Cluster Storage for Commodity Computation

Russell Glen Ross

Wolfson College

A dissertation submitted to the University of Cambridge
for the degree of Doctor of Philosophy

December 2006

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university. No part of this dissertation has already been, or is currently being submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words.

Summary

Standards in the computer industry have made basic components and entire architectures into commodities, and commodity hardware is increasingly being used for the heavy lifting formerly reserved for specialised platforms. Now software and services are following. Modern updates to virtualization technology make it practical to subdivide commodity servers and manage groups of heterogeneous services using commodity operating systems and tools, so services can be packaged and managed independent of the hardware on which they run. Computation as a commodity is soon to follow, moving beyond the specialised applications typical of today's utility computing.

In this dissertation, I argue for the adoption of *service clusters*—clusters of commodity machines under central control, but running services in virtual machines for arbitrary, untrusted clients—as the basic building block for an economy of *flexible commodity computation.* I outline the requirements this platform imposes on its storage system and argue that they are necessary for service clusters to be practical, but are not found in existing systems.

Next I introduce Envoy, a distributed file system for service clusters. In addition to meeting the needs of a new environment, Envoy introduces a novel file distribution scheme that organises metadata and cache management according to runtime demand. In effect, the file system is partitioned and control of each part given to the client that uses it the most; that client in turn acts as a server with caching for other clients that require concurrent access. Scalability is limited only by runtime contention, and clients share a perfectly consistent cache distributed across the cluster. As usage patterns change, the partition boundaries are updated dynamically, with urgent changes made quickly and more minor optimisations made over a longer period of time.

Experiments with the Envoy prototype demonstrate that service clusters can support cheap and rapid deployment of services, from isolated instances to groups of cooperating components with shared storage demands.

Acknowledgements

My advisor, Ian Pratt, helped in countless ways to set the course of my research and see it through, demanding rigour and offering many helpful suggestions. Major inflection points in my work always seemed to follow meetings with him. Steve Hand also helped me throughout my time at Cambridge, offering sound advice when needed and sharing his mastery of the research game. Jon Crowcroft read drafts of my dissertations, even taking time from his Christmas holiday to send me feedback.

I was fortunate to be in the Systems Research Group during a time when it was full of clever and interesting people doing good research. Andy Warfield, Alex Ho, Eva Kalyvianaki, Christian Kreibich, and Anil Madhavapeddy have been particularly rich sources of conversation and camaraderie.

I might not have survived the whole experience without Nancy, whose love and constant support provided a foundation for everything else.

Table of Contents

Chapter 1

Introduction

This dissertation presents the design and implementation of a distributed file system. The design is motivated by the requirements of a new environment that emerges at the intersection of three trends: the renaissance of machine virtualization as a tool for hardware management, the increasing use of commodity hardware and software for server applications, and the emergence of computation as a commodity service.

Providing computing services for hire is nothing new, but previous offerings have always been on a limited scale, restricted to a specific type of application or tied to the tool stack of a single vendor. Web and email hosting services are commonplace, and many applications like tax preparation software and payroll management are available as hosted services. Numerous frameworks for hosted computing have been proposed [Ami98, Vah98, Tul98] and commercially implemented [Ama06, Sun06, Kal04], but these require using a specific distributed framework or middleware that introduces a "semantic bottleneck" between the application and the hosting environment [Ros00a]. The only way to make hosted computing a commodity is to convince vendors and customers to agree on a common set of interfaces so that applications can be easily moved from one host to the next and an open marketplace can emerge.

Seeking universal agreement on a new application environment is a daunting task, but one that can be circumvented by using an existing platform that already enjoys wide acceptance: the PC. Machine virtualization technology makes it possible to emulate a standard PC with low overhead, while still controlling the resources clients have access to and giving them a secure, reliable environment. Using virtual machines as basic deployment units gives applications access to standard tools and operating systems, and lets them move more easily from a local testing environment to a remote host, or between competing vendors. A combination of flexibility, compatibility, and manageability makes virtual machines stand out as a commodity hosting container.

In this dissertation, I propose that a platform for a commodity computation service be built on clusters of commodity hardware using virtual machines as the management unit. Designing for clusters introduces scale at the implementation level that platform providers can exploit to reduce costs, support a wider range of applications, improve resource utilisation, and foster an ecosystem of intermediate service providers. Instead of managing a complete tool stack demanding a diverse range of expertise, vendors can specialise in hardware provision and management while still achieving the scale to make it worthwhile and leave room for service differentiation. More specialised services and middleware can exist as a value-added service from the same vendor, or third-party providers can layer their services above the virtualized hardware and sell their software and expertise to end users. In this way, vendors of hosted services can focus their efforts on their core expertise without having to build a widely-distributed network or branch out into hardware management as a prerequisite.

Numerous research and engineering challenges are posed by this environment, but this dissertation focuses on providing storage to hosted virtual machines. Unlike other cluster file systems that focus on parallel computation and other scientific workloads, the Envoy file system proposed here is optimised for the requirements of commodity computation. Managers must be able to migrate running services to balance load and make efficient use of hardware resources, so storage cannot be tied to a specific host. Most virtual machines require private boot images, but some also need to coordinate their efforts through shared storage, with flexible control over shared access. The storage system must offer efficient shared access to images with strong consistency guarantees, but it must also accommodate many private images per machine across thousands of machines.

Forcing clients to start from scratch with each deployment and upload an entire operating system image would be costly and slow, and hosting multiple file system images per machine adds a new multiplier to scaling requirements, particularly capacity scaling [War05]. Expanding the definition of the environment to include a base of commodity software as well as a commodity hardware interface offers a solution to both problems. A few well-known software distributions can be installed by the vendor and offered to clients as templates that they can customise according to their own requirements. By deliberately injecting redundancy into client images, the storage demands can be reduced to a few template images plus the customisations and other data unique to each client. Besides easing storage demands and making inter-client caching more effective, this ties the cost of commodity computation to the degree of customisation required. Clients using standard tools enjoy lower deployment costs than those requiring extensive customisation, encouraging standardisation through economic pressures without imposing artificial restrictions.

Storage systems are normally measured by their performance, but their suitability to the environment they serve is equally important. If clusters of untrusted virtual machines are to succeed as a commodity computation platform, they must be served by a storage system optimised for the expected workload and with adequate features to make management practical.

1.1 Contributions

The thesis of this dissertation is that clusters hosting virtual machines provide a viable platform for commodity computation, and a file system optimised for that environment can support the commodity computation model by providing useful management features and scaling to accommodate arbitrary numbers of file system images under expected workloads.

The first contribution of this work is a definition of the requirements of a flexible commodity computation environment. Commodities are homogeneous enough to make suppliers interchangeable, and have a large enough market for economic forces to ensure that the cost to clients is directly related to the marginal costs of the product or service. I argue specifically for clusters of commodity hardware as the basis of a computation platform, with computation in a virtual machine as the product offered to customers.

The second contribution of this dissertation is the design and prototype implementation of a file system designed for the outlined environment. It builds on previous work on cluster storage systems, but addresses the management needs and scaling characteristics of a platform supporting many independent clients. It also exploits its environment to use the cheap storage available on commodity machines, reduce complexity by integrating with the virtual machine structure, and reduce capacity requirements by explicitly acknowledging the redundancy resulting from a template-based deployment model.

1.2 Outline

The remainder of this dissertation is structured as follows:

Chapter 2 discusses the relevant background, including the state of commodity computation and the developments in machine virtualization that underpin this work. Special attention is given to the extensive body of work on storage systems, with a focus on how storage systems are shaped by their intended environments, and how they relate to the new environment proposed here.

In Chapter 3 I define the problem of flexible commodity computation, and argue for virtual machines hosted by clusters of commodity hardware as the platform for a computation economy. The first part of the argument is that virtual machines decouple hardware management from software management, encouraging transparent competition and specialisation that is not tied to a particular application domain. The second point is that clusters take advantage of economies of scale and allow unrelated clients to share hosting, where careful management can balance the demands of diverse users and make efficient use of resources. The chapter concludes with a discussion of the role of the storage system in this environment.

Chapter 4 presents Envoy, a file system for clusters of virtual machines. Like many cluster file systems it builds a distributed file system above a separate object storage layer. Envoy partitions management of the global namespace along hierarchical lines, assigning control of a *territory* to the machine that uses it most, and territory boundaries are dynamically updated in response to runtime conditions. Objects are cached and served directly by the machine that controls the associated territory, eliminating distributed cache coherency protocols while still permitting cache sharing between clients.

The prototype implementation of Envoy is described in Chapter 5 and evaluated in Chapter 6. The basic functionality of Envoy is fully implemented, with file systems exported to clients using a Linux implementation of the 9p protocol from the Plan 9 operation system. The evaluation tests the performance and scalability of the prototype, validating the overall design and suggesting possible improvements in specific areas.

Finally, Chapter 7 concludes and discusses directions for future investigation.

Chapter 2

Background

This dissertation builds on the rich body of research and systems that have preceded it. In this chapter I present relevant background material to establish the context of the current work. First comes a discussion of commoditisation within the computer industry and how it relates to the proposal in Chapter 3 for a commodity computation platform. Next, the current state of machine virtualization—an important component of the proposed platform—is presented with an emphasis on recent systems for the x86 architecture. The chapter then turns to influential storage systems and other work that relates to the Envoy file system described in later chapters.

2.1 Commodity computing

Commoditisation relates to computing in two distinct ways, both of which are relevant here. The first is the general consolidation of PC hardware into a series of commodity parts, which are cheap enough and powerful enough to take over many applications that were the domain of specialised "big-iron" hardware in the past. The second is the push to offer computation as a service instead of as a product, where billing is based on work done rather than on hardware and software delivered. This dissertation proposes using clusters of commodity hardware as a platform for offering commodity computation services.

2.1.1 Commodity big iron

The scale of the PC market ensures that the best the semiconductor market has to offer is available for PCs. The difference between cheap commodity hardware and expensive mainframes and supercomputers is in scalability, reliability, and manageability. High-end equipment is designed to offer high bandwidth on buses and I/O channels and to scale through highly-parallel configurations. Redundant components mask some classes of faults, and others can be isolated and repaired without interrupting the operation of unaffected components. While these features could be offered in commodity machines, they add too much cost to be worthwhile in environments where service interruption is more annoying than expensive.

Networks of workstations

The proposal for Networks of Workstations (NOW) was an early argument for using networks of smaller computers for large problems instead of seeking parallelism through specialised hardware [And95a]. While individual components are less performant, the aggregate capacity of a large collection of machines is immense. Commodity disks may be slow, but accessing them in parallel yields high bandwidth as well as high capacity. Reliability and fault tolerance can be designed as a property of the combined system, rather than being engineered into individual hardware components. This has the advantage of transferring the expense from a per-unit manufacturing overhead to a one-time cost in the software design.

PCs have encroached on the workstation market and the two terms are largely synonymous now. The same trends that prevailed in the mid 90s for workstations continue today, however. Switched networks lead to aggregate bandwidth that scales with the number of hosts. Single-threaded performance on cheap PCs is a significant fraction of that on the fastest server hardware. Disks continue to grow in capacity, but their performance improves at a much slower rate. Performance is best scaled through parallelism, and a collection of many machines that all have cheap disks has the potential to outperform even the fastest single controller or collection of high-performance disks attached to a single machine.

Clusters

While workstations in an organisation can be profitably pooled for large jobs, clusters are specially built for that purpose and eliminate the workstation role from a node's list of duties. The basic principles behind clusters were laid down in the 1960s [Amd67], but the power and affordability of recent commodity hardware has brought clusters to renewed prominence. The same hardware used in high-street PCs powers some of the most powerful computing sites in the world [Top06].

Dedicated clusters have a few advantages over more loosely-organised structures. Nodes are typically housed in a controlled environment with redundancy built into power supplies, network links, and cooling equipment. While many clusters grow incrementally, they are still relatively homogeneous, being built from batches of identical hardware. They are also professionally managed and monitored with central control over all resources, unlike looser amalgamations of machines.

As parallel computing vehicles, clusters are mainly used for large "embarrassingly parallel"[1] problems that can be neatly decomposed into discrete tasks. Google has been prominent among commercial users of commodity clusters, and their search engine architecture is an example of a highly-parallel task. The search index can be cloned to serve many concurrent transactions, and partitioned to split processing for a single query over multiple nodes [Bar03b]. Scientific workloads, simulations,

[1] See http://en.wikipedia.org/wiki/Embarrassingly_parallel

and graphics rendering farms are all characterised by repetitive computations over huge data sets, and are similarly well-suited to cluster implementation. Beowulf is a popular system specifically designed for addressing these kinds of tasks on clusters of commodity hardware running Linux or other Unix-like operating systems [Rid97].

These problems are generally large enough to warrant custom software solutions as well as dedicated hardware resources. Management software for clusters usually assumes that the application is trusted and cooperative. Systems like TranSend [Fox97] and CARD [And97] require application software to be divided into discrete application units that the monitoring software can direct. Migrating tasks from node to node and compensating for failed nodes requires cooperation from the application, or interaction with a transaction interface that exposes application semantics. Beowulf [Rid97] is a lower-level toolkit, providing libraries of useful tools for building a custom cluster application.

For some classes of problems, clusters dominate as the architecture of choice, with clusters of commodity hardware increasingly winning over clusters of specialised hardware. This dissertation does not seek to supplant them for inherently parallel tasks, but rather to extend their advantages into other problem domains. While clusters are a sound choice for problems requiring the dedicated effort of thousands of machines, they can be an equally viable platform for problems too small to require the undivided attention of even a single node. Clusters are normally built and located for a specific user and a specific task, but they can also be built where conditions are best—perhaps near network interconnect points or where cheap resources abound or a favourable regulatory environment exists; strategic choices can give vendors a competitive edge. The advantages of scale can be brought to bear on small and medium-sized problems, not just on the largest of tasks.

2.1.2 Computation for hire

Computer hardware is cheap enough that most organisations can afford to buy systems with sufficient power for their needs. However, because of the complexity of managing computer systems and the ongoing operating and maintenance costs, this is not always the best way to proceed. Numerous avenues exist for clients wanting to hire computation services instead of owning and managing their own hardware and software.

Replacing the machine room

The primary motivation for outsourcing computation as considered here is to reduce or eliminate the role of the machine room. Dedicated server hardware comes with expenses that may outweigh the benefits of local ownership and control.

The most obvious costs are direct: physical space required, electricity consumed, and heat produced. Commodity hardware is cheap enough that these runtime costs may exceed the capital cost of the hardware over its service lifetime [Bar05]. Hardware must either be located at a company site, where appropriate facilities must be installed and internet connectivity hired, or at a dedicated hosting facility, where managing it is more complicated and may require sending staff to a remote location.

Maintaining servers also requires qualified staff, another significant expense. The skills required to manage computer systems are unrelated to the core mission of many organisations, forcing them to develop a sizable technical staff to support operations that have little to do with technology. Running a parallel group outside an organisation's core expertise may lead to additional inefficiencies.

These costs are even less palatable when the computation facilities are only needed intermittently. Unpredictable demand may force planning based on peak requirements, but many costs are fixed and do not drop during periods of low demand. When asked about the expenses of their large computing infrastructure, Jeff Bezos of Amazon claimed that cost from lack of utilisation dominated the costs of power, servers, and people [Bez06].

Hosted services

Some services are generic enough to be hosted remotely by specialised providers, who make money by offering the same inexpensive service to many clients. Basic applications like email, payroll management, web hosting, and other common services have already attracted commercial interest. Hosting them locally brings all the negative aspects of managing an on-site machine room, while adding little or no value over using a remote provider.

Hosted software eliminates the need for the client to buy or manage dedicated hardware, and the application is managed by specialists who can amortise fixed costs over a large customer base. Upgrades do not require work from the client, making it practical to keep all customers synchronised and further reduce support costs. The unpredictable demand of individual clients can become predictable in the aggregate, allowing more efficient hardware use.

Similar arguments apply to storage outsourcing [Ng02], for those wishing to hire managed storage while retaining local control of applications. Multi-site backups, capacity planning, and other domain-specific concerns can be turned over to a specialist, leaving local staff to focus on the problems unique to the organisation.

Hosted services fit well with the present proposal, which takes the concept a step further by splitting management of hardware from management of software. In the service cluster model advocated here, service providers can focus entirely on their applications, leaving hardware management to a dedicated hosting service. Alternatively, clients can hire software management services from one vendor and hardware hosting directly from another, isolating the specific value offered by a software vendor from the basic operating costs that all software applications require.

Utility computing

Application hosting is most practical for widely-used services, where few differences exist between clients. The proposal in this dissertation is a form of utility computing, where hardware resources are made available for custom applications at a cost related to resource consumption. *Utility computing* refers to a range of computing models, with the common characteristic of on-demand resource provision. Hosted services as described above, and grid computing as described below, fall under the umbrella of utility computing.

Existing work focuses mainly on large applications where abundant resources are needed, but not necessarily over a long enough period to justify building a suitable hardware solution. An example is rendering feature-length animated films, which requires vast computing power and complex interactions, but only until the project is finished. Vendors with sufficient resources can manage them like batch processors on early mainframes, assigning hardware to a series of applications that have been suitably prepared for the environment [Wil04]. HP's Utility Data Center (UDC) [Kal04] uses virtualization to manage services, much like the service clusters described in Chapter 3, but focuses on large projects that can be individually adapted to the UDC environment. Amazon's Elastic Compute Cloud (EC2) service uses the Xen virtual machine manager to offer hosted computing at large or small scales, with billing by the instance-hour and by bandwidth used [Ama06]. Like the current proposal, it relies on commodity software environments to minimise the changes required to migrate from a locally-owned machine to a hosted service.

The Collective project also uses virtualization to isolate services, and proposes turning the desktop PCs of an organisation into a pool of utility computing hosts to simplify administration [Sap03]. To end users, interactive applications and services appear as *virtual appliances* that can be instantiated on a local PC. This allows specialised staff to manage desktop software as well as machine-room applications. Desktop PCs act somewhat like terminals for managed software, but terminals that also host computation for their own clients. In some sense, this is the reverse of other utility computing environments; it delivers managed software services to local hardware instead of offering hosted hardware to run client applications.

Grid computing

Grid computing [Fos03, Sun06] seeks to combine resources in the wide area to provide on-demand computation. Utility computing gets its name from basic utilities like power, water, and sewer service, offering a common service to many clients who pay according to what they use. Grid computing styles itself after power grids, which provide a meshed infrastructure to match power producers with consumers across the boundaries of a single utility.

Pooling the resources of many organisations has the potential to put vast resources at the disposal of clients, but it also introduces new problems. Unlike the cluster environments typical of utility computing, grid applications must communicate over the wide area with its low bandwidth and high latency. This makes the grid model most suitable for highly parallel CPU-intensive jobs, which can be parcelled into discrete units and distributed to remote nodes. Moving from a single, trusted management environment makes security considerations more complex as well.

Grid computing requires special middleware, though using virtual machines to support more general-purpose clients has been proposed [Fig03, Zha04]. Even then, poor connectivity makes it slow and expensive to transfer large data sets or share data extensively between nodes. The distribution model of grid computing makes it poorly suited to interactive applications or those that require close cooperation between nodes.

Existing utility computing solutions do little to address the needs of an organisation with a relatively small computing infrastructure hoping to stay out of the hardware management business. A few applications can be outsourced to a service provider, and large jobs can be run on hired hardware, but these systems are only applicable to a small subset of the applications running in a typical organisation's machine room. To achieve the scale necessary to make an offering profitable, vendors have focused on hosting widely-used applications and projects that are already large. This differs from the current proposal, which aims to make hosted computation a commodity service practical for a wider range of client applications.

2.2 Virtualization

Resource virtualization makes it possible to put familiar tools and interfaces in new environments without substantial modifications. Virtual machine abstractions allow standard operating systems to share hardware even when they are designed to assume exclusive control. Virtual disk abstractions provide a block interface like that of a standard disk, but without tying it directly to a single piece of hardware. Virtualization is used for a variety of reasons, but the relevant purpose here is for resource management and sharing.

2.2.1 Virtual machine monitors

IBM VM/370 [Gum83] first virtualized hardware to support legacy code without modification. Modern commercial systems like VMWare [Dev98] and Microsoft Virtual Server [Mic06] are often used for similar purposes, allowing standard operating systems to run unmodified as guests in new environments. This is useful for testing labs, which can test applications against a range of software configurations on limited hardware, and for end users who need access to applications from multiple software platforms. While this allows access to a wide range of guest software, it comes at a performance cost on some hardware—including the standard x86 platform—which was not designed with virtualization in mind.

Virtual machine monitors (VMM) work by imposing a thin software layer between the operating system and the bare hardware. Unlike emulators or simulators, VMMs allow code to execute directly on the hardware whenever possible. Certain operations break the illusion of isolation, however, particularly those involving I/O, memory protection, and interrupt management. The VMM must intercept these operations and simulate the desired effects on the virtual machine. If the hardware does not allow the VMM to directly intercept such operations, it must do so using software means, such as code rewriting or emulation. The overhead of such virtualization techniques is highest when I/O operations are frequency, as is common in server applications.

The Xen hypervisor uses *paravirtualization*, where the operating system is explicitly ported to the hypervisor environment. By eliminating expensive virtualization techniques for I/O operations, paravirtualization leads to low overhead even for network and database applications with heavy I/O components [Bar03a]. This does not affect userspace applications, whose I/O operations are already virtualized by the operating system, so unmodified binaries can still be used with the modified operating system. The down side is that paravirtualization does not provide support for systems that are not modified to support it, including legacy systems. The combination of high performance and support for modern commodity systems makes it suitable as a host for a managed platform like the one proposed here. Recent updates to the x86 platform support full hardware virtualization [Ada06], allowing unported systems to run as well, although explicit support for the paravirtualized environment is still beneficial in reducing overhead.

Virtualization allows a commodity computation platform to isolate hosted services from each other and allow them to be managed without explicit cooperation at the application level. Multiple guest operating systems can run on the same hardware, and virtual machines can be migrated transparently to balance load, collocate related services, or free a machine for servicing [Cla05]. Virtual machines can even be frozen and migrated to remote sites [Sap02], a deployment strategy used by vMatrix [Awa02] to migrate and replicate specialised internet services.

The IBM zSeries [IBM06] platform provides a managed hosting environment for paravirtualized Linux, but it does so on specialised mainframe hardware and ties clients to a single vendor solution. The availability of fast, secure virtualization for cheap commodity hardware opens up the possibility of applying these techniques to create an open commodity market for computation.

2.2.2 XenoServers

A public computing platform requires more than just hosting technology. The XenoServers project [Ree99, Kot04a] explores the infrastructure required to host network services on a public computing platform. The Xen hypervisor started as part of the XenoServers project, which motivated its emphasis on performance over compatibility with legacy code. Application binaries can run without modification in a paravirtualized system, so the modifications Xen requires only affect the operating system.

The XenoServers project also addresses the problem of helping clients find a suitable host [Spe03]. Network services may have specific connectivity requirements, especially when being deployed for use by a small group or an individual. For example, a game server connecting a group of players requires low latency to a specific set of clients to support real-time interaction. To support a large number of hosting providers, XenoServers also defines the infrastructure for billing through a trusted third party.

This dissertation is not directly linked to the XenoServers project, but it is related. XenoServers provides much of the general framework for public computing, but it does not specify the architecture for hosts beyond the use of the Xen hypervisor. In this work I propose using clusters of commodity hardware as the basis for public computing, and I offer a storage system suitable for that environment. The Envoy file system proposed here also supports a deployment strategy similar to the one we described as part of the XenoServers project [Kot04b].

2.3 Storage systems

Storage is a fundamental component of all general-purpose computer systems. A combination of high storage density, random access, and low cost has made magnetic storage on rotating platters the dominant medium for durable storage. The time to access a random byte of data from a disk is typically measured in milliseconds, while for DRAM this time drops roughly six orders of magnitude to a nanosecond scale, and that gap is continuing to grow.

Because disks are so slow, storage systems are designed around the goal of accessing the disk as little as possible, and favouring sequential access to avoid costly seek delays. Effective use of cache is vital to this goal, and specialised storage systems tuned to specific system architectures and expected workloads are worthwhile because of the potential speed gains over more general systems.

Storage systems have been studied extensively, and one purpose of this section is to survey related work that has influenced this dissertation. A second purpose relates to the thesis of this work, which argues that an environment can only succeed with an appropriate storage architecture. The works discussed here are presented in the context of the environments they support, in part to establish a pattern of symbiosis between storage architectures and computation environments. Envoy, the file system introduced in this dissertation, is discussed briefly in this section to relate it to prior work, but details of its design are reserved for later chapters.

2.3.1 Local file systems

Disks are mechanical devices and their performance is limited primarily by the need to physically position the disk head over the correct part of the platter to access data. File systems designed for local disks achieve performance by minimising the physical movement required. Disk access can be reduced through caching, and hardware latency can be minimised by arranging data on the disk to minimise movement. Correctness and reliability tend to trump performance as design considerations, however, and the wide variety of possible workloads makes it difficult to pick clear winners from competing designs.

The Berkeley Fast File System (FFS) was the first to optimise data layout for performance by clustering related information. Block metadata is distributed across the disk to be near the file data it describes, file metadata is grouped for files in the same directory, and blocks in the same file are grouped whenever possible [McK84]. *Clustering*, as this design is called, exploits concepts that extend beyond local file systems: the same expectations of correlated access can be exploited to reduce latency when accessing data across the network [Ame02].

File system tracing reveals how files are accessed in real systems [Ous85]. Most files accessed are small, but size distributions are skewed enough that most data transferred is from large files. Writes are less common than reads, and most files are short-lived. Files access is typically sequential, and most files are read from or written to in their entirety. These trends have proved resilient over time, though the scale has increased and the largest files in typical systems have grown much larger [Rue93, Gib98b, Ros00b].

As cache sizes grow, more requests can be satisfied without consulting the disk, and designs can assume that many read requests can be served from memory. All writes whose effects are not quickly undone have to go to disk eventually, so despite being less frequent overall, writes requests can dominate the mix of operations that penetrates the cache and reaches the disk. Furthermore, updates often involve metadata changes as well, potentially requiring multiple costly disk seeks even for small updates.

Log-structured file systems (LFS) address this problem by borrowing from database design and making the entire file system an append-only log. Writes are gathered and written as sequential chunks on disk, with relevant metadata rewritten instead of updating existing structures directly [Ros91]. *Journaling* file systems apply the same idea to other file system types, logging only the intent to update metadata. Once the log is committed, the conventional structures can be updated asynchronously while still guaranteeing durability in the face of a system crash [Hag87, Swe96, Twe98]. Breaking the chain of synchronous metadata commits is also the objective of *soft updates*, a technique that involves careful rearrangement of data in the buffer cache to permit delays and reordered writes [Gan94].

Studies comparing FFS with LFS [Sel95] and journaling with soft updates [Sel00] reveal a complicated picture. Transaction workloads force frequent synchronous writes to support their own semantics, negating the benefits of aggregated writes, and updates to clustered file systems reduce the ordered-write problem enough to keep it competitive with LFS for small file updates. Very different strategies can lead to comparable results, but for all local file systems it is careful attention to the motion of the disk head that leads to good performance.

2.3.2 Client-server file systems

If local file systems can be characterised by how they manage disk head motion, distributed file systems are dominated by concerns of data placement and cache management. For a single host, cache management is easy. The OS has a monopoly on the disk and can reconcile any concurrent requests. Complications are mainly concerned with deciding when to commit writes to disk to ensure durability in the face of a system crash. Distributed systems must also consider consistency between caches on multiple machines. If a cache delivers an out-of-date version of a file, the application may be led to produce incorrect results.

The Sun Network File System (NFS) was the first widely used file system for sharing files across hosts [San85]. NFS serves many clients from a single access point (typically a server or a dedicated storage appliance [Hit94]), which hosts the persistent and canonical version of a file. Cache policy is unspecified, with no explicit support from the server. Clients generally cache reads and writes in memory and check with the server before relying on old cache entries (typically in the range of 10s of seconds). Thus an update made by one client is only detected by another when the first has sent the update to the server and the second has checked for an update.

Clients can send a constant stream of `stat` requests to check for updates, but they cannot hasten a delayed update from another client, so they can never be assured of having the latest version of a file. An update to the protocol helped reduce traffic somewhat by performing implicit `stat` requests and including the results with common operations [Paw94, Cal95], but the fundamental problem remains as clients still delay writes to the server. The latest update, NFSv4, includes features to improve cache management for uncontended data by *delegating* complete control of individual files to clients and *revoking* the delegation when other clients seek concurrent access, at which point it reverts to the consistency semantics of earlier versions [She03].

Other client-server systems address the problem in different ways. The Andrew File System (AFS) [Sat85, How88] uses the client's disk as a persistent cache with *close-to-open* semantics. In this scheme, the cache is validated at file open time, and changes forwarded to the server at file close time. To reduce validity checks at file open time, a client can be given a *callback*, meaning that it can assume the file is current unless explicitly notified by the server. These changes improve scalability by enlarging the effective cache size on clients and reducing the load on the server, but they still leave open the possibility of conflicting updates by multiple clients. Coda extends AFS to explore the problem of conflict resolution, opening up access to mobile users and allowing clients to continue operating when disconnected from the network [Sat90, Mum95]. DEcorum goes the other way, extending AFS to strengthen cache consistency using *token passing*—a scheme where each file has a single logical token that a client must obtain before it can write to the file [Bur88]—as well as to interoperate better with existing file systems and reduce recovery time after a crash [Kaz90].

The Sprite team observed that—despite the popularity of NFS—few applications explicitly address inconsistencies introduced by the file system, so loosening consistency guarantees in favour of performance gains is a dangerous tradeoff. They sacrificed some performance for full cache consistency by disabling caching for files under contention [Bak91, Nel88, Wel91]. Since concurrent access is relatively rare [Kis91], this does not pose a significant problem for overall performance.

In all distributed file systems that permit sharing, there must either be a canonical version of the file (or block [McG98]) or some way to reconcile conflicting updates [Kis91]. In the former case, some participant is usually nominated as the owner of a particular file through a lease [Gra89], token passing [Bur88, Man94, Kaz90], or some other scheme. Any other host wanting access to the latest version must coordinate through the owner. Ownership may go to the host that provides storage, the one actively using the file, or a management host that connects the two [Bla93, Kel94]. In Envoy, ownership goes to an active user, which then acts as a synchronous server to other users. Unlike Sprite, the principal user can continue to cache the file locally and share its cache with other concurrent users.

Another possibility is to disallow write sharing. The Cedar file system [Sch85] makes all shared files immutable, and turns the problem into one of versioning [Gif88]. Venti takes this a step further by storing all file versions permanently and addressing them by a hash of their contents [Qui02]. Since files are never changed or deleted, the store collects a complete history of all historical states of the file system. In workstation environments, it is possible for storage capacity to grow faster than storage is consumed, making this a feasible system, or past versions can be selectively removed as in the local file system Elephant [San99].

A less drastic approach is to permit snapshots [Hit94], then mark historical versions as read-only with no requirement for cache coordination [War05]. Envoy employs this dichotomy between active and read-only file versions, implementing cache management only for mutable objects. It does not coalesce identical read-only objects like Venti or Farsite [Dou02]. It could be extended to do so using a lazy, asynchronous process, but it reduces the need by promoting file system forking with copy-on-write as a management tool to avoid creating many of the duplicates in the first place.

Where snapshot systems typically use copy-on-write to transparently combine old data with new, some systems allow explicit stacking of file systems. Spring [Kha93] allows file systems to be layered with optionally synchronised updates as a mechanism for extending functionality by layering in new features. Plan 9 [Pik90] allows any file system mount to be layered over another and their contents combined to give each user a custom view of local and remote file systems [Pik92]. The copy-on-write NFS server I developed for the XenoServers project [Kot04b] allows layering instructions to be put in a file in any directory, directing the server to immediately reconfigure the user's view of the file system. These systems can be used as a way to fork a file system, by sharing the common base image and capturing changes in a private layer. Over time this can lead to complex hierarchies of layers, and such systems rely on the semantics of their backing file systems. Envoy can be used in conjunction with a stacking layer, but it already provides explicit support for snapshots and file system forks.

2.3.3 Serverless file systems

While creative caching can alleviate the problem somewhat, all client-server architectures have inherent scaling problems. As a single point of contact for all clients, a server is subject to load that grows in proportion to the number of clients, and it also represents a single point of failure. In addition, the duties of a server tend to make it unsuitable for other uses, so such an architecture calls for a dedicated server. As workstations have grown in power, harnessing their excess capacity to cooperate on large problems has become increasingly attractive [And95a].

In xFS, the traditional roles of a server are split and distributed to the clients to yield a serverless architecture [And95b]. Hosts can act as clients, managers that coordinate data placement, and/or storage servers that provide disk space, similar to the file system of the earlier LOCUS distributed operating system [Wal83]. Cache coherency is achieved through an explicit consistency protocol where conflicting client requests are detected and managed.

In addition, the xFS team observed that modern networks make retrieval from a peer's cache faster than from a disk [Dah94a], so sharing and coordinating cache across hosts can yield benefits over independent local caches [Dah94b]. The resulting protocol was so complex that the team had to employ a formal protocol verifier to get a working implementation [Wan98]. While a valid way to manage complexity, reducing complexity through design may be preferable, especially in storage systems where correctness is paramount.

Farsite also targets a workstation environment, but assumes that participating machines are not trusted [Ady02]. This requires encrypting data and using complex Byzantine agreement protocols instead of trusting hosts that have been assigned management roles. It also calls for a higher replication factor to guard against malicious attacks as well as hardware failures [Dah94b], and makes cache sharing between hosts less practical. These restrictions are imposed by the environment, again highlighting the importance of matching storage design to expected conditions.

The downside of having workstations double as servers is that the server function is not completely isolated from the other activities of the workstation. Server load is normally determined by the aggregate activity of many clients instead of that of a single unpredictable user, and a user may also switch a workstation off without notice. While a server can also fail unexpectedly, careful administration generally makes this an infrequent event. Extra redundancy is necessary to make files *available*, even when they are still reliably stored on a workstation that has been powered down.

The same trends that lead to excess capacity in workstations make dedicated servers cheap and powerful, without the additional complexity of a heterogeneous management environment. Despite numerous studies showing feasibility [Bol00, Dou99, Dou01] and systems developed and tested [Ady02, Wal83], no serverless file system has seen widespread use for general-purpose computing.

While Envoy has similar goals for serving a location-independent file hierarchy to many untrusted clients, putting it in a trusted cluster environment changes the assumptions significantly. Hardware virtualization allows malicious clients to coexist on hardware with trusted server processes, and the complexity of Byzantine failure models can be avoided. Managed hardware also means that replication factors can be planned around hardware failure rates and load balancing without worrying about hosts being switched off by desktop users.

2.3.4 Wide-area file systems

Carrying the idea of distributed file systems to the logical extreme leads to global file systems, running on hosts throughout the world and serving millions of clients located anywhere. Latency and available bandwidth become major concerns in this environment, and the inability to trust hosts forces widely-distributed file systems—as was true with Farsite in a smaller setting—to focus mainly on managing replicas for availability and reliability.

Systems with servers

Ficus [Guy90] and Echo [Bir93] link servers together to form a single, global file hierarchy, with transparent navigation between the discrete volumes that make up the system. In Echo, entire volumes are replicated in tightly-synchronised groups with one *primary* server and one or more *standby* servers. A token-passing scheme allows clients to cache files locally but still maintain global consistency [Man94]. Ficus relaxes the synchronisation requirements in favour of optimistic concurrency, where conflicts must be resolved after being detected. Volume replicas are loosely synchronised, and each may hold copies of only a subset of the files logically contained in the volume. Updates can be made to any file that has at least one replica available [Pop90], permitting continued operation in the face of network partitions or other failures, similar to Coda [Kis91].

An early version of xFS [Wan93] also follows a two-tier model, where loose clusters of nearby hosts work together and share a single *consistency server*. The consistency server acts as a proxy for the group when communicating in the wide area, and as a coordinator for operations within the cluster. Requests are served from the cache of a local host when possible, and forwarded to a remote cluster when necessary. Consistency is maintained through tokens that permit local caching for multiple readers or a single writer. To reduce the state that must be tracked, tokens cover entire groups of files.

JetFile [Grö99] and Pangaea [Sai02a] rely on pervasive replication with little overall structure. JetFile uses specialised servers for a few metadata functions, but most operations happen directly between clients. In both systems clients maintain replicas of all files they are interested in, and optimistic concurrency control permits disconnected operation. They differ in how updates are propagated. Pangaea maintains a replica graph for each file and pushes updates to other clients [Sai02b], while JetFile makes extensive use of IP multicast to locate replicas and announce changes, and clients pull updates when required. When conflicting updates are detected in either system, the versions must be reconciled explicitly, generally using a last-writer-wins policy.

Peer-to-peer systems

Relying on trusted servers restricts the audience for a wide-area file system to large organisations and service providers who can maintain widely-dispersed networks. Peer-to-peer systems use resources volunteered by participants on large numbers of machines.

The simplest systems are read-only file distribution schemes. Bittorrent tracks all clients with a central manager, but data blocks are transferred mainly between clients [Coh03, Pou05], which request blocks they have not yet received from peers that have already downloaded them. Avalanche [Gka05] uses network coding to decrease the incidence of “rare” blocks that make it difficult for clients to complete the last steps of a file download. Both systems feature capacity that grows with the number of users, without consuming excessive bandwidth at the server. Such systems are mainly useful for sharing the cost of distributing static content with those who benefit from it, and not for general-purpose storage needs.

Wide-area file systems can also be built using *distributed hash tables* (DHT) such as CAN [Rat01], Pastry [Row01a], Chord [Sto01], and Tapestry [Zha01], which provide distributed lookup services on peer-to-peer networks. CFS [Dab01] and PAST [Row01b] implement read-only systems using content-based addressing, similar to Venti [Qui02], but using an underlying DHT to locate object replicas. Other systems implement mutable file systems over similar substrates, storing data blocks, files, or content-based fragments [Rab81] as immutable objects distributed across the network. Pasta [Mor02] stores metadata in special index blocks, each of which is associated with an asymmetric cryptographic key. The key is used to locate the index block (instead of using a hash of the contents as for normal data blocks) and to protect changes to it, allowing the index to change while retaining a unique static ID. Eliot [Ste02] stores metadata outside the immutable substrate, creating a separate, writable system that references the read-only data indexed by the DHT. Ivy [Mut02] stores all user changes to the user's log, which is then made available to others through the DHT, and each user consults as many logs as necessary to construct a coherent view of the file system.

Peer-to-peer file systems all suffer from the transience of users. Volunteered resources can be withdrawn without notice, so high levels of replication are required to ensure accessibility and availability [Bla03, Rab89]. Latency is also high in the wide area, so further replication and caching is necessary to make performance acceptable. OceanStore [Kub00] was designed as a global network of highly-connected clusters that cooperate closely, with additional clustering at the file level for groups of files that are regularly accessed together. The prototype, called Pond [Rhe03], uses Tapestry to organise virtual resources (data blocks and manager nodes), but it also forms localised clusters of participants to implement Byzantine agreement protocols without the high latency typical of DHTs. These systems are tiered to reclaim some of the benefits of locality while still providing a global service.

Envoy has some structural parallels with these systems. While latency between machines in a cluster is much lower, Envoy still caches data on disk and in memory near the client. Locality is pursued at the machine level to aggregate the storage requirements of a set of virtual machines, with data ultimately replicated and spread throughout the cluster as storage objects.

The relationship of the present work to global storage systems is more than just a passing architectural resemblance, however. Envoy and the service clusters that host it are designed for commercial providers that locate computational resources near storage and fast network access. Instead of trying to hide the distance between users and data, service clusters share the goal of XenoServers [Ree99] to move computation to a resource-rich environment. As a basic platform, service clusters backed by Envoy storage can form the building blocks of global service networks, including storage services.

2.3.5 Cluster storage systems

The path from local file systems to wide-area file systems is generally progressive, with concerns about cache management, trust, latency, replica placement, and reliability growing at each step. Clusters have networks with low latency and high bandwidth, and large numbers of trusted, centrally-managed hosts. Their mixture of characteristics from the largest and smallest of systems partially explains their popularity as replacements for traditional, monolithic supercomputer architectures.

With a large number of redundant components, clusters have the potential to be highly reliable (in the aggregate) with abundant bandwidth. One of the principal drivers of cluster storage systems is avoiding bottlenecks. While this is important in all systems, the vast aggregate disk and network bandwidth available to a large cluster makes it easy to overwhelm any single component [Hos04]. In addition, the opportunity cost of wasted resources is potentially very high, so cluster systems must make good use of the combined storage capacity and I/O bandwidth available from the array of component machines.

Storage layers

A common approach for cluster storage systems is to divide the problem into two distinct layers: one that manages the physical disks, balancing load and capacity and handling the addition and loss of disks, and a second layer that builds a file system above an abstract block- or object-level interface provided by the first.

The Multi-Service Storage Architecture divides storage into two distinct services in order to support a wider range of client. The lower level gives clients direct access to unstructured objects, while still enforcing access restrictions and offering concurrency control. The higher level offers a complete type system for composing objects and defining relationships between them. Above these layers, applications can build a wide range of traditional file systems, databases, and storage of structured multimedia objects, all supported by a common infrastructure [Bac91].

Zebra [Har93] stripes data across multiple storage servers in a network version of RAID [Pat88], but uses a single file manager to coordinate metadata. Zebra is based on the Sprite LFS implementation, and clients cache data and metadata the same as they would in Sprite. The file manager assumes the role of a standard file server, but all log operations are striped and clients direct data requests directly to the storage servers. Swarm [Har99] removes the file manager to present a stand-alone layer that exports striped logs to clients, which can be used to implement local file systems or other high-level services.

Petal [Lee95, Lee96] pools storage devices to export sparse virtual block devices to clients. Persistent state is distributed across the servers, and global state is updated using a distributed agreement protocol that tolerates node failures. By caching a small amount of metadata, clients can direct most requests directly to the correct storage server, updating metadata lazily when it proves to be out-of-date. Petal maintains a global map of servers for each virtual device, optionally using *chained declustering* for redundancy. Chained declustering [Hsi89] allows simple load redistribution when nodes fail and makes catastrophic failures more likely to damage a few virtual disks extensively than cripple many virtual disks with small corruptions.

Frangipani [The97] builds a file system over a Petal virtual disk. It takes advantage of the large, sparse address space to simplify data structures, with large regions reserved for inodes, allocation bitmaps, logs, small blocks, and large blocks. As with most layered systems, the physical layout is not determined by the virtual layout, so many of the layout strategies of local file systems are inapplicable [Ste05]. Instead, the layout of files is simple, with a map tracking up to sixteen 4 KB blocks for small files, spilling into one of 2^{24} large blocks that can store files up to 1 TB. File systems can be shared by multiple clients; a distributed lock manager coordinates caching and prevents corruption from concurrent access.

Object-based storage systems [Fac05, Mes03] provide an object-level interface to disk space. Instead of exporting virtual disks to clients, they manage storage as collections of objects with attributes, which generally correspond directly to files and directories in complete file systems. Network-attached secure disks (NASD) drop storage servers and embed management functions directly in the storage device to save costs [Gib97, Gib98a], while metadata is controlled by a separate server. They require a separate metadata server to manage security and coordination for shared storage, but cryptographic techniques make these metadata operations largely asynchronous. Synchronous communications are mainly confined to direct requests from clients to the storage devices.

FAB [Frø03, Sai04] refines the ideas found in Petal and introduces the idea of *storage bricks*—dedicated storage modules made from commodity components that can be combined to form a storage pool. FAB randomly assigns each data segment to a set of storage bricks and uses majority voting to manage replicas. This allows it to tolerate failures and add new bricks without pausing and temporarily ignore overloaded servers. It also supports fast snapshot operations without a centralised coordinator [Ji05].

The Self-* Storage project aims to extend brick-based storage to automate many aspects of configuration and management of storage in order to reduce administrative hassle and costs [Gan03]. Like the AutoRAID storage device [Wil95], the Ursa Minor prototype [AEM05] focuses on optimising data layout and failure tolerance for different workloads and requirements. Using a unified infrastructure it supports multiple storage policies and can convert existing data while still operating normally.

Storage layers are complementary to the goals of Envoy. Indeed, the Envoy prototype's primitive storage layer is a stub intended to be replaced by a system much like those described here. Envoy assumes a reliable and performant storage layer, and builds a complete distributed file system optimised for large numbers of untrusted virtual machines above it.

File systems

Built on storage layers that can deliver parallel access to many disks, cluster file systems are left mainly with problem of managing metadata and maintaining cache consistency. Lustre [Lus06] follows the model proposed by the NASD group [Gib98a], with clients contacting storage devices directly for object-level data access. Metadata is controlled by a centralised server with a failover replica.

The Google File System [Ghe03] focuses on a specific workload and drops the POSIX file system interface. It, too, splits metadata management from data storage, but is further optimised for large files with sequential reads and append-only writes. Random reads and writes are supported, but the overall emphasis is on high throughput rather than low latency. In this environment, data caching has little value and metadata is minimised by managing files as sequences of large, 64 MB chunks. Chunks are stored as normal files using a Linux file system on commodity hardware.

Clusters are widely used as replacements for supercomputers, and many cluster file systems are tuned to scientific-computing workloads. Like the Google environment, they often require high sustained throughput from large files. Unlike workstation environments where read-write sharing is rare, scientific computing makes frequent use of parallel processes writing to different parts of the same large file [Wan04]. GPFS [Sch02] is optimised for large deployments where any centralised coordination point is unacceptable. Based on a block-level storage system, it supports a high degree of concurrency by replacing the centralised metadata manager with a distributed lock manager with byte-range locking, and using extensible hashing to support large directories that can be queried and updated concurrently. In addition, it allows write sharing for non-conflicting metadata updates and pushes most of the communication burden for token revocation to the client triggering it, further reducing synchronous metadata operations in the lock manager.

Many commercial solutions use dedicated storage area networks (SAN) to connect hosts with storage devices over a high-speed switching fabric. As with most of the systems described here, SANs let clients connect directly to storage devices and rely on a separate layer to mediate sharing and form a file system. Storage Tank [Men03] uses manually-configured partitions of the namespace to distribute control between metadata servers and relies on the block-based interface provided by SANs, but the overall structure is similar to systems like GPFS.

Ceph [Wei06] is also tuned to scientific cluster workloads, but is based on object storage. It, too, distributes metadata management to avoid bottlenecks, but instead of using distributed locks and having clients directly modify metadata structures, it builds a distributed version of the metadata manager common in other object-based systems. Object replicas are placed according to a special hash function, allowing clients to compute the location of an object instead of having to consult the metadata manager. Ceph handles the problem of heavy write sharing in scientific workloads by augmenting the POSIX interface to allow weakened consistency semantics. This works for scientific computing, where applications are generally custom-written anyway, but is less helpful when clients use commodity tools. Like Envoy, Ceph divides management of the hierarchical namespace according to runtime activity, but it does so purely for load balancing within the distributed metadata service [Wei04]; no server benefits more than another from controlling a particular region of the namespace. Envoy, in contrast, delegates metadata management to the client dominating access to that part of the file system, making the placement of metadata management a matter of absolute performance as well as load distribution. It also differs from Ceph's distributed scheme by using a time-based protocol to prioritise changes and promote stability (see Section 4.4).

While Envoy is also built for clusters, it is designed for many independent clients with more traditional workloads, rather than large, scientific systems bringing the power of thousands of machines to bear on a single data set. Envoy creates a single hierarchical namespace, but it is arranged as an administrative tree with discrete, client-level file systems as leaves. Envoy is optimised to manage large numbers of these images, with full administration of a private image being managed by the machine using it, and control of shared images being distributed among the participating clients according to runtime demand. In this way, Envoy—with its model of a cluster being host to more clients than machines—is largely complementary to the systems described here, which attempt to make the cluster act like a single, large machine. Instead of centralising metadata management and then introducing distribution schemes to overcome the limitations of centralisation, Envoy partitions the namespace according to runtime demand and distributes metadata control to the clients, or more precisely to a secure virtual machine hosted on the same node as the client.

2.3.6 Virtual machines

A few storage systems have been designed specifically for virtual machine environments. Operating systems hosted on virtual machines can use conventional file systems implemented on private block devices. These can be physical partitions assigned to individual virtual machines, or virtual block devices accessed through a virtualized driver. In addition, network-facing client protocols such as NFS operate the same on virtual machines as on real machines.

My experience with two systems influenced the design of Envoy. The first was CoWNFS, a stacking file system that uses copy-on-write techniques and user-controlled layers to emulate snapshots and file system forks [Kot04b]. CoWNFS operates as a userspace NFS server running in an administrative virtual machine and exporting customised storage views to clients. A private, writable layer can be stacked over a read-only template to capture changes and isolate them from other users. Sharing is possible between clients through the NFS protocol, but access is limited to files already available in the namespace of the administrative VM.

The second system, Parallax [War05], exports private virtual block devices to clients. Like CoWNFS and Envoy, Parallax operates through a server in the administrative VM on each host. Template images with fork and snapshot support assist rapid deployment of clients, using a copy-on-write mechanism as clients diverge from their starting templates. Parallax operates at the block level with no support for sharing between clients, except when clients inherit read-only blocks from the same template. A distributed block-storage layer makes use of cheap commodity disks in the host machines for persistence and location-transparent access.

Ventana [Pfa06] is a file system for virtual machine environments with similar goals to Envoy. Like Parallax and CoWNFS, both systems provide a single server for each machine, which clients access through a virtual network or block device. Ventana also supports snapshots and forks of file system images, and tracks versioning at the per-file level. Like Envoy, it uses an object-based storage layer, but objects are immutable and changes are tracked through successive file versions, similar to JetFile [Grö99]. Ventana uses a centralised metadata server to track file versions (unlike JetFile, which announces new versions through IP multicast) and to manage image branches. It offers loose consistency, communicating with clients over the NFSv3 protocol and requiring nodes to check with the metadata server on each access to bound cache divergence. Persistent caching at each node permits disconnected operation for clients, and any resulting conflicts are managed by applying all client changes to the repository after making a snapshot. To lessen the bottleneck of a centralised metadata server, Ventana allows file system images to be marked as private and all metadata to be cached at the node. Envoy targets clusters of virtual machines where centralised servers are impractical. Metadata control is distributed, and manual configuration of private and shared images is rejected in favour of dynamic algorithms that adapt to runtime behaviour.

Virtual machines have also been used to support grid computing [Fig03]. GVFS [Zha04] extends NFS with userspace proxies that implement persistent caching, file prefetching, and per-file metadata hints, allowing compressed transfers and other deviations from standard NFS semantics to better support the wide-area grid architecture. Like other virtual machine storage systems, GVFS exploits the redundancy in file system images cloned from a template, using symbolic links to introduce some transparency in the cloning process and facilitate shared caching.

2.4 Summary

Commodity computing refers to two basic trends. In the first, commodity hardware is used in place of specialised devices for an increasing range of tasks, particularly at the high end, where clusters of workstations have replaced specialised supercomputers for many demanding applications. In the second, computation is treated as a commodity service rather than just as a product to be purchased and used. *Utility computing* refers to any metered computation services, and is also called *on-demand computing*.

Machine virtualization is a technique for multiplexing hardware for use by several *virtual machines*, each with its own operating system and tools. Originally used on mainframes decades ago, virtualization is now popular on commodity hardware, where the Xen virtual machine monitor and others like it allow even I/O intensive server tasks to be hosted with low overhead. Xen came from the XenoServers project, which uses it to build a public computing platform where anyone can buy or sell computing resources on an open market.

Local file systems gain performance through caching and by minimising disk head movements. Client-server file systems must coordinate client caching and address coherency in the presence of concurrent access by multiple clients. Control over data can be delegated to clients, or conflicting updates can be reconciled later. Serverless file systems have the same problems but lack a centralised server to coordinate, and must either assign server-like roles to participants or use agreement protocols to ensure consistency.

In the wide area, latency and reliability are major problems, particularly with donated resources. Some systems form localised clusters within the larger system to simplify communications and to reduce latency problems, or an explicit structure may be built into the network. Some peer-to-peer systems use overlay networks for lookup services and routing, while others form networks around individual objects or use multicasting to coordinate updates and locate objects.

Cluster file systems are often built with a separate storage layer, which provides a simple abstraction to the data pool. Object-based systems use objects that roughly correspond to files and directories, while others provide virtual disks. Clusters have vast aggregate resources, so redundancy and parallelism are used both to increase performance and to tolerate component failure, which is commonplace with large numbers of nodes. Moving centralised metadata control out of the critical path or replacing it with distributed locking or other distribution schemes helps prevent bottlenecks from choking highly-parallel data paths. Explicit support for virtual machines is only beginning to be developed in storage systems.

Chapter 3

Service Clusters

The standardisation of the shipping container revolutionised the logistics industry, which in turn has had a significant impact on global commerce over the past 50 years. Building a box hardly seems like the stuff of revolutions, but the cheap availability of foreign goods and easy access to global markets that characterises modern economies owes much to exactly that. The significance of the container is illustrated partly by what it offers, namely, flexibility and efficiency, but equally significant is what it does not offer. It is not an engine, a vehicle, a route, a company, a service, or any of the things necessary to transport goods from one place to another. Instead, it is a neutral, common ground. Those wishing to ship something can pack according to well-known dimensions using widely-available tools, and those offering transport services can use trains, ships, trucks, or whatever method of transport and whatever routing system allows them to offer competitive service while making a profit.

The computer industry is in need of a shipping container. While the computer hardware industry is increasingly viewed as a commodity business, computation as a commodity service is still in its infancy. The components are all there: PCs are powerful, networks are fast, disks are big, operating systems are flexible and efficient, and everything is cheap. The

Top 500 supercomputer list is dominated by clusters make from commodity hardware [Top06], and companies such as Google have used commodity hardware to solve large commercial problems instead of relying on special-purpose hardware that is more powerful and more reliable, but also much more expensive [Bar03b, Ghe03]. These efforts have been highly successful, but they still revolve around using commodity hardware to build platforms that are customised for a particular task or class of tasks. Like oil tankers and passenger trains, they are efficient and well-suited to their intended markets, but not easily adapted to other kinds of clients.

Containers have succeeded for several reasons. They are generic enough to support an enormous range of cargo, but rigid enough to pack tightly together and stack neatly. They can be pulled individually behind trucks, strung together on trains, or packed onto huge cargo ships for ocean transport. They can be moved and loaded easily and quickly using cranes, and they can move from one ship to the next without any changes. Clients can fill as many containers as they need and only pay according to what they use.

Many of these same characteristics would help to make commodity computation a reality. Clients should be able to deploy a wide range of computation and communication services, but vendors should also be able to manage them in a generic way. Clients should be able to use their own commodity machines and software to implement services, but have them run equally well in a large, commercial setting. Likewise, deployment costs should be low and procedures simple, and redeployment costs should not create onerous barriers to changing vendors. Finally, small services using very few resources and those spanning many dedicated machines should be able to coexist without interfering with each other and with clients paying according to what they use.

In this chapter I argue that a platform of *service clusters* approximates this ideal, using commodity hardware and commodity operating systems and tools, isolated and managed using virtual machine monitors. The *service* is the unit of management, allowing arbitrary tasks to coexist, each isolated in a virtual machine. While I do not prescribe a particular operating system or set of tools, I do suggest that vendors select a small set of common, commodity platforms to offer as templates, and that clients and vendors can both benefit from adhering to them. Clients can then package their custom services as everything that differs from a standard platform and send it to the vendor that can host it according to their needs.

Not every task is best served by a common platform—oil tankers will continue to excel at transporting oil—but many jobs that run on custom installations today could be better implemented as commodity computation tasks that are cheaper for clients to run and profitable for vendors to host. Clusters are well suited to large, parallel problems, and are generally built in response to a specific need; organisations develop cluster expertise because they have problems that demand it, and the specifications of the installation are driven by the budget and the demands of the application. This chapter argues for the reverse approach: build an efficient cluster architecture and make it suitable for a wide range of applications.

Finally, I do not propose a complete solution. Instead, I argue for the essential characteristics that distinguish service clusters from other large-scale uses of commodity computers, while leaving much of the general infrastructure to others. I focus on the storage needs of the platform and identify how a suitably designed file system can use the commodity hardware in a cluster to cheaply and easily support deployment and management of services built on commodity tools.

3.1 Commodity computation

Services are harder to package as commodities than goods. The quality of oil, the purity of precious metals, the strength of steel, and the composition of building materials can all be objectively measured. Compatibility with standards, quality of workmanship, energy consumption, and feature sets make consumer electronic devices comparable, and even food can be graded and compared, at least at the level of basic ingredients. Services can also be commodities, but only when competing offerings can be compared on price and quality, and when customers can move freely between providers without prohibitive lock-in effects.

3.1.1 Flexible commodity computation

Commodity computing describes a range of systems for exploiting the cheap and plentiful computing power available in commodity PCs. Instead of building expensive, speciality servers to tackle complex problems, commercial users and researchers are increasingly harnessing the power of many smaller, cheaper machines to achieve the same end. Because of the massive economies of scale in the PC market, the aggregate power that can be had from a group of cheap PCs is much greater than what the equivalent funds could purchase in more powerful, specialised server hardware.

Efforts to use commodity hardware for large computation tasks are orthogonal to the goal of treating computation power itself as a commodity. While clusters of commodity machines may be part of the underlying implementation of a commodity computation service, using specialised server hardware is also a viable option. When considering a service as a commodity, the methods used to offer the service are left to the provider, and innovation through proprietary techniques may prove a competitive advantage. From the customer's perspective, it is only the quality of the service rendered and the cost that matter.

Utility computing, also called *on-demand computing*, describes the business model of providing computing services and charging based on use of resources. This may apply to specialised services such as databases or web hosting platforms, or to any service where usage is metered and clients pay based on what they use rather than the capabilities that are available. By itself, utility computing answers only part of the problem. Customers are isolated from the fixed costs, risk of component failure, and administrative expenses associated with hardware ownership, but they may be restricted in the range of services offered or applications accommodated.

Just as early mainframes required customised software development and incurred porting costs with each new iteration or change of vendor, utility computing systems subject clients to the tools and infrastructure requirements of their hosts.

Flexible commodity computation is a form of utility computing that allows standard commodity operating systems and tools as well as customised services to be deployed quickly and cheaply, with the basic environment providing commodity operating systems and tools rather than a specialised platform. Deployment costs are proportional to how far a client deviates from a standard, commodity environment, e.g., a standard Linux distribution, rather than the total amount of software their application requires to operate. By providing standard tools and standard environments, one flexible commodity computation service can be swapped for another without significant barriers such as porting costs and deployment costs, making computation itself a fungible commodity that can be used for a wide range of tasks, large and small.

3.2 Service containers

The first step in commodity computation is packaging computation jobs in manageable units, without unnecessarily restricting what clients can do. Service containers must balance the conflicting goals of isolating clients from each other and preventing bad behaviour, and supporting the maximum range of legitimate client activities. This section argues for isolating services in virtual machines and encouraging clients to use commodity software.

3.2.1 Decoupling hardware and software

The provider of services and the trader of physical commodities resemble each other the most when the services of multiple providers are essentially interchangeable, which requires agreement about not only what is to be done, but what is being acted upon. Cars of the same make and model can be repaired by a wide range of mechanics. Shipping firms can offer to transport a container of a specific size and weight between two points for a specific cost. Before the container was standardised, loading and unloading procedures would vary based on what was being shipped and how well it packed next to the goods of other customers. This would in turn affect the cost structure and tie together two services that are more efficient when separated and optimised individually: loading and shipping.

Likewise, a web services platform may offer compelling services for its specific domains, allowing clients to host their web applications easier and cheaper than they could with their own hardware and software stack, but doing so would conflate two issues that could be better optimised individually: providing and managing the hardware resources, and managing the software infrastructure for web applications. The expertise required for these two parts of the problem may be quite different, and combining them forces clients to choose a package deal when they may be better served by independent choices. The skills of hardware managers may also be put to better use serving the needs of multiple software platforms at a larger scale, not just accommodating clients of a particular class of web services.

Any domain-specific middleware will necessarily be limited, and coupling the efficiency of a shared hardware infrastructure to a specific application domain will limit the economies of scale that could otherwise be achieved by more specialised providers. True commodity computation will divorce the application domain from the provision of a hosting platform, allowing specialists to excel in serving their respective functions. A platform that supports only a restricted domain of applications is offering a computation service, but it is not offering computation itself as its product. Attempting to port a service from a client's own machines to a hosted service provider to a competitor's platform may reveal how far each provider is from offering generic computation as a product.

The most flexible platform available to clients is wholly-owned and managed hardware. The PC has proved to be extremely adaptable and capable of hosting an enormous range of applications. Giving clients a commodity hosting platform that approximates the flexibility of a standard machine gives them access to familiar tools and maximum latitude in designing their applications, without requiring them to conform to a specific middleware structure or use custom programming interfaces. It also protects the client from being locked-in to a single vendor through dependence on proprietary software.

Partitioning physical machines using a virtual machine manager and giving clients access to entire virtual machines pushes the dividing line between vendor- and client-management as close as possible to the hardware itself. Constraints still remain to give vendors control of the hardware and the ability to manage security concerns, but machine virtualization currently represents the state of the art in minimally decoupling control of the hardware from the concerns of the software stack.

3.2.2 Decoupling unrelated services

As hardware gets more capable, individual hosts can accommodate multiple applications. Organisations that wish to make efficient use of hardware investments must measure or estimate the requirements of each application and map them to machines in a way that maximises the use of resources without overtaxing individual nodes. Managers are left with the choice to under-utilise hardware resources, explicitly address load balancing in the applications, or manually allocate resources and re-balance as necessary. Each has its costs and its advantages.

Modern virtualization managers like Xen have low enough overhead to justify partitioning a machine even absent concerns about security and control of the machine [Bar03a]. By putting each application in its own virtual machine, the issue of hardware allocation can be separated from the design and administration of the software itself. By assigning one application to one virtual machine, VMs and the services they contain can be migrated as individual management units in response to runtime demand, without the explicit cooperation of the application.

Decoupling unrelated services separates the problems of load balancing and maximising resource utilisation from the problems of software installation and deployment. Services contained in virtual machines become generic units that can be managed with generic tools, ignoring many of the intricacies of the actual software package. Isolating applications from each other is useful even when they are hosted on the same machine, as configurations may conflict. When vendors certify operating system platforms for software applications, they often stipulate that the application must have a dedicated environment to rule out untested interactions with other applications.

Isolating services in their own virtual machines also has the potential to increase security. While the same operating system and tool chain may be used, it can be stripped to include only those services and drivers necessary to support a single task. By being deployed with a minimal set of supporting software, a service can reduce its risk of being compromised by flaws in unneeded software packages.

The disadvantage of this deployment model is that extra resources are consumed. In addition to the application software, operating systems and supporting libraries must be part of each service container. Overhead that is shared in a traditional environment is duplicated in each VM when services are partitioned in this way. This is a cost, but not one without reward: it buys flexibility and the potential for automation and simplification of management. Tailoring the runtime environment to the specific task can reduce the memory and CPU overhead without significant re-engineering, and suitable storage strategies can reduce the storage redundancy that otherwise results from increasing the number of complete with operating system images to support the additional VMs.

3.2.3 Supporting commodity tools

Any suitably designed framework can separate the management and control of hardware from that of software; this is one of the basic functions of operating systems. Similarly, balancing the demands of applications against the capabilities of hardware is a recurring theme in system design. Achieving both of these aims while permitting the use of a wide range of standard, commodity tools and operating systems precludes custom frameworks, however. Virtualization and a discipline of packaging applications into minimal service containers brings these capabilities to existing applications without the expense of porting to a new software platform.

Using commodity software when appropriate brings many of the same advantages as commodity hardware. Commodity operating systems and tools are cheap, powerful, and under constant development, so features and improvements accrue over time. Just as using commodity hardware allows users to benefit from the scale and competition of a thriving market, relying on commodity software gives access to the benefits of industry-wide testing and development efforts driven by competition and a large, demanding user base.

An important characteristic of commodity software tools is that they are widely used, and the most popular can be easily identified. Even without a formal process, standards emerge over time and change slowly, both in proprietary and open source software communities. Computation providers can streamline deployment and reduce costs by offering a small set of file system images based around *de facto* standards, complete with an operating system and standard tools. Clients can then customise an image to support a specific service, and deploy it with little additional effort. The setup time needed, bandwidth consumed, and storage required to customise the image are related to the degree of customisation required, not to the overall complexity of the service. As supporting tools become more sophisticated and capable, and as the base of standard software evolves, more intricate services can be deployed without increasing the deployment costs.

The users best served by these base images are those whose needs are met entirely by commodity software. Deploying a DNS server or a web server requires little more than configuration and some content, all of which can be transferred using standard tools on a private virtual machine. Offering standard base images as an option does not restrict clients to what is provided, however. The architecture favours the use of standard tools, but it does not prevent users from starting from scratch. As is true in many product domains, departing from the standard is discouraged only by the higher cost. As is also true in many kinds of product fabrication, making a custom image incurs some one-time costs; using that image as the base for many service instances makes it possible to amortise that cost.

3.3 Service clusters

One of the enduring goals of systems research is to provide a good platform for running applications. Even early systems such as Multics were explicitly intended as infrastructure for higher-level computing services, seeking "continuous operation in a utility-like manner, with flexible remote access," with requirements such as "convenience of manipulating data and program files, ease of controlling processes during execution and above all, protection of private files and isolation of independent processes" [Cor65]. Four decades has seen much progress, but similar goals are still applicable.

The first part of the problem of commoditising computation is packaging tasks into manageable units. As argued above, service containers are a flexible way to isolate applications from the machines that host them and from unrelated tasks with which they may share hardware. While service containers can be deployed on individual machines, networks of workstations, or other groups of hardware, it is in cluster environments that they find their natural home. *Service clusters* are clusters of commodity machines managed centrally to support the deployment of arbitrary service containers, either as isolated instances or as groups of interacting services.

3.3.1 Economies of scale

The most obvious benefits of hardware clusters are related to scale. Quantity purchases generally lead to better prices, and dedicated facilities can be streamlined for a single purpose to eliminate waste, e.g., temperature control, lighting, and physical layout can be optimised entirely for the hosting machines, rather than for a mixed environment of machines and people. Fixed costs can be amortised over large numbers of nodes, and running costs can be negotiated for bulk quantities.

Scale also makes it possible to devote resources to system design that would be impractical for smaller deployments. Staff can devote all their time to managing the lower levels of the computation stack—from hardware up to the virtual machine—and to optimising the platform without specific applications in mind. Facilities can also be located away from client buildings to take advantage of favourable business environments, high-speed internet connections, and available staff. Scale and access to a wide range of clients also increases the potential rewards for improving efficiency and service quality.

Hardware can be added to clusters incrementally, which eliminates the need for an accurate forecast of the lifetime demands of the system. In addition, clusters of machines can achieve much greater overall scale than even the largest single machines. Scalability over time and absolute scalability are also features of non-clustered distributed systems, but those lack the high-speed interconnects and coordinated administration possible in managed environments.

The use of commodity hardware also allows rapid machine acquisition and cheap prices. Incremental scaling means that newly added hardware can always take advantage of the best price/performance ratios and benefit from the constant downward pressure on prices in a competitive market [Fox97]. Commodity disks are relatively unreliable, but they are also large and cheap and offer a good source for storage capacity [Pat88, War05] that comes standard with most machines.

The independence of nodes in a cluster offers redundancy that can be exploited for both reliability and availability. This is necessary not only to exceed the expectations of a single system, but to match it as well. Having many parts that can fail independently offers a much higher probability that at least one will fail than that any single component will fail, and a cluster that does not tolerate some component failures will quickly become unusable [Bir93].

3.3.2 Heterogeneous workloads

Over-subscribing capacity is a common practice for businesses that offer a fixed level of service, but expect some users to use only part of what is offered. Airlines overbook flights with the expectation that some passengers will forfeit their places, allowing the airline to capture revenue based on the promised service, not the service delivered. Some web hosting services over-subscribe their hardware capacity, a practice that allows them to provision based on expected average use rather than maximum potential use.

Since clusters can be expanded incrementally, observed average use can become the metric for established providers, especially at large scale. It may be prudent to hold some reserve capacity to handle bursts in usage, but even burstiness becomes more predictable at large scales. The more diverse the services using a cluster, the less likely an external event will trigger a burst in usage that overwhelms the capacity of the entire cluster. Heterogeneity and varied workloads may be difficult to manage at a small scale because they are difficult to anticipate and plan for, but at larger scales their uncorrelated fluctuations become a benefit.

Random variations in activity must be accommodated, but periodic fluctuations in demand can sometimes be planned for and exploited to make better use of resources. Predictable events like business hours, holidays, academic calendars, and other stable cycles can significantly influence some service workloads. Planning for these by pairing complementary services in a dynamically configured cluster can help minimise idle resources and reduce expenses. Systems used heavily during business hours could share with systems used by game servers, assuming that the latter are used more during leisure hours than on company time.

3.3.3 Central management

Clustering groups of machines together enhances scalability and resilience to failed components compared to a single system, but it does not simplify application design. New failure modes, networked interconnects, the lack of shared busses, and the lack of shared memory and processor control all change the way systems must be designed. The simplicity of a single system is lost in a cluster, but some of its features can be retained or at least emulated. Clusters, unlike wide-area distributed systems, are generally physically close to each other and managed under a single administrative domain. Physical security and the level of trust placed in each node is increased as a result.

Centralised administration and high-speed communication (via shared memory and inter-process communication) are two advantages of traditional servers. Clusters typically put components on a single site with high-speed local networking and a secure physical location. The machines are all owned and administered by a single organisation and can be built with appropriate cooling systems and redundant power supplies and network links. Systems designed to prevent or avoid any centralised control usually do so for privacy or legal reasons, neither of which is compelling in the service cluster environment. On the contrary, some degree of central administration is an essential feature of service clusters. The owner of the cluster needs to control access and admission to the service pool as well as monitor the services that are running to ensure that they do not exceed their alloted resources. Given physical proximity and central control, central administration adds convenience without introducing unnecessary penalties in flexibility.

To further emulate the desirable characteristics of centralised servers, clusters must ensure global access to data in the storage system from any constituent node. This is best achieved through a single, global name space with a completely coherent view of all files in the cluster. Trading coherence for performance represents a failure of global access, as concurrent services effectively create private views of data, requiring a separate mechanism for restoring the consistency that the file system violated [Bir93]. While partial coherence is sufficient for some applications, it exposes differences between local and distributed systems and weakens the guarantees that the file system offers. This requires planning for all system designers, even those that ultimately determine that the weaker guarantee can be safely ignored [Wal94].

While an important goal of virtualization is to isolate services on the same machine from each other, they do still share hardware. If the hardware fails, or if the virtual machine manager crashes, all of the services hosted on that machine will also fail. The rest of the cluster can continue operating, but other nodes that were cooperating with the failed machine may still be affected. To minimise that impact, server functions can be located on the same node as the clients that use them. This is the principle of *fate sharing*, where the loss of server processes that fail when a machine goes down mainly affects clients that also went down with the machine.

Service clusters bring together a wide range of clients under central management, making it possible to monitor and model the behaviour of unrelated services and plan resource allocation with more information than a single client could provide. Unrelated clients may have complementary resource requirements, e.g., one demands many CPU cycles and another much memory, which a cluster manager can detect and exploit in mapping services to physical nodes. This applies to characteristics observed over time as well, such as cycles of demand driven by external factors like the time of day or day of the week. Putting a wide range of services from a wide range of clients together in a single cluster allows administrators to make decisions informed by pertinent, observed data, coaxing out optimisations that would not be available to clients acting on their own.

3.3.4 Supporting a software ecosystem

Service clusters have the ambitious goal of replacing the machine room with hired resources. To achieve this they must support a similar range of activities and offer tangible benefits compared to privately owned and managed hardware. As a first step, commodity computation functions as a direct replacement for owned and managed hardware resources, supported by maximum flexibility in the environment and access to a full set of standard and customised tools with minimum overhead.

Packaging computation tasks as service containers yields advantages for administration that could be equally useful in private machine rooms. Isolating applications in fine-grained protection domains and maximising resource utilisation through allocation and migration at the service level benefits hardware owners and application writers alike. A resource for hire with the same level of flexibility and control offers new possibilities for service providers that narrower service offerings do not.

With low-level services available, third parties can offer services appropriate for end users. Instead of offering packaged applications, sending consultants to assist with deployment, or hosting software themselves on their own hardware, software vendors can sell their services on a service cluster platform. Web hosting, group-ware, email hosting, payroll services, etc., all exist currently as managed services, coupling the software services with the hardware services. If desired, clients can separately negotiate each aspect of a hosted service—the hardware hosting and the software management—and retain greater control over their own data.

In addition to end-user services, a service cluster economy could support an entire ecosystem of intermediate services. While standard software installations as base images form an important part of flexible commodity computation, they will not be appropriate for everyone. Some clients may wish to purchase database services, running on the same service cluster as the client's application but managed by a third party. Scalable and widely-distributed web hosting middleware could be offered by a vendor that buys hardware resources as needed from a range of service clusters, then sells simple packages to individual clients. Expertise in building distributed services and managing complex software has value that need not be coupled with hardware management.

In an ecosystem of hosted software, service clusters are the base environment upon which other services build. Clients may require only a single service container or they may hire a large amount of capacity, either to fill their own needs or to export their higher-level services to other users. To support these different usage patterns, service clusters must address the needs of shared groups of services as well as services in isolation.

3.4 Storage for service clusters

The design and intended applications of service clusters put specific demands on the storage system. While many of the storage requirements have been explored individually in other settings, the combination is unique, and existing systems only partially address the needs of this environment. This section discusses those requirements, and how they influence the design of Envoy, a storage system for service clusters.

Service clusters are a flexible platform for implementing arbitrary services, and the storage system that supports them must be similarly flexible. While the design must not impose unnecessary restrictions on what clients are allowed to do, it can draw on expectations about how they will behave to guide optimisations. The high-level goal is to predict and optimise for the most common client demands, while degrading gracefully as they stray from expected patterns. In addition, some specific requirements are derived directly from the needs of the environment.

3.4.1 Running in a cluster

Service clusters are centrally managed clusters of physical machines, each of which hosts any number of client services on private VMs. One of the chief advantages of clustering independent services is that resources can be alloted based on average requirements even though many individual clients will be decidedly non-average. As actual demands are observed, the manager can periodically migrate running services to new hosts and redistribute the load to even out the demands on a single machine. To facilitate transparent migration, the storage system must not tie images to a single machine. Even private images used only by a single service must be location independent within the cluster, or capable of moving without forcing a restart or excessive disruption.

Incremental scaling is another important aspect of cluster environments. Using commodity hardware gives owners access to the best price to performance ratios, and building the cluster gradually in response to demand allows them to track the desired part of the curve as it evolves over time. The storage system must be capable of accepting new hosts without undue disruption to those already running. Unlike peer-to-peer systems [Bla03] or networks of workstations [And95a], clusters are under central control and are used for a single purpose, so the addition and loss of machines is an exceptional event rather than the norm. Such events must be accommodated, but they need not be optimised for.

Another issue related to scale is that crash recovery must be localised. Transparent failover is not necessary, and recovering gracefully from failures rather than completely masking them [Bak94] is acceptable in this environment (note that hardware may fail as well; services that require specific reliability guarantees must implement redundancy at a higher level). Clusters are expected to run in machine rooms with well-provisioned machines that are properly managed, so failures are not expected to be frequent, but they are clusters of commodity hardware and failures will occur regularly for large installations. Confining the effect of a crash to a small area minimises the damage, and restricting the disruption to participants with overlapping interests (sharing files, cache, etc.) is even better.

Besides imposing specific requirements, a cluster of commodity machines provides useful resources for designing a storage system. Commodity disks are big and cheap, and having them distributed throughout the cluster provides a natural source of distributed storage capacity. Machines may reserve some capacity for administrative use, for booting the machine, or for providing swap space for clients, but most is available for use by the storage system.

These disks are not of premium quality, but can be expected to perform reasonably well. A disk can be no more available than the machine to which it is connected. Other hardware faults besides disk failures can disable a node, so the storage system must provide redundancy across machines to tolerate failures even with reliable storage. Since that cannot be avoided, combining disks into a RAID within a single node would only significantly increase the reliability of only that node, not the system as a whole. Envoy assumes that disks are independent, and that the failure of a disk will disqualify the entire machine until it is fixed or replaced.

Commodity software is even more important than commodity hardware in supporting flexible commodity computation. By being presented with a choice of standard platforms and being charged according to how much they deviate from those basic starting points, most clients will be expected to draw heavily from common file system images. This can be considered a basic property of service clusters and the storage system must be designed to exploit it. Independent virtual machines introduce another scalar factor in scalability demands that could otherwise lead to excessive capacity requirements [War05]. Encouraging duplication in file system images can be a benefit for the file system, however, because it allows Envoy to overlap caching even with unrelated clients.

3.4.2 Supporting heterogeneous clients

There is an inherent tension in service clusters between the goal of supporting the widest range of client applications possible on the one hand, while isolating them from each other and preventing unauthorised activities on the other. To address the latter concern, the storage system must be resilient to arbitrary client behaviour or misbehaviour. Implementing the file system as a cooperative service run by the client would expose it to Byzantine failures and considerable additional complexity. Instead, Envoy exploits the virtual machine architecture to isolate the cluster-wide file system management from client code, just as individual clients are isolated from each other.

Combining file system functions at the physical machine level is not an arbitrary choice nor merely an artifact of using a virtual machine container. The host machine represents a new layer in the storage hierarchy that would otherwise be present in a cluster. Just as an operating system can aggregate the requests of unrelated processes, a file system manager can bring together the activity of all virtual machines hosted on the node, which may have no direct relationship or even awareness of each other. The failure of a client need not interfere with the continued operation of other clients or other machine-level nodes, and the vocabulary of the client is restricted to the protocol with which it connects to the file service, minimising the damage that a misbehaving client can inflict on the rest of the system. The approach described here is not a requirement of the environment, but the security and performance characteristics enabled can be considered minimum requirements for the intended use.

Containing and restricting clients is only one side of the struggle in service cluster design. The other part is providing services with as much flexibility as possible. For the storage system, this means that clients must have as much control over file access as possible. Specifically, client operating systems must be able to both create and override their own file access restrictions. For private file system images, turning over complete control over access to clients would be sufficient, but arbitrating shared access to images by multiple clients requires a more hands-on approach. To support truly flexible scenarios, Envoy's security model must accommodate both.

Clients also vary in their longevity. Short-lived tasks will only thrive as part of the commodity computation ecosystem if they are cheap enough and can be deployed quickly enough to be competitive with the end-user owned and managed equivalents. Services that are tied to human activity may need to accommodate a human's impatience and short attention span. Interactive services or those that respond to other external conditions may need to expand over multiple instances in response to changing conditions, only to shut down excessive instances when a spike in demand subsides. These scenarios require a lightweight deployment mechanism that can rapidly produce not only standard base images, but also forked copies of custom file system images produced by clients. Forking running virtual machines is beyond the scope of this work, but file system support for the process is essential and relevant.

Long-lived clients have their own requirements, which must also be addressed in the storage system. Reliability can be considered a basic requirement of general-purpose storage systems, but additional support for backups and historical snapshots is also crucial especially for long-lived tasks. Runtime snapshots give a stable version of a changing file system image that can be backed up offline, used for reverting changes, examined for debugging purposes, or analysed for forensic purposes after a service has been compromised [Kin03, Whi04]. Only clients can determine the right tradeoff between the extra storage costs invoked by snapshots (which retain files that would otherwise be deleted) and the convenience of a detailed history, so Envoy must allow each client to dictate its own snapshot policy.

Envoy must be able to provide a private, bootable image for each service that launches. Platforms for embarrassingly parallel problems and other homogeneous service platforms can install basic software at each node and rely on a shared file system only for shared data. The profile of service clusters admits this possibility for starting the virtual machine manager and other administrative software, but individual services cannot be tied to a specific machine. Even the possibility of supplying a range of standard base images on each node and using a stacking file system to export virtual private images falls apart as clients fork highly-customised images, leading to inconsistency and excessive management complexity. A better solution is to require that the storage system be able to supply globally accessible images and location transparency, but do so in a way that accommodates the common case (many private images used by a individual services that migrate infrequently) with good performance. The convenience of global accessibility and transparent mobility must not cost much when it is not used much.

Service clusters are a platform for flexible commodity computation, which may take the form of distributed services as well as self-contained computation processes. Private images may be the most common case, but transparent file sharing is just as necessary for larger services. This leads to the problem of controlling shared access as mentioned above, and also to the necessity of managing concurrent file access. Past studies have concluded that runtime contention is quite rare [Kis91, Wel91], but the potential is always present in a shared file system. Envoy aims for perfectly consistent file sharing, i.e., any set of concurrent reads and writes from multiple clients appears to follow a logical sequence, and it does so with the further requirement that scaling of shared images be limited only by the extent of overlapping access. Subsets of a shared space that are used by only a single client should perform like private images, again supporting a range of access patterns but optimising for the most common case.

3.4.3 Local impact

In normal clusters, the throughput of the whole system is the paramount concern. In service clusters, good aggregate performance is still important, but the cluster owner is no longer the primary client, and clients are concerned primarily with the performance of their individual services.

Envoy is designed to encourage *local impact*, meaning that the resources consumed directly or indirectly by a service should be as close to that service as possible. If not in the same VM, then on the same machine, or on another machine that has some specific reason to yield its resources to a remote service.

By extension, this principle leads to a topology that is shaped according to runtime conflicts. When there is no reason to suspect contention, machines will prefer to assume complete control over the storage in active use by their client services. If two machines must explicitly coordinate their access to storage, they are treading on overlapping or neighbouring storage and implicitly declaring that a conflict is likely to occur.

3.5 Summary

Flexible commodity computation is a form of utility computing based around commodity tools and standards, allowing fast and cheap service deployment. Packaging services in virtual machines helps with commoditisation by abstracting the hardware interface, encouraging skill specialisation, and allowing services to be managed without explicit client cooperation. Dividing individual services into individual *service containers* allows finer-grained management and isolates unrelated services from each other. Using standard tool sets and operating system distributions lets services be defined as a set of changes to a base configuration, making deployment cheaper and making it easier to move services from one provider to another.

Service clusters use clusters of commodity machines to manage service containers as a commodity computation platform. Clusters bring economies of scale in hardware purchasing and management, and amplify the benefits of innovative administration and specialised skills. By pooling unrelated services, cluster managers can benefit from heterogeneous workloads to smooth resource demands and maximise the use of available hardware. As a basic computing platform, a service cluster can host an ecosystem of service providers.

The storage requirements of service clusters differ from other clusters. Clients are not trusted, and each one typically needs a private boot image as well as shared storage. The storage system must handle private data quickly and provide consistent caching for shared data. Localising impact increases scalability by reducing contention for shared resource, and improves performance by reducing latency.

Chapter 4

The Envoy File System

This chapter outlines the design of Envoy, a file system to support flexible commodity computing in service clusters. This chapter focuses on the architecture and general features, along with the main algorithms and administrative considerations. Details about protocols and implementation decisions are reserved for the next chapter, which discusses the prototype implementation.

The chapter starts with a discussion of the environment, including the expectations placed on the hardware and tools outside the scope of this dissertation. It then discusses the architecture of the entire system and how individual operations are supported by it. A discussion of how private and shared images are presented to and managed by clients is followed by discussion of how they are managed internally to optimise the overall system. The chapter concludes with a discussion of failure and recovery concerns.

4.1 Assumptions

Envoy is designed for service clusters, with the needs of flexibly commodity computation informing its assumptions about security and client demands.

Service clusters are assumed to have well-provisioned networks. High-speed local-area networking makes communication between nodes in the cluster cheap and fast, and switched interconnects make communication between pairs of nodes possible without significantly affecting the rest of the cluster. Redundant connections reduce the impact of failures, so network partitions—while still possible—are expected to be rare.

Nodes are also assumed to fail independently. Clusters are expected to run in machine rooms with redundant power sources and ample cooling. While commodity hardware is prone to failure, well-managed hardware can still be reliable and well-engineered clusters can isolate failing nodes, preventing groups of machines from failing together.

Service clusters run jobs for untrusted clients, but the environment itself is assumed to be trustworthy. Virtual machine managers isolate clients and prevent many malicious forms of behaviour. In particular, the network is assumed to be secure, with address spoofing and packet sniffing by clients prevented by the virtual machine managers and related systems. In addition, Envoy's own services can run in a secure environment, isolated from untrusted clients.

This work also assumes that other aspects of service-cluster management are provided by suitable solutions. Procedures for billing, managing client sessions, balancing load, allocating resources, etc., are omitted from this dissertation. It is further assumed that Envoy can cooperate with management software when necessary, though specific details are ignored. Envoy provides mechanisms to support client file system image management, but it does not prescribe procedures or management practices, as these must necessarily depend on other aspects of the system in addition to the storage system.

4.2 Architecture

Services access Envoy using a client-server file system interface. Each physical machine in the service cluster runs an administrative virtual machine

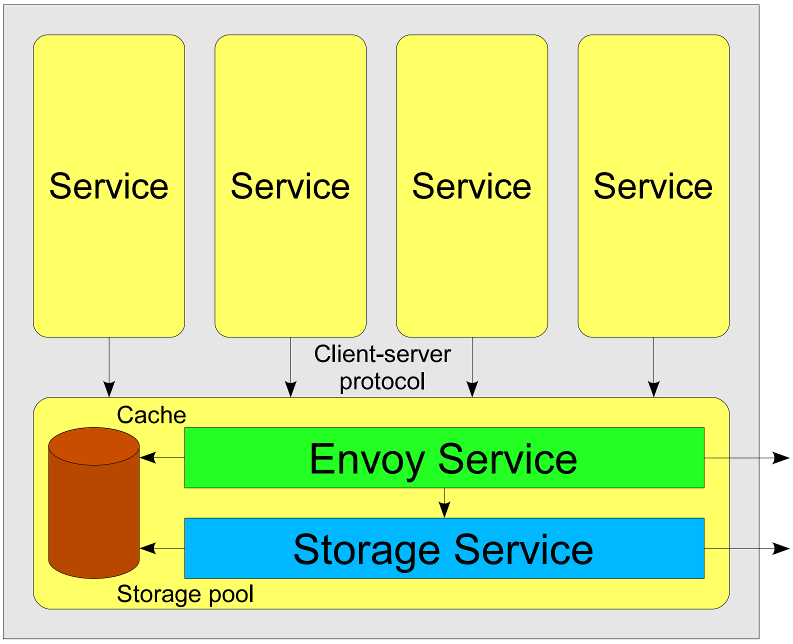

Figure 4.1: Each physical machine has a single administrative VM that hosts the Envoy services. This VM exports a network file system protocol to other service VMs running on the same machine.

that manages the storage processes for all services on that machine. These processes partition the local disk between a contribution to the shared storage pool and a local persistent cache as well as provide a standard interface allowing clients on the machine to access the file system. Figure 4.1 illustrates a typical machine. Client access to the system is restricted to a simple, well-understood client-server protocol, and a trusted server process acts as a proxy to the complete system [Sha86]. Byzantine failure from clients is less of a problem, because they have a limited interface to the system and hold no trusted data.

The file system management processes join a cluster-wide service that is comprised of two primary layers, as illustrated in Figure 4.2. Storage is managed by the lower level, which allows a small set of basic file operations on objects. The storage interface is stateless and the storage service makes no attempt to prevent or manage concurrent requests or to enforce any kind of security policy. Objects are extents of bytes with a small set of attributes.

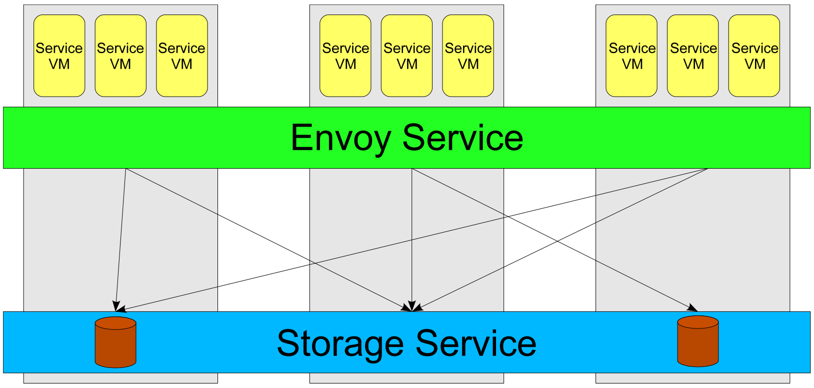

Figure 4.2: The envoy service coordinates access to provide a single, coherent view of the distributed file system. It relies on the storage service, which provides a repository of objects referenced by unique identifiers.

On top of the storage service is the envoy layer, which builds a hierarchical file system out of objects, coordinates access to files, provides authentication and access control services, manages caching, and exports a standard network file system interface for services to access.

In the remainder of this section I detail the functionality and requirements of these systems and consider the tradeoffs of various design decisions.

4.2.1 Distribution

Figure 4.3 depicts a commonly-used storage arrangement using a series of dedicated file servers to handle the needs of many clients. This architecture is successfully used in many settings, and, despite alternatives developed over the years, is still the dominant storage model in practical use.

The client-server model has obvious flaws when applied to clusters with many transient clients. Data placement decisions must balance space requirements and expected access rates in order to avoid overloading a particular server. Rebalancing—a disruptive and time consuming job—may be necessary in response to added clients, added servers, added disks, variations in client workload, and accumulation of data over time.

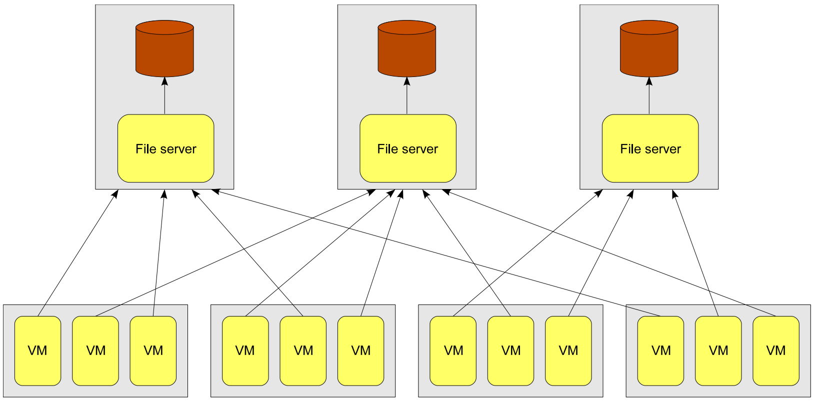

Figure 4.3: A popular storage solution for groups of clients involves a series of dedicated servers. Content on the servers is carefully managed to distribute storage demand and transaction load between the servers.

With the service cluster model, the problem is made even worse. To make efficient use of increasingly powerful hardware, each physical machine may host many services, each of which requires a boot image as well as access to the data relevant to its intended task. Dividing each machine means that there may be an order of magnitude more virtual machines than physical machines, putting excessive demands on a centralised storage infrastructure [Hos04]. Single-purpose services may also be short lived and demand may vary by time of day, making manual balancing impractical.

The client-server model has not endured as long as it has simply for lack of alternatives, however. It has many strengths that can inform the design of a more distributed architecture. With a single server managing shared data, concurrent access can be managed simply through explicit leases and cache invalidation, centralised caching with synchronous access, or through a lock manager. Whatever the mechanism for resolving conflicts, a centralised server is ideally suited to detecting and responding to concurrent requests because it is the point on the access graph at which all requests converge. The consistency of data that has reached the server is as good as its backing store.

The simplicity of a server is also a virtue. A failstop model for reliability can generally be assumed, backups are relatively straightforward, the server is typically dedicated to a single task or is shared with other trusted services, and the semantics are simple to define in terms of client behaviour[1].

The chief faults of a centralised system are the introduction of a single point of failure and the inability to scale beyond the network and disk bandwidth that can be hosted by a single server. While these limits are unacceptable at large scales, they are quite serviceable for small groups of clients.

For clients that are sharing data, it is difficult to improve on a centralised server. For sufficiently overlapping data sets, any consistent model will degrade to something resembling a server during periods of contention because all interested clients will have to synchronise their access to the contended bits. A single arbitrator will ultimately oversee each bit of data, whether it is a traditional server, a lease-holding client, or a quorum of co-operating peers.

Sharing cache space is also a benefit of consolidated control of shared data. As the performance gap between disk access and memory access continues to grow, efficient use of available cache space becomes increasingly important. Once again, a single centralised cache fails the scalability requirement, but a shared cache for a smaller set of clients with overlapping data needs provides attractive properties. Accessing a cache across a high-speed network can be faster than accessing a local disk [Dah94a], so in a cluster setting, organising data to maximise the aggregate cache is more valuable than organising it to localise access [Fra92]. Stated another way, the effective cache size of the combined cluster is greater when redundant entries are consolidated, and maximising the combined cache size to avoid disk seek penalties is becoming more important than avoiding the network hops that arise from using a shared cache.

While a server cannot handle an unlimited number of clients, it can serve many clients under typical workloads. In the case of overlapping

[1] NFS versions 2 and 3 have notoriously complicated consistency semantics, but this is almost entirely due to client policies. NFS server semantics are straightforward.

requests from different clients, a shared cache on a shared server can outperform a series of unshared servers that each must retrieve the same data from disk, despite the overhead of network latency. Envoy is designed to move file management to the client's machine when there are no apparent conflicts, but to pick one participant to control files that are shared and act as a server to the others.

This principle of localising control where possible, but reverting to a simple, well-understood client-server model when sharing is necessary is fundamental to Envoy. It leads to fate sharing among clients with overlapping interests in the areas of performance, resource usage, and failure recovery. In each case, services with overlapping resource demands cooperate directly with each other and disinterested parties are not involved.

The overall distributed architecture of Envoy is based around partitioning control of the file system to put data and metadata management as close to interested clients as possible. Entire file system images or parts of images that are used exclusively by a single service (or a group of services hosted on a single physical machine) are managed directly by the envoy service on the same machine. Where sharing occurs, the client with the highest demand retains direct control and acts as a proxy for other clients accessing the same storage, thus sharing a single cache and avoiding complicated coordination protocols.

4.2.2 Storage layer

The objective of the storage layer is to provide a simple, stateless interface for accessing objects. The storage layer provides redundancy to enhance availability and reliability, and distributes objects to balance the load on individual servers. Like many of the cluster file systems described in Section 2.3.5, Envoy uses an object-based interface to storage [Fac05]. Disk layout policies are left to individual storage nodes, and replica placement is left to the storage system.

Envoy puts a memory cache and a persistent disk cache between the object storage layer and the file system layer. File system traces show that with large caches, the operations that go to storage are dominated by writes and non-sequential operations [Rue93], and traditional layout optimisation in layered systems is of questionable value [Ste05], so the envoy layer does not take an active role in object placement decisions.

Numerous strategies are available for distributing objects across the cluster, including random distribution, chained declustering [Hsi89], partitioning based on object ID ranges, collocating objects created together, etc. These can be managed through a separate service [Gib98a], by storing maps on the servers and caching lookup tables on the client nodes [Lee96], or by using a mapping function that allows clients to compute the appropriate storage server based on the object ID and other metadata [Wei06].

The contributions of this dissertation are at the file system level, not at the object storage level, so the details of object storage are omitted here and readers are referred to the relevant literature as discussed in Section 2.3.5.

4.2.3 Envoy layer

The envoy layer forms a file system from the objects provided by the storage layer, coordinating and caching access to the file hierarchy, and exporting a client-server protocol to client services. The entire cluster shares a global, hierarchical namespace, but clients typically mount a subtree from the hierarchy and treat it as a complete file system.

Territories

A single instance of the envoy service runs on each physical machine, and the global name hierarchy is divided among participating instances in the cluster. When a given instance takes responsibility for some part of the

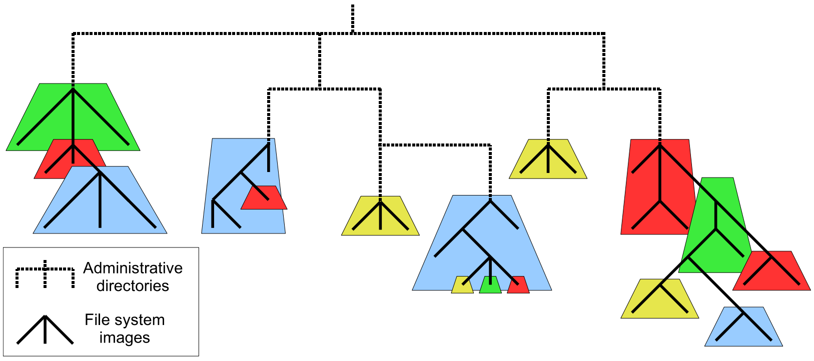

Figure 4.4: Envoy's namespace is hierarchical, and participating machines claim branches of the tree as *territories* to be managed locally. Branches within a territory can be split off into new territories to be managed by a other machines. The overall system structure is a federation of territories.

namespace, it is said to *own* a *territory* covering the relevant subtree of the hierarchy, as illustrated in Figure 4.4. All operations within local territories are handled locally. Storage objects may be cached locally both in memory and in the persistent cache, which is used exclusively for territories local to the machine.

This partitioning of the global namespace and the resulting federation of constituent parts is what gives Envoy its name. When clients request operations that stray from local territories, the requests are handed off to the envoy on the appropriate machine. It follows that each instance must know not only the boundaries of its own territories, but how to find the envoy for neighbouring territories, i.e., those that can be reached by a single directory traversal (up or down) from a local territory.

The envoy service is stateful, and tracks not only its territories and neighbours, but the state of all files and directories in use by its client services. When a client navigates beyond the boundaries of a local territory, requests are forwarded directly to the envoy that owns the neighbouring territory. If further navigation moves beyond the boundaries of the neighbour's territories, the neighbour does not forward it to the next envoy, but instead bounces the request back to the originator with the address of the envoy that can answer the request.

Under this system, two envoys maintain a direct relationship with each other only when they are immediate neighbours in territory ownership, or when one is serving requests for a client of the other. It follows that if territories are alloted such that the owner of a territory is also its most active user, traffic on an envoy instance will generally be dominated by its local clients.

Often, the best that can be achieved in a steady-state system is to have the owner of a territory be the envoy driving a plurality of traffic, not a majority. Sometimes this is an inevitable consequence of overlapping client demands, but often some further gerrymandering of the territory boundaries can improve access locality. Since the needs of the clients and the needs of the envoys are generally aligned, a practice that in politics usually serves those in power at the expense of those they represent serves both equally well in file systems.

Files

Files and directories can be mapped easily to objects as provided by the storage layer. Files are stored as objects with a set of attributes, and directories as files with special semantics and a different interface. Special files are stored as normal objects with special contents, accessible through the interfaces appropriate to the file types.

Unix file systems are organised around *inodes*, which organise the contents of a file and its attributes, but not its name. Envoy employs a similar structure, with file contents and attributes separate from the name hierarchy. Objects have numeric identifiers like inodes, but a file can be backed by different objects during its lifetime, so an object ID is not suitable for directly identifying a file.

Directories are files containing listings of other files. In Envoy, directories are managed at the block level, with each block containing some number of entries. An entry consists of a file name, the object ID that links to its contents and attributes, and a flag indicating the file's copy-on-write status. When this flag is set, the object is considered read-only, and will be cloned before any changes are committed to the file's contents or attributes. This process is completely opaque to clients.

Special files, such as device nodes and symbolic links, are stored as regular files whose contents follow a defined format. For symbolic links, the file contents are the target of the link, for devices they are an ASCII string identifying the major and minor device numbers, etc.

4.2.4 Caching

The cache in a distributed file system must balance performance with consistency and durability. While maintaining a coherent view of the file system that is tolerant to software and hardware faults, a cache should reduce latency, improve throughput, and increase overall capacity. The latter is achieved by reducing network and disk congestion and freeing input/output channels to absorb additional load.

Enhancing performance is the primary reason for employing a cache in the first place. The growing gap in performance between main memory and disk makes effective cache management critical, as a random disk access is five or six orders of magnitude slower than a similar memory access. The service cluster environment does not provide much insight into the expected access patterns of its constituent services, as one of the purposes of the environment is to support arbitrary clients. While application-specific cache hints are not available to aid in cache replacement decisions, service clusters do imply many file system images with overlapping content. Many images are private to a single client, while others may be accessed by multiple concurrent services.

In a 2002 interview, Eric Schmidt of Google observed that for seek-intensive workloads, DRAM can be cheaper to deploy than disks [Spr02]. The seek time of a single disk cannot be improved significantly, so increasing disk performance requires adding redundant spindles. With many mirrored disks, many seeks can proceed in parallel and a random request can be satisfied fastest by the disk whose head position happens to be nearest the requested datum. Because the of the large performance gap, Google found it cheaper and faster to store their entire web search index in DRAM, which can serve many requests quickly, than to create enough replicated disks to handle the same transaction load.

While Google's implementation revolved around an inherently parallel task [Bar03b], it can still inform the design of a cache solution for Envoy. A single, commodity machine cannot hold as much memory as would be required for an index of the web, but by considering the aggregate capacity of a cluster instead of focusing on the capabilities of a single machine, they arrived at the surprising but sensible conclusion that "it costs less money and it is more efficient to use DRAM as storage as opposed to hard disks". Finding data in a local cache is ideal, but with a high-speed network connecting machines in a cluster, it is faster to query the cache of another machine than a locally-attached disk [Dah94a], suggesting that it would also be prudent for the design of Envoy to rely on the combined cache of the cluster as well as the cache of individual nodes.

Envoy is designed to compromise between the competing goals of maximising local cache hit rates and maximising the aggregate cache capacity of the cluster. Two design features are particularly relevant to addressing these goals. The first is that all client requests are served synchronously by the envoy service without the aid of a local cache. Instead they rely entirely on the shared cache hosted by the envoy service in its private virtual machine. The local envoy directly services all requests—local and remote—for territories it owns, so the entire cluster caches at most a single copy of a given file. Multi-level caching is most effective when lower levels are significantly larger than higher levels (as with the persistent cache compared to the in-memory cache), otherwise the lower level ends up shadowing the higher level and contributing little to overall hit rates [Mun92].

This could potentially strain the envoy that owns a particularly popular file, as it funnels all traffic for that file to a single node. For light to moderate sharing, this is not an issue, and in practice the envoy will be accessing the file mainly from its cache and can handle significant traffic. For extreme instances of sharing, services should use explicit network-facing protocols instead of relying on the file system as a poor man's distributed shared memory system.

The second design feature has a more complex impact on the aggregate cache capacity. Territorial borders are drawn along boundaries in the namespace hierarchy, but because of the copy-on-write mechanism in snapshots and file system forking, multiple names may refer to the same underlying storage layer object. This only happens when the object (but not necessary the file) is read-only, so cache consistency need not be considered, but it does mean that multiple envoys may cache the same underlying object. While this mechanism introduces redundancy in the cluster-wide cache, it also has the potential to consolidate cache entries within a single envoy instance. If multiple file system images have files backed by the same object, they will occupy the same place in the persistent cache as well as the in-memory cache.

Cache utilisation is most effective when clients on a single machine use file system images forked from a common template, and the more complete the template image the more likely it is that services will rely on common rather than custom-installed files. This fits nicely with the stated goals of flexible commodity computation, where both the host and the client gain from using the most popular commodity tools. The host by reducing client footprint and increasing capacity, and the client by reducing deployment costs and maximising performance through increased cache hits.

4.2.5 Data paths for typical requests

To summarise the architecture of Envoy, consider the data paths followed by typical file system requests as depicted in Figure 4.5. The best case is retrieval from in-memory cache on the same machine and is designed to be the most common. Extra steps are required in progressively less-common operations until the worst case, where a request travels from a client to the local envoy service, is forwarded to a remote envoy, misses the local cache and is forwarded to a storage server instance where the data is retrieved from disk.

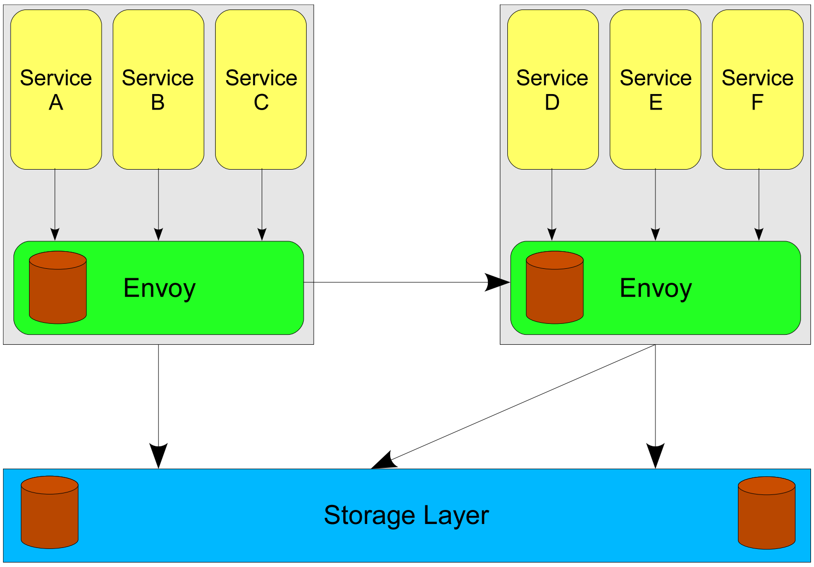

Figure 4.5: File system requests proceed from service VMs to the local envoy service. A request from a local territory may be filled by the local in-memory cache, the local disk cache, or by a single network hop to a storage server. A request for a foreign territory adds a single network hop in each case, as the local envoy acts as the client to a remote envoy.

Read operations

The best case is a request for hot data in a local territory. In this case, data can be served from the in-memory cache of the local envoy server. With a fully optimised implementation using Xen or a similar VM environment, this data transfer can occur with a single data copy from the cache to a data page, and that page can then be swapped directly to the client VM via page table manipulation. Since the client's OS does not keep a cached copy, that page can likewise be passed on directly to the client application. While the prototype is not this optimised, the design permits a very lightweight operation involving a single data copy and some metadata manipulation.

Warm data from a local territory follows a similar path, prefaced by retrieving the requested data from the local persistent cache (on disk) into the in-memory cache. With large, cheap commodity disks, the persistent cache can easily hold several operating system images and typical application suites. Typical Linux installations occupy no more than a few gigabytes, and even that includes many supporting files that are rarely used and may never be referenced in common service deployments [Gib98b]. If services have forked from standard base images as proposed, it is realistic to assume that most operating system and standard applications files will be available in the local cache hierarchy for a service being deployed on an active node [Klo02].

When the local cache fails to deliver, the envoy service must retrieve requested data from the storage layer. In the prototype the persistent cache holds only complete objects, so an entire object must be transfered before the envoy service can begin fulfilling requests from the local cache. If the object is replicated, it can be retrieved through parallel transfers from multiple storage servers for higher throughput. The cache implementation could also be refined to store ranges of bytes instead of just entire objects, which would complicate bookkeeping but permit faster access to partially-transferred files. It would also allow partial caching for files that are too large for the cache. Sequential access to large files merely scrubs the cache when using a least-recently-used eviction policy, but other access patterns could benefit from caching regions of the file, and non-sequential access to large files is becoming more common [Ros00b].

Operations in territories outside local control add an extra network hop between the local and remote envoys for all operations. The data is not stored in the local cache, so locality of reference does nothing to remove this network penalty. It does offer another optimisation opportunity (the data, once received from the network, can be passed to the client application without any further copying) but this is minor compensation for a guaranteed latency penalty.

Fortunately, this penalty need not be too great nor too common. The remote envoy handles the request just as it would one from a client local to it, including caching, so referential locality does improve performance from the cold- and warm-cache cases. In addition, service clusters have high-speed local area networking across switched connections. Finally, because of the way territories are decided, in a steady state system foreign envoy requests generally imply some degree of sharing. While relatively uncommon in itself, runtime sharing requires *some* form of synchronous network communication to guarantee consistency, so Envoy's goal of reducing synchronous inter-node traffic to cases of either concurrent or infrequent access seems a reasonable one.

Write operations

Write operations are less common than reads, but much of the complexity in file system design comes from supporting them. While a good cache satisfies many read requests from memory quickly and with no correctness concerns (provided coherency is maintained in the case of distributed systems), write operations cached in memory raise concerns about durability. If the server acknowledges a write operation as being complete but has only committed it to an in-memory cache, then there is a window of vulnerability before the data is stored to disk. If the system crashes in this time, it will lose data that the client expects to be resilient to crashes. In an isolated client, this may be acceptable. The client will simply be forced to restart from the state that was committed to disk and will lose a bounded amount of work. It is particularly problematic for distributed systems and others with external side-effects, however, where other participants may cue subsequent actions on the premise that a write has been successfully and durably committed to disk. The problem is further exacerbated in a commodity hardware environment where failures are routine.

At the other extreme, one can commit all writes to disk before completing the transaction. This makes it clear to the client when a write operation has been consummated, and it is free to either wait for the acknowledgement or proceed asynchronously with explicit knowledge of the risk it is assuming. While this is a simple and appealing model, it ignores two important realities. The first is that the default action for most commodity operating systems is to cache writes and acknowledge them immediately while delaying the disk write. Changing the expected performance characteristics of a basic operation like writing to disk would severely affect the performance of many standard tools in a negative way and not provide an environment friendly to commodity software. The second is that most files created are short-lived temporary files that are soon deleted [Ous85], so synchronously writing them to disk introduces not only unnecessary latency but also unnecessary disk contention.

Several intermediate possibilities exist. Instead of having write requests proceed directly to the storage layer, the local persistent cache could be used as a staging area, with write requests being committed locally and then forwarded to the storage layer after some delay. This would do little to improve performance, however, as synchronous disk access is slower than synchronous network access, so this would not eliminate the slowest link in the event chain. Specialised hardware with involatile memory could also act as a staging area, giving good write performance while retaining durability. The latter approach violates our goal of using widely-available commodity hardware, however, and neither approach is resilient to hardware faults that result in the entire node failing.

Another approach is to only guarantee synchronous durability when explicitly requested by the client, using the equivalent of the Unix `fsync` system call. This matches the semantics of local file systems, and thus what most software is written to assume. It is not without faults, however, as the popularity of high-level scripting languages and middleware frameworks (especially for network services, exactly the types of clients service clusters are designed to support) means the connection between application actions and disk operations is often obscured. Requiring low-level controls to get correct behaviour may be a popular compromise, but it is not ideal. Recent results suggest that the latency of synchronous writes can be effectively hidden by the operating system, supporting the choice to support synchronous semantics at the distributed system level [Nig06].

The solution Envoy employs is based on exploiting the cluster environment. While commodity hardware is expected to fail occasionally, simultaneous failure of multiple machines is still rare, provided that the nodes are sufficiently isolated from each other in terms of power and cooling. Since service clusters are intended for professional hosting environments, it is reasonable to assume that hardware faults occur in isolation. With that assumption, durability is less about committing data to disk and more about redundancy. A write request is considered final when it is in the memory of all of the storage servers that will eventually commit it to disk. If the envoy server fails, the storage servers are unaffected. If one or more of the storage servers fail before committing the data to disk, the recovery mechanism must restore consistency using the most up-to-date of the replicas. Having storage servers potentially out of sync due to a failed asynchronous write in this scenario is fundamentally no different from having one fail while trying to satisfy a synchronous request. In both cases, the inherent asynchrony of the network means that replicas may be out-of-sync with each other. Only the degree of the problem changes. This scheme allows temporary files to be written and deleted before reaching the disk, but does not do so by gambling with durability for some arbitrary time window, which may or may not be suited to the client's workload [Ros00b].

4.3 File system images

A single, hierarchical namespace unites all the participants in an Envoy cluster, but for management purposes there are two distinct levels. The administrative file tree starts at the root of the namespace and has as its leaves the file system *images* that are normally accessed by clients. Imposing this additional structure in the tree codifies the intended usage pattern, which simplifies administration and allows for some simple but effective optimisations.

4.3.1 Security

Service clusters must accommodate a heterogeneous collection of clients, some of which may trust each other, but most of which do not. To support standard operating systems and tools, Envoy must support familiar semantics, including granting complete control over private images, while also accommodating shared images that grant limited access to various clients.

In addition, clients themselves may wish to arbitrate access to shared storage, rather than expecting the cluster owner to manage credentials for any combination of services.

Enforcement

Simple security is one of the benefits of the service cluster model. Physical machines are controlled by trusted software that isolates clients from each other and from the administrative tools. Hosting the cluster in a managed environment similarly secures physical access to the machines and makes a trusted environment possible. Virtual network devices connect client VMs within a machine, and they also route network packets from clients to the rest of the cluster and the outside world. With well-defined and controlled boundaries, packets cannot be spoofed within the cluster and network addresses can be an accurate and reliable indication of the origin of network data.

For Envoy, this means that client access to the file system can be strictly isolated to the client-server interface exported by a client's local envoy. The clear lines of trust provided by the environment make enforcement of security policies relatively simple. Envoys communicating with each other can be authenticated by their network address as well as by the credentials they provide, as can connections between envoys and storage servers. Encryption of traffic is possible, but since the cluster contains only trusted machines, the threat of packet sniffing is much less than in other environments.

Policy

Access to files is controlled at two levels. The first governs entry to the file system. When a client mounts a directory, be it a file system image or an administrative directory, it supplies its credentials and a pathname relative to the global root of the file hierarchy. The supplied credentials determine the client's identity and maximum privilege level for access to the requested directory and all of its descendents. The second level manages access to individual files according to the identity the client supplied at mount time and the permission attributes of the files.

Administrative directories support all normal file operations, and configuration is done through ordinary files with special formats and naming conventions that are recognised by the envoy service. A credential file can grant access to descendents of the directory that contains it, descendents that may be image roots or further levels of the administrative hierarchy. Permission granted at one level cannot be revoked at a lower level; credentials are established at mount time and are unaffected by subsequent configuration changes. Special files lose their meaning within file system images, so credentials apply to entire images or groups of images.

A noteworthy feature of this system is that clients can manage their own stable of images and the credential files that govern access to them. A set of mutually-trusting clients can share a single credential file and a pool of images with minimal structure, or a client can create a deeper hierarchy of images with fine-grained credentials granting limited access to less trusted clients. A client's credentials may specify its identity or permit it to assume any identity at mount time. This lets Envoy arbitrate shared access between clients or grant them root-like control over the identity space of an image.

File-level credentials are stored as file attributes. A client's identity is established at mount time, and after that access follows the semantics of the client-server protocol used to access Envoy. Unix-style user/group/world permissions and fine-grained access control lists can both be stored as file attributes and enforced by the server. The system used by a particular implementation is largely determined by what the client access protocol supports. In the case of the Envoy prototype, the 9p protocol dictates Unix-style permissions. Depending on the credentials supplied at mount time, superuser privileges can be granted or denied to a client to support the Unix `root` user or emulate NFS-style root squashing [Paw94].

A few examples are illustrated in Figure 4.6:

(a) The cluster owner creates a variety of templates with well-known operating system installations and grants all clients access to fork them as needed. He gives clients individual credentials that give them control over a private branch of the namespace, which they can use as they see fit. Clients can retain exclusive access to their branches, or they can create new credential files to control sharing among their own VM instances and between other clients in the same cluster.

(b) An isolated client registers with a host and is given credentials granting full control over an empty administrative directory. In addition, credentials are supplied that let her fork several image templates, access that is equivalent to letting her mount those templates as a read-only user. She forks a Linux distribution and assumes root privileges over what is now a private image. After configuring the image, she forks it again to support two cooperative VMs. User accounts on her virtual machines access storage through the kernel, which uses her single set of master credentials to authenticate any arbitrary user, but mounts them as individual users with access limited according to Unix-style permission control.

(c) A service provider starts with a similar empty directory, but creates a few levels of directories. He, too, forks a Linux distribution, but customises it with applications fully configured to support a specific kind of server. He forks his own images to prepare several variations that will be useful for different clients. These images are collected under a single directory, where he creates a file with custom credentials to give his clients access to fork the templates he has prepared. In addition, a different directory with different credentials governs access to a shared image that his clients use to share data with each other. They can mount the shared image with limited access, but they can get full read-only access to the templates, or fork one and assume full control over the copy.

(d) A distributed service starts out with the same empty directory. The manager prepares a custom template from a forked FreeBSD image, then forks it many times for different instances of the service. Each

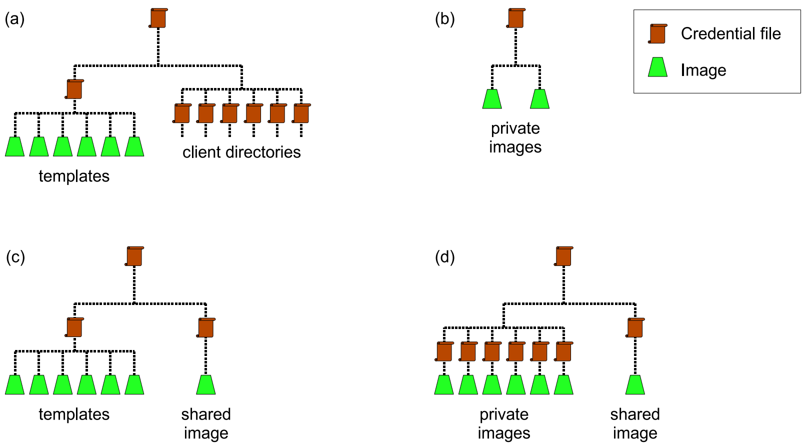

Figure 4.6: Credential files give clients control over access to images and assignment of user IDs. Within images, normal file system semantics govern access to individual files. (a) This two-part access control lets cluster managers grant limited access to template images while giving clients control over their own branches of the namespace. Clients can use their own credentials or grant more limited control in their own hierarchies. (b) Clients may use their own credentials to access images, (c) create their own custom templates for resale along with shared data areas, or (d) create a mixture of private images and shared images.

> instance is assigned randomly-generated credentials that give it full control over a single cloned image, limited access to a shared data image, and nothing else. Each private image is housed in its own directory to isolate credentials, but all of those directories can be accessed by the manager with her master credentials.

Client-owned hierarchies can be used to support many scenarios, from simple single-user images to complex interactions between different clients with limited trust. Templates can be customised and shared, and images can be opened for sharing with client-configured but Envoy-enforced access limitations, or clients can take full control of images and grant access to data through their own network-facing protocols. Flexibility is derived from enforcing normal file access semantics, but letting clients control access to images and manage the assignment of identities within those images. Client-managed credentials can apply to administrative directories as well, letting cluster users further delegate management responsibilities to their own clients.

4.3.2 Forks and snapshots

To support the rapid deployment of services, standard installations of commodity operating systems and software can be provided to clients as a starting point. This is accomplished in Envoy by creating a small set of well-known template images and allowing clients to fork them as a starting point for their own private images. Using copy-on-write techniques, many clients can diverge from a single image, sharing unchanged files and consuming resources only for the changes made.

The copy-on-write mechanism in Envoy works at the level of an object in the storage system. Files and empty directories are leaves in the object tree, with non-empty directories acting as interior nodes that branch out. Modifications to privately-owned objects can be made directly to those objects, but changes to a shared object can only be made by cloning it and applying the changes to the copy.

An object clone has a different object ID than the original, so directory links to the object must be updated, which in turn may require cloning the directory object. Changing a leaf in the file system tree requires cloning a path from a writable ancestor to the object itself, a process that resembles modifying an immutable tree structure in a functional language: an entirely new tree is produced that links back into the old tree wherever possible.

Image versions

While snapshots could be taken at any point in the file hierarchy, a few additional rules simplify management. The root of the namespace is always writable, as are all administrative directories, i.e., those that are not descendents of a directory whose reserved name marks it as an active image root or a snapshot of one. Existing snapshots are always immutable, and new ones can only be taken from the root of an image, not from arbitrary points in the client's directory structure.

A new image is started by creating a specially-named directory in an administrative area that does not already contain an image. Naming conventions distinguish the active version of the image and identify its snapshots. The naming scheme restricts the history of an image to a linear series of snapshots taken over time. For management purposes, the entire collection is considered a single image with an active head and a series of historical versions.

Forking an image

The word "snapshot" implies that a copy of the state of an image is frozen and set aside as through an external observer captured a view of the entire active system, but in reality it is the active version that is frozen and set aside, while a newly-created head takes the place of the old one and diverges from it over time. The continuity of the active image is preserved through sleight-of-hand as active file state is transparently transferred from the old head to the new.

The process of forking an image is the same, except that it starts with a snapshot already frozen as part of the history of another image. To fork a running image, a snapshot must first be taken, after which the forked image and the new head of the old image can both start from the same immutable tree and move forward independently.

Snapshots and territories

Because of the copy-on-write mechanism, changing a single file may require modifying a path all the way back to the root of the image, potentially crossing territory boundaries and involving multiple envoys in the process. This would violate the goal of local impact for normal file operations and complicate the protocol between envoys. To simplify matters, Envoy requires that the root of each territory be writable (unless it is part of a read-only snapshot). The worst case is reduced from cloning a path back to the root of the image to cloning a path to the root of the territory, making it always possible for write operations to be completed by a single envoy.

To satisfy this requirement, the snapshot operation must clone a path to each territory exit after freezing the territory. In practice this process works in reverse, starting at the leaves of the tree and working toward the root of the image. Child territories are frozen and their roots cloned, then parent territories are frozen and paths from their roots to their children cloned. Eventually, the root object of the image is replaced by a clone; the old object becomes the root of the snapshot and the new clone becomes the root of the active head of the image.

This process simplifies writes, but it also creates unnecessary cloning for territories that do not serve any write requests before the next snapshot. This is a tradeoff between simplicity of design and space efficiency. Cheap and ample disk space is one of the motivating advantages to using commodity hardware, so this tradeoff is consistent with the goals of the Envoy file system.

4.3.3 Deleting snapshots

Storage space is cheap and plentiful, and systems such as Venti are designed to keep a complete history of all files ever created [Qui02]. This may be appropriate for some workstation environments, where the increase in storage capacity can out-pace typical data creation rates, but deleting files permanently is still a necessity for many other users. In service clusters, users are charged according to the resources they use, so they must have the flexibility to completely remove old files. Also, when clients leave a particular service cluster, the owner may wish to reclaim the space for future use, as there is little incentive to keeping it around on behalf of a client that is no longer paying. Also, concern for privacy and compliance with data retention laws may require data to be completely removed.

The only backups mechanism in Envoy is the snapshot operation. Since anything created and deleted in the window between two successive snapshots is no longer accessible anywhere, it is deleted immediately. This is easily detected through the copy-on-write mechanism, which identifies all files that were created since the most recent snapshot. All files with copy-on-write flags (either explicit in a directory link or implicit through an ancestor's link) are backed by objects referenced by one or more read-only snapshots, so envoys never delete these objects when their corresponding files are deleted.

The problem comes when trying to delete old snapshots, as it can be difficult to determine which storage objects are only referenced by that snapshot.

An obvious solution is to implement reference counting in the storage layer. This has the advantages of simplicity, accuracy, and immediacy. The main disadvantage is that it puts the performance burden in the wrong place: every time a directory object is cloned (a frequent operation when a snapshot is followed by write requests) the reference count of all objects in that directory must be incremented. This requires updating all storage replicas of all files in the directory, making the copy-on-write mechanism expensive in order to support an infrequent operation that is not timing critical.

The copy-on-write mechanism in Envoy is similar to the one in Parallax, as is the problem of deleting snapshots [War05]. The problem was simpler in Parallax, however, which uses a copy-on-write radix tree to map logical blocks to physical blocks in a virtual block device. The virtual block numbers do not change between snapshots, so comparing the physical blocks mapped to by two successive snapshots reveals which blocks from the first were unlinked during the lifetime of the second.

With Envoy and its hierarchical file tree, the problem becomes more complex. While it is still a simple matter to compare the object IDs of directory entries to detect changes between snapshots, it is harder to distinguish between the effects of objects being cloned and objects being deleted and created. In Parallax, the two radix trees are recursively walked in parallel, and changes always denote cloned blocks. In Envoy, files and directories can be renamed, so there is no easy way to match an object in an old snapshot with a cloned version of the same object in a newer snapshot. The static virtual block IDs in Parallax are replaced by variable names in Envoy, making the process of comparing two snapshots more difficult. Supporting hard links makes it even harder to define a one-to-one correspondence between object references in two successive snapshots, since multiple references may exist to a single object.

This potentially messy problem can easily be solved by brute force. Instead of imposing extra runtime overhead for normal file operations or attempting to walk two file systems and identify matching files within them, it is simple and practical to gather a complete list of objects referenced by an image. Using 64-bit object IDs, such a list takes 8 megabytes for every million files. A 1999 study of workstations at Microsoft found an average of 13,309 files on each of 10,568 machines, with the subset running NTFS (the newest file system studied and the one with the largest average number of files) averaging 24,229 files over 3,332 machines [Dou99]. Average file counts will no doubt continue to grow, but anecdotal tests found the number still in the low millions for typical, modern desktop Linux installations.

Gathering a complete list of objects and sorting it for each snapshot makes the identification of object creates and deletes a simple matter of scanning two catalogues in order and finding differences. It requires some storage space to implement, but not much. Deleting snapshots is not a critical-path operation and for typical image sizes the brute force approach requires little enough space that that no amount of reduction would be worth more than a very small complexity increase.

This asynchronous process bears some resemblance to the cleaner of log-structured file systems [Ros91], which asynchronously recovers disk space from the tail of the log. The snapshot-delete problem has a critical difference, however: Envoy runs in a cluster environment, where the asynchronous process can run on a different node to prevent the interference caused by the cleaner under some workloads [Sel93, Sel95]. In addition, successive snapshots will typically have much overlap in the directory objects referenced, so the machine-level cache can absorb much of the traffic.

A sorted file listing all object IDs referenced in the snapshot can easily be stored in the administrative directory that contains the image. To delete a snapshot, the objects of it and its immediate successor and predecessor are compared in order. Each object that appears in the snapshot but disappears in the successor can be safely deleted from the storage layer. To make it safe and resilient to crashes, the envoy service ensures that no clients are currently accessing the snapshot, then it unlinks the root of the image first, and deletes the list of object IDs last. At recovery time, an object ID file found without a corresponding image indicates that a crash occurred before the operation completed, and it can be safely restarted. The image itself isn't necessary at this stage, and as long as the cleanup process can tolerate objects having already been deleted, it can work entirely from the object lists.

The only other complication in this process is image forks, where multiple successors may exist for a single snapshot. Forking an image does not directly affect the snapshot used as the starting point, so the easiest way to detect forks it to log them. The log is consulted before any snapshot delete attempt, and deleting images that have more than one immediate successor is not permitted.

A few other corner cases are worth mentioning. The current version of an image can be deleted using the same procedure, but it must be made read-only or client access must be disallowed before gathering the list of object IDs (note that access is normally only forbidden when deleting starts; the catalogues of predecessors and successors must be assembled as well as those of images marked for deleting, but normal access can continue during this process). Otherwise, the normal procedure suffices, with the successor object catalogue taken to be empty. The log of image forks must also account for deletes, so that entire trees of images can eventually be pruned back to the root if desired.

4.4 Territory management

The Envoy file system model is based on the idea of presenting a single, large file tree connecting arbitrary images, but providing incentives to use it in ways that can be exploited to provide good performance and scalability. The service cluster model makes it simple to isolate control of the file system from the clients who use it, while still keeping synchronisation logic and caching on the same machine most of the time. Providing a global namespace gives a great deal of flexibility, but inferring usage patterns and collocating ownership of branches of the tree with the clients that use them yields short data paths and good performance.

4.4.1 Design principles

A variety of approaches to distributing territory ownership are possible, and only a realistic usage model drawn from empirical study of real-world deployments can accurately inform optimum choices. Lacking that, territory management in Envoy is designed with a few goals in mind.

The first is to favour optimising long-term patterns over tracking short-term trends. High-speed switched networks minimise the penalty for serving a request from a remote envoy compared to handling it on the client's envoy, and the growing gap between memory and disk performance makes flushing the cache to support a territory realignment continually more expensive. Based on this and on the past success of client-server file systems, Envoy favours slow evolution of the namespace topology to capture steady-state client behaviour. Instead of attempting to track each change in runtime usage patterns, it offers a client-server model that gradually optimises itself over time by collocating servers with clients.

The second is to avoid complexity whenever possible. This applies to the runtime behaviour of the system as well as the algorithms and implementations that drive it. For debugging, recovery, and runtime analysis, territories with simple boundaries that do not change frequently are preferred. Painting control of the namespace tree in broad strokes makes it easier for humans to comprehend and analyse, minimises perverse cases that can threaten correctness and the success of recovery operations, and makes global logging of changes practical. With this in mind, Envoy favours using a few territory divisions to give good results over making many divisions in an attempt to approach optimal results.

4.4.2 Special cases

Token passing is a popular way to coordinate access to file system objects. Before a client can access a file, it must be granted an appropriate *token*, and the token must be transferred to a second client before it can operate on the same file. Multiple reader/single writer tokens may permit some concurrent access, but synchronous token transfers are frequently necessary before a request can be satisfied. While these operations can be optimised, fundamentally they operate on a pessimistic model analogous to locking in shared memory schemes.

Territories in Envoy are related to leases and token systems, but they are based on an optimistic model where large groups of files can be granted to an owner on the assumption that sharing is uncommon. Synchronous token transfers are avoided; every request from a client can be processed immediately either by the client's envoy or by a direct, already-established link to the owner, with complete access to the owner's cache. Performance is best when the territory's owner is on the same machine as the client, but because of the cluster environment the penalty of an extra network hop for remotely-owned territories is not excessive.

The decision to cede all or part of a territory to another envoy is always made by the current owner. While a client's envoy may be able to recognise the client's ongoing demand for a particular region of the file tree, only the envoy that manages it can account for all clients that are accessing it and act based on complete information. Territories form a tree overlaid on the file system tree, and territory transfers are always driven by the parent of the territory being transferred. The parent only initiates a transfer when the owner of the territory requests it, leading to the first special case in territory realignment: when a territory is dormant, it is ceded to its parent. Dormancy is determined through the general mechanism described below, but this is highlighted as a special case because no other envoy can detect a territory that has fallen out of use. Territories are also ceded when an envoy is shutting down.

Another special case is based on the expectation that most images (especially those used as boot images) will be used by only a single client: when a client mounts an image that is not in active use, the image is immediately ceded to that client's envoy. The first client to mount an image may not be the one that will use it the most, and the dynamic algorithm would sort it out eventually anyway, but this heuristic avoids a warm-up period of degraded performance for the most common usage pattern. For services with no sharing, this is sufficient to completely localise non-administrative traffic. This is also an example of how imposing a little structure on the file tree can not only simplify administration, but also improve performance.

4.4.3 Dynamic boundary changes

Control of a private image is handed over the the relevant envoy when the client mounts it, but control of shared images is initially given to the *first* client to access it, which may not necessarily be the heaviest user. Envoys observe the access patterns of their local clients and compare them to forwarded traffic from remote envoys, periodically ceding control of parts of territories, or even handing over control of an entire territory in an attempt to improve locality of access.

Territories are transferred in response to observed usage, with one of two goals: to improve locality or to simplify the territory layout. The latter occurs when traffic to a territory falls below the *idle threshold*, and the benefits of local ownership are not considered worth the extra cost in topological complexity. In this case, the territory is handed to the neighbouring envoy with the most boundaries in common with the old owner, either the parent or the owner of one or more child territories.

Traffic is monitored by the number of requests from each remote envoy, with all client requests combining to form the owner's contribution. A single value for each participant is computed as the total number of requests, exponentially decayed over time with a configured half-life. This approach allows envoys to continually monitor load and react not only to the presence of imbalances, but to their severity as well. Sub-optimal layouts are addressed more urgently when traffic volumes—and the potential benefits of optimisation—are high, with a slower response given in low traffic, where waiting can confirm that the trend is lasting and that a fix is likely to be worthwhile.

For simplicity, only a single new territory is created in each realignment; if two branches of a territory need to be ceded to a remote envoy, they will not be combined into a compound transfer, but will instead be evaluated and transferred independently. To aid in accurately predicting the effect of a given transfer, the traffic value for a directory combines requests for its descendents with those for the directory itself. To the extent that recent traffic trends continue, the combined traffic values for a particular object summarises the overall effect of ceding it as the root of a new territory.

For each object in a territory, the owner considers ceding a new territory rooted at that object to each envoy that has driven traffic to that branch. The harm to the local envoy and the benefit to the remote envoy are weighted equally by subtracting local traffic from that envoy's traffic, yielding the expected benefit of the transfer. Requests from third-party envoys will be unaffected, as they will just be transferred from one remote envoy to another. In this way, the expected benefit of each territory change (including transferring control of the entire territory) can be considered and compared with the alternatives.

Before actually initiating a transfer, two conditions must be met: the transfer must be the one that will yield the maximum expected benefit, and the urgency of the proposed change must be sufficient to justify the disruption of a boundary change. A highly beneficial transfer is considered urgent, but if the improvement is modest then it is delayed to encourage stability in the topology and discourage thrashing of the cache. A simple linear scale has the urgency requirement decreasing as time elapses since the most recent boundary change affecting the envoy. A minimum delay and the idle threshold guard against extreme cases.

Scanning an entire territory after each request would be prohibitively expensive, but the scheme outlined here can be approximated in a straightforward fashion. As each request is recorded and the traffic value for affected objects updated, the envoy computes the expected benefit of transferring that object to the originator of the request. This process is repeated as the request is recorded for parent directories all the way to the root of the territory. The maximum benefit observed is compared to the time elapsed since the most recent transfer, and the envoy decides if a new transfer is warranted.

With this implementation, a transfer that was rejected at the time of a request for insufficient urgency may become viable as the territory ages, and the envoy will not notice it in the absence of a new request to trigger a re-evaluation. While this violates a strict interpretation of the procedure, it does so only in the absence of traffic from the remote envoy, a condition that casts doubt on the efficacy of the transfer anyway.

4.5 Recovery

Putting commodity computation in a managed environment with well-provisioned hardware reduces the rate of node failure, but all computer systems are subject to the hazard of failure, whether from hardware faults or software problems. A viable recovery procedure is essential for any storage system, and minimising the disruption to other nodes is also important in a cluster environment.

The controlled environment has its advantages, however, in that the expectation of failure for a given node is low enough that it can be treated as an exceptional condition, rather than a routine part of operation. This stands in contrast to distributed systems running on machines owned and managed by a wide range of people, whether peer-to-peer systems on volunteered machines or networks of workstations in a corporate environment, where the needs and habits of the node owners preempt the interests of the whole system.

As a commodity platform, failure of an individual node can reasonably disrupt the services running on it. Uptime guarantees and other reliability requirements must come from higher-level services, which may be implemented as a series of clients on a service cluster or across multiple clusters. The latter is necessary for the most stringent requirements anyway, since catastrophic failures such as natural disasters may affect all machines at a location no matter how well cared for. Since prevention of failure is impossible, and failover capabilities for all clients would be complex and expensive, a more practical approach is to assume that nodes will occasionally fail, and seek to minimise the disruption to the rest of the cluster while restoring the service of the lost node as quickly as possible.

The basic assumption implicit in Envoy's recovery model is that the failure of a machine or any of its parts (including management software) may result in lost service to the clients hosted on that machine. Moving outward, envoys that were interacting with the failed machine should be able to recover fully with some disruption, while nodes with no overlapping interests should be unaffected.

The other philosophy that drives failure recovery is that it should be as simple as possible. Recovery scenarios are both difficult to predict and difficult to simulate. In a distributed system, interactions involving multiple participants can be complex enough when they work, and an unanticipated failure at an unexpected time can often lead to conditions that are hard to anticipate. Enumerating cases that must be handled is an error-prone process that can become intractable with too many sources of faults. Even when complex failure cases are correctly identified, simulating them under realistic conditions to test equally complex recovery code is another difficult and error-prone process. Ongoing field testing is dominated by correct behaviour (one hopes) so recovery code is rarely exercised as well as normal code paths. Finally, because recovery procedures are only invoked in response to failure, they represent the last line of defence against lost data and the last chance to retain the trust of users, a critical element in storage systems.

Some of the most successful systems reflect these concerns. Database systems typically log transactions in a simple, append-only structure before applying changes to complex data structures. At recovery time, no attempt to diagnose the exact conditions of failure is necessary—instead the log can be replayed to complete committed transactions or even resume an interrupted recovery process. Similar journaling schemes are increasingly common in modern file systems for many of the same reasons. Careful ordering of writes can ensure that a crash at any stage leaves the system in a sane state, offering the same principle benefit: recovery from a wide range of failure conditions can proceed even without knowing what actually caused the failure.

4.5.1 Prerequisites

Some amount of redundancy of runtime state is necessary to allow complete recovery after a node failure. The fate of a client is already tied to that of its local envoy, so envoys track the state of all active file handles for local clients, including those files that are owned by remote envoys. The local envoy acts as a proxy server for remote files, so it can easily peek at request and reply messages to observe state updates. For local and remote file handles, an envoy tracks the full pathname of the file, the user credentials used to access the file, and any state related to file status and position required to support the client-server protocol used by the client.

A client's envoy can track its file handle state locally, but file data is not duplicated for forwarded requests. Duplicating runtime state can prevent disruption, but preventing data loss requires redundancy in written data and attributes. The write-through persistent cache in Envoy answers this requirement by ensuring that data has reached the storage servers for all replicas before reporting a write operation as complete. As long as enough storage servers are able to write the data to stable storage, the crash of an envoy node will not affect the stability of data, nor its immediate availability to the recovery process.

Depending on the storage layer implementation, a window of vulnerability may exist between the time the first and last storage servers have been notified of an update, during which an envoy crash would result in an inconsistent state between replicas. The problem resembles that of a storage node failure, and in both cases recovery of the storage node would necessitate it "catching up" with missed transactions. This is fundamentally a part of the storage layer design, not part of the envoy service.

4.5.2 Recovery process

With runtime state and file data available, it is possible for envoys that were interacting with a failed node to recover fully from the disruption, with temporarily degraded performance as the only effect visible to clients. The affected envoys include all those with which the crashed node had any kind of relationship, including the owners of the parents of its territories, the owners of the children of its territories, remote envoys accessing files in its territories, and remote envoys to which it forwarded requests for local clients.

The first step in recovery is recognising that something has gone wrong. Envoys monitor the status of their neighbours in the territory tree as well as that of the envoys with which they share files. These connections can be monitored closely without imposing a significant burden on the network, as they are always unicast messages between small sets of hosts. Normal interactions can double as heartbeat messages most of the time, and explicit messages are only necessary on otherwise idle connections.

Once a node fails, the territories it owned and their descendents are immediately dissolved and annexed into the parent territory. In-flight operations no longer act under the authority of a territory owner and are suspended. Forwarded operations between two otherwise-healthy envoys are rejected with a suitable error so that the client's envoy is made aware of the failure. Parents notify children recursively, resulting in a pool of envoys that hold file handles for their clients but have nowhere to send transactions. The envoys then reconnect each file handle to a territory owner by walking from the root of the file system tree to find the new owner. Unlike normal directory navigation requests, these traversals always succeed, even if directory permissions have recently changed. Envoys can nominate themselves to reclaim ownership of territories, or they can leave it to the normal demand-driven process to sort out boundaries over time.

In keeping with the goal of simplicity and blanket coverage of failure cases, this handles failures that occur in the middle of territory ownership transfers as well as simpler, steady-state cases. The canonical version of the file tree is maintained by the storage system, not by the soft state of territory ownership. Once the ownership tree is corrupted by a failure, the corrupt branch is pruned and re-grown from its former root. Similarly, the canonical version of a file handle is maintained by the client's envoy, not by the territory owner. Envoys champion the needs of their own clients by restoring their file handles to working status and resuming their suspended operations, while the operations initiated by failed envoys are forgotten and their files implicitly closed.

If certain write operations were in progress when the failure occurred, it is possible that orphaned objects may result. The operations can still succeed, but objects that were created but not linked to may be lost to the system. Snapshots and the copy-on-write mechanism expose particular vulnerabilities with their leaf-to-root clone-and-update procedure, potentially losing chains of cloned directory objects if interrupted by an ill-timed failure. This does not result in data loss, but it may leave an occasional object that cannot be reached through the file tree. The potential for minor capacity loss is regrettable, but the likelihood and magnitude of the problem are both low and can be ignored.

4.5.3 Special cases

Throwing away damaged soft state and rebuilding it instead of trying to patch it makes recovery from a node failure simple and uniform across a range of failure conditions. An important step in the process depends on the ability of newly-ostracised envoys to rejoin the collective by starting at the root of the tree and pushing file handles back down to their appropriate locations. Two problems can inhibit this process: failure of the root node, or a change in the path from the root to the target node.

Every node that joins the Envoy system must know how to locate the root node. Whenever a client attempts to mount a file system image, it specifies it as the path from the root of the global namespace to the root of the image. Locating the root initially may be part of the startup process for an envoy, or it may integrate more closely with a higher-level service-cluster management tool. Other management services must exist to instantiate and monitor client VMs, and it is sensible to have Envoy coordinate directly with those services. Monitoring the root node and anointing a replacement when it fails could fall to the management tools as well, or envoys could instead be started with an ordered list of root nodes, the next taking over when the previous fails.

The rest of the recovery process need not be changed for the root node, as the existing procedure would suffice. It may prove worthwhile to treat it as an exception, however. In the prototype, all of the top-level administrative directories are managed by a single envoy, with the transition to individual images being the first point at which territory ownership changes are allowed. While this is just a simplification for the prototype, an implementation with no such restriction would probably follow that pattern quite often, as most clients would mount an image and then never interact with administrative directories again. The root node may end up with many more children than a typical node, and dissolving all territory boundaries would disrupt the whole cluster instead of localised regions of related clients. As an optimisation, the root node could log territory changes that it observes into a regular file with a well-known name (periodically checkpointing by dumping a complete list). Its successor would then start in the normal way, read the log file, and then contact each child to inform it of the change. This would also require that children of the root be aware of their special status so that they would not dissolve their branch of the tree upon detecting the root node failure.

Renaming a directory always throws file handles for its descendents temporarily out-of-sync. A recursive update procedure ensures that this does not last too long, but node failure could interrupt the propagation of the update, or the update could happen after a descendent fails but before other affected nodes have fully recovered; in either case the file handles that they use to reconnect would have incorrect pathnames. Renaming high-level directories is already quite rare, and having it coincide with a node failure is quite unlikely. It would be reasonable to simply report a stale file handle to the client when this happens, forcing it to locate its files again and re-open them. Alternatively, the messages used to propagate renames down the territory tree could be adapted to implement a two-phase commit protocol. This would also prevent other potential race conditions that might come up when renaming directories.

Besides envoy failure, storage node failure and network partitions can also disrupt a running system. Both problems are the province of the storage layer. The occasional failure of a storage node is expected and is one of the reasons for replication of all stored objects. A failure can be tolerated and repaired without disrupting the running system (other than possible performance degradation), though the details of recovery are part of the object storage system and omitted from this dissertation.

The network partition case is more serious, as it resembles the simultaneous failure of many nodes. The most important consideration is to prevent permanent data loss, which means preventing conflicting updates that cannot later be resolved. By relying on the storage layer, this can be easily resolved: envoys can only make binding changes in the storage layer when they can contact a majority of the replicas. If a small part of the network is isolated, it will soon fail until connectivity is restored. If the larger portion retains enough storage servers, it can continue on uninterrupted, but it may also happen that both parts of the network are disabled until the problem is resolved. Network partitions are a traditional bane of distributed systems, but they are less of a concern in cluster settings, because of redundancy in the network itself. Redundant network interfaces, routers, and switches make it unlikely that groups of machines will be isolated, so requiring a repair before resuming normal operation is reasonable. Using SCTP [Ste00] instead of TCP gives built-in multihoming support, so isolated network failures can be tolerated without the envoy service taking any explicit action.

4.6 Summary

The Envoy file system is a distributed storage system designed for service clusters. It assumes a cluster environment with well-maintained machines and sufficient redundancy to prevent correlated component failures.

Envoy runs a service on each node in a trusted administrative virtual machine, which exports file services to clients on the same node. A separate storage layer provides an object-level data abstraction, which the envoy layer uses to compile a file system. Control of the file tree is divided into *territories*, or subtrees which may themselves be further subdivided. A territory is *owned* by one envoy node, which maintains a persistent and in-memory cache for it and acts as server to all clients that access it. Client requests for remote territories are forwarded by the local envoy. Relationships between envoys are maintained only when they have neighbouring territories, or when one is handling client requests on the other's behalf.

File system images are subtrees within the global namespace that are treated as management units. Lightweight snapshot and fork operations use copy-on-write techniques to enhance performance and encourage sharing even between unrelated clients. After a snapshot, all objects become globally read-only and can be cached anywhere in the cluster without further coordination. Each writable object is owned by a single envoy that coordinates all access to it, and clients share a node-level cache, so consistency is guaranteed.

Control of an image is initially handed to the envoy of the first client that mounts it, but territories may be migrated, split, and re-merged in response to demand. A greedy algorithm monitors client usage and finds the most urgent transfer, or the one that will benefit the remote node the most while hurting the current owner the least. The level of urgency determines how quickly the transfer will happen, with low urgency migrations being delayed to promote stability and effective cache use, while more pressing moves are made quickly.

The fate of clients is tied to their local envoys, which maintain copies of all their runtime file system state. The failure of an envoy dissolves all territories below it in the hierarchy, forcing descendents in the tree to re-join the active system by walking from the global root down to the locations of files that are still active. Node failure only requires action from envoys that had some kind of working relationship with the failed node.

Chapter 5

The Envoy Prototype

Even the simplest file systems involve complex interactions between numerous hardware and software components, and distributed file systems compound that complexity. While all modern designs tend to draw heavily on prior work, the particular combination of classic components and new innovations that make up a new system like Envoy can only be validated with empirical results.

I have implemented a prototype of Envoy to make testing possible and to permit a level of refinement in the design that is only possible with practical experience. This chapter first details the goals and parameters of the prototype, then discusses aspects of the implementation that shed further light on the design, expose its limitations, or show how implementation choices have shaped the prototype.

5.1 Scope and design coverage

The goals of the prototype are more modest than the goals of the Envoy design. The design addresses the storage needs of a complex system, and its suitability for the problem at hand is determined partly by how accurately the problem was described. Including appropriate features addresses some of the needs of service clusters, and only production experience can fully determine how successful some features are. Because this dissertation addresses only the storage aspect of the commodity computation problem, a complete evaluation must be deferred.

Other aspects of the design and its suitability for the intended task can be validated by testing an implementation of the file system, even in the absence of a complete service cluster environment. The prototype was implemented to allow empirical testing of the basic structure of Envoy, and to expose practical issues that could lead to refinement of the design. The design presented in the previous chapter reflects many lessons learned from the prototype implementation, and the next chapter presents measurements from using it.

The remainder of this chapter describes the artefact itself. The Envoy design calls for a client-server file system interface for connecting individual clients to the distributed system, but it does not require a specific one. The prototype is written to support a specific interface, and the rest of the prototype takes design cues from that interface. While the storage layer structure is not specified in the general design, a prototype supporting the general interface and redundant layout required by Envoy was necessary for testing; its implementation is also described here.

5.1.1 Implementation shortcuts

The prototype is designed to validate the basic design and performance of the Envoy system. Features with little effect on performance, such as authentication schemes and recovery procedures, are omitted because their implementation would prove little. Other shortcuts in the implementation supported faster implementation at the expense of runtime performance. For a prototype this is an appropriate tradeoff, as validation of the overall design is more important than particular benchmark results. A few such shortcuts are described here.

Synchronisation

The prototype uses a simplified concurrency model based on a few assumptions. Concurrent transactions are permitted and executed on individual processor threads, but only one is allowed to execute at any given time. A global lock is held by the running thread, and is released whenever a network or disk operation is invoked. This simplifies shared-memory management considerably, and is justified as long as the processing time for transactions is dominated by I/O latency.

Deadlock is avoided by forcing transactions to release all held locks when an attempt to acquire a new one fails. The transaction then waits on a queue and is restarted when the contended object becomes available. This forces transactions to acquire all necessary locks before introducing any side-effects that cannot easily be rolled back, but in exchange it allows locks to be acquired in any order and simplifies lock management. Multiple-reader/single-writer locks are also available for controlling access to territories, so transaction queueing is mostly limited to concurrent operations on a single file.

Caching

The prototype runs under Linux or other Unix systems, and takes advantage of the buffer cache of its host rather than implementing its own cache. This is a deliberate choice rather than a shortcut, as it takes advantage of the ongoing refinement of the host operating system. This is in line with the goal of taking advantage of commodity hardware and software, where continuing improvements are expected over time. The cache can be managed by managing the VM that hosts the envoy services.

Garbage collection

The prototype is implemented in C and uses the Boehm-Demers-Weiser conservative garbage collector [Boe93]. Garbage collection is unusual in file system implementations, and may come at a performance cost, but it is useful for simplifying and speeding up the implementation of the prototype.

5.2 The 9p protocol

Clients access Envoy using a client-server file system protocol between the virtual machine of the client and that of the envoy service. While a custom protocol would offer the greatest flexibility, it would also make the implementation considerably more complicated and would have little value in demonstrating the viability of the design.

5.2.1 Alternatives

With Linux as the client operating system of choice, a few prominent protocol options were available but ultimately rejected for the prototype.

NFS version 3

The NFS protocol is popular, simple, well-understood, and has a robust implementation. Most of the complexity is implemented in the client drivers, so servers can be quite simple [San85, Paw94, Cal95]. I also had experience with the protocol after implementing a stackable, copy-on-write file system (CoWNFS) to support service deployment in XenoServers [Kot04b].

For user mode servers generally and Envoy in particular, however, the stateless model of NFS complicates implementation. File handles assigned by the server and given to the client are expected to be immutable and always available (there are no explicit open and close operations), making it difficult to maintain pairings between active files and the envoy instances that own them. With transparent copy-on-write, these handles must either be cached indefinitely or tied to some other file identifier, such as its complete pathname. Changing territory boundaries makes caching impractical, and allowing higher-level directories to be renamed makes the latter difficult to make robust.

Security under NFSv3 is based on Unix user and group IDs, which proves to be an inflexible mechanism when diverse clients become involved. The ID space must be shared between clients that share file systems, or the server must provide a mapping service to reconcile the differences. While not an insurmountable problem, this can get unnecessarily complicated when a client mounts multiple file system images, many clients share some images, and parts of those images are forked from standard base images.

The caching model of NFS is entirely under the control of the client driver, which generally sends frequent `stat` requests to check if its cache is still current. Write operations are generally asynchronous, however, and the lack of control would have defeated the cache coherency guarantees that Envoy seeks to achieve. Caching at the client level also prevents consolidation of duplicate caches at the physical machine level, another of the aims of the Envoy design.

NFS version 4

The most recent update to NFS addresses many of these concerns by introducing a stateful model and using leases to manage client caching [She03]. These client delegations could work well with Envoy, allowing private files to be cached by the client and shared files to be held back and cached by the envoy, though in the latter case the consistency semantics are no better than in earlier NFS versions. More flexible file handles and explicit state management are also a better match. NFSv4 would be a viable candidate for a future implementation, but at the time the Envoy prototype implementation was started, it was still relatively immature and the supporting tools were complicated to use and poorly documented.

AFS

The Andrew File System is another popular client-server system with a mature implementation [Sat85, How88]. It employs persistent caching at the client, which is a poor fit for the Envoy model, and it is not intended as a general-purpose protocol for implementing custom servers. Adapting it to a system with very different semantics and basic assumptions would negate the benefits of using an established protocol and implementation.

FUSE

The FUSE driver and tools support custom userspace file systems under Linux. The FUSE interface is modelled after the Linux VFS interface and exposes some of its complexity, but the main point against using FUSE for the Envoy prototype is that it is not a network-facing protocol. A userspace tool for the client would be required that would in turn connect to the envoy service across the virtual network device. Besides adding an unnecessary layer of indirection, this would complicate using Envoy as a root file system.

Custom Envoy client driver

Another possibility would be to implement a custom kernel driver for the Envoy client as well as the Envoy server. This would permit an exact match between the expectations of the client and the semantics supported by the server. It would also permit optimisations precluded in standard drivers by the need to support a diverse range of servers.

Using a custom driver would have disadvantages, among them the need to divert implementation time away from the server or to extend the total time required. A custom driver would not be available in standard software distributions, either, requiring additional deployment effort for end users. Finally, a custom protocol would make evaluation more difficult by making it harder to isolate the costs of the Envoy deployment model from the costs of the protocol and client implementation. Being able to compare Envoy performance against other servers using the same client driver may penalise the overall performance, but it also allows a more detailed analysis of the results.

9p

The Envoy prototype is implemented using the 9p protocol from the Plan 9 operating system [Pik90, Pik92], which is specified in section 5 of the Plan 9 manual [Pik95]. Plan 9 breaks from Unix semantics in many ways, so the Linux port of 9p has been extended to better support Unix semantics [Hen05]. The 9p protocol is intentionally simple. Individual connections are established for each user, and file ownership is tracked using user and group names, not numeric identifiers. The protocol includes no explicit support for client-side caching beyond the availability of file version tagging that could simplify NFS-style cache validation checks.

The 9p protocol is based around the thirteen messages listed in Table 5.1. Authentication is achieved through an arbitrary sequence of reads and writes to a special file handle established with the `auth` message, and after the server is satisfied the client can follow-up by attaching to a specific point in the file hierarchy. Access to the rest of the hierarchy is through `walk` messages that move new or existing file handles through the namespace, and normal operations that can then access files and directories through the associated file handles.

5.2.2 Mapping Envoy to 9p

For basic file operations, an envoy acts like a 9p server to the client. The client operating system connects over a virtual network device that connects its VM to the VM hosting the envoy service. While the prototype uses a standard TCP connection for communicating, 9p supports any transport layer and could use an interface for virtual machine environments that swaps memory pages between VMs instead of copying data

Message name	Description
`version`	Initial handshake; establish message size limits.
`auth`	Authenticate a user to mount a specific path name.
`flush`	Cancel a pending request.
`attach`	Mount a specific path for a specific user.
`walk`	Navigate directories and/or clone a file handle.
`open`	Open a file or directory.
`create`	Create a file or directory.
`read`	Read bytes from a file or entries from a directory.
`write`	Write bytes to a file.
`clunk`	Release a file handle and close the file.
`remove`	Delete a file or empty directory.
`stat`	Read attributes for a file.
`wstat`	Change a file's attributes.

Table 5.1: The messages in the 9p protocol. `version` and `auth` prepare a connection for an `attach` request, which establishes a single file handle. `walk` can clone or move existing file handles, giving access to the rest of the file tree. `flush` relates to an individual transaction, but all other messages operate relative to an active file handle.

through the standard network protocol stack. While this has not yet been implemented for 9p servers, it is a potential optimisation that could reduce the additional cost of retrieving data from the machine-level cache over retrieving it from an OS-level cache in an individual VM.

File handles and state management

When a file is owned remotely, the local envoy acts as a proxy server and forwards requests to the appropriate envoy. Like client connections, forwarded transactions are tracked relative to file handles owned by specific users.

File handles are tracked by small positive integers, which are unique in the context of a specific connection. When acting as proxies, envoys map client identifiers to *remote identifiers*, which are unique for a given envoy instance. This allows envoys to combine all requests from their constituent services when contacting a remote envoy, effectively making the envoy appear as a single client. By making the identifiers unique for all outgoing requests from the envoy instead of for each local-remote envoy pair, no re-mapping is required when file handles migrate from one remote envoy to another during territory realignments.

Security

Individual users mount the file system in 9p, and file identifiers are subsequently tied to a single user. The server authenticates a specific user and authorises the creation of a specific file handle at mount time, and additional file handles are produced by cloning existing ones and walking the directory structure to find the desired files.

This system is simple and matches the distributed layout of Envoy well. Envoys trust each other, so authentication need not be repeated as clients cross territory boundaries. Identifying individual users instead of authenticating hosts and leaving user management to them also makes shared images easier to manage. A user or process on one client can connect to a shared image using its own credentials, without requiring administrative access even on its own virtual machine. Likewise, the administrative user can be restricted and treated like a normal user on a shared volume without requiring any specific trust in the client's operating system.

9p does not specify the protocol for authenticating users, but instead offers a mechanism for arbitrary exchanges to take place before a user succeeds in attaching an image. This allows flexibility in the server implementation, which may rely on the network address (which can be securely controlled in service clusters) in addition to security protocols that exchange credentials. A generic implementation is possible within the bounds of 9p, as is closer integration with the cluster's administrative services.

Caching

9p does not explicitly manage caching, and the Linux client driver does not implement any client-side caching. This supports Envoy's model of consolidated node-level caching by forwarding all requests immediately to the envoy. No further configuration is necessary for clients.

Because all requests cross the virtual network device between a client VM and the envoy service, the efficiency of the protocol is important to the performance of the file system. 9p permits any transport layer to be introduced under its message protocol, so an implementation that avoids copying data through the network stack and instead swaps physical memory pages between two virtual machines is viable. The prototype does not implement this, but 9p would permit it as an optimisation.

5.3 Storage service

The storage layer is implemented in two parts, called the top and bottom halves.

The bottom half is implemented in a stateless storage daemon hosted on each physical server. While the storage daemon instances form a collective pool of storage, they do not communicate directly with each other. At the local level, each storage manager is unaware of any global state, and responds blindly to incoming requests from the envoy layer. Instances do not attempt to balance load, resolve conflicting requests, or manage redundancy, nor do they monitor which object IDs they considered valid. Instead, they provide a thin, simple storage service for numbered objects with attributes.

To make these servers more useful, the top half of the storage layer is implemented in the envoy daemon. It is responsible for mapping an object ID to the set of storage server instances that host the referent object. Combined with the persistent cache, the storage layer top half provides a simple procedural interface to the storage layer, where objects are named by unique IDs. The top half is responsible for creating and locating replicas, detecting and masking/recovering from failures, allocating new object IDs when needed, and reading and writing data and attributes.

Message name	Description
`reserve`	Claim a range of unallocated object IDs.
`create`	Create a new object.
`clone`	Copy an existing object.
`read`	Read a byte range from an object.
`write`	Write a byte range to an object.
`stat`	Query an object's attributes.
`wstat`	Modify an object's attributes.
`delete`	Remove an object.

Table 5.2: The messages in the storage service protocol. `reserve`, used only by the master of a replica group, returns a range of previously unallocated object IDs. `clone` copies an object, and if it is a directory it also sets the copy-on-write flag for all entries in the new copy. All operations except `reserve` are stateless, requiring an object ID as well as operation-specific parameters.

An object ID is considered a globally unique name for that object and all replicas are identified by the same object ID. One object server from a replica group is nominated as the master, with the added responsibility of allocating available ranges of object IDs to individual envoy instances.

The remainder of this section describes the protocol used by the storage servers, and discusses the implementation of the object store.

5.3.1 Protocol

The storage server protocol is implemented as an extension of the 9p protocol. It uses the same RPC mechanism and shares the same hand-shaking messages. Like 9p, it does not define an authentication protocol, but instead provides a framework for implementations to exchange credentials. In its intended setting, authentication could also be based solely on IP addresses, since the entire network is controlled and virtual machines can easily be prevented from spoofing their addresses. Table 5.2 lists the messages unique to the storage server.

The *clone* operation, which copies an object from one given ID to another, is the only procedure that differentiates regular file objects from directory objects. When the attributes indicate that an object being cloned is a directory, the storage server will expect it to follow a particular format (described in Section 4.2.3) and will set the copy-on-write flags within each block as it makes the copy. It is not essential that this functionality be part of the storage service; indeed, the clone operation itself could be implemented in terms of the other procedures. It is merely a performance optimisation to avoid extra network round trips when possible. When a clone operation requires copying a file from one storage server to another (when the new object ID is allocated to a different server than the existing object) a more conventional sequence of reads and writes must occur.

The other operations are straightforward, though it is worth noting that they differ from the related 9p operations in an important way: they require an object ID as a parameter instead of an active file handle. Envoys can make and break connections to storage servers as needed without losing any state. A pool of connections to frequently used storage servers is an obvious optimisation, but is not strictly necessary.

5.3.2 Implementation

The storage service is implemented as part of the same executable as the envoy service. The two must be run as separate processes even on the same virtual machine, but since they share an executable image the operating system can map the read-only segments of the executable to the shared memory blocks.

Combining these two services in one implementation is convenient because they share a lot of code. The protocols for 9p clients, the storage service, and the envoy service are implemented as one single set of messages, only some of which are considered valid for each connection type. This allows shared code for network management, concurrent message dispatch, and numerous lower-level libraries. The code for storing objects on disk is also shared by the envoy service for managing the persistent cache.

Disk layout

Each object is stored as a file in a normal Linux file system. Objects are stored in a hierarchy of directories, arranged to create a radix tree indexed by the object ID. In this scheme, a 64-bit object ID is broken into a series of smaller bit sequences, each of which (as a hexadecimal string) names a directory in the path to the object.

The least-significant bits of the object ID are used in constructing the file name. Each node name in the tree need only identify the few bits that distinguish the branch of which it is the root—the entire path taken together identifies the full ID of the object.

The number of bits partitioned into each directory level is significant. It is chosen to be the largest value n such that a directory with 2^n entries named by hexadecimal strings encoding n bits each will fit in a single disk block on the underlying file system. This structure turns the file system into a crude radix tree with block-sized nodes, and uses the OS buffer cache for the indexing structure in place of a custom cache.

The bit-splitting process starts with the least-significant bits and proceeds to the most-significant bits, ensuring that if any level of the index has a smaller fan-out factor than any other, it will be the root. This is for two reasons: squandering space in a single root block is preferable to wasting an equivalent factor of space in every leaf node, and less critically, changing the number of bits in the object ID for an existing object store can be done by manipulating a few entries at the root and leaving all other nodes untouched.

Directories and metadata

The number of bits from the object ID stored in the file name at the leaf node is also different from the higher-level directory nodes. While some of the object attribute fields are provided by the backing file system, others fields are stored as strings in the file name itself. File names are of fixed length, and encode both the bits that distinguish the object from its immediate neighbours and object attributes that are not easily stored as file attributes.

As with the interior nodes of the radix tree, the number of files in a leaf directory is chosen to keep the directory in a single disk block. When the system looks for an object, it reads the entire directory that will hold that object and caches the results. This gives it the precise file name for the object (along with the metadata stored in that name) and also primes the buffer cache with the directory block, so when the object file itself is accessed it will be located through the cached directory. In this way, extra attributes can be accessed from the file names without incurring extra disk seeks.

The attributes stored in the file names are the file mode and names of the user and group that own the file. The mode (including access permissions and the file type) cannot be freely read from and written to in its entirety in normal file systems, so encoding a device or a directory as a normal file requires storing the type somewhere else. While access permissions could be stored in the standard mode attribute, they would be honoured by the underlying file system and would prevent unfettered access by the storage manager, particularly when it runs as an unprivileged user. User and group names in Envoy are stored symbolically, while most Linux file systems store them as numeric IDs, and again the file system's recognition of their semantic meaning would interfere with the storage manager. Storing any of these fields as a prefix to the actual file contents would violate block alignment assumptions that many clients make about file data, so an external mechanism was desirable.

While access to files in the object store is stateless, the access patterns of normal file system use are applicable to objects. Because the persistent cache only stores complete files, entire files are typically read from the storage server in sequence, a common pattern for client operations as well. The storage service caches open handles to the most recently accessed files, both to avoid having to re-open them on each request and to avoid giving the backing file system a false hint that the file is no longer needed. Open file handles are kept on a strict least-recently-used basis, as are the cached lists of file names in the leaf-node directories mentioned above.

Caching

Envoy considers a write request finished when it is in the memory of the storage servers for all replicas. While this opens a window during which the data could be lost by a system crash, it is unlikely that multiple servers would crash during the same window (assuming that nodes in the cluster are sufficiently isolated from each other). The storage server immediately passes all requests through to the underlying file system, and relies on the OS buffer cache to implement delayed writes. This is the simplest approach to implementation (use someone else's) but is also a sensible choice. Little would be gained by a custom solution, and this approach takes advantage of continuing improvements in the operating system. Indeed, taking advantage of continuing improvements in commodity hardware and software is one of the primary motivations for this work.

5.3.3 Limitations

The storage server implementation is meant to be as simple as possible while still supporting realistic workloads. With the intention that a production design would take advantage of other work in object storage systems, the prototype neglects several important characteristics of a complete system.

Despite the absence of these essential features, the storage prototype does implement replicated storage with all the characteristics necessary to support the envoy layer. As that is sufficient for the stated purposes of this dissertation, the prototype is also considered sufficient.

Security

While the mechanism for authenticating envoys at connection time exists, no authentication is attempted in the prototype. From a performance standpoint, a challenge-response scheme or something similar would be a startup cost that is ignored in the prototype. It would be largely hidden by even a moderately-sized connection pool, but it would be present. In a controlled environment like a service cluster, a scheme based on IP addresses would be faster, simpler, and nearly as secure. IP addresses could be assigned according to the role of the owner, so traffic from client VMs could be easily identified by all, and the VM hosting envoy and storage services could employ a firewall to prevent storage servers from ever seeing client requests.

Replica groups

The top half of the storage layer, i.e., the part that maps object IDs to storage servers, is essentially absent from the prototype. For any large deployment, a suitable disbursement of objects across storage servers is essential. Different degrees of failure protection may be desirable as well, with increased replication being available to clients at a premium cost.

Redistribution

When nodes are added or removed from the cluster, objects must be moved and copied to maintain a suitable replication factor. While this could be controlled by a centralised service, no mechanism currently exists for copying directly between storage servers (other than systems such as `rsync` that bypass the storage service and go directly to its object store). These capabilities are critical to a system that will change over time, which is a basic characteristic of any realistic cluster deployment.

Failure recovery

Recovery of a crashed envoy node is addressed in Section 4.5, but recovery of a failed storage node is an equally important problem. While redundancy allows the envoy layer to continue serving files, changes made during the downtime must be propagated to the restored node to ensure consistency, and nodes that do not recover must be replaced or their stored objects redistributed.

These basic requirements of an object storage system are best addressed in a coherent design. The reader is referred to the work described in Section 2.3.5 for examples of how this could be done.

5.4 Envoy service

The envoy service runs on each node to form a single, distributed service across the cluster. It exports a view of the global namespace by acting as a 9p server to VM clients running on the same physical machine. Each instance assumes responsibility for specific territories, or branches of the namespace tree. The name *Envoy* comes into play in two respects: it negotiates contact between local clients and the larger system, and acts as representative for its local territories to other nodes in the cluster and their constituent clients.

This section starts by describing the protocol used for inter-envoy communication, and then proceeds to discuss some internals in the envoy implementation. This includes details about basic file operations and navigation between envoy instances, the copy-on-write mechanism behind the fork and snapshot operations, and the coordination of state during territory boundary realignment.

For operations that require the synchronous cooperation of multiple envoy instances, dependency cycles and deadlock become a concern. To minimise these issues, Envoy is designed with a top-down locking protocol where synchronous operations only directly involve immediate neighbours

Message name	Description
`snapshot`	Take a snapshot of a territory.
`nominate`	Request that the parent transfer a territory.
`grant`	Give the target control of a territory.
`revoke`	Reclaim control of a territory.
`migrate`	Inform an envoy of file handles that have moved.
`walkremote`	Change directories across a territory boundary.
`statremote`	Query file attributes across a territory boundary.
`closefid`	Release a file handle that has walked to a new host.
`renametree`	Inform descendents of a directory rename.

Table 5.3: The messages in the envoy service protocol. The first five support the major state management operations, while the remainder handle cases where an operation affects multiple territories.

in the tree of territories, and owners of parent territories always initiate and coordinate transactions with those lower in the tree. While operations with wide-ranging impact may require communicating with every node in the cluster, they never require a cycle in the connectivity graph and are not prone to distributed deadlock.

5.4.1 Protocol

As with the storage service, the envoy service protocol is implemented as a series of extensions to 9p. The integration is closer at the envoy level, however, as forwarded transactions from remote clients use the standard 9p messages when appropriate, depending on the extensions listed in Table 5.3 for territory management and to handle operations that straddle territory boundaries.

Envoys establish connections with each other only when they need to, specifically when they own neighbouring territories or when transactions from one envoy's clients involve a territory owned by another envoy. Neighbouring territories always form a tree structure, with one territory in each pair (and by extension its envoy) being considered the parent of the other.

New connections between envoys must be authenticated in a way similar to connections between envoys and storage servers. The handshake messages from 9p are used, with the protocol string in the `version` message identifying the connection as one between envoys so that the appropriate range of messages will be understood. Authentication can proceed through an exchange of credentials, or the controlled environment of the service cluster can be exploited and network addresses can be used to identify authorised service hosts.

9p messages from Table 5.1 are all considered valid coming from other envoys, with the exception of the `walk` and `attach` messages. They are replaced by an augmented version of `walk` called `remotewalk` that accommodates territory boundary changes. Transactions still proceed in relation to normal file handles, with the further restriction that an envoy will never act as a proxy for another envoy, only for clients.

5.4.2 Data structures

Unlike the stateless storage servers, envoy instances must track two main classes of state related to the two main roles an envoy assumes, namely those of 9p file server and territory owner.

Territories and claims

An envoy may own zero or more territories, each of which is a branch of the global file tree, possibly with exits to child territories owned by other envoys as depicted in Figure 5.1. When a territory is given to an envoy through a `grant` message, it is also given the object ID that currently backs the root directory or file (a territory can be a single file) and a flag indicating if the object is writable. A territory may cover a snapshot image, in which case all objects are read-only. The root of the territory is never flagged as copy-on-write, so thaw operations (described in Section 5.4.3) can always operate locally.

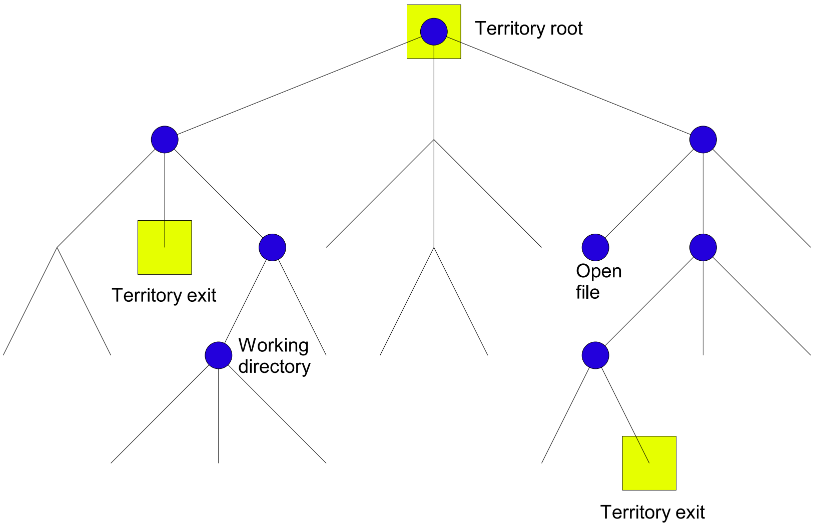

Figure 5.1: Territories are trees, and their boundaries (yellow squares) are defined by the territory root and the exits, which point to child territories owned by remote envoys. Claims (blue circles) represent individual storage objects, and within a territory they form a tree overlaying a subset of the territory, with leaves at each territory exit and at each active file or directory.

Overlaid on the file tree within a territory is another tree of *claims*, or references to storage layer objects. Claims are maintained for all file handles, territory exits (or more precisely, the immediate parent directories of exits), and the path from any other claim back to the root of the territory. It is possible for multiple claims to exist for a single storage object, but only when it is named by multiple nodes of the file system tree. In this case the claims must all be read-only or copy-on-write, and thus the associated objects are read-only, making coordination for accessing the object unnecessary. Despite being referenced by multiple claims, the file cache will recognise it as a single object and avoid duplication

File handles

File handles in 9p are identified by small positive integers picked by the client, so the envoy interprets them in relation to the incoming connection. This gives each client a separate ID space, preventing confusion when resolving file handle IDs to file handles. All client handles represented by a

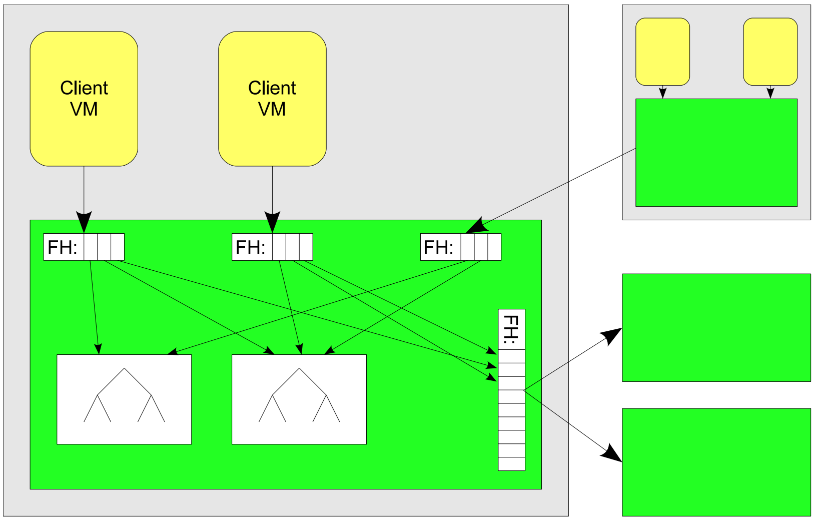

Figure 5.2: File handles for incoming connections are grouped and distinguished by the source, be it a client VM or a remote envoy. Each handle is tied to a specific claim in a local territory. A Client handle may also be forwarded to a remote envoy, in which case it is assigned an entry from the envoy's single remote file handle pool.

single remote envoy are merged into a single ID space, so the envoy receiving the requests treats the entire envoy and all the clients it represents as a single client, as illustrated in Figure 5.2. Since user credentials are associated with individual file handles, this does not confuse security handling. Envoys-as-clients are not quite like regular clients, as their file handles are never re-forwarded; instead, when a file handle moves to a new territory the handle is sent back to the original envoy, which can forward it directly to the appropriate target.

When forwarding client requests to remote territories, an envoy remaps the client ID to an envoy-specific remote ID. A single pool of remote IDs serves all outgoing connections, regardless of their targets, which simplifies territory migration as described in Section 5.4.6. The envoy also maintains a local file handle stub for remote files, which is kept up-to-date by peeking at the requests and responses for remote operations. In this way, state for local clients is lost when an envoy fails, but remote envoys preserve the state for their clients and can use it to recover from the failure.

Persistent cache

The persistent cache is the other major source of state in an envoy. On-disk objects are managed using the same code as the storage service uses for managing the object store, with the same disk structure and the same reliance on the underlying file system for in-memory caching.

The significant difference between the cache that an envoy maintains and the object store tended by a storage server is that the envoy must determine which objects are properly synchronised with the storage layer. Validating an unknown entry (one found in the persistent store but whose status is unknown) can be accomplished with a simple metadata comparison, so it is worthwhile keeping old objects even when the envoy has ceded the territory that referenced them, as it may eventually come back. Objects in the cache are removed immediately when the same object is deleted from the storage layer, but otherwise objects are cleared using a least-recently-used policy. Envoy guarantees that objects referenced by a local territory will never be changed by a remote envoy, so once an object has been validated in the context of local territory, it's validity need not be refreshed until the object has left local control.

5.4.3 Freezing and thawing

Synchronisation at the object level is governed by an important invariant: an object in the system may be referred to by exactly one name in the hierarchical namespace, or it must be read-only. The storage layer makes no attempt to detect or enforce the read-only case, so it is left to the envoy layer to ensure that this invariant is preserved. To make this straightforward, objects can transition from being writable to read-only, but they can never go back. Note that this refers only to objects in the storage layer, not to the access control of the file system interface.

The same invariant makes cache management simple: the envoy that owns a given territory can cache it without any invalidation concerns, and read-only objects can be safely cached at any number of envoy nodes without fear of interference. Objects in the cache become invalid only when a territory boundary changes, but even then objects that are part of a ceded territory are not explicitly flushed from the cache. Keeping them active helps in two cases: a read-only object may still be referenced by another name in a local territory, and the cache entry (in-memory or on-disk) may still be useful if a file backed by that object is later returned to local control. In the latter case the object must be verified to match the version in the storage layer, but this can be done with a lightweight metadata comparison.

The single mutable/multiple immutable dichotomy in the storage layer is tracked in the envoy layer through the copy-on-write flag in directory entries. The immutable property is not directly assigned to objects, but instead is imbued by the link from directory to file object. Furthermore, when the immutable object is itself a directory, the property is applied recursively to all of its children, overriding the individual copy-on-write flag in the link to each child. To be regarded as mutable, an object must be reachable by a path from the root of the global namespace to the directory entry linking to the object without traversing any set copy-on-write flags.

One immediate consequence of this is that taking a read-only snapshot of a directory and its descendents requires only setting the copy-on-write flag in the link to it, an operation known as *freezing*. Once a directory or file is frozen, the storage layer objects that back the branch rooted at that point are considered immutable. To simplify bookkeeping, this operation is only performed from the root of client images when a snapshot operation is requested, with an exception discussed in Section 5.4.5.

The complementary operation is called *thawing*. While the freeze operation works at the root of a subtree and affects it in its entirety, thawing aims to leave as small a footprint as possible; it is always performed with the goal of modifying a particular file. To thaw a file, the owning envoy starts walking up the line of its ancestors until it finds one that is already thawed. As the root of the namespace cannot be frozen (one can consider it as having a single, implicit link from the envoy service itself, but there is no mechanism provided to set the copy-on-write flag on this implicit link), this search is guaranteed to succeed. From there the envoy walks back to the target file, *cloning* intermediate directories as it goes. In addition to making a copy of the directory as its name implies, cloning sets the copy-on-write flag of every directory entry in the copy, effectively transferring the copy-on-write property from the single link leading into the directory to all the links leading out of the directory. The implicit property that each child inherited becomes explicit after being passed down a generation. After thawing each directory for mutability, the parent is updated to reflect both the new object ID and the cleared flag. Eventually the intended target itself (be it a file or directory) is cloned, its immediate parent link updated, and the file is fully thawed.

Thawing resembles the procedure for modifying a value in an immutable tree in a functional language. In addition to changing the value itself, the path from that item to the root of the tree must be copied if the item is to be reachable from the root. In the thawing operation, it is only the root of the immutable subtree in which the item resides that must be copied. While the analogous procedure in a functional language copies everything exactly except for the path being changed, the thaw operation must clear the copy-on-write flag as it goes, and thus must push it down from parent link to sibling links at each level in order to preserve the immutable status of the unaffected branches.

The prototype clones an entire object, but it could be optimised to store deltas or otherwise compress changes, especially for directories and other metadata [Sou03].

5.4.4 Read operations

Most basic file and directory operations have a straightforward implementation. The need to cross territory boundaries makes some more complex, however, as they must coordinate with remote envoys to complete.

Reading files and attributes

The most straightforward operations are `read` and `stat`, which read the data and metadata of files, respectively. These requests can be filled using the data paths described in Section 4.2.5. The only complication involved is handling the `atime` attributed, which tracks the last time the data from a file was accessed by a `read` or `write` operation. Read operations always proceed through a single envoy, so this attribute can be tracked accurately: the envoy can send a message with the time stamp (to ensure that it is consistently applied to all replicas, even if all clocks are not in sync or network latencies between the different storage servers vary) to all replicas in the storage layer which can then update the attribute.

The intended meaning of the `atime` attribute becomes obscured when applied to frozen files, however. Since attributes are part of the object, akin to inodes in traditional Unix file systems, changing the attribute will affect all files that are backed by that same object. Those instances may be in read-only snapshots of the same file system image, or in common files available in an unrelated file system image. In the former case, the read-only property of the snapshot is broken, and in the latter case the isolation of the two images is compromised. Updating the `atime` attribute would give an unrelated client using the same template file system image the ability to loosely track file accesses, which could represent a security risk.

Accurately tracking the `atime` attribute is problematic with frozen files, and if it is only updated on some files (the set of which may change over time as successive snapshots are taken), it is too unreliable to be useful. It also generates network and disk traffic for every read, even those that can be satisfied from the cache. For these reason, the `atime` attribute is present in Envoy for compatibility, but it effectively mirrors the modified time attribute.

Reading from directories

Reading from directories introduces two complications. The first—that successive `readdir` requests require state that must be transmitted from the client each time or transferred to another envoy when territories are realigned—is just an implementation issue. The second—that a directory may span multiple territories—requires the cooperation of all affected envoys. The list of contents for a single directory is considered an atomic unit when drawing territory boundaries, but the files and directories named may be remote. If the client-server protocol used to export the file system to a service only requests names, then the implementation footprint resembles that of file reads. If the response includes attributes as well—as with 9p—additional requests must be forwarded to the respective envoys for all files that are across a boundary in order to ensure consistent results. These requests are sent directly by the envoy that owns the directory, so if the client is remote, `readdir` may require the cooperation of three or more envoys to complete a single request.

Multi-step directory navigation can also create a star pattern of requests, but they always centre around the client's local envoy. Navigating down a series of directories—*walking* in 9p parlance—may involve crossing a territory boundary at each step in the extreme case. Because an envoy confines itself to knowledge of its local territories and their immediate boundaries, it cannot always predict the endpoint envoy for a navigation. Even if it could, intermediate steps must always be taken to allow permission checking at each level. When a remote territory must be consulted, the client's envoy forwards all remaining steps in the `walk` request to the remote owner, which proceeds as far as it can with the navigation. It may return one of three results: if the result is successfully completed, the two envoys store any state necessary to handle future requests; if the result is a failure, an appropriate error code is returned; if the navigation reaches another territory boundary, the partial result comes back along with a pointer to the envoy that must be consulted to continue the navigation. The client's envoy then repeats the procedure with the remaining navigation steps.

Because these operations are all asynchronous, and because the navigated directories may be owned by remote envoys that are not even neighbours to the client's envoy, walk may encounter transient failures. Between the time a remote envoy returns instructions for forwarding the remainder of a request and the time the next step is attempted, the target territory may migrate to a new owner and the request will fail. Because the envoy that sent the forwarding instructions was an immediate neighbour of the target, its information must have been correct at the time, and careful coordination can ensure that by the time the target bounces the request back with a failure notice, the referrer knows of the change to its immediate neighbourhood. The solution is simply to restart the walk request from the beginning, which also handles problems that occur when walk operations overlap with recursive updates due to higher-level directories being renamed.

Generally, having territory boundaries closely aligned with demand benefits everyone, but if multi-step directory navigations are frequent, their performance will be hurt by frequent boundary crossing. Caching walk results can be done safely with a few precautions. First, note that a navigation that succeeds one time and fails another implies one of three things: one of the steps was deleted or renamed, permissions changed somewhere, or a different user requested the navigation. The prototype caches walk results keyed by the path traversed and the user that requested it. With the assumption that directory renames and permission changes are uncommon (particularly across boundaries determined by locality of reference—these changes are probably most common in a region being actively modified by a single user, not in higher-level directories whose descendents are in active use by different clients), envoys broadcast notification of such changes down the hierarchy when they occur, invalidating cached navigation results. The notification of permission changes is implemented as a special case of directory renaming where the directory is given the same name it had before, which is sufficient to invalidate the cached walk results.

Attributes at the endpoint of the navigation are always confirmed directly with the owner, as are all of the final steps that occurred on territory owned by that same envoy. With a hot cache, multi-step directory navigations are satisfied from the cache of the client's envoy and a single step to the envoy hosting the target of the navigation.

5.4.5 Write operations

Redundancy in soft state and dependencies in permanent state make write operations more complex. Some updates must be made in a specific order to minimise the damage done by a system crash, and others must be managed carefully to preserve the coherency of the cache and to keep metadata in different nodes synchronised.

Writing data and attributes

Like the corresponding read operations, writes to files and changes to file metadata are largely a matter of directing the request to the appropriate envoy. The first change to a file since a snapshot or fork will first require the file to be thawed, but subsequent changes can be made directly to the local cache entries and propagated to all replicas in the storage layer. As discussed in Section 4.2.5, changes are considered complete when they have been received by all storage servers, but not necessarily committed to stable storage.

To ensure consistent time stamps, the modification time is determined by the envoy that owns the file and transmitted to the storage layer along with the data being written. While the clocks of machines within a cluster can be synchronised within reasonable bounds, network latencies and variations in storage server load levels would make it difficult to rely on strict synchronisation for consistent time stamps. The envoy is a natural location for deciding on the canonical time for its territories.

Thawing a file requires walking back in the file tree until an already-thawed directory is found. To bound the complexity of this procedure and to avoid deadlocks, the root of every local territory is always thawed unless it is part of a read-only snapshot image. While thawing may require cloning multiple levels of the directory hierarchy, this confines the direct impact to a single envoy. It also preserves the top-down rule for synchronous inter-envoy operations by preventing an envoy from demanding a coordinated change from its parent in the tree of territory ownership.

Deleting files

The inode-like disconnect between directories and the files they name means that one envoy may own a territory consisting of a single file, while another envoy owns the directory containing that file. Most operations involve either the file itself or the directory, but not both, so it is obvious which envoy should serve the request. Removing a file or directory is a case where the possibility of divided ownership complicates the implementation.

Deleting a file affects both envoys in this case, but the operation needs to appear atomic to clients. The normal rule in Envoy is that the parent should drive bilateral operations, meaning in this case that the owner of the directory should coordinate the removal with the owner of the file. This is a mismatch with 9p, where removal is an operation initiated on a file, not on the directory that contains it.

A higher-level consideration ultimately drives the design of the delete operation, however. Removal can only succeed on files or empty directories, so the envoy that owns the target of a successful delete will have nothing left to own afterward. Instead of trying to atomically coordinate a multi-step operation between two envoys, the parent instead revokes the child envoy's ownership and reclaims it before proceeding.

Since 9p initiates removal with the file to be deleted and not its containing directory, this requires a minor slight-of-hand to maintain the top-down coordination rule. A `nominate` request is sent from the file's envoy to that of the parent directory, requesting that it reclaim the removal target. Since `nominate` operations are implemented strictly in terms of names (not 9p file handles) this request cannot trigger a deadlock, and the `remove` transaction running on the child envoy effectively becomes a passive observer while control of the territory is transferred. After the transfer is complete, the envoy aborts the transaction and starts it over. Since the file is no longer local, the envoy will either forward the request (if it happens to be the client's local envoy) or reject it, forcing the client's envoy to redirect it to the new owner. `nominate` requests do not return until all state has been transferred, so the restarted transaction can proceed immediately.

Deletes happen in four basic steps. The first checks that the file is a suitable candidate for deletion, namely that it is a file or an empty directory. The second verifies that the client has permission to remove it from the parent directory, and the third actually removes it. In the case where the territory ownership must change, the envoy owning the file can complete the first step, but it does not know if the second step will succeed and blindly proceeds to initiate the transfer of control. This could cause unnecessary activity if the permission check fails, but in practice the Linux driver used in the prototype does an explicit check before making the delete request, so the possibility of failure is remote: it would indicate an unlikely race condition and would not compromise the correctness of the operation even then.

The fourth step in deleting a file is to actually delete the object that backs it and reclaim the storage space. This is simple enough to do, but deciding when to do it is slightly more complicated. An object can only be deleted when no more names link to it, and to support Unix semantics it should also be preserved until all clients accessing it have closed the file. The former case is easy to detect: if the link to the file was marked as copy-on-write, it shares a link with at least one snapshot and should not be removed. Otherwise, it was created or cloned since the most recent snapshot and can be completely removed.

To support familiar Unix semantics, the envoy instance waits until all file handles are closed before deleting the file. 9p is stateful, so this is straightforward, but it does create a few corner cases that must be dealt with. A new file with the same name could be created, so the envoy cannot continue to track the object by its former name. Without a name the file does not belong to any territory, so it becomes an orphan. Opening a file and deleting it is often done to provide a temporary scratch file that will delete itself if the process dies, but it may still be long-lived. The envoy must still be prepared to thaw it if it was frozen at delete time, migrate it to another envoy in response to demand or to the owning envoy shutting down, and delete the storage layer object even in the event of a failure. The prototype does not address all of these issues, but it does support arbitrary use of the file until the last file handle is closed, at which point the object is deleted from the storage layer if appropriate.

Renaming files

Renaming a file exposes the same coordination issues as deleting a file. In 9p a rename request is part of a `wstat` operation, which treats the file name as a property of the file object rather than of the directory that contains it. With rename the issue is even worse than with delete, because up to three objects and three envoys may be involved: one owning the directory, one owning the file to be renamed, and the third owning an old file with the target name that will be implicitly deleted when the rename completes. To further complicate the issue, Unix programs assume that rename-with-delete will complete atomically. Note that the original 9p protocol does not support this (it calls for a failure if the target name is already in use) but the Linux implementation does support this arrangement, and many clients demand it.

As with delete, the prototype simplifies the problem by consolidating ownership of all participating objects under a single envoy before making any changes. Unlike the delete case, the renamed file exists afterward and may be a non-empty directory. If it was owned by a different envoy before, that envoy probably drives most of the traffic to it and annexing it into the parent territory may be a poor move for matching ownership with usage. One solution would be to transfer control back to the original owner after the rename completes. It would have to re-validate some cache entries, but they would still be mostly unchanged, and even the in-memory cache would still be usable after a single metadata check on each file. A second possibility would be to engineer a distributed version of rename for these cases, driven by the parent (as with all synchronous operations) and handling transient failures appropriately. A third approach (the simplest and the one taken by the prototype) is to assume that such renames are rare and do nothing. Correct behaviour is still maintained, and eventually traffic patterns will force the owner to cede the territory once again if appropriate.

Renames of directories present another difficulty that cannot be safely ignored. Territory boundaries are all managed using directory and file names, so renaming a directory that is ancestor to other territories throws management state out of sync. On the assumption that renaming high-level directories is relatively rare, the prototype handles them by recursively propagating rename events from parent territory to child territory and from owner envoy to client envoy, rather than trying to assign immutable aliases for directories or some other similar scheme. In handling an uncommon corner case, minor overhead is preferred to complexity.

In addition to updating territory and file handle names, these recursive rename messages flush the walk cache described in Section 5.4.4. While a more fine-grained invalidation could be implemented, in practice an occasional flush has very little effect on performance. As a special case, this message can be used to rename a directory to the same name, forcing a flush of the walk cache for descendent territories but not changing anything else. This is useful when directory ownership or attributes change and cached `walk` results may no longer be valid. Explicit cache invalidation for this uncommon case makes it possible to maintain an accurate cache indefinitely without any overhead for the normal case.

Hard links

Unix file systems allow multiple names to refer to the same file. While symbolic links merely contain the name of their target, hard links allow multiple links from directories to resolve to the same inode. File operations are coordinated internally with reference to the inode, not the file path, so handling concurrent access is no different than if the clients all opened the same file through the same name. Because inode numbers are only meaningful in the context of a file system, hard links are not permitted across file system mount points. This mechanism is widely used both for aliasing, where two names are given to the same file, and as a way of moving a file without copying it, where the original name is unlinked after the alias has been created. 9p does not support hard links, but the Linux extensions to it do.

While Envoy's structure of names and object IDs resembles the Unix system of file paths and inodes, it does not behave quite the same way. Access coordination in Envoy is done with reference to file names, not storage layer object IDs. When the latter are shared, the objects are assumed to be read-only and can be cached freely on different nodes with no attempt to synchronise access. Different references to the same object ID, even within the same file system image, may be split into different territories and owned by different envoy instances.

For this reason, Envoy offers only partial support for hard links: a file must be frozen before additional names can be linked to it. If future changes are made to the file through any of its references, the changed versions will be thawed independently and will diverge.

The implementation of the hard link operation is also unique in that the client's envoy intercepts the request before passing it on to the owning envoy if the link is being created in a foreign territory. The Linux extensions to 9p implement hard links by walking to the target of the link, then creating the new link as a file whose contents name the file handle of the referent. Full file handle state for the client is only available to the client's envoy—not to a remote envoy that owns the file—so it uses the handle to freeze the target file (if necessary) and re-writes the request to include the object ID instead of the file handle. The envoy (remote or local) can then link the new name to the object.

Hard links also complicate snapshot removal by allowing objects to be frozen outside of whole-image snapshots. If an object is created and a hard link made to it before a snapshot is taken, it's copy-on-write flag will make it appear just like objects inherited from a previous snapshot. If all links to it are subsequently deleted or cloned, the corresponding storage object will not be removed because the normal mechanism will assume a link still exists from a previous snapshot. To ensure that these objects are deleted correctly, a special file is created in the root of the image listing qualifying objects, i.e., those frozen to make a hard link possible. This hidden file is consulted at snapshot delete time to augment the list of files created since the last snapshot. The file is overwritten rather than cloned after a snapshot, and its contents ignored if the copy-on-write flag indicates it has been carried over from a previous snapshot.

5.4.6 Modifying territory ownership

Misaligned boundaries do not prevent correct behaviour, nor do they introduce a heavy performance penalty. The algorithms to trigger changes are designed to promote good long-term layout choices based on steady-state traffic patterns. Cache effects penalise excessive changes, while the low network latency in a cluster environment reduces the cost of relying on a remote envoy even when local ownership would be preferable. In centralised server systems, all file access is routed across a high-speed LAN to a server with many clients. In Envoy, this pattern is also quite serviceable, even when it is not ideal, and Envoy is designed to favour gradual, long-term changes over a quick response in short-term traffic patterns.

Realigning territories requires careful coordination between multiple envoys. In addition to agreeing on new boundaries, state related to active file handles has to be migrated and updated. This includes both the territory owner's state and that of the envoys representing clients, which need to be directed to the new owner. *Inflight* operations—those that have been initiated by a client but not completed—must also continue and succeed despite concurrent territory changes.

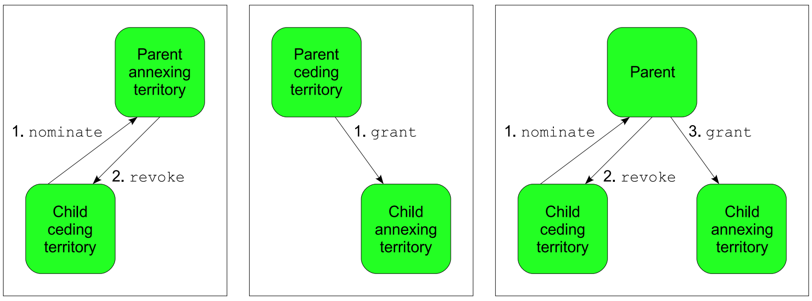

Figure 5.3: Territory migrations are always driven by the parent in the namespace tree, which can annex a neighbouring territory, cede territory to another envoy, or coordinates the transfer of a territory from one neighbour to another. For a child to initiate a transfer, it must send a `nominate` request to the parent, which then carries out the transfer.

Like other operations involving multiple envoy instances, territory migration is driven by the owner of the parent territory, as illustrated in Figure 5.3. If an envoy wants to merge a territory with its parent, or transfer it to another specific envoy, it sends a `nominate` request to the parent, giving the root path of the territory and the envoy to which it should be transferred. If a territory owner wants to cede part of a territory to another envoy or annex a neighbour that is a child of a local territory, it begins the process unilaterally. In practice, annexing is only done in response to a nomination from the child, but the process is always driven in a strictly top-down fashion.

Transferring state

When an envoy is granted a new territory, the first thing that it needs is a description of the boundaries. A territory is just a branch of the global namespace hierarchy, so it is uniquely identified by its root pathname. In addition to that, the parent envoy sends a list of *exits* from the territory. An exit is a link to an already-established territory that branches off from the one being granted. Since the boundary description may include any number of exits, multiple messages may be required to transmit the full set.

The new owner also receives a full set of active file handles for the new territory. With the 9p protocol, a file handle may identify an opened or unopened file, a file or directory used for `walk` or `stat` calls, or a directory in the process of being read. To facilitate territory changes and transparent crash recovery, the envoy must be able to recreate any state needed to continue a directory read given the number of bytes returned so far in the sequence of reads. Transfers that interrupt `readdir` sequences are uncommon, so the prototype simply starts over from the beginning and throws away enough data to find the appropriate place in the directory when the need arises.

Every element of state transferred to the new owner, including boundaries and file handles, has a counterpart maintained by a partner envoy that must be notified of the changes. The exit through the root comes from the parent, which already knows about the transfer (since it initiated it), but the boundary exits and file handles require that notifications be sent to their respective envoys. The previous owner is a special case, and since it is also identified as part of the `grant` message, the new owner can safely assume that the old has completely adjusted to the new order of things and not explicitly notify it of each file handle and shared border the two have in common. Some neighbouring territories and file handles may be local to the new owner; the goal of territory migrations is to bring the territory to the main user, so this is an expected and hoped-for case, requiring neighbouring territories to be merged and remote file handles to be made local. For all other cases, the grant recipient sends a notification to the other envoy immediately, grouping multiple entries where possible.

The other half of the process is having a territory ownership revoked. As with all such operations, the process is coordinated by the parent. The child responds to the revoke request by transferring all necessary state to the parent, whether the territory is being merged into its parent or transferred to a third envoy. In the latter case, the child is given a hint about where the territory is going to end up and is expected to update its file handles and territory boundaries accordingly.

A revoke operation works much like a grant operation in reverse: the envoy receives a `revoke` message identifying the root of the territory, and it responds by sending back the current borders and active file handles. It updates its own state for local file handles that become remote, but otherwise leaves notifications to the new owner as described above. This is important for synchronising the transfer with client requests, as detailed below.

Together with `nominate` messages, grant and revoke transactions give full flexibility for arranging territories. An envoy may cede part of its territory by granting it to a remote envoy and establishing a parent/child relationship with it. An owner may likewise cede its territory to the parent or to a third envoy by issuing a nominate request. Compound requests that carve up territories in more detail are not directly supported, though some could be simulated through sequences of transactions. If traffic patterns remain constant, future refinements will yield a similar result.

In general, complicated control decisions made by a central player are avoided in preference to localised choices. The owner of a territory is the only envoy that knows the recent history of requests, and is thus best suited to make realignment decisions.

Inflight operations

Top-down coordination of synchronised transactions helps prevent deadlock, but other synchronisation issues still exist in territory boundary changes. File operations may be at any stage of completion when the transfer begins, and all must complete successfully without interrupting the client.

The prototype is designed to favour simplicity over other virtues in complex transactions, and this is no exception. Each participant in a migration waits for all inflight transactions directly involving the territory in question to complete before locking it and queueing up any further incoming requests. When the transfer is complete, the lock is released and the queue examined. Significantly, a revoke transaction always completes

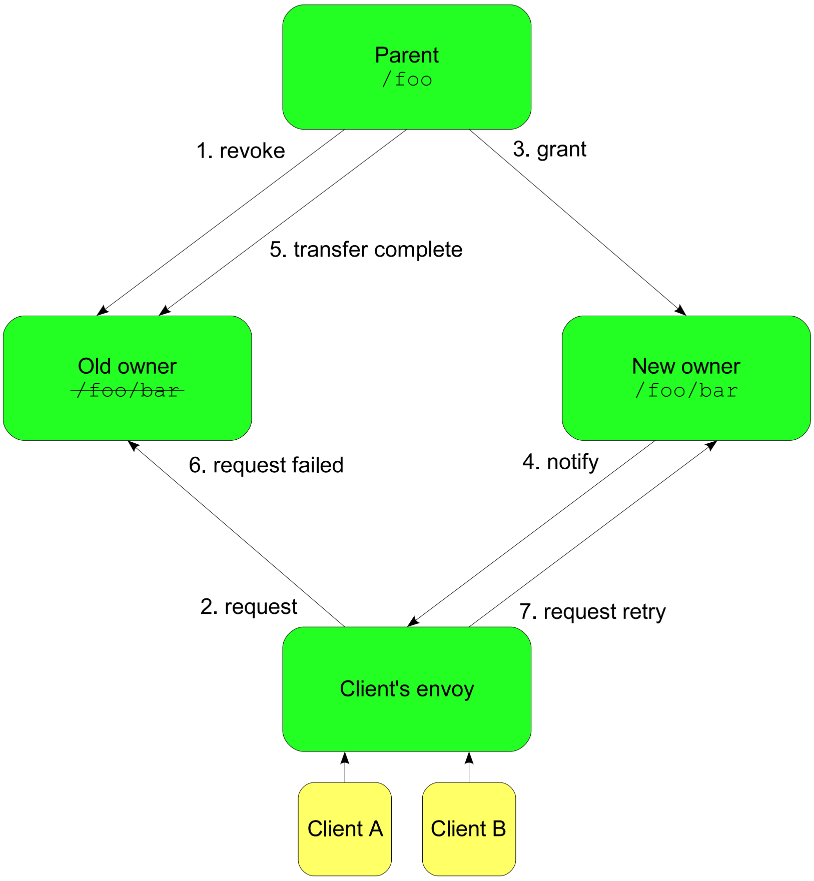

Figure 5.4: The sequence of events when a client file request conflicts with a territory transfer.

after a grant (when the two are paired), and only after all other envoys involved have been notified of the change.

For some requests this will result in nothing more than a pause, while for others the immediate consequence is a failed result. The envoy may no longer recognise a file handle that has been transferred, and can do no more than send the request back to its originator to try again. Forwarding it as a special case would be an option, but a dangerous one. The message could still end up at the wrong place due to bad timing (and a second transfer of the target), it would involve just as many network round trips (originator to old owner to new owner vs. originator to old owner followed by originator to new owner), and the new owner would reject the request anyway as coming from the wrong owner (recall that remote file handles are associated with the client's envoy).

Figure 5.4 illustrates a request that conflicts with a territory transfer. A request coming from a client's envoy that reaches its target before the territory is locked is allowed to run to completion. If it arrives after the lock, then it is suspended (this is true if the territory owner is the same as the client's envoy, or if it is remote). The revoke transaction only completes after the corresponding grant has completed and the client's envoy has been notified of the update, so when the old owner bounces the request back, the client's envoy knows the new forwarding address and can immediately retry the request with the new owner. If a request comes after a particular file handle has been updated but before the grant has completed, the new owner simply holds the request until the transfer transaction is complete, at which point it can safely start processing requests, even before the revoke operation has been finalised.

5.4.7 Limitations

The prototype include some limitations that are artifacts of the implementation, rather than being inherent to the Envoy model. These are problems that can be solved with some routine engineering, and would need to be addressed in a production system. For the prototype, however, solving them adds little value at the expense of extra complexity.

Envoys as failure points

The first issue is that a failing envoy loses all state for its local clients. This makes the failure of a single service affect all services on the same physical machine without an option for recovery. While services also depend on other node-wide management software, including the virtual machine manager, this is a dependency that could be removed.

The 9p protocol does not include any provision for recovering from a server failure, and the TCP connection used between the client and the envoy cannot be recovered transparently using standard network stacks. Both issues can be resolved by introducing a proxy between the client and the envoy (much as the envoy serves as a proxy for connections to remote envoys) that tracks the state of file handles, and is capable of restoring that state to a restarted envoy. Better yet, the client driver could be modified to maintain this state, and a recovery protocol introduced to synchronise and validate the state with a restarted envoy.

Migrating clients

A related problem is restoring state for VM instances that have been migrated to a different machine. With the current prototype, those instances will continue to send all requests to the envoy on the old machine, as they will not be aware of the migration. The envoy could easily transfer the state by employing a mechanism similar to that used to transfer territories, but a way to detect the migration would be necessary, probably through direct cooperation with the management process invoking the migration. The IP address of the new envoy would be different than the old, so some cooperation from the client would also be necessary to restart the TCP connection. Tricks involving the virtual network device connecting the client to the envoy may also be possible, but it is unlikely that the additional transparency gained would be worth the complexity introduced.

A related problem is virtual machines that are suspended to disk and restarted later, on either the same or a different physical machine. This can be viewed as another form of client migration, but from the client's perspective it more closely resembles the condition when the envoy fails and restarts. The solution proposed for that—clients retaining copies of file handle state and envoys validating and accepting it—would also be applicable here.

Control of administrative directories

For simplicity, territories control is only transferred starting at an image root, so all administrative directories are controlled by a single envoy.

Traffic to these levels of the hierarchy are normally confined to mount requests and administrative actions, and for the latter most of the work for snapshot creation and removal is still done by territory owners.

Still, nothing precludes this area from being split up. Tracking authentication credentials and resolving the target of fork requests would become a bit more complex, but not substantially so.

The more serious issue is that the prototype cannot migrate control of the root to another envoy. The standard mechanism for migrating territories is applicable, but in addition all envoys need to be notified of the change so that mount requests (which always start from the root of the namespace) could be properly routed. Straightforward solutions exist, but they were of little value in accomplishing the goals of the prototype and so were not pursued.

5.5 Summary

The Envoy prototype uses the 9p protocol to export the file system to clients, and uses extensions of 9p for its internal messages. The file system and cache system of the underlying operation system are both exploited to avoid duplicate work and to gain access to optimised systems.

The storage layer is simple placeholder that exports a stateless object interface. Objects are stored as files in the host file system, with extended attributes stored as part of the file name. Redundant storage nodes are supported, but object placement is hard-coded and no recovery facilities are implemented.

The envoy layer creates a file system from storage objects. Territories are a basic management unit, and the root of each territory is always made writable to prevent requests from involving multiple envoy nodes. Deadlock is prevented by following a top-down locking discipline along the territory hierarchy when multiple envoys are involved. Some operations require cooperation between multiple envoys, but others can be localised by changing territory boundaries before performing the requested action. Careful coordination ensures that dynamic territory re-alignments do not interfere with pending requests.

Chapter 6

Evaluation

Envoy is designed based on assumptions about a platform that does not yet exist. This limits how the system can be evaluated in two ways. The first is scalability. While common bottlenecks can be avoided and the architecture examined for potential scalability limitations, a system without a large-scale implementation can never be fully tested for issues that only appear at large scales. The best that can be achieved is to identify the most likely sources of problems and extrapolate the results of testing on a smaller scale. The architecture can also be evaluated against assumptions about the workloads it is intended to support, which leads to the second limitation: workloads cannot be accurately forecast. Predicting and simulating client access patterns is a difficult problem even for existing environments [Gan95], and the problem is amplified for service clusters. As a general purpose platform, service clusters are intended to support a wide range of existing workloads and to enable a flourishing ecosystem of computation services that create entirely new workloads. The design and the evaluation must necessarily rely on assumptions about how the system will be used, and those assumptions limit the applicability of the results.

In this chapter I evaluate the Envoy prototype with three principal goals: to measure the impact of specific design choices, including the basic distributed organisation and cache layout, to assess the scalability of the system, and to evaluate Envoy's ability to support the types of workloads expected in service clusters.

6.1 Methodology

The absence of a realistically sized service cluster limits the scope of system-level testing and the usefulness of many application-level results. The performance side-effects of interacting components and the access patterns of different clients are particularly difficult to predict. The performance characteristics of the storage layer depend on design elements and implementation choices beyond the scope of this dissertation. These factors combined with the poor public availability of many comparable systems limit direct comparisons to previous work.

Building and evaluating a prototype of the Envoy file system is not a futile exercise, however. The artefact may reveal little about real-world usage and behaviour, but measuring it can still do much to justify or condemn the design. This section starts by outlining assumptions about service clusters that influence the design, how the design reflects them, and how measuring the prototype helps to evaluate the decisions made. It concludes with a description of the testing environment and the benchmarks used.

6.1.1 Assumptions and goals

Envoy's success in achieving its stated goals is evaluated partly through argument and comparison to other work as described in previous chapters. Other aspects can be tested directly on a small installation and the results extrapolated to larger configurations. While not comprehensive, this approach is consistent with Envoy's role as one part of a complete service cluster environment. Further research and engineering will be necessary before service clusters can be fully evaluated on their own merits. Some of the assumptions that underpin them can be tested directly, while others must rest on arguments and an appeal to past and future work.

Independent clients

Though efficient sharing is important, individual clients accessing unshared data dominates most workloads. Even when data sets overlap, time may divide access from different clients, effectively yielding exclusive access that is transferred from one client to the next over time. Envoy seeks to capture these patterns by subdividing territories to distribute file ownership and control, and by dynamically updating the boundaries over time in response to changing usage. When this is successful, or when access is exclusive in the first place, Envoy resembles a simple client-server system serving files from an administrative virtual machine to a client virtual machine on the same host.

Since one of Envoy's goals is to create this intra-host configuration whenever possible, the evaluation starts by considering system performance under this circumstance. Maximising raw performance is not the focus of the prototype implementation, but comparing it to systems with a similar topology creates baseline expectations for a production system, and examining absolute performance figures helps confirm the sanity of the overall design.

Section 6.2 evaluates the performance of Envoy as a file server within a single host. Composite performance numbers are compared to other client-server systems, and tests using different configurations help to demonstrate where performance costs are embedded in Envoy. In addition, performance for remote hosts is evaluated to show that the drive to localise traffic benefits performance as well as scalability.

Shared images

Section 6.3 turns the evaluation to shared data. The emphasis of this section is not on performance—which is dependent largely on runtime topology as evaluated in Section 6.2 and the quality of the implementation—but rather on the behaviour of the system under shared workloads. In effect, Section 6.3 evaluates Envoy's ability to localise traffic, and Section 6.2 explores the benefits and costs that result from its success or failure.

Some aspects of Envoy's dynamic behaviour cannot be adequately tested in the limited environment available for this evaluation. The Envoy design prizes simplicity and stability in the layout of territories, both to simplify testing and recovery, and to encourage cache sharing even in the presence of runtime conflicts. The latter goal is achieved by delaying territory transfers until traffic is clearly dominated by a remote participant or until a more slight imbalance has persisted for a longer period of time. Resisting change allows the cache on the owning host to serve all participants at the expense of a network hop from the remote client to the envoy. Rearranging boundaries more eagerly may reduce network hops, but it creates some migration latency and the new host may have to prime its cache before service can resume at full speed.

While the parameters controlling dynamic behaviour can be configured, the ideal balance of stability and layout optimality can only be adequately determined with realistic workloads. Oscillating demand could be best handled by reacting quickly and varying the territory layout in lockstep with traffic changes, or being reluctant to change in order to maximise cache efficacy on one of the hosts may prove better. In the absence of a larger test system and realistic workloads to evaluate specific parameters, this evaluation relies on artificial traffic loads that can be controlled and varied to test the response of the system.

Scalability

Envoy's scalability is not evaluated directly. To do so convincingly would require a large cluster with a diverse client base, and direct testing on a reduced scale would prove little. The presumed scalability of the system depends on successfully localising as much traffic as possible. If all requests are satisfied by the same physical machine that hosts the client, then a cluster's capacity to host clients grows linearly with the number of machines added to the system. If most requests are handled locally and neighbouring hosts are consulted mainly to resolve runtime conflicts, then scalability is limited primarily by the degree of runtime sharing. Envoy's goal of scalability is based on a design that attempts to approximate that limit.

The storage layer is another source of traffic and inter-host dependence, but it is not fully specified in this work and cannot be evaluated fairly. A large persistent cache on each host reduces the direct load on the storage layer just as it does for AFS [Sat85], but by coupling storage nodes with envoy hosts and using switched networking, the overall transaction capacity of the storage layer should scale with the number of hosts anyway, and the number of hosts limits the number of clients. The systems considered in Section 2.3.5 have already demonstrated the scalability of similar storage architectures; conflict management and the coordination of metadata differentiate each system, but in Envoy that burden is not left to the storage layer.

Features

Rapid and inexpensive deployment of services requires support from the file system, and is one of the principal motivations for Envoy. The demands of a software ecosystem are difficult to predict, and Envoy's suitability for this environment is likewise difficult to assess. Features allowing rapid cloning and snapshots of file system images contribute to the flexibility of service clusters and are intended to promote the use of commodity software, but their actual impact in a production system is a matter of

machine	**druid**
processor	AMD Opteron 240 1.4GHz (×2)
memory	6 GB
disk	Maxtor Diamond Plus 9 160GB SATA/133 7200 RPM
network	Tigon3 BCM95704A7 10/100/1000BaseT Ethernet
machine	**skiing**
processor	AMD Opteron 250 2.4GHz (×2)
memory	4 GB
disk	Seagate Cheetah 10K.6 73GB Ultra320 SCSI 10000 RPM
network	Tigon3 BCM95703A30 10/100/1000BaseT Ethernet
machine	**moonraider**
processor	Intel Xeon 2.4GHz with Hyper-threading (×2)
memory	2 GB
disk	Western Digital Caviar 120GB EIDE ATA/100 7200 RPM
network	Intel 82545EM Gigabit Ethernet Controller

Table 6.1: Evaluation machines used throughout this chapter. druid is the root server for most tests, while skiing is the remote client for most comparisons. moonraider is used when more than two nodes are required, and both skiing and moonraider host the storage layer services except where noted.

speculation. The speed with which images can be cloned and new services launched is measured in Section 6.4.

6.1.2 Test environment

Three test machines, named *druid*, *skiing*, and *moonraider*, are referenced in this chapter. Each runs SUSE Linux Enterprise Server 10 configured to use Xen. Virtual machines are suffixed with a number, where zero is always the administrative VM that controls hardware directly. Other VMs access devices through virtualized interfaces. Each administrative VM is configured with 512 MB of memory, and client VMs are given 256 MB. The machines themselves are described in Table 6.1. All network connections use gigabit ethernet, and all three machines have more RAM than is allocated to the running virtual machines.

Clients access Envoy using the 9p client driver from version 2.6.18 of the Linux kernel, back-ported to the 2.6.16 kernel included in the Linux distribution. When the test requires it, client VMs are booted from Envoy, but for other tests they are each given a private partition. All services on each machine, including all VMs, Envoy's persistent cache, the storage layer object repositories, and file systems exported by NFS and 9p servers, are run from partitions on a single hard drive on each machine.

druid-0 is the root server for all Envoy tests, and the only server for NFS and 9p tests. Additional Envoy nodes run on skiing-0 and moonraider-0, and Envoy clients on each machine connect to their respective nodes unless otherwise noted. NFS and 9p clients on other machines connect to the server on druid-0 in order to compare network overhead and other effects that arise when using multiple machines. The storage layer in this setup consists of two storage instances hosted on skiing-0 and moonraider-0. Additional tests move the moonraider-0 instance to druid-0, but this is noted in the text when it happens.

Benchmarks

Bonnie is a benchmark designed to expose performance bottlenecks [Bra96]. Bonnie is used in this evaluation to provide some baseline performance numbers and to compare Envoy with other file systems. The Linux driver for 9p was also tested using Bonnie, and that evaluation compares some of the performance characteristics of 9p with NFS [Hen05]. Some similar comparisons are made here, but the emphasis is on Envoy and its architecture rather than the 9p protocol and its driver.

Bonnie runs in six phases, all involving a single file several times larger than the available memory. It starts by writing the file a character at a time, then it scans the file a block at a time, writing a change back to each block as it goes. The next pass reads the file a block at a time, then it rewrites it (without reading) using blocks, followed by a final pass reading the file a character at a time. Finally, it creates three child processes that seek randomly in the file in parallel.

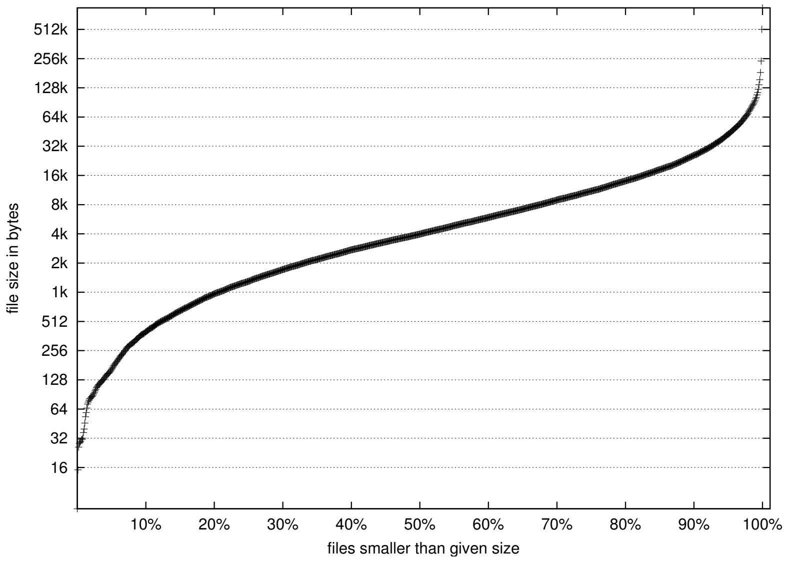

Figure 6.1: The distribution of file size in the Linux kernel source tree. 18,703 out of the 20,208 files are 32k or smaller and can fit in a single 9p protocol message.

Bonnie does not exercise metadata operations, and the character operations test aspects of the system that are not relevant here. Bonnie also fails to flush all buffers to disk between tests, leading to possibly skewed results in some cases. Nevertheless, it provides a useful starting point in evaluating Envoy that allows comparison with other published results.

To further explore the design of Envoy, a more controlled test is necessary. The Linux 2.6.18 kernel source tree provides a large block of data that can be used for various tests. Figure 6.1 shows the distribution of file sizes in the source tree. The kernel has 1238 directories, each holding an average of 16.32 files (not including subdirectories). The distribution is skewed, with a standard distribution of 31.95 and the largest directory holding 713 files. The median size is 8 files.

The largest file is 853k, but 92.6% of the files can be transmitted within a single 32k message using the 9p protocol. File sizes are significant particularly for read tests when files must be retrieved from the storage layer. Large files are read in parallel from the two storage servers, but for most files a single request to a randomly chosen storage node is sufficient, so parallel reads do not make a significant impact on the results. The prototype implementation also reads the entire file from the storage layer and writes it to the persistent cache before answering any requests from the client. For these reasons, latency will likely affect Linux kernel read tests more than transfer rates.

For write tests, the file tree is extracted from a compressed `tar` archive hosted on a local disk partition. For each test, the compressed archive is first loaded into memory to prime the cache. When compressed, the archive is small enough to fit in memory on the clients, and decompression is quick enough to have negligible effect on the results. The uncompressed archive is 229 MB, and the extracted tree occupies about 350 MB of disk space due to internal fragmentation.

Reading all the files back into a new (uncompressed) `tar` archive provides one read test. The archive is piped to `/dev/null` to avoid incurring any local storage costs. `rsync` is also used to read the files, again piping all results to `/dev/null`. `rsync` differs from `tar` in that it reads multiple files concurrently, reducing the overhead of latency from synchronous read requests. Both tests scan all directories and read metadata for each file in addition to reading file contents.

Three different cache settings are relevant for Envoy. The first is a cold cache, which requires retrieving all data and metadata from the storage layer. The second is a warm cache, where data is available from the node-local persistent cache, but not in memory. For cold and warm cache tests, all servers are restarted and all partitions remounted to ensure that in-memory buffers are empty. This also forces Envoy to verify persistent cache data with the storage layer before using it, but this is a quick metadata operation that adds little overhead to the results. Hot cache results pull data from the in-memory cache, but server processes are still restarted between tests and the storage layer must still be consulted to verify data validity.

For `npfs`, a multi-threaded userspace 9p server used for comparison testing, a cold cache environment corresponds most closely to the warm cache setting for Envoy. `unfsd`, a userspace NFS server, has client-side caching with three basic configurations, where data is retrieved from the server's disk, the server's memory, or the client's local cache memory respectively. In the final case, the server is not restarted and the client not remounted to avoid flushing the cache. Individual tests are accompanied by details of the cache status for each server.

6.2 Independent clients

Much of the complexity in distributed file system design is related to data sharing, but sharing is less common than individual clients accessing private data. A service derived from a stand-alone machine typically requires a private boot image, and lightweight fork operations in Envoy encourage the cloning of entire images for private use by related and unrelated service instances.

To succeed as a distributed file system on a large scale, Envoy must provide a useful service to isolated clients that obviates the need for additional storage services for most clients. While there is no technical or administrative distinction between private and shared images, Envoy's design and runtime behaviour under these two conditions can be evaluated separately. This section focuses on Envoy's performance and behaviour in the absence of sharing.

6.2.1 Performance

The prototype is not optimised, and absolute performance is not the focus of this evaluation. Nevertheless, it is helpful to establish baseline performance numbers to show that Envoy is viable as a file system. For context, the results are compared to `npfs`, a multi-threaded userspace 9p server, and `unfsd`, a single-threaded userspace NFS server. Userspace servers have additional overhead when compared to in-kernel implementations, but the penalty is assumed to be similar for all of the implementations tested. Standard benchmarks do a poor job of isolating the costs involved in Envoy's data paths, and most of the evaluation in this chapter eschews them in

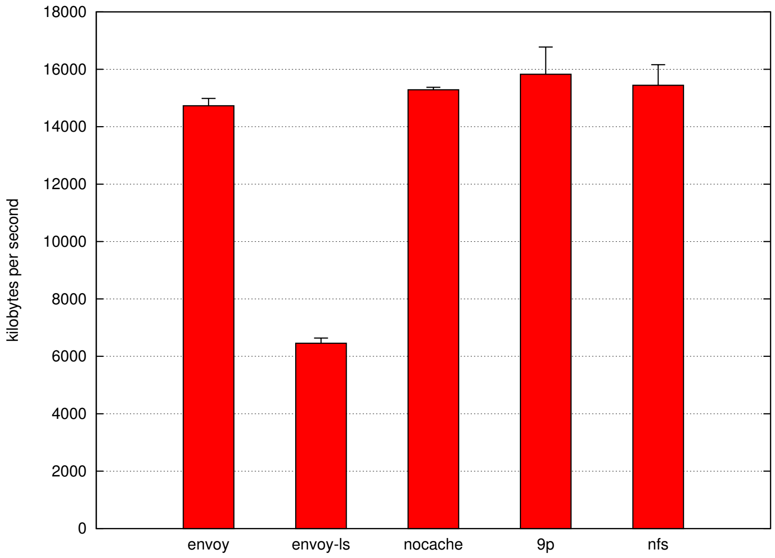

Figure 6.2: Bonnie benchmark results for block writes on a 2 GB dataset. Results for Envoy, Envoy using a local storage server instance, Envoy with no persistent cache, a userspace 9p server, and a userspace NFS server are compared. Error bars show the standard deviation over ten runs.

favour of more controlled tests, but this section starts with the Bonnie benchmark to provide some raw performance figures comparable to those from other systems.

Figure 6.2 shows the block write results of the Bonnie benchmark on a 2 GB dataset. Envoy is competitive with the 9p and NFS servers, despite writing to two storage servers and the local persistent cache. Bonnie does not flush buffers to disk after writing, so this represents a conservative result for Envoy. When the write is complete, Envoy has flushed the data to two storage servers and can tolerate a node failure without loss, while the 9p and NFS servers both have a dirty cache on a single node with no replicas.

For sustained writes, the disk is the primary bottleneck, as demonstrated by the result labelled *envoy-ls*, which has one of the two storage servers on the same machine as the envoy. In this case, data must be written twice using the same disk, once for the cache and a second time for the storage instance. In addition, the in-memory cache is split between the two functions, cutting its effective size in half. The *nocache* figure is

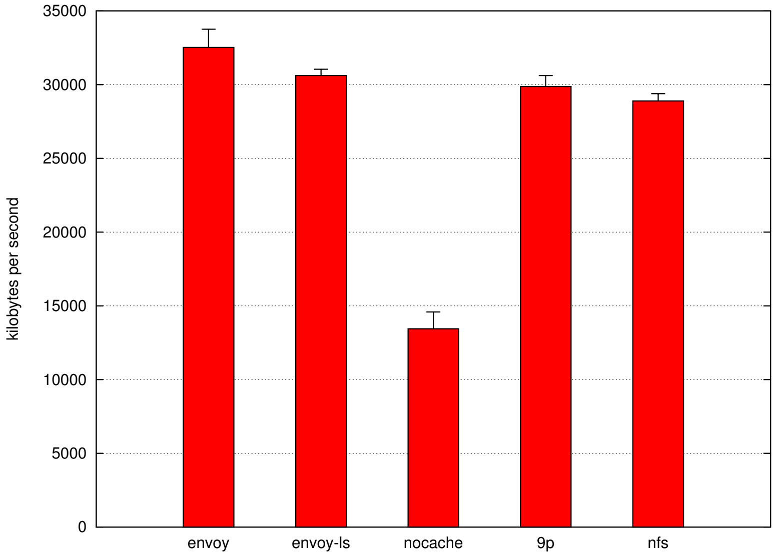

Figure 6.3: Bonnie benchmark results for block reads on a 2 GB dataset. Results for Envoy, Envoy using a local storage server instance, Envoy with no persistent cache, a userspace 9p server, and a userspace NFS server are compared. Error bars show the standard deviation over ten runs.

an Envoy server with the persistent cache disabled, a change that has little effect in this test because the writes are performed in parallel on each node.

Envoy nodes all host storage servers, and it is reasonable to assume that each node generates a similar amount of traffic on average, but sustained writes are not typical of average workloads. Storage is distributed throughout the cluster, so it is not likely that a node with a write-heavy client will also host storage for other write-heavy clients at the same time; envoy-ls represents a worst-case scenario rather than a common case.

Figure 6.3 shows the block read results from the same benchmark. Bonnie's test uses a single file, so the Envoy servers are reading data from a single file in the persistent cache. All of the tests except nocache involve transfers from a single file in one VM to a client in another VM, with similar overall results. As in the write tests, the results are dominated by disk speeds.

With no persistent cache, Envoy must fetch each block from one of the storage servers. The prototype limits files to one outstanding transaction

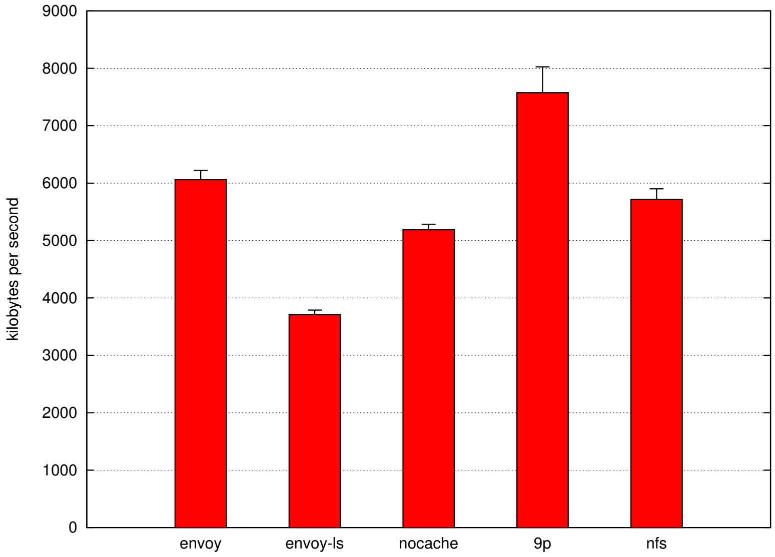

Figure 6.4: Bonnie benchmark results for block re-writes on a 2 GB dataset. Each block in the dataset is read, modified, and written back in sequence. Results for Envoy, Envoy using a local storage server instance, Envoy with no persistent cache, a userspace 9p server, and a userspace NFS server are compared. Error bars show the standard deviation over ten runs.

each, so parallelism is not exploited in this test, and requests are randomly distributed to the two storage servers, so neither ends up with a sequential read through the entire file. A more sophisticated storage layer would likely reduce the penalty of omitting the persistent cache, but in normal operation most data is expected to be in the persistent cache anyway.

Bonnie's re-write test reads the same file in sequence, writing back a change to each block as it is encountered, with results displayed in Figure 6.4. The Envoy results reflect the write penalty for a local storage server and the read penalty for no persistent cache, with no surprises. The poor showing from NFS relative to 9p is due to an artifact in the test that interacts with the client-side cache. The rewrite test is preceded by the write test with no explicit sync operation to flush the write cache to disk. All of the re-write tests start with a dirty cache, but NFS adds the client's cache to that of the server, leaving more writes from the previous test that are billed to the current one.

The results from testing with Bonnie show Envoy to be competent at serving files in a simple client-server configuration between two virtual machines. Results are comparable to a simple 9p or NFS server for operations on datasets larger than the in-memory cache size. In these tests, the speed of the disk appears to be the primary bottleneck, which is the expected result for basic operations involving bulk data transfer with few metadata operations. The remainder of this section explores the results in more detail to reveal the effect that the network and virtual machine architecture have on performance and how caching, both in-memory and on-disk, influences performance.

6.2.2 Architecture

An important goal of the Envoy design is to localise data and metadata control when possible, and minimise the involvement of disinterested nodes when not possible. Private images are always controlled by the envoy instance on the same machine as the client using it, and territories in shared images are managed by the most active participant. Besides reducing collateral impact in a bid to improve scalability, localising control and caching has direct performance benefits for clients.

As described in Section 4.2.5, requests follow one of several paths to completion depending on whether or not the required data is cached and whether ownership is local or remote. The impact of network and virtual machine layout is measured here, with four layouts. *druid-0* is an administrative virtual machine that hosts all server processes, and owns all territories in the case of Envoy. *skiing-0* is an administrative VM on a different machine, hosting Envoy server processes but no others. *druid-1* and *skiing-1* are client VMs on the two machines. These tests focus on hot cache results to prevent disk access time from masking other sources of overhead.

For the 9p and NFS results, druid-0 acts as the server in all instances, and results are included for all four client locations. Envoy clients attach to their local envoy instance in each case, so for clients on skiing, all requests are forwarded on to the envoy instance running on druid-0.

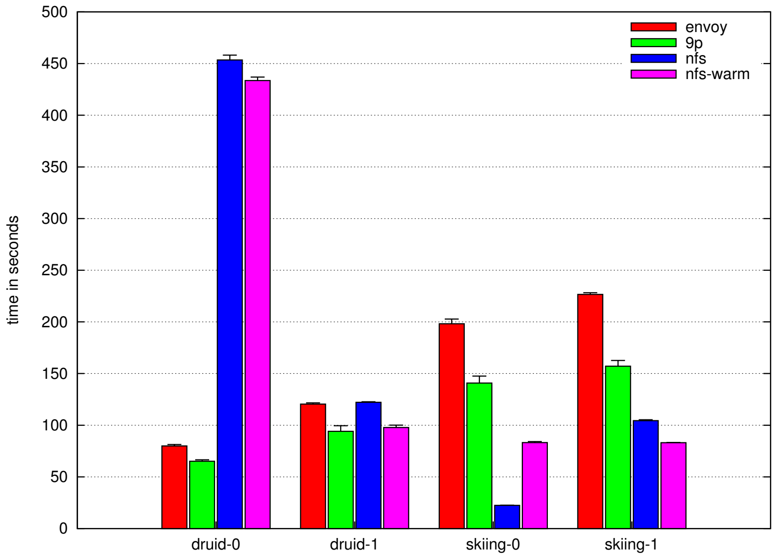

Figure 6.5: Time to `tar` the Linux kernel source tree with a hot cache over various data paths. NFS results are included for a hot client-side cache and for a hot server-side cache only (labelled nfs-warm). Error bars show the standard deviation over ten runs.

Figure 6.5 shows the time required to `tar` the Linux kernel source tree from each client location with a hot cache. In each case, the file data is available from the buffer cache of druid-0, so the differences are largely due to network, protocol, and implementation overhead. The Envoy and 9p results show a significant difference between druid-0 and druid-1 results, which comes from the virtual network device that connects the two VMs. Much of this overhead could be eliminated through a Xen-specific transport between the VMs combined with optimisations in the 9p driver to eliminate data copying, as suggested in Section 4.2.5.

Envoy is consistently slower in this test than in the Bonnie read test, which measured bulk operations from a single file. The `tar` test exercises metadata operations much more heavily, which exposes the extra overhead of a prototype-quality implementation of Envoy. For the skiing-0 and skiing-1 results, requests are considered and forwarded at the skiing-0 envoy instance, so overhead is inherent to the design. For druid-0 and druid-1, however, both 9p and Envoy are using the same protocol to deliver the same data across the same network paths, and the data is coming from the buffer cache in each instance. Optimisation of the prototype would probably result in something closer to parity between these results.

The NFS results for druid-0 are an exception caused by the client-side cache. Because the client and server must compete for cache space, the cache is not sufficient to hold the file tree and disk access is required. While seemingly an unfair circumstance for the NFS results, this result is included to illustrate the benefit of shared caching. Allowing a larger unified cache on each host instead of having each client provide its own smaller cache reduces this kind of cache duplication and increases the effective cache space available cluster-wide. While the local cache can improve performance, it can also cause more disk access and reduce both performance and the overall transaction capacity of the cluster.

A second anomaly caused by the client-side cache is the result for skiing-0, where the entire file tree fits in the client-side cache. The displayed result is only applicable when the test is repeated within the 30-second window in which the Linux NFS client considers data valid. The test completes in an average of 22.38 seconds and makes this possible, but if a delay is introduced between each iteration, the average time nearly doubles to 44.05 as each cache entry must be validated against the server.

Figure 6.6 supplements these results with the same tests with the in-memory cache flushed before each iteration. Data is resident in the persistent cache in the case of Envoy, so for each of the three servers the data is coming from the same disk. NFS suffers from the same double-buffering issue on druid-0, but otherwise its client-side cache proves very helpful for the metadata-intensive operations in this test. This is clearest in the jump to a remote machine, where 9p must consult a remote server for many operations that are absorbed by the client-side cache in NFS. If the optimisations already suggested prove insufficient, an approach like that of NFSv4—where a limited token-passing scheme lets clients access unshared files without fear of conflicts but reverts control of shared files to the server [She03]—may be useful for Envoy, allowing it to retain strong consistency guarantees while allowing local caching for unshared files. Testing with a

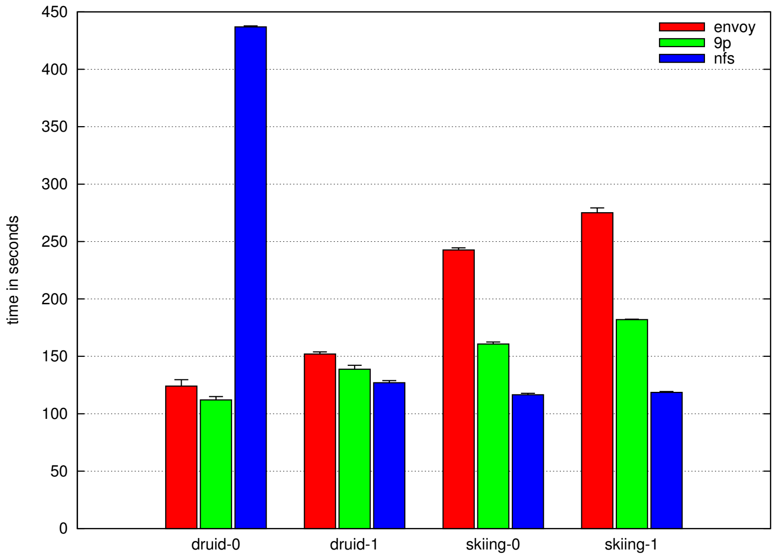

Figure 6.6: Time to `tar` the Linux kernel source tree from the persistent cache (for Envoy), with the in-memory cache cold for all servers. Error bars show the standard deviation over ten runs.

representative workload on an optimised implementation of Envoy would better inform such a choice.

These tests do show a clear performance benefit to coupling control of territories with the clients that use them, particularly in the difference between Envoy results for druid-1 and skiing-1. Comparing the two graphs also shows that Envoy is faster serving warm data controlled by the same machine than hot data from a remote machine, suggesting that territory boundaries can be relatively fluid without destroying performance.

Figure 6.7 compares using `tar` and `rsync` to read the Linux kernel source tree from cache using Envoy and 9p. `rsync` reads asynchronously from multiple files at one time, whereas `tar` reads one file to completion before starting the next. The increase in time required for both tests between druid-1 and skiing-1—which are the two important client locations—is reasonably consistent. Envoy sees a time increase of 88% for `tar` and 85% for `rsync`, while 9p sees 67% for `tar` and 63% for `rsync`.

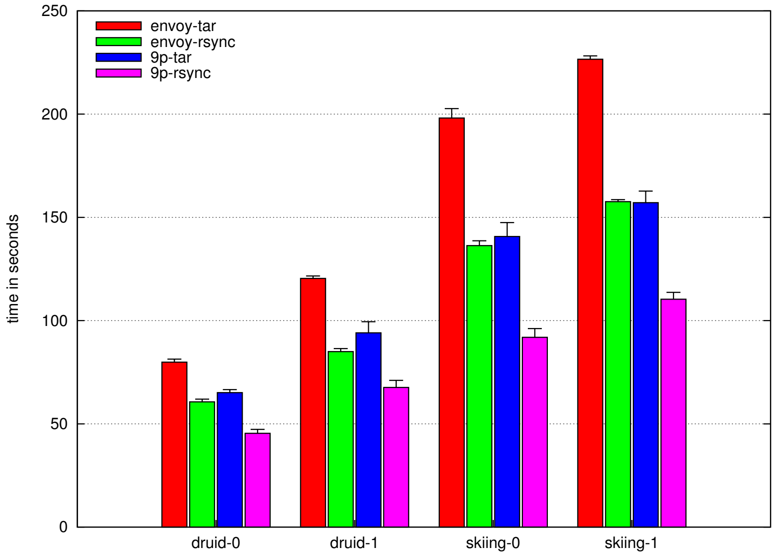

Figure 6.7: Time to read the Linux kernel source tree using `tar` and `rsync` after priming the in-memory cache with all file data. Error bars show the standard deviation over ten runs.

Connecting skiing-1 directly to druid-0 instead of to the envoy instance on skiing-0 yields increases of 40% for both tests over the respective druid-1 results, or a reduction of 34% and 33% compared to the same tests using the envoy on skiing-0. The average times—168.97 for `tar` and 118.71 for `rsync`—compare more favourably to the simpler 9p server, taking 8% longer for both tests.

Envoy uses the 9p protocol to connect to clients, and it also uses an extended version of it between envoy instances. Comparisons between Envoy and a simple 9p server using similar data paths and cached results demonstrates that the 9p server is consistently faster, but performance degrades at a similar rate for both systems as obstacles are introduced to the data paths. The 9p server passes most requests directly to the file system, while Envoy is a more complex system that tracks additional state and offers more features. Nevertheless, the correlation between the two optimistically suggests that the Envoy prototype could optimised to a performance level close to that of the 9p server without introducing any architectural changes. A roughly 33% overhead beyond network costs for remote transactions is imposed by the forwarding system, but this is a reasonable tradeoff for simplified recovery and the possibility of removing

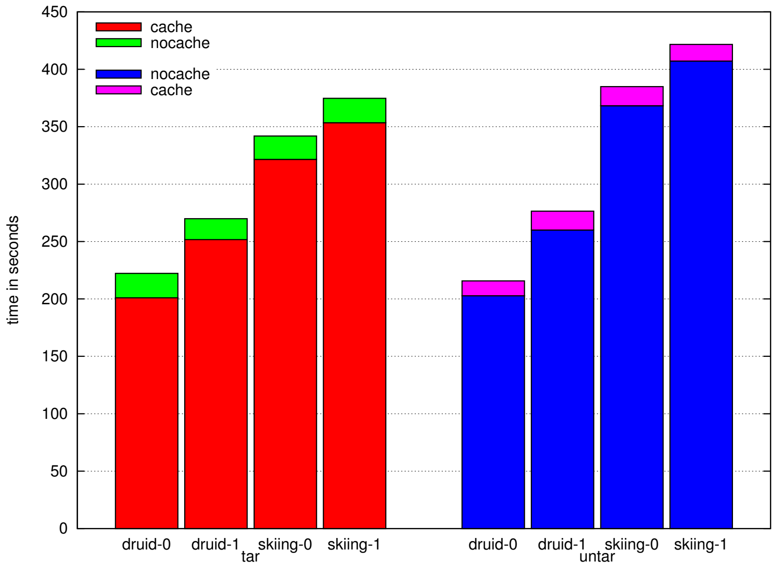

Figure 6.8: Cold cache read and write tests with the persistent cache enabled and disabled. `tar` times are faster with the cache, even though it requires writing all data to the local disk. `untar`, the write test, is faster with the cache disabled.

forwarding overhead completely through territory migration. Any consistent system will require coordination for overlapping requests; Envoy's territory scheme ensures that sharing can be coordinated and still benefit from an in-memory cache.

6.2.3 Cache

In addition to the data paths imposed by the architecture, Envoy requests interact with different cache states. When the cache is *hot*, data is available from the in-memory cache on the envoy that owns the relevant territory. A *warm* cache holds data in the local on-disk cache, but not the in-memory cache. When data must be retrieved from the storage layer, the cache is said to be *cold*, and for all tests the storage servers themselves are restarted and data partitions remounted to empty in-memory buffers.

For all Envoy tests, the server was restarted between each iteration, reseting the server's internal state. For both hot and warm cache tests, the server must initially validate cached objects against the storage servers, but can then proceed to use them from the local cache. Testing found this to

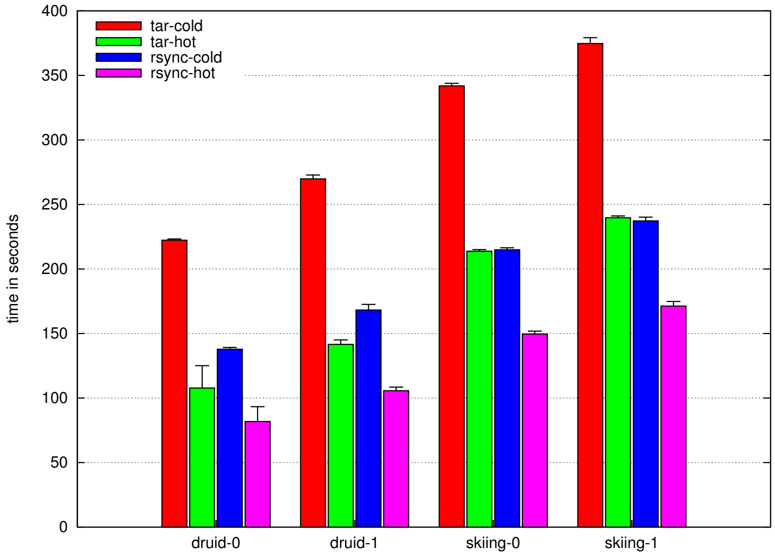

Figure 6.9: `tar` and `rsync` tests run with the persistent cache disabled. The cache status (hot or cold) refers to the in-memory cache on the storage servers. Error bars show the standard deviation over ten runs.

have little effect on the results, so all figures presented here use the more conservative method.

The persistent cache stores objects on the same node as the envoy that uses them, which is generally a performance win. For write operations, however, it simply adds another write destination for data that is already being sent to multiple storage nodes. The `untar` part of Figure 6.8 shows modest overhead for write operations, since the writes proceed in parallel with those on the storage nodes.

Reads in the `tar` test are faster with the cache, even though all data must come from the storage layer in either case, and enabling the cache requires writing everything to the local disk in addition to returning it to the client. This is because the disk is otherwise idle, and most of the write operations can be absorbed by the buffer cache and proceed asynchronously. In return, having the data in memory on the local node prevents additional read operations from being directed back to the storage layer. The in-memory cache is provided by the host operating system through operations on the object store, so disabling the persistent cache disables the in-memory cache as well. Even though the test is primarily based on write

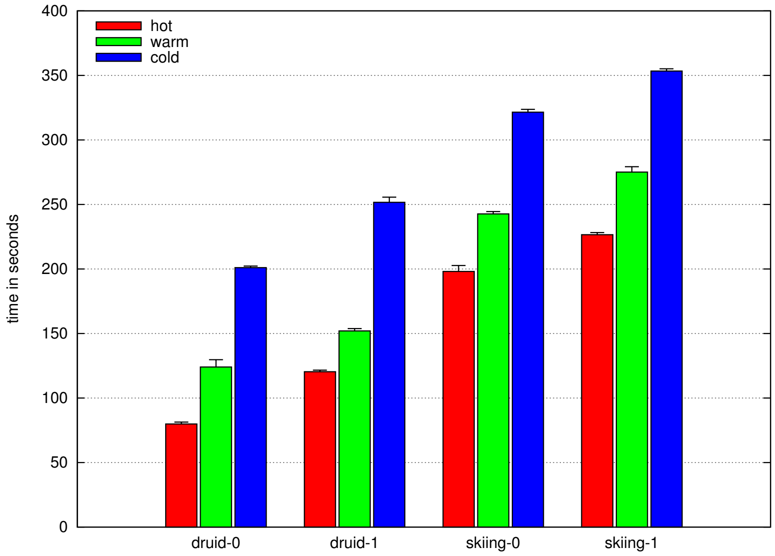

Figure 6.10: `tar` test results for hot, warm, and cold cache states on all client locations. Error bars show the standard deviation over ten runs.

operations, disabling the cache results in a 41% increase in the number of messages sent to the storage layer in this test.

Even with the persistent cache disabled, there are still other cache factors to consider. Figure 6.9 examines read test results with the persistent cache disabled, varying the status of the in-memory cache on the storage servers. As expected, a hot cache improves performance considerably. The improvement in cold cache results moving from `tar` to `rsync` is particularly dramatic, because `rsync` is able to schedule reads from both storage servers at once, whereas `tar` leaves one completely idle while waiting for the other.

Figure 6.10 compares the `tar` test run on Envoy's three basic cache states. As demonstrated earlier when the persistent cache was disabled, many requests are absorbed by the in-memory cache even when it starts out empty. Micro-benchmarks would probably show a greater difference between the three cache states, but reading a series of files is a common and realistic workload. In addition to the performance gained from caching objects on each host, the load on storage servers is reduced. Particularly when storage servers and envoy servers are hosted on the same nodes, localising traffic as much as possible helps both performance and scalability.

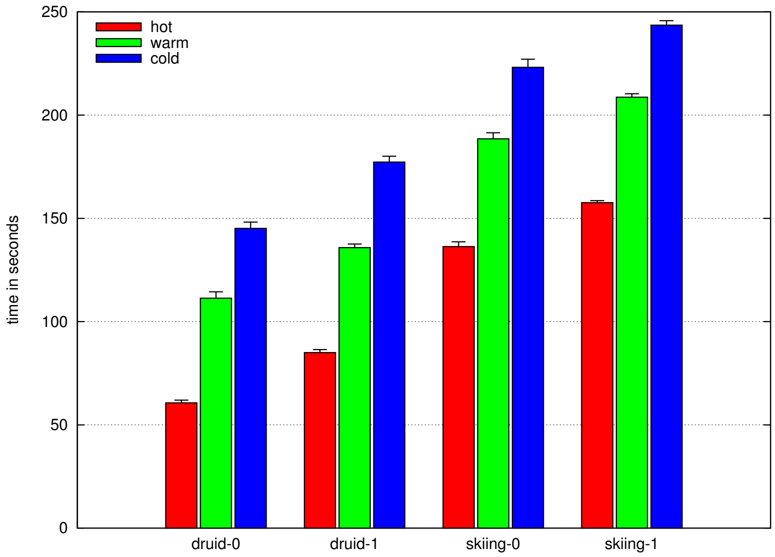

Figure 6.11: `rsync` test results for hot, warm, and cold cache states on all client locations. Error bars show the standard deviation over ten runs.

A cold cache is at less of a disadvantage over a warm cache using `rsync`, as demonstrated in Figure 6.11. Overlapping requests employ both storage servers in parallel, making better use of additional disk spindles. As with other tests, there is a clear advantage to accessing data on the local node. In the `rsync` results, a hot cache with a remote client (skiing-1) is faster than a cold cache with a local client (druid-1), but not by a wide margin. In the more common hot and warm cache scenarios, network latency and the cost of forwarding requests outweigh differences in cache performance.

Many cluster file systems use parallel transfers to increase throughput on large file transfers. Nothing prevents Envoy from doing so except for the current implementation. The Envoy prototype does transfer large files from all available storage servers in parallel when transferring them to the local persistent cache, but it transfers entire objects at file open time before answering any read requests. Because of this simplistic implementation, parallel reads serve to reduce latency at file open time rather than increase throughput at file read time.

If the transferred file fits in the buffer cache, then reads that commence after the transfer completes will find a hot cache and proceed quickly, but large files will incur an unfortunate performance penalty, particularly in the common pattern of sequential file reads. In addition to the lag of transferring the entire file to the local node, clients will have to wait for much of the file to be written to the local object store, and then sequential reads will create congestion on the disk as the file is read back into memory from the beginning and forces the remaining dirty blocks to be written to disk. The throughput advantage of striped reads will be completely lost.

A more refined implementation could cache partial results, and feed striped reads from the storage layer back to the client in real time, perhaps issuing read-ahead requests to keep the queue full. Similarly, reads that saturate the local disk could be supplemented with streamed reads from the storage layer, making use of multiple disks. More sophisticated write policies could also improve storage layer performance. With a large persistent cache at each node, a log-structured storage layer may be profitable, since write requests are likely to dominate storage layer traffic. Log-structured systems suffer under certain workloads, particularly when application semantics demand frequent checkpointing. With Envoy, writes are considered stable once they have reached enough storage nodes, so checkpointing would not need to interfere with the storage layer's bulk write policies as they do on local file systems. The results examined here show clear advantages to the simplistic persistent cache implemented in the prototype, but it leaves much room for improvement in conjunction with a fully-developed storage layer.

Client-side caching was advantageous to NFS in some tests, but proved little benefit in others. Envoy, which eschews client-side caching in favour of a larger shared cache at the machine-level, does surprisingly well in many tests despite the increased traffic to the envoy service. Some form of client-side caching would be essential for a production system, particularly when hosting shared libraries and other memory-mapped files. Achieving the right balance between cluster-wide cache capacity and performance at local nodes will require further research.

6.3 Shared images

When file system images are shared, Envoy manages territory boundaries dynamically as described in Section 4.4. Territories are split and merged in response to runtime demand, with control being granted to the machine that is generating the most traffic for a portion of the file tree. When the current owner generates slightly less traffic than a remote node, it waits between changes for a time proportional to the degree of imbalance in request traffic. Severely skewed loads trigger more rapid changes, while minor sub-optimalities are tolerated longer to establish a longer-term trend and avoid thrashing the cache.

The dynamic algorithm is configured by specifying the two endpoints of the urgency/delay line. The minimum delay between migrations is paired with the minimum traffic imbalance that will trigger a migration after waiting out that delay. At the other end, the smallest imbalance that will ever prompt a change is paired with its minimum delay. Urgency is defined by the number of requests to a region of the territory, decayed over time with a configured half-life.

Determining suitable values for these parameters requires knowledge of workloads and the demands of the specific environment. If stability is valued over a fine-grained territory layout, the system can be made more reluctant to enact changes. If workloads change more rapidly, a more eager policy can be adopted. The previous section compared the performance of local and remote clients, and concluded that local control yields a performance advantage as expected. Private images are always ceded to the client's envoy at mount time, so it is only in the presence of workloads with actual sharing that the dynamic algorithm comes into play.

The previous section explored the performance of the Envoy prototype under various condition, and this section turns to dynamic behaviour under shared conditions. Migrating active state between nodes is a quick operation, similar to the metadata operations discussed in the next section. The prototype also completes the operation that triggers a territory change before executing the transfer, so any pause happens between client requests. Write operations are always committed to the storage layer before returning, so no cache flush needs to take place, either. Cache effects will create some extra delays as the new owner loads active objects that were already in memory on the previous host, but the results of the previous section suggest that these will largely be offset by the improvements derived from local control.

6.3.1 Dynamic behaviour

A series of typical sharing scenarios were laid out and simulated, and the response of the dynamic territory algorithm observed. For the tests, a half-life of five seconds was used to give a gradual and predictable response to simulated traffic. An urgency of 100 triggers a change after five seconds, while a modest threshold of five triggers a change after 120 seconds.

To generate load, a script walks to and opens files at random within a subtree of the Linux kernel sources at a specified rate. Each operation sends multiple requests to the server, walking from the root of the tree (the Linux 9p driver generates a new requests for each step in the directory traversal), opening the file, and immediately closing it. The script can generate load in multiple trees and can ignore subtrees within a target region.

The prototype is instrumented to output a report after each transfer, including a graph of the complete territory structure at each end of the transfer. After being combined and plotted using the `graphviz` graph-drawing package [Gan00], the output represents a live snapshot of the territories and active claims in the running system.

Home directories

One common pattern in using existing file servers is to share a file system divided into home directories. While multiple users share a single image, in practice their activities are largely partitioned into specific subdirectories.

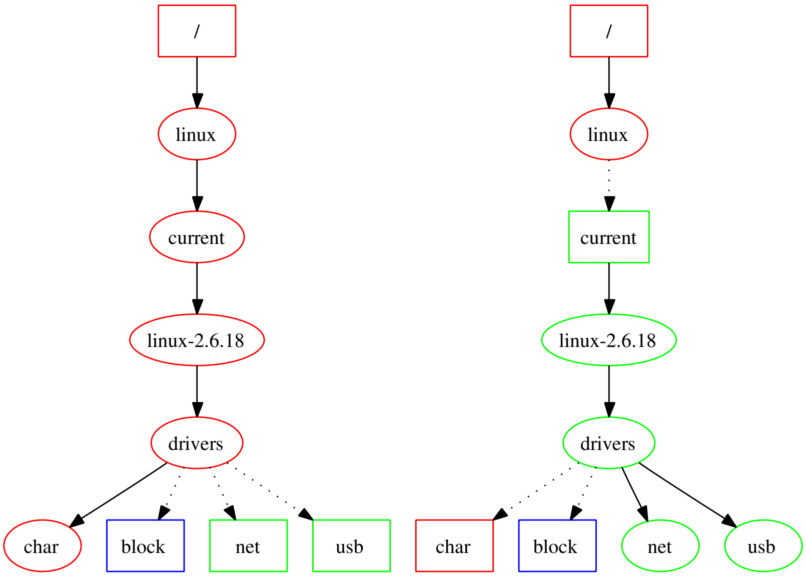

Figure 6.12: The territory graph when sharing an image divided into home directories. Rectangles denote territory roots, and colours indicate the host that owns a directory. druid (red) is the root server and directs its requests to `char`, skiing (blue) accesses `block`, and moonraider (green) sends requests to `net` and `usb`. The tree on the left depicts the setup, while the one on the right shows the actual layout created by Envoy.

As the root server and host to the first client to mount the image, druid-0 starts out owning the entire territory. Four directories within the Linux tree are assigned to clients on the three servers, which start opening two randomly chosen files per second in each directory. The tree on the left of Figure 6.12 depicts the territory tree as planned, with the four directories at the bottom acting as home directories. The tree on the right is the actual result from testing this configuration on Envoy.

While druid had its own client traffic, it also had requests coming at an equal rate from skiing and twice as many from moonraider, which was assigned two directories. The envoy on druid had the options of ceding control of one directory to skiing and the other two to moonraider, or transferring control of the whole image to moonraider and allowing it to make further splits. The imbalance as viewed at the root of the image would be equal to that at the root of each target directory (note that moonraider's doubled traffic is offset by local traffic when viewed from the root, making the combined urgency equal to that at the root of each

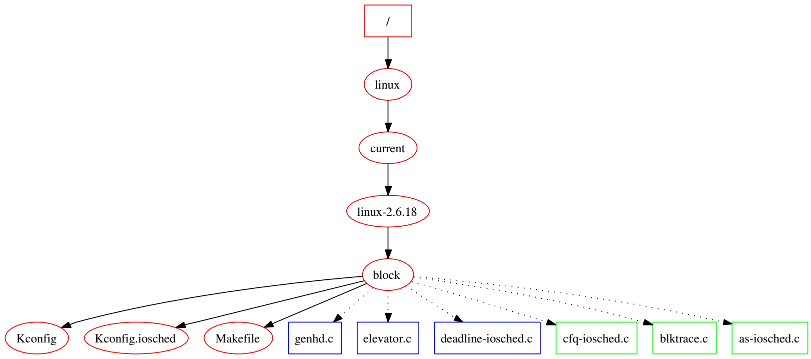

Figure 6.13: The territory graph when sharing a directory hosting privately-accessed files. Rectangles denote territory roots, and colours indicate the host that owns a directory. Clients on each of the three servers were given three files each within one directory to access at a constant rate. The tree shows the final steady-state result produced by Envoy.

target directory), except that each client request traversed from the root of the tree. The extra overlapping traffic near the root tipped the balance in favour of ceding control of the whole territory to moonraider, which then waited until enough time had elapsed to transfer control of the two remotely-accessed directories to their respective hosts.

The order in which transfers occur affects the final layout. Had moonraider introduced its requests one directory at a time, druid would have retained control of the top-level directories. The greedy algorithm used in Envoy does not guarantee optimal layouts, but it does try to produce useful behaviour with simple rules.

Log files

Directory boundaries are not always the ideal place to partition shared files. Multiple clients may also access individual files that are stored in a common directory. A common example of this is in log files that collect data and results from multiple servers in a shared directory. For this situation, there is no sharing within the files themselves.

Figure 6.13 shows the result of this test, where three files were assigned to each client. It took longer to achieve the final layout with this test for two reasons. Instead of accessing random files in a directory, clients accessed individual files, so fewer requests were generated. Navigation steps leading to the file went through shared directories and did not contribute to the urgency of transfers. In a more realistic setting, the client would likely hold an open handle for each file yielding a similar effect, so this difference is a artifact of the test script. The second reason is that more transfers had to take place, with enforced minimum delays between each. druid was the only host to initiate transfers, and each one involved the same root territory, so the process was completely sequential.

After reaching a steady state, no further traffic between envoys would be necessary if each client maintained an open handle to each file. True to Envoy's goal, having the files in the same directory would be convenient but not prevent individual machines from handling file traffic locally. With no actual sharing taking place and a consistent traffic load, no overhead for coordination is necessary.

Sharing files

The next test involves actual sharing between clients. All three hosts accessed random files within the same directory at the same rate. The expected result would be to have no territory changes, as no remote client would generate more traffic than the initial owner. An artifact of the implementation makes this dependent on the configuration parameters. Directory navigation results are cached on each envoy, even for remotely-owned territories. To minimise stale cache effects, the final target of each 9p `walk` requests is contacted directly if it is remote, even when a cache entry exists. Since the Linux 9p driver issues a separate `walk` request for each directory, this means that the navigation cache is effectively disabled for remote hosts. It is active for locally-owned territories, however, which effectively reduces the number of steps for local clients but not for remote clients. Because of this, remote clients appear to generate more requests despite the uniformity that is carefully maintained by the clients.

With a slight urgency boost for remote clients, an equal sharing situation can have one of two effects on territory boundaries. If the idle threshold is high enough, the extra navigation traffic generated by the test will not be enough to trigger a transfer, and control will remain with the initial owner indefinitely. If the idle value is too small, however, control will eventually be passed to one of the remote hosts, which will then face the same situation. Since this thrashing happens on a slow time scale (nearly 120 seconds between transfers in this particular test), the detrimental effects on caching are minimal. One might argue that periodically passing control between equally active clients is ultimately fairer, if not an absolute win in terms of system-wide performance and capacity.

6.4 Image operations

Service clusters must be able to easily manipulate file system images to achieve their promised flexibility. Providing templates with commodity software and allowing clients to easily clone and customise them makes service instantiation a relatively lightweight operation. When combined with functionality in the virtual machine monitor to suspend, clone, and resume running services, this could allow rapid instantiation of common services. Envoy provides support for rapid forking of file system images, along with the ability to take snapshots of active images. This section measures the performance of these two management operations, along with the time and space required to deploy new services.

6.4.1 Forks

New images can be created from scratch, or they can be forked from an existing image. Forks always start from a snapshot of an existing image, and snapshots are always read-only. With the copy-on-write mechanism described in Section 4.3.2, this makes forks a fast and efficient operation. Even if the template snapshot is in active use, only the root of the image needs to be copied to distinguish the new image from the old.

The time required to fork an image was measured in two ways. In the first, the server was instrumented to record the time spent completing the operation. Over fifty iterations, the average fork took 339 μseconds with a standard deviation of 32 μseconds. Over the same fifty iterations, the client VM observed the time required to fork the template and create a new file in the new image. `/usr/bin/time` reported 10 milliseconds for each iteration, which corresponds to its timer resolution.

6.4.2 Snapshots

Forking an existing template is fast because few operations are involved. Snapshots can be taken of active images, which may involved cloning storage objects as well as copying runtime state. If an image has been divided into multiple territories, the snapshot operation will span multiple envoy instances and will required cloning a path from the root of the image to each territory root, as discussed in Section 4.3.2. Forking an active image also involves taking a snapshot and then performing a fork operation, so cloning an active system is dependent on the snapshot mechanism as well.

In the simple case, snapshots are nearly as fast as forks. When an image is inactive, taking a snapshot requires cloning the root, adding the new snapshot to the parent directory, and updating the link to the active view of the image. Over fifty iterations, an instrumented server reported an average of 1258 μseconds per snapshot with a standard deviation of 108 μseconds.

To evaluate more complex snapshot cases, a tree of territories involving three machines was created. Control of a Linux kernel source tree was divided such that every directory was the root of a territory owned by a different envoy than its parent, i.e., every territory was forced to cede control of subdirectories equally between the other two servers. druid-0 was the root server and hosted 439 territories, skiing-0 hosted 421, and moonraider-0 hosted 379 territories, for a total of 1239 territories. This is far beyond what one would expect in a normal deployment for a single image, but serves to exercise the snapshot mechanism under extreme conditions.

After constructing the territory tree, fifty snapshots were taken of the kernel image in an average of 13.71 seconds each with a standard deviation of 3.13 seconds, or approximately 11.1 milliseconds per territory in the tree. Most of this time was spent cloning objects to ensure that the root of each territory was writable in the active image after the snapshot completed. Under Envoy's consistency semantics, this requires submitting changes to the storage servers before considering the operation complete, but not flushing their buffers to disk. If the snapshots are done in rapid succession, the storage server buffers cannot absorb the traffic and disk write times increase the time. After a warmup run of several snapshots, fifty more were done without pause in an average of 73.13 seconds each with a standard deviation of 26.8 seconds, or approximately 59 milliseconds per territory. The high variability comes from the lack of coordination between the client operations and the buffer cache on the storage servers.

The snapshot timings reflect a combination of network messages, internal state updates, and storage object updates. The bulk of the time is spent doing storage object updates, though this is not measured directly. To approximate this, another operation can be measured that performs similar state updates and has the same network message profile, but does not involve updating storage objects. Renaming the root directory requires recursively notifying child territories of the pathname change, which makes it a suitable choice. Fifty such operations complete in an average of 96.7 milliseconds each with a standard deviation of 1.09 milliseconds, approximately 78 μseconds per territory.

	cold cache	hot cache
client requests/bytes	70,016/ 1,264,742	71,648/ 1,296,578
client responses/bytes	70,016/79,457,018	71,648/81,323,281
storage requests/bytes	2,764/ 288,858	2,059/ 285,429
storage responses/bytes	2,764/ 17,476,789	2,059/ 675,196

Table 6.2: Messages and bytes transferred when virtual machines boot from forks of the same Linux template image. The first was booted with a cold cache, and the second after restarting Envoy but not clearing the in-memory or on-disk cache. In the hot cache case, objects had to be validated against the storage server because of the server restart, hence the similar message counts, but data could be retrieved from the cache.

6.4.3 Service deployment

Besides making service deployment quick, an important goal of snapshot and fork operations using copy-on-write is to encourage the use of commodity software and exploit the redundancy that results. Objects inherited from a common template or a previous snapshot can be read by any envoy without coordination, as higher-level semantics guarantee that the object will not be modified again. This allows cache sharing for read-only objects between unrelated services. Besides cutting down the cache required on a given node, this makes it likely that common template objects will already be loaded into the persistent cache of typical nodes, allowing the most common objects to be served locally and reducing the cluster-wide load on storage servers.

To measure this effect, a template image was prepared with SUSE Linux Enterprise Server 10, and a virtual machine booted from a fork of this image with a cold cache. Another VM booted from a second fork of the image, this time with the cache primed from the first one. Boot times as measured with a stopwatch were about 50 seconds in either case, with the hot cache a few seconds faster (compared to about 30 seconds when booting from a local partition). Boot times were hurt by the lack of client-side cache, particularly for shared libraries loaded as memory-mapped files. While optimisations could make node-level caching sufficient for most conventional file operations, support for executables and shared libraries is rather poor in the current cache model.

Table 6.2 shows the results collected by an instrumented version of Envoy. The server was restarted between tests, so the hot cache results include message traffic to validate cached objects with the storage servers. Booting the second image pulled less than 4% of the bytes retrieved from the storage layer when starting the first image, showing a heavy overlap between read-only data in the two instances. Combined with the lightweight snapshot and fork operations, this suggests that service instances can be created and deployed quickly and with little impact beyond the local machine.

6.5 Summary

The prototype is evaluated on a small deployment to test and validate design decisions, to explore possible bottlenecks, and to explore the dynamic behaviour of the system.

Performance is measured on static system layouts. Envoy compares favourably with userspace NFS and 9p servers on bulk reads and writes, but is slower when more metadata operations are involved. Comparison to 9p for similar data paths suggests that significant overhead is due to an unoptimised implementation, but other costs are due to the architecture. Forwarding requests to a remote envoy costs roughly 33% more than connecting the client directly to the remote server. Caching makes predictable improvements in read performance, though the local persistent cache hurts write performance in many cases.

In the presence of sharing, dynamic territory management works largely as predicted for common sharing cases. Shared images with privately-accessed directories and files are successfully partitioned along appropriate boundaries. When files are shared and accessed at similar rates by multiple clients, the system is reluctant to transfer ownership, though slight imbalances can trigger changes over a longer time scale.

Image-level operations are fast, with performance limited mainly by the amount of disk activity they incur. Booting multiple virtual machines from forks of a common template image demonstrates a high degree of sharing. Traffic to the storage layer for the second is only 4% of that of the first by bytes transferred. While the lack of per-client caching is acceptable for many operations, it hurts the performance of memory-mapped shared libraries enough to significantly slow down the boot process.

Chapter 7

Conclusion

This dissertation concludes with a discussion of opportunities for future research and a summary of the work presented here.

7.1 Future Work

Storage clusters are only partially specified in this work, and additional research is necessary to make them a reality. Even ignoring the storage layer, virtual machine tools, cluster management tools, and economic modelling that are outside the scope of this dissertation, several aspects of Envoy's design suggest avenues for future research.

7.1.1 Caching

Envoy eschews client-side caching in favour of a unified node-level cache. This works well for some workloads, but poorly for others, particularly for memory-mapped files. Besides being part of Envoy's cache consistency mechanism, this design allows consolidation of redundant cache entries that would otherwise exist on multiple clients. While template images encourage object sharing, many objects will still be used privately by only one client at a time, where issues of consistency and consolidation do not apply. Two questions remain unanswered: how much of the performance penalty from moving the cache outside the client's VM can be optimised away, and how could a hybrid cache that uses client-side caching with token-passing for some data and a consolidated cache for the rest be constructed to balance competing concerns of individual client performance and aggregate cache capacity and performance.

7.1.2 Territory partitioning

Envoy uses a simple greedy algorithm to decide on territory boundary changes, with a unique time-based mechanism to promote stability while still permitting minor optimisations over time. Further testing with realistic workloads would improve understanding of the algorithm's behaviour, and may suggest improvements. The existing implementation could be enhanced to make more sophisticated recommendations, for example keeping a subtree and migrating the rest of the territory whereas now it can only recommend the converse. Also, finding optimal configuration parameters that balance the benefits of locality with the costs of moving to an unprimed cache in realistic conditions remains future work.

7.1.3 Read-only territory grants

Territories in Envoy divide and subdivide file system images to align ownership with access. Currently, a single owner monopolises each territory and maintains its cache, resulting in performance penalties for remote clients. Implicit sharing is available through image forking, but since most file data is never modified, additional sharing of territories could be enabled. Territories could be extended to permit read-only grants to multiple nodes, which are revoked and consolidated when write operations are requested, effectively extending write-exclusive/read-shared locks to the territory level. It remains to be seen whether or not this would provide significant benefits for realistic workloads.

7.1.4 Storage-informed load balancing

Envoy entrusts territory management decisions to the current territory owners because they have the most information to make good allocation decisions. Similarly, the file system in a service cluster would be a good place to collect information about client behaviour to inform load balancing and service placement decisions. Envoy's structure already detects clients with overlapping interests in active images, and could also infer common interest in objects shared through image forking. Gathering and using this information could make it possible to optimise system-wide service layout to minimise storage-layer demand and enhance the value of shared caching, as well as increasing the aggregate performance and capacity of the service cluster.

7.2 Summary

This dissertation argues for the commoditisation of computation services using virtual machines as containers hosted on clusters built from commodity hardware. In particular, it addresses the storage demands of this environment, a need not adequately filled by existing systems. After an introduction to the problem in Chapter 1, Chapter 2 discusses relevant background work in commodity computing, machine virtualization, and storage systems.

Chapter 3 describes the proposed environment in more detail, and argues its merits as well as describing its requirements. *Service containers* are defined as virtual machines customised from template images of commodity software, and *service clusters* are cluster environments optimised for deploying client services in a flexible way. This combination permits management of untrusted client software in ways that maximise the use of hardware resources and separate hardware management from software deployment.

The Envoy file system is introduced in Chapter 4 and its implementation described in Chapter 5. Envoy runs in a trusted virtual machine on each node, which exports a standard client-server file system interface to locally-hosted clients. A global namespace tree includes file system images that can be managed and shared under client control. Control of images is given to the envoy node on the same host as the client, and control of shared images is partitioned into *territories* by a dynamic process that responds to runtime demand. In a stable state, nodes need to communicate with each other only when the interests of their respective clients overlap. Lightweight snapshot and fork operations allow simple image management, and encourage implicit sharing of underlying objects even when no relationship exists between clients.

Evaluation of the Envoy prototype in Chapter 6 demonstrates that baseline performance in the most common configuration is comparable to simple client-server file systems. The forwarding architecture imposes a 33% overhead for remote requests in addition to added network latencies, making it beneficial to optimise territory boundaries but not overly expensive to tolerate sub-optimal layouts with light traffic. Caching, both in-memory and on-disk, improves read performance, though on-disk caching comes at a minor cost for write operations. The dynamic territory algorithm successfully optimises common sharing scenarios. Booting multiple services from forks of a common template image yields implicit object sharing that results in a significant reduction of traffic to the storage layer.

Envoy supports the storage needs of service clusters, allowing flexible and efficient management of storage and optimising for expected usage patterns, with an architecture that promotes scalability by localising storage management and encouraging cache sharing.

References

[Ada06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, December 2006. 15

[Ady02] 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 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 1–14. Boston, Massachusetts, December 2002. 23

[AEM05] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay Y. Wylie. Ursa Minor: Versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 59–72. San Francisco, California, December 2005. 30

[Ama06] Amazon Web Services. `http://aws.amazon.com/`, November 2006. 1, 12

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Spring Joint Computer Conference*, pages 483–485. Atlantic City, New Jersey, April 1967. 8

[Ame02] Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. Group-based management of distributed file caches. In *Proceedings of 22nd IEEE International Conference on Distributed Computing Systems (ICDCS 2002)*. Vienna, Austria, July 2002. 18

[Ami98] Elan Amir, Steven McCanne, and Randy H. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 178–189. Vancouver, British Columbia, Canada, September 1998. 1

[And95a] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995. 7, 22, 52

[And95b] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–126. Copper Mountain Resort, Colorado, December 1995. 22

[And97] Eric Anderson and Dave Patterson. Extensible, scalable monitoring for clusters of computers. In *Proceedings of the 11th Systems Administration Conference (LISA '97)*, pages 9–16. San Diego, California, October 1997. 9

[Awa02] Amr A. Awadallah and Mendel Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW 2002)*. Boulder, Colorado, August 2002. 15

[Bac91] Jean Bacon, Ken Moody, Sue Thomson, and Tim Wilson. A multi-service storage architecture. *ACM SIGOPS Operating Systems Review*, 25(4):47–65, October 1991. 28

[Bak91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In

Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91), pages 198–212. Pacific Grove, California, October 1991. 21

[Bak94] Mary G. Baker. *Fast Crash Recovery in Distributed File Systems.* Ph.D. thesis, University of California at Berkeley, 1994. 52

[Bar03a] 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 Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177. Bolton Landing, New York, October 2003. 15, 42

[Bar03b] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003. 8, 37, 69

[Bar05] Luiz André Barroso. The price of performance. *ACM Queue*, 3(7):48–53, September 2005. 10

[Bez06] Jeff Bezos. Q&A: Amazon's Jeff Bezos at Web 2.0. `http://www.informationweek.com/showArticle.jhtml?articleID=193700471`, November 2006. 11

[Bir93] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corp. Systems Research Center, Palo Alto, California, September 1993. 24, 47, 49

[Bla93] Matthew A. Blaze. *Caching in Large-scale Distributed File Systems*. Ph.D. thesis, Princeton University, January 1993. 21

[Bla03] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems (HotOS IX)*. Lihue, Hawaii, June 2003. 26, 52

[Boe93] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming*

Language Design and Implementation, pages 197–206. Albuquerque, New Mexico, June 1993. 101

[Bol00] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS '00)*, pages 34–43. Santa Clara, California, June 2000. 23

[Bra96] Tim Bray. Bonnie. `http://www.textuality.com/bonnie/`, 1996. 147

[Bur88] Michael Burrows. *Efficient Data Sharing*. Ph.D. thesis, University of Cambridge Computer Laboratory, December 1988. 20, 21

[Cal95] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995. 20, 102

[Cla05] 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 '05)*, pages 273–286. Boston, Massachusetts, May 2005. 15

[Coh03] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*. Berkeley, California, June 2003. 25

[Cor65] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of AFIPS Fall Joint Computer Conference*, volume 27, pages 185–196. June 1965. 45

[Dab01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215. Chateau Lake Louise, Banff, Canada, October 2001. 26

[Dah94a] Michael Dahlin, Thomas Anderson, David Patterson, and Randolph Wang. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating*

Systems Design and Implementation (OSDI '94), pages 267–280. Monterey, California, November 1994. 23, 63, 69

[Dah94b] Michael Dahlin, Clifford Mather, Randolf Wang, Thomas E. Anderson, and David A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS '94)*, pages 150–160. Nashville, Tennessee, May 1994. 23

[Dev98] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent 6397242, October 1998. 14

[Dou99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS '99)*, pages 59–70. Atlanta, Georgia, May 1999. 23, 84

[Dou01] John R. Douceur and Roger Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th Symposium on Reliable Distributed Systems (SRDS 2001)*, pages 4–13. New Orleans, Louisiana, October 2001. 23

[Dou02] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of 22nd IEEE International Conference on Distributed Computing Systems (ICDCS 2002)*. Vienna, Austria, July 2002. 21

[Fac05] Michael Factor, Kalman Methand Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *Proceedings of the Second International IEEE Symposium on Emergence of Globally Distributed Data*. Sardinia, Italy, June 2005. 29, 64

[Fig03] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of 23nd IEEE International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 550–559. Providence, Rhode Island, May 2003. 13, 34

[Fos03] Ian Foster and Carl Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2nd edition, November 2003. 13

[Fox97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 78–91. Saint-Malo, France, October 1997. 9, 46

[Fra92] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 596–609. Vancouver, British Columbia, Canada, August 1992. 63

[Frø03] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. FAB: Enterprise storage systems on a shoestring. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems (HotOS IX)*. Lihue, Hawaii, June 2003. 29

[Gan94] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60. Monterey, California, November 1994. 19

[Gan95] Gregory R. Ganger. Generating representative synthetic worloads: An unsolved problem. In *Proceedings of the 21st International Conference on Technology Management and Performance Evaluation of Enterprise-Wide Information Systems, Computer Measurement Group*, pages 1263–1269. Nashville, Tennessee, December 1995. 141

[Gan00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience (SPE)*, 30(11):1203–1233, September 2000. 165

[Gan03] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* Storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003. 30

[Ghe03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Bolton Landing, New York, October 2003. 30, 37

[Gib97] Garth A. Gibson, David Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS '97)*, pages 272–284. Seattle, Washington, June 1997. 29

[Gib98a] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 92–103. San Jose, California, October 1998. 29, 30, 65

[Gib98b] Timothy J. Gibson, Ethan L. Miller, and Darrell D. E. Long. Long-term file activity and inter-reference patterns. In *Proceedings of the 24th International Conference on Technology Management and Performance Evaluation of Enterprise-Wide Information Systems, Computer Measurement Group*, pages 976–987. Anaheim, California, December 1998. 18, 72

[Gif88] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, 1988. 21

[Gka05] Christos Gkantsidis and Pablo Rodriguez. Network coding for large scale content distribution. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Commications Societies (INFOCOM 2005)*, volume 4, pages 2235–2245. Miami, Florida, March 2005. 25

[Gra89] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 202–210. Litchfield Park, Arizona, December 1989. 21

[Grö99] Björn Grönvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 251–264. New Orleans, Louisiana, February 1999. 25, 33

[Gum83] Peter H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983. 14

[Guy90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus replicated file system. In *Proceedings of the USENIX Summer 1990 Conference*, pages 63–71. Anaheim, California, June 1990. 24

[Hag87] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 155–162. Austin, Texas, November 1987. 19

[Har93] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 29–43. Asheville, North Carolina, December 1993. 28

[Har99] John H. Hartman, Ian Murdock, and Tammo Spalink. The Swarm scalable storage system. In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems (ICDCS 1999)*, pages 74–81. Austin, Texas, June 1999. 28

[Hen05] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space: Using 9P2000 under Linux. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 83–94. Anaheim, California, April 2005. 105, 147

[Hit94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246. San Francisco, California, January 1994. 19, 21

[Hos04] Andy Hospodor and Ethan L. Miller. Interconnection architectures for petabyte-scale high-performance storage systems. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, pages 273–281. College Park, Maryland, April 2004. 28, 62

[How88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, February 1988. 20, 104

[Hsi89] Hui-I Hsiao and David J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. Technical Report CS TR 854, University of Wisconsin, Madison, June 1989. 29, 65

[IBM06] Linux on IBM System z. `http://www.ibm.com/servers/eservers/zseries/os/linux/`, November 2006. 15

[Ji05] Minwen Ji. Instant snapshots in a federated array of bricks. Technical Report HPL-2005-15, HP Laboratories, 2005. 30

[Kal04] Mahesh Kallahalla, Mustafa Uysal, Ram Swaminathan, David E. Lowell, Mike Wray, Tom Christian, Nigel Edwards, Chris I. Dalton, and Frederic Gittler. SoftUDC: A software-based data center for utility computing. *IEEE Computer*, pages 46–54, November 2004. 1, 12

[Kaz90] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig Everhart, W. Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum file system architectural overview. In *Proceedings of the USENIX Summer 1990 Conference*, pages 151–164. Anaheim, California, June 1990. 20, 21

[Kel94] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–131. San Francisco, California, January 1994. 21

[Kha93] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 1–15. Asheville, North Carolina, December 1993. 22

[Kin03] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 223–236. Bolton Landing, New York, October 2003. 55

[Kis91] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 213–225. Pacific Grove, California, October 1991. 21, 24, 56

[Klo02] Andrew J. Klosterman and Gregory Ganger. Cuckoo: Layered clustering for NFS. Technical Report CMU-CS-02-183, Carnegie Mellon University, October 2002. 72

[Kot04a] Evangelos Kotsovinos. *Global Public Computing*. Ph.D. thesis, University of Cambridge Computer Laboratory, November 2004. 16

[Kot04b] 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 (WORLDS '04)*. San Francisco, California, December 2004. 16, 22, 33, 102

[Kub00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher 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 IX)*, pages 190–201. Cambridge, Massachusetts, November 2000. 26

[Lee95] Edward K. Lee. Highly-available, scalable network storage. In *Proceedings of the 40th IEEE Computer Society International Conference*. March 1995. 28

[Lee96] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 84–92. Cambridge, Massachusetts, October 1996. 28, 65

[Lus06] Lustre. `http://www.lustre.org/`, November 2006. 30

[Man94] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems (TOCS)*, 12(2):123–164, May 1994. 21, 24

[McG98] A. McGregor and J. Cleary. A block-based network file system. In *Proceedings of the 21st Australian Computer Science Conference*, pages 133–144. Springer, Perth, Australia, February 1998. 21

[McK84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, August 1984. 18

[Men03] Jai Menon, David Pease, Robert Rees, Linda Duyanovich, and Bruce Hillsberg. IBM storage tank—A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003. 31

[Mes03] Michael Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, August 2003. 29

[Mic06] Microsoft Virtual Server. `http://www.microsoft.com/windowsserversystem/virtualserver/`, November 2006. 14

[Mor02] Tim D. Moreton, Ian A. Pratt, and Timothy L. Harris. Storage, mutability and naming in Pasta. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pages 215–219. Springer-Verlag, London, United Kingdom, 2002. 26

[Mum95] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 143–155. Copper Mountain Resort, Colorado, December 1995. 20

[Mun92] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems or your cache ain't nothin' but trash. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 305–313. San Francisco, California, January 1992. 69

[Mut02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 31–44. Boston, Massachusetts, December 2002. 26

[Nel88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):134–154, February 1988. 21

[Ng02] Wee Teck Ng, Bruce Hillyer, Elizabeth Shriver, Eran Gabber, and Banu Özden. Obtaining high performance for storage outsourcing. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*. Monterey, California, January 2002. 11

[Nig06] Edmund B. Nightengale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–14. Seattle, Washington, November 2006. 74

[Ous85] John K. Ousterhout, Herve Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, pages 15–24. Orcas Island, Washington, December 1985. 18, 73

[Pat88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116. Chicago, Illinois, June 1988. 28, 46

[Paw94] Brian Pawlowski, Chet Juszczak, and Peter Staubach. NFS version 3: Design and implementation. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 137–152. Boston, Massachusetts, June 1994. 20, 77, 102

[Pfa06] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 353–366. San Jose, California, May 2006. 33

[Pik90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer UKUUG Conference*, pages 1–9. London, United Kingdom, July 1990. 22, 105

[Pik92] 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. Le Mont Saint-Michel, France, September 1992. 22, 105

[Pik95] Rob Pike. *Plan 9 Programmer's Manual, Volume 1.* Bell Laboratories, Murray Hill, New Jersey, 1995. 105

[Pop90] Gerald J. Popek, Richard G. Guy, Thomas W. Page Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the First Workshop on the Management of Replicated Data*, pages 5–10. Houston, Texas, November 1990. 24

[Pou05] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS '05)*. Ithaca, New York, February 2005. 25

[Qui02] 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 '02)*. Monterey, California, January 2002. 21, 26, 82

[Rab81] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, May 1981. 26

[Rab89] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989. 26

[Rat01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172. San Diego, California, August 2001. 26

[Ree99] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accountable execution of untrusted programs. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 136–141. Rio Rico, Arizona, March 1999. 16, 27

[Rhe03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 1–14. San Francisco, California, March 2003. 26

[Rid97] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of PCs. In *Proceedings of the 1997 IEEE Aerospace Conference*, volume 2, pages 79–91. Aspen, Colorado, February 1997. 9

[Ros91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 1–15. Pacific Grove, California, October 1991. 18, 84

[Ros00a] Timothy Roscoe and Bryan Lyles. Distributing processing without DPEs: Design considerations for public computing platforms. In *Proceedings of the 9th ACM SIGOPS European Workshop*. Kolding, Denmark, September 2000. 1

[Ros00b] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54. San Diego, California, June 2000. 18, 72, 75

[Row01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed*

Systems Platforms (Middleware 2001), pages 329–350. Heidelberg, Germany, November 2001. 26

[Row01b] Antony I. T. 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 Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201. Chateau Lake Louise, Banff, Canada, October 2001. 26

[Rue93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 405–420. San Diego, California, January 1993. 18, 65

[Sai02a] Yasushi Saito and Christos Karamanolis. Pangaea: A symbiotic wide-area file system. In *Proceedings of the 10th ACM SIGOPS European Workshop*. Saint-Emilion, France, September 2002. 25

[Sai02b] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 15–30. Boston, Massachusetts, December 2002. 25

[Sai04] Yasushi Saito, Svend Frølund, Alistair C. Veitch, Arif Merchant, and Susan Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 48–58. Boston, Massachusetts, October 2004. 29

[San85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer 1985 Conference*, pages 119–130. Portland, Oregon, June 1985. 19, 102

[San99] 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 Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123. Charleston, South Carolina, December 1999. 21

[Sap02] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, Massachusetts, December 2002. 15

[Sap03] Constantine Sapuntzakis and Monica S. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Operating Systems (HotOS IX)*. Lihue, Hawaii, June 2003. 12

[Sat85] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, pages 35–50. Orcas Island, Washington, December 1985. 20, 104, 145

[Sat90] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990. 20

[Sch85] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, pages 25–34. Orcas Island, Washington, December 1985. 21

[Sch02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244. Monterey, California, January 2002. 31

[Sel93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 307–326. San Diego, California, January 1993. 84

[Sel95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference*, pages 249–264. New Orleans, Louisiana, January 1995. 19, 84

[Sel00] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84. San Diego, California, June 2000. 19

[Sha86] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proceedings of 6th IEEE International Conference on Distributed Computing Systems (ICDCS 1986)*, pages 198–204. Cambridge, Massachusetts, May 1986. 60

[She03] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. RFC 3530, April 2003. 20, 103, 156

[Sou03] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Greg Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58. San Francisco, California, March 2003. 122

[Spe03] David Spence and Tim Harris. XenoSearch: Distributed resource discovery in the XenoServer open platform. In *Proceedings of the 12th IEEE International Symposium on High-Performance Distributed Computing*. Seattle, Washington, June 2003. 16

[Spr02] Tom Spring. PCWorld.com interview with Eric Schmidt: Three minutes with Google's Eric Schmidt. `http://archives.cnn.com/2002/TECH/internet/02/01/interview.eric.schmidt.idg/index.html`, February 2002. 68

[Ste00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. RFC 2960, October 2000. 97

[Ste02] Christopher A. Stein, Michael J. Tucker, and Margo I. Seltzer. Building a reliable mutable file system on peer-to-peer storage. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 324–329. Osaka, Japan, October 2002. 26

[Ste05] Lex Stein. Stupid file systems are better. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS X)*. Santa Fe, New Mexico, June 2005. 29, 65

[Sto01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160. San Diego, California, August 2001. 26

[Sun06] Sun grid compute utility. `http://www.sun.com/service/sungrid/`, November 2006. 1, 13

[Swe96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 1–14. San Diego, California, January 1996. 19

[The97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237. Saint-Malo, France, October 1997. 29

[Top06] TOP500 supercomputer sites. `http://www.top500.org/`, November 2006. 8, 37

[Tul98] Patrick Tullmann and Jay Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 111–117. Sintra, Portugal, September 1998. 1

[Twe98] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo (LinuxExpo '98)*, pages 25–29. Durham, North Carolina, May 1998. 19

[Vah98] Amin Vahdat, Tom Anderson, Mike Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the 7th IEEE International Symposium on High-Performance Distributed Computing*, pages 52–63. Chicago, Illinois, July 1998. 1

[Wal83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed file system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 49–70. Bretton Woods, New Hampshire, October 1983. 22, 23

[Wal94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994. 49

[Wan93] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 71–78. Napa, California, October 1993. 25

[Wan98] Randolph Y. Wang, Thomas E. Anderson, and Michael D. Dahlin. Experience with a distributed file system implementation. Technical Report UCB/CSD-98-986, Computer Science Division, University of California at Berkeley, January 1998. 23

[Wan04] Feng Want, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, pages 139–152. College Park, Maryland, April 2004. 31

[War05] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS X)*. Santa Fe, New Mexico, June 2005. 3, 21, 33, 46, 53, 83

[Wei04] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*. Pittsburgh, Pennsylvania, November 2004. 32

[Wei06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320. Seattle, Washington, November 2006. 31, 65

[Wel91] Brent B. Welch. Measured performance of caching in the Sprite network file system. *Computing Systems*, 3(4):315–342, 1991. 21, 56

[Whi04] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Using time travel to diagnose computer problems. In *Proceedings of the 11th ACM SIGOPS European Workshop*. Leuven, Belgium, September 2004. 55

[Wil95] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 96–108. Copper Mountain Resort, Colorado, December 1995. 30

[Wil04] John Wilkes, Jeffrey Mogul, and Jaap Suermondt. Utilification. In *Proceedings of the 11th ACM SIGOPS European Workshop*. Leuven, Belgium, September 2004. 12

[Zha01] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California at Berkeley, April 2001. 26

[Zha04] Ming Zhao, Jian Zhang, and Renato Figueiredo. Distributed file system support for virtual machines in grid computing. In *Proceedings of the 13th IEEE International Symposium on High-Performance Distributed Computing*, pages 202–211. Honolulu, Hawaii, June 2004. 13, 34