requests.

Figure 6.12 plots the cumulative distributions of the completion time of several parts of the agreement process: I describe these series below. For comparison it also plots the distribution of RTTs between all pairs of nodes in the experiment.

Each request is sent to a manager chosen in the usual way according to liveness and latency estimates. On receiving the request, the manager initiates a corresponding Paxos proposal, either proposing the operation itself, or forwarding it to the node that it believes is leader. The series labelled 'time to reach consensus' records the time taken between the manager receiving the operation from the client and the *same* manager learning that the Paxos protocol has agreed a final sequence number for the operation.

As each manager receives confirmation that the operation has been agreed, it updates its copy of the metadata, then sends a reply to the client that issued the operation. In this experiment, requests are 'nops' that do not materially change the metadata's contents but otherwise behave as normal operations. This means that the metadata is written after applying the update then reinserted into the DHT as usual. The time that the client measures between issuing the request and receiving a reply from the first manager is recorded for the series labelled 'first response to client'.

Figure 6.12 shows that the agreement process for most operations completes in two communication steps, *i.e.* one round trip. The median latency is 170ms, with a 90th percentile of 610ms. The transmission delay to and from the client means that the median operation latency from a client's perspective is 285ms, having a 90th percentile of 1,605ms. As a proportion of corresponding points on the all-pairs RTT distribution, this median is 95%

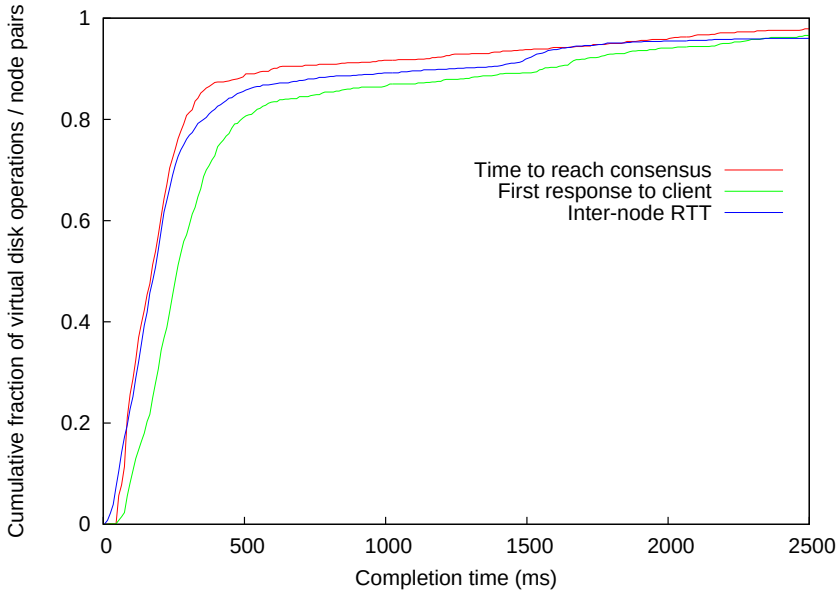


Figure 6.12: The cumulative distribution of completion time of virtual disk operations. The series show both the time to achieve consensus between managers and the client-perceived latency.

of the RTT and the 90th percentile 130%. These results are achieved by sending requests to the nearest manager and accepting the first reply.

Not only does *Xest* satisfy virtual disk operations with low latency, but also the need for such operations is infrequent. A client needs to block only when it initially requests a lease on a virtual disk, then later to transfer the lease if it subsequently migrates. In the example considered in the previous section, the first node must make two such operations, and each subsequent node one. All other virtual disk operations can be performed asynchronously.

6.4.2 Migration

My final experiment demonstrates *Xest*'s ability to migrate a client request stream between machines. *Xest* was deployed over the same set of fifty nodes as used in the previous section. At one node, a kernelbuild file system was instantiated then trained on the kernelbuild1 trace. After completing group reallocation, the five managers of the associated virtual disk were allowed to completely retrieve the contents of the resulting snapshots and advertise themselves as sources.

Once the pointer reinsertion process had reached a steady state, the virtual disk was forked and replay of the trace kernelbuild2 commenced. Both the initial node and every subsequent node were chosen randomly from those nodes that were not managers or had previously replayed any portion of the trace. After running for six minutes at a node, *Xest* is notified of

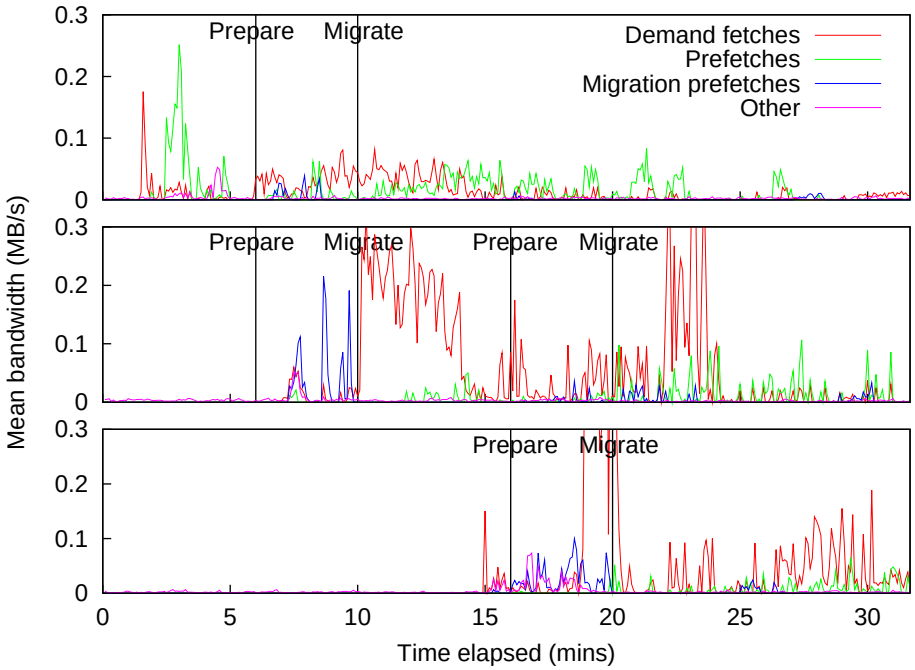


Figure 6.13: The bandwidth profile at 5 second sample intervals across three nodes as a virtual disk is migrated across them. Each graph's Y axis ranges from 0 to 0.3MB/s. The vertical red lines indicate, as labelled, the times at which migration was notified to the source node, and at which migration was then enacted.

the destination of the next migration, and preparations begin. After a further four minutes, the migration is enacted, and the process repeats.

Figure 6.13 shows the bandwidth profile of three nodes during the period in which the virtual disk is migrated between them in turn. At each step, the source node learns that an active volume is to be migrated, and sends over the next four minutes three working set notifications to its peer. The target node ceases to be quiescent and issues three spikes of migration prefetches, corresponding to these notifications. After each migration has taken place and the source's volume has been closed, it is subject to fetch requests, prefetch and demand, both from the peer to which the virtual disk was migrated and to the virtual disk's managers.

Figure 6.14 illustrates the steps taken to enact a final migration request, and is annotated with times recorded for the migration between the first two nodes shown in Figure 6.13. One node is in Cambridge, UK and the other Passau, Germany, a 40ms round-trip. The process takes a total of 20.6 seconds, of which 5.6s is due to the suspension and resumption of the trace. The total outage time, the period for which the volume is inaccessible, is only 18.2s.

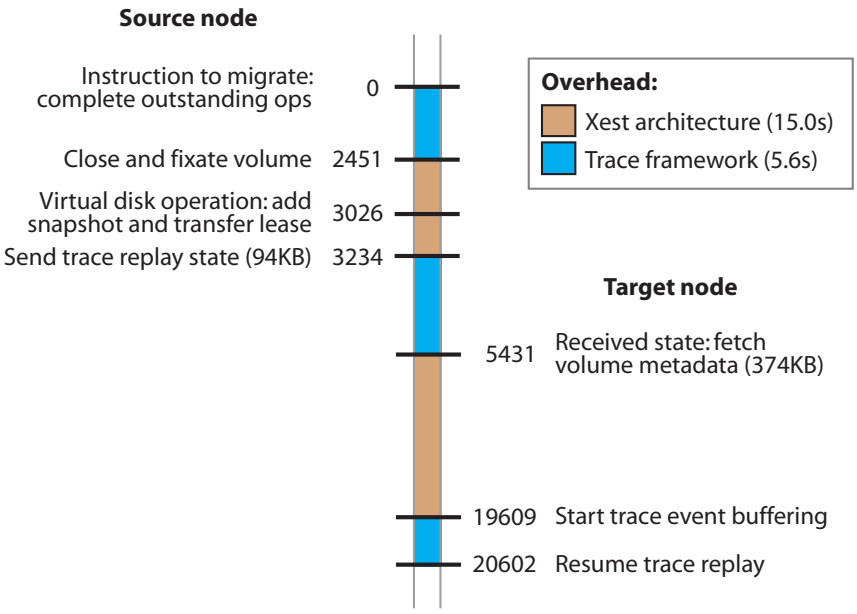


Figure 6.14: The steps taken in migrating access to a volume. Times given are in milliseconds and refer to the recorded times from the first transfer shown in Figure 6.13.

The transfer of the volume metadata from the source node was the single step consuming the longest time, 14.2s, in fact the majority of the outage. Since this metadata had only just been fixated and was only available from that single node, striping it from multiple sources was not possible. This volume's metadata's size of 374KB stands well against its total file system of 1.6GB. However it serves to emphasise the discussion on volume metadata scalability in Section 3.3.3. Adjustments in the metadata's format, overlapping its transfer with previous steps such as virtual disk operations, and improvements in the quality of grouping may reduce this delay in future.

6.5 Summary

In this chapter I evaluated the *Xest* prototype implementation. By measuring the group reallocation process I showed how, despite the limitations of the trace data that I produced, *Xest* can produce grouping arrangements that reduce the time spent stalled in cold cache conditions in comparison to fetching single objects. In addition, although only prefetching implicitly, it performs well in comparison to a conventional prefetcher.

I demonstrated how advertising whole groups makes my approach practicable for large volumes of storage, and how sources and metadata for groups can be rapidly located and retrieved. I showed that the latency imposed by the virtual disk manager's agreement pro-

tocol is practicable and demonstrated that a virtual disk containing a large file system can be migrated in around 20 seconds.

In the next chapter, I conclude this dissertation by summarising my contributions and outlining future work.

Chapter 7

Conclusion

In this dissertation I have described the design and implementation of *Xest*, a distributed file system developed to meet the storage requirements of the XenoServer platform. To mitigate the effects of high latency and large volumes of storage on its performance, *Xest*'s architecture is organised around groups, sets of related files and directories built by gathering data on client access patterns. My evaluation demonstrates that my design is practical given slow networks and cold caches: conditions anticipated for migrating virtual machines. Further, it shows that *Xest*'s use of grouping improves the performance of its fetch protocol and the scalability of its addressing protocol.

In Section 7.1 I summarise this dissertation's contributions then explain how they satisfy my thesis statement. In Section 7.2 I conclude with a number of suggestions for further work.

7.1 Summary

In Chapter 1 I described how performance constraints imposed by wide-area latency motivate the XenoServer platform, in which strategically-located servers multiplex network services inside whole-system virtual machines. I set out the challenge of building *Xest*, a storage substrate for services deployed on the platform, and proposed the technique of grouping as a means of obtaining good performance in this environment.

In Chapter 2 I discussed the requirements for *Xest* in more detail and surveyed related work. I presented key aspects of distributed file systems' architectures and examined previous efforts to mitigate scale and slow networks using access patterns.

Chapter 3 described the architecture of *Xest*'s storage management components. This presentation is a significant contribution of the dissertation. First, it details how the abstraction of the *Xest* volume allows a conventional hierarchical file system structure supporting snapshot and fork operations to be derived from an underlying set of objects arranged into

groups on the basis of their access relationship. Second, it shows how a minimal amount of consistent read-write metadata, virtual disks, can be used to construct a global namespace and facilitate the appearance of migration. Third, it describes how storage management tasks are performed on the basis of whole groups but can selectively reduce their granularity under evidence of poor clustering.

In Chapter 4 I discussed the group reassignment process, the dissertation's second main contribution. I presented novel structures for storing and accumulating statistics inside *Xest*'s conventional file system layout, and set out a new incremental and partial algorithm for reassigning objects to groups.

In Chapter 5, I addressed the difficulty of evaluating a wide-area file system whose clients are a set of migrating virtual machines. I describe how I built a trace capture toolkit and replay framework to allow *Xest* to be deployed over a real distributed test bed by simulating only the external workload.

Finally, in Chapter 6, I evaluated the *Xest* prototype. I measured in detail both its grouping and storage management components, and demonstrated the good performance of my implementation.

These contributions substantiate the thesis statement that I set out in Section 1.3.

My first claim is that *Xest*'s design satisfies the XenoServer platform's novel storage infrastructure requirements. I summarised those requirements in Section 2.1.2. Support for nomadic, isolated virtual disks and disk image management is provided by the interfaces I describe in Sections 3.3.1 and 3.6.2. Section 3.3.4 demonstrates how *Xest* uses copy-on-write to exploit commonality between virtual disks. The other requirements are environmental: slow networks, large storage volume, and dominance of cold cache conditions. I have demonstrated the suitability of my design in these respects by carefully constructing a wide-area experimental testbed and evaluating *Xest* at a large scale and in cold cache conditions.

My second claim is that the application of grouping to *Xest* can reduce the duration for which requests are stalled in cold cache conditions. This is evidenced by the results in Section 6.2.2 and Section 6.2.3 that demonstrate a net reduction in total time spent stalled relative to the no prefetching case: 13% for the intensive recompilation of the Linux kernel and 17% during a virtual machine's boot process.

It is arguable that further refinement to the grouping algorithm is necessary to make the process more robust over a broader range of file system activity. However, my results indicate that even sparse statistics and the resulting poor clusterings do not result in significantly poorer performance, on account of *Xest* reverting to more fine-grain behaviour after detecting this.

7.2 Future work

I highlight here a number of directions for future research.

Further tools for managing virtual disks. Administrators may benefit from additional tools to manage the potential proliferation of file system images generated during the lifetime of a set of distributed services. One useful primitive might merge two or more snapshots into a new volume forked from each, in a new virtual disk. The existing volumes remain available, and conflicts in the resulting file system hierarchy may be detected and resolved offline, in contrast to file systems that use optimistic consistency semantics.

Storage reclamation. *Xest* facilitates significant sharing of file system content between a volume and its copy-on-write ancestors. However further incidental commonality will develop between the contents of sibling and unrelated volumes. An automated background process that detects and coalesces identical fixated objects could eliminate storage duplication and benefit their discovery and caching by providing additional sources.

Further, snapshots inside a virtual disk may be pruned in order to free space consumed by the groups local to them while retaining historical state, as the Elephant file system does [Santry99].

Distinction between read and write workloads. In building access records *Xest* does not distinguish between reads and writes. A fruitful area of future research lies in incorporating this information into the existing grouping structures. Organising groups by objects that are written together may reduce the number of new groups introduced with each snapshot as a result of copy-on-write. Further, prediction of complete object overwrites avoid wasted prefetch bandwidth.

Online group reassignment. The group reassignment process currently runs at the point that a volume has been closed and is about to be fixated. The main disadvantage of this approach is that a service relying on a volume must be taken offline in order for its grouping arrangement to reflect the most recent access patterns. Few aspects of the methodology set out in Chapter 4 would need to change to perform reassignment while a volume is writable; the main obstacles are implementation issues concerning locking and journalling.

Improvements to cache advertisement. A number of techniques may improve group location latency or reduce the cost of advertising groups, including caching hot pointers along the lookup path and using variable TTLs that adapt according to storage status, group hit rate and cache size.

A distributed deployment over Xen. The evaluation methodology presented in Chapter 5 uses simulated client workloads in order to measure *Xest* running over a distributed test

bed. A future deployment over a large set of machines running Xen would provide longer, real-life workloads of distributed services with which to better evaluate *Xest*'s performance.

Towards a production deployment. A production deployment of *Xest* may be run according to the same economic model as the XenoServers platform itself. An organisation rents virtual machines at each XenoServer in which to deploy *Xest* and pays for the resources that the nodes use. A centralised management service will monitor nodes' liveness, issue credentials and adjust resource provisioning. In turn, *Xest* nodes will account for their client's usage, passing the data back to the management service for billing purposes.

A major research challenge is how to multiplex access to *Xest* to the client virtual machines while retaining their isolation. Isolation necessarily compromises the performance benefits that can be yielded by sharing: incorporating any shared service into the platform involves extending resource isolation into the service itself, or running the service separately in each client's virtual machine.

One solution may be to split *Xest* into two halves. A back-end would run in a separate virtual machine and a front-end in each client virtual machine. Most file system activity would involve only the client's front-end. The back-end would interact with remote nodes and fairly partition the resources contented for by local clients, such as fetch bandwidth and cache space.

Bibliography

[Adya02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. *FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment*. In Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), pages 1–14, 2002. (p 24)

[Akamai03] Akamai. *Akamai EdgeComputing*, 2003. White paper. http://www.edgecomputing.org/wp/. (pp 14, 20)

[Amer01] Ahmed Amer and Darrell D. E. Long. *Noah: Low-Cost File Access Prediction Through Pairs*. In Proceedings of the 20th IEEE International Performance, Computing and Communications Conference. IEEE, 2001. (p 31)

[Amer02a] Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. *Group-Based Management of Distributed File Caches*. In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), 2002. (p 33)

[Amer02b] Ahmed Amer, Darrell D. E. Long, Jehan-François Pâris, and Randal C. Burns. *File access prediction with adjustable accuracy*. In Proceedings of the International Performance Conference on Computers and Communication (IPCCC), 2002.

[Anderson04] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. *Buttress: A toolkit for flexible and high fidelity I/O benchmarking*. In Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST), pages 45–58, March 2004. (pp 42, 117)

[Anderson95] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. *Serverless Network File Systems*. In Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 109–126, 1995. (p 24)

[Apache07] Apache. *HTTP server project*, 2007. http://httpd.apache.org/. (p 112)

[Aranya04] A. Aranya, C. P. Wright, and E. Zadok. *Tracefs: A File System to Trace Them All*. In Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST), pages 129–143, 2004. (p 109)

[Awadallah02] Amr Awadallah and Mendel Rosenblum. *The vMatrix: A Network of Virtual Machine Monitors For Dynamic Content Distribution*. In 7th International Workshop on Web Content Caching and Distribution, 2002. (p 20)

[Barham03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. *Xen and the art of virtualization*. In Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 164–177, 2003. (p 20)

[Bavier04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. *Operating System Support for Planetary-Scale Network Services*. In Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI), 2004. (pp 20, 105)

[Blaze92] Matt Blaze. *NFS Tracing by Passive Network Monitoring*. In Proceedings of the Winter USENIX Technical Conference, pages 333–343, 1992. (p 109)

[Boichat03a] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. *Deconstructing Paxos*. SIGACT News, 34(1):47–67, 2003. (p 74)

[Boichat03b] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. *Reconstructing Paxos*. SIGACT News, 34(2):42–57, 2003.

[Bolosky00] 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, 2000. (p 99)

[Bonnie96] Bonnie, 1996. http://www.textuality.com/bonnie/. (p 108)

[Broder03] Andrei Broder and Michael Mitzenmacher. *Network Applications of Bloom Filters: A Survey*. Internet Mathematics, 1(4):485–509, 2003. (p 25)

[Castro97] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. *HAC: Hybrid Adaptive Caching for Distributed Storage Systems*. In Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 102–115, 1997. (pp 35, 59)

[Chun03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. *PlanetLab: an overlay testbed for broad-coverage services*. SIGCOMM Computer Communication Review, 33(3):3–12, 2003. (pp 20, 38, 105)

[Chutani92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. *The Episode File System*. In Proceedings of the Winter USENIX Technical Conference, pages 43–60, 1992. (pp 27, 38)

[Clark05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. *Live Migration of Virtual Machines*. In Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI), 2005. (pp 20, 28, 70)

[Cohen03] Bram Cohen. *Incentives Build Robustness in BitTorrent*. In Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems, 2003. (p 26)

[Dabek01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. *Wide-area cooperative storage with CFS*. In Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 202–215, 2001. (p 24)

[Dabek04a] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. *Vivaldi: A Decentralized Network Coordinate System*. In Proceedings of the ACM SIGCOMM Conference, 2004. (p 41)

[Dabek04b] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. *Designing a DHT for low latency and high throughput*. In Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI), 2004. (p 139)

[Dahlin94] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. *Cooperative Caching: Using Remote Client Memory to Improve File System Performance*. In Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), pages 267–280, 1994. (p 24)

[Day93] Mark Day. *Object Groups May Be Better Than Pages*. In Proceedings of the Workshop on Workstation Operating Systems, pages 119–122, 1993. (p 32)

[Day95] Mark Day. *Client Cache management in a Distributed Object Database*. Technical Report MIT-LCS-TR-652, Massachusetts Institute of Technology, 1995. Ph.D. dissertation. (p 32)

[Dean04] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), pages 137–150, 2004. (p 106)

[Denning80] Peter J. Denning. *Working Sets Past and Present*. IEEE Transactions on Software Engineering, SE-6(1):64–84, January 1980. (p 70)

[Douceur99] John R. Douceur and William J. Bolosky. *A large-scale study of file-system contents*. In Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 59–70, 1999. (pp 26, 111)

[Eaton99] Patrick Eaton, Dennis Geels, and Greg Mori. *Clump: Improving file system performance through adaptive optimizations*, 1999. Class project report. UC Berkeley. (p 34)

[Ellard03a] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. *Passive NFS Tracing of Email and Research Workloads*. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST), pages 203–216, 2003. (p 109)

[Ellard03b] Daniel Ellard, Jonathan Ledlie, and Margo Seltzer. *The Utility of File Names*. Technical Report TR-14-03, Harvard University, 2003. (p 31)

[Ellard03c] Daniel Ellard, Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Margo Seltzer. *Attribute-based Prediction of File Properties*. Technical Report TR-05-03, Harvard University, 2003.

[Foster97] Ian Foster and Carl Kesselman. *Globus: A Metacomputing Infrastructure Toolkit*. International Journal of Supercomputer Applications, 11(2):115–128, 1997. (p 20)

[Geels01] Dennis Geels. *Space-Optimized Markov Chain Model for File Prefetching*, 2001. Unpublished. (p 31)

[Gray00] Jim Gray and Prashant Shenoy. *Rules of Thumb in Data Engineering*. In Proceedings of the 16th International Conference on Data Engineering, pages 3–12, 2000. (p 21)

[Griffioen94] James Griffioen and Randy Appleton. *Reducing file system latency using a predictive approach*. In Proceedings of the Summer USENIX Technical Conference, pages 197–207, 1994. (pp 30, 31)

[Griffioen95] James Griffioen and Randy Appleton. *Performance Measurements of Automatic Prefetching*. In Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, 1995. (p 16)

[Hand03] Steven Hand, Tim Harris, Evangelos Kotsovinos, and Ian Pratt. *Controlling the XenoServer Open Platform*. In Proceedings of the 6th IEEE Conference on Open Architectures and Network Programming, 2003. (pp 14, 19)

[Hibler03] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. *Fast, Scalable Disk Imaging with Frisbee*. In Proceedings of the USENIX Annual Technical Conference, pages 283–296, 2003. (p 26)

[Hitz94] Dave Hitz, James Lau, and Michael Malcolm. *File System Design for an NFS File Server Appliance*. In Proceedings of the Winter USENIX Technical Conference, pages 235–246, 1994. (pp 27, 28, 38)

[Howard88] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. *Scale and Performance in a Distributed File System*. Transactions on Computer Systems, 6(1):51–82, 1988. (p 23)

[Jarvis73] R.A. Jarvis and E.A. Patrick. *Clustering Using a Similarity Measure Based on Shared Near Neighbors*. IEEE Transactions on Computers, C-22(11):1025–1034, 1973. (pp 82, 131)

[Joukov05] Nikolai Joukov, Timothy Wong, and Erez Zadok. *Accurate and Efficient Replaying of File System Traces*. In Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST), pages 337–350, 2005. (pp 111, 117)

[Jung02] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. *DNS performance and the effectiveness of caching*. IEEE/ACM Transactions on Networking, 10(5):589–603, 2002. (p 139)

[Kaashoek96] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. *Server operating systems*. In Proceedings of the 7th ACM SIGOPS European workshop, pages 141–148, 1996. (p 33)

[Katcher97] John Katcher. *PostMark: A new file system benchmark*. Technical Report TR-3022, Network Appliance Inc., 1997. (p 108)

[Kernighan70] B.W. Kernighan and S. Lin. *An efficient heuristic procedure for partitioning graphs*. The Bell System Technical Journal, 49(2):291–307, 1970. (p 34)

[Kistler91] James J. Kistler and M. Satyanarayanan. *Disconnected Operation in the Coda File System*. In Proceedings of the 13th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), volume 25, pages 213–225, 1991. (p 32)

[Kotsovinos04] Evangelos Kotsovinos, Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand, and Tim Harris. *Global-scale service deployment in the XenoServer platform*. In Proceedings of the First Workshop on Real, Large Distributed Systems, 2004. (pp 27, 38)

[Kotsovinos05] Evangelos Kotsovinos. *Global public computing*. Technical Report UCAM-CL-TR-615, University of Cambridge Computer Laboratory, 2005. Ph.D. dissertation. (pp 14, 19)

[Kozuch02a] Michael Kozuch and M. Satyanarayanan. *Internet Suspend/Resume*. In Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, pages 40–46, 2002. (p 29)

[Kozuch02b] Michael Kozuch, M. Satyanarayanan, Thomas Bressoud, and Yan Ke. *Efficient State Transfer for Internet Suspend/Resume*. Technical Report IRP-TR-02-03, Intel Research Pittsburgh, 2002. (p 29)

[Kroeger01] Thomas M. Kroeger and Darrell D. E. Long. *Design and Implementation of a Predictive File Prefetching Algorithm*. In Proceedings of the USENIX Annual Technical Conference, pages 105–118, 2001. (pp 31, 89, 108)

[Kroeger96] Thomas M. Kroeger and Darrell D. E. Long. *Predicting File-System Actions From Prior Events*. In Proceedings of the USENIX Annual Technical Conference, pages 319–328. USENIX, 1996. (p 31)

[Kubiatowicz00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. *OceanStore: an architecture for global-scale persistent storage*. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 190–201. ACM Press, 2000. (p 25)

[Kuenning97a] Geoffrey H. Kuenning. *SEER: Predictive file hoarding for disconnected mobile operation*. Technical Report UCLA-CSD-970015, University of California, Los Angeles, Computer Science Department, 1997. Ph.D. dissertation. (pp 32, 82, 99)

[Kuenning97b] Geoffrey H. Kuenning and Gerald J. Popek. *Automated Hoarding for Mobile Computers*. In Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 1997. (pp 16, 31, 32)

[Kumar93] P. Kumar and M. Satyanarayanan. *Supporting Application-Specific Resolution in an Optimistically Replicated File System*. In Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems, pages 66–70, 1993. (p 26)

[Lamport98] Leslie Lamport. *The part-time parliament*. ACM Transactions on Computer Systems, 16(2):133–169, 1998. (pp 41, 74)

[Lei97] Hui Lei and Dan Duchamp. *An Analytical Approach to File Prefetching*. In Proceedings of the USENIX Annual Technical Conference, 1997. (p 31)

[Mazières01] David Mazières. *A toolkit for user-level file systems*. In Proceedings of the USENIX Annual Technical Conference, pages 261–274, 2001. (p 42)

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. *A fast file system for UNIX*. ACM Transactions on Computer Systems, 2(3):181–197, 1984. (pp 31, 34)

[Moreton02] Tim Moreton, Ian Pratt, and Timothy Harris. *Storage, Mutability and Naming in Pasta*. In Proceedings of the International Workshop on Peer-to-Peer Computing at Networking 2002, Pisa, Italy., May 2002. (pp 24, 28)

[Morris86] James Morris, Mahadev Satyanarayanan, Michael Conner, John Howard, David Rosenthal, and F. Donelson Smith. *Andrew: a distributed personal computing environment*. Communications of the ACM, 29(3):184–201, 1986. (p 23)

[Muir04] Steve Muir. *The Seven Deadly Sins of Distributed Systems*. In Proceedings of the 1st Workshop on Real, Large Distributed Systems (WORLDS), 2004. (p 106)

[Mummert96] Lily Mummert and M. Satyanarayanan. *Long Term Distributed File Reference Tracing: Implementation and Experience*. Software — Practice and Experience, 26(6):705–736, 1996. (p 117)

[Muthitacharoen01] Athicha Muthitacharoen, Benjie Chen, and David Mazires. *A Low-bandwidth Network File System*. In Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 174–187, 2001. (p 28)

[Nagle84] John Nagle. *Congestion control in IP/TCP internetworks*. RFC 896, 1984. (p 62)

[Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. *Caching in the Sprite Network File System*. ACM Transactions on Computer Systems, 6(1):134–154, 1988. (p 23)

[Olson99] Michael Olson, Keith Bostic, and Margo Seltzer. *Berkeley DB*. In Proceedings of the 1999 Summer USENIX Technical Conference, 1999. Available at http://www.oracle.com/berkeley-db/. (p 42)

[Oppenheimer05] David Oppenheimer, Brent Chun, David Patterson, Alex Snoeren, and Amin Vahdat. *Service Placement in Shared Wide-Area Platforms*. In Poster session of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 2005. (p 64)

[Ousterhout90] John K. Ousterhout. *Why Aren't Operating Systems Getting Faster As Fast As Hardware?* In Proceedings of the Summer USENIX Technical Conference, pages 247–256, 1990. (p 15)

[Park04] KyoungSoo Park and Vivek S. Pai. *Deploying Large File Transfer on an HTTP Content Distribution Network*. In Proceedings of the 1st Workshop on Real, Large Distributed Systems (WORLDS), 2004. (p 26)

[Patterson93] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. *A Status Report on Research in Transparent Informed Prefetching*. ACM Operating Systems Review, 27(2):21–34, 1993. (pp 30, 31)

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. *Informed Prefetching and Caching*. In Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 1995. (p 30)

[Pawlowski94] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. *NFS Version 3: Design and Implementation*. In Proceedings of the 1994 Summer USENIX Technical Conference, pages 137–152, 1994. (pp 22, 23, 26)

[Petersen97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. *Flexible Update Propagation for Weakly Consistent Replication*. In Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 1997. (p 26)

[Pike92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. *The use of name spaces in Plan 9*. In Proceedings of the 5th ACM SIGOPS European Workshop, pages 72–76, 1992. (p 27)

[Policroniades04] Calicrates Policroniades and Ian Pratt. *Alternatives for Detecting Redundancy in Storage Systems Data.* In Proceedings of the 2004 USENIX Annual Technical Conference, pages 73–86, 2004. (p 28)

[Quinlan02] Sean Quinlan and Sean Dorward. *Venti: a new approach to archival storage.* In Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST), 2002. (p 27)

[Quinlan91] Sean Quinlan. *A Cached WORM File System.* Software — Practice and Experience, 21(12):1289–1299, 1991.

[Rabin81] Michael O. Rabin. *Fingerprinting by random polynomials.* Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981. (p 28)

[Ramasubramanian04] Venugopalan Ramasubramanian and Emin Gün Sirer. *The Design and Implementation of a Next Generation Name Service for the Internet.* In Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pages 331–342, 2004. (p 139)

[Rhea02] Sean Rhea and John Kubiatowicz. *Probabilistic Location and Routing.* In Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Society, pages 1248–1257, 2002. (p 25)

[Rhea03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. *Pond: The OceanStore Prototype.* In Proceedings of the USENIX Conference on File and Storage Technologies (FAST), 2003. (pp 15, 25, 42)

[Rhea04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. *Handling Churn in a DHT.* In Proceedings of the 2004 USENIX Annual Technical Conference, Boston, Massachusetts, 2004. (pp 39, 60, 64, 106)

[Rhea05a] Sean Rhea. *Event-Driven Programming, Asynchronous Input/Output, and the Bamboo DHT*, 2005. Available at http://bamboo-dht.org/async-tutorial/. (p 42)

[Rhea05b] Sean Rhea. *OpenDHT: A Public DHT Service.* Technical Report, University of California, Berkeley, 2005. Ph.D. Dissertation. (pp 39, 60)

[Rhea05c] Sean Rhea, Byung-Gon Chun, John Kubiatowicz, and Scott Shenker. *Fixing the Embarrassing Slowness of OpenDHT on PlanetLab.* In Proceedings of the 2nd Workshop on Real, Large Distributed Systems (WORLDS), pages 25–30, 2005. (pp 61, 63, 106, 139)

[Roselli00] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. *A Comparison of File System Workloads.* In Proceedings of the USENIX Annual Technical Conference, pages 41–54, 2000. (p 108)

[Rowstron01a] Antony Rowstron and Peter Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.* In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329–350, 2001. (p 24)

[Rowstron01b] Antony Rowstron and Peter Druschel. *Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility.* In Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 188–201, 2001. (p 24)

[Ruemmler91] Chris Ruemmler and John Wilkes. *Disk Shuffling.* Technical Report HPL-91-156, Hewlett-Packard Laboratories, 1991. (p 34)

[Saito02] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam. *Taming aggressive replication in the Pangaea wide-area file system.* In Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), 2002. (pp 15, 25)

[Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. *Design and Implementation of the Sun Network Filesystem.* In Proceedings of the Summer USENIX Technical Conference, pages 119–130, 1985. (p 23)

[Santry99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. *Deciding when to forget in the Elephant file system.* In Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 110–123, 1999. (pp 27, 72, 151)

[Sapuntzakis02] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. *Optimizing the Migration of Virtual Computers.* In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), pages 377–390, 2002. (p 28)

[Schlossnagle00] Theo Schlossnagle. *Daquiri web server performance analysis tool*, 2000. http://www.omniti.com/~jesus/projects/. (p 112)

[Shah04] Purvi Shah, Jehan-François Pâris, Ahmed Amer, and Darrell D. E. Long. *Identifying Stable File Access Patterns.* In Proceedings of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST), pages 159–163, 2004.

[Shriver01] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. *Storage Management for Web Proxies.* In Proceedings of the USENIX Annual Technical Conference, pages 203–216, 2001. (p 33)

[SPEC99] Standard Performance Evaluation Corporation. *SPECWeb99*, 1999. http://www.spec.org/osg/web99. (p 107)

[Spence03] David Spence and Tim Harris. *XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform.* In Proceedings of the 12th IEEE

Symposium on High Performance Distributed Computing (HPDC), June 2003. (p 20)

[Steere97] David C. Steere. *Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency*. In Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 1997. (p 30)

[Still04] Michael Still. *Userspace filesystems with FUSE*. In Proceedings of the Ottawa Linux Symposium, 2004. (pp 38, 106)

[Stoica01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications*. In Proceedings of the ACM SIGCOMM Conference, pages 149–160, 2001. (p 24)

[SYSmark02] SYSmark, 2002. http://www.bapco.com/techdocs/. (p 108)

[Tait91] Carl D. Tait and Dan Duchamp. *Detection and Exploitation of File Working Sets*. In Proceedings of the11th International Conference on Distributed Computing Systems, pages 2–9, 1991. (p 99)

[Tsangaris90] Manolis M. Tsangaris and Jeffrey F. Naughton. *Amnesia: a stochastic access model for object stores*. Unpublished Manuscript, University of Wisconsin-Madison, 1990. (p 86)

[Tsangaris91] Manolis M. Tsangaris and Jeffrey F. Naughton. *A Stochastic Approach for Clustering in Object Bases*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 12–21, 1991. (pp 31, 34, 85)

[Tsangaris92] Manolis M. Tsangaris and Jeffrey F. Naughton. *On the Performance of Object Clustering Techniques*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 144–153, 1992. (pp 34, 35, 86, 128)

[Vahdat02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. *Scalability and Accuracy in a Large-Scale Network Emulator*. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), pages 271–284, 2002. (p 106)

[Veitch01] Alistair Veitch, Erik Riedel, Simon Towers, and John Wilkes. *Towards Global Storage Management and Data Placement*. In Proceedings of the 8th Workshop on Hot Topics in Operating Systems, pages 184–184, 2001. (p 21)

[VMWare02] VMWare Inc. *VMWare Workstation*, 2002. White paper. http://www.vmware.com/. (p 20)

[Vogels99] Werner Vogels. *File system usage in Windows NT 4.0*. In Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 93–109, 1999. (pp 28, 51, 111)

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. *A system for adaptive disk rearrangement*. Software — Practice and Experience, 20(3):225–242, 1990. (p 34)

[Warfield05] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. *Parallax: Managing Storage for a Million Machines*. In Proceedings of the 10th Workshop on Hot Topics in Operating Systems, 2005. (p 28)

[Wilkes04] John Wilkes, Jeffrey Mogul, and Jaap Suermondt. *Utilification*. In Proceedings of the 11th ACM SIGOPS European Workshop, 2004. (p 20)

[Williamson05] Mark Williamson. *Extreme Paravirtualisation: beyond arch/xen*. University of Cambridge Computer Laboratory. PhD proposal, 2005. (pp 22, 38)

[Yaghmour00] Karim Yaghmour and Michel R. Dagenais. *Measuring and Characterizing System Behavior Using Kernel-Level Event Logging*. In Proceedings of the USENIX Annual Technical Conference, pages 13–26, 2000. (p 109)

[Yue73] P. C. Yue and C. K. Wong. *On the Optimality of the Probability Ranking Scheme in Storage Applications*. Journal of the ACM, 20(4):624–633, 1973. (p 34)

[Zhao04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. *Tapestry: A Resilient Global-scale Overlay for Service Deployment*. IEEE Journal on Selected Areas in Communications (JSAC), 22(1):41–53, 2004. (p 25)

[Zhu05] Ningning Zhu, Jiawu Chen, and Tzi-cker Chiueh. *TBBT: Scalable and Accurate Trace Replay for File Server Evaluation*. In Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST), pages 323–336, 2005. (p 117)