Second Year Report

Russ Ross <rgr22@cl.cam.ac.uk>
University of Cambridge Computer Laboratory

31 January 2005

1 Introduction

During my second year as a PhD student I have achieved a few significant milestones in my research and in my personal life; I have also modified my research focus. In this report I present a few brief highlights from the past year together with a more detailed description of my current research efforts.

1.1 Copy-on-write NFS server

One project detailed in my first year report[1] was a copy-on-write NFS server[2] that I wrote for use in the XenoServers[3] project. At that time it was in an incomplete state, sufficient to prove the concept but not suitable for practical use. In the months that followed I continued my implementation efforts and built a working system with usable performance characteristics.

Once I reached full functionality in the implementation, I started benchmarking and profiling it with disappointing results. I built a Linux 2.4 kernel on a mounted partition as a basic test, and found that building on a file system mounted with `cownfsd` slowed the process (which is normally CPU bound) by a factor of 2 or 3 compared to a normal userspace NFS server, and the server process was consuming unreasonable amounts of CPU time. The daemon was also prone to crashing unpredictably.

Profiling revealed that the RPC[4] library[5] I used (implemented by a third party in OCaml) was largely to blame. I explored the tools for interfacing between OCaml (the implementation language) and C, and found that I could write a code generator to automate the conversion between the C data structures generated by the standard Unix `rpcgen` tool and the corresponding structures in OCaml, including interfacing with the garbage collector.

The result was a server with more predictable behavior and fewer crashes, and performance on par with a normal userspace NFS daemon written in C. The overhead of providing copy-on-write functionality and writing the server in a garbage-collected high-level functional language was measurable, but quite reasonable.

The server played an integral part in our proposed XenoServers deployment strategy as outlined in a submission to OSDI. A revised version of this paper was later accepted to the WORLDS conference[6].

1.2 Marriage

During the summer I took some time off my studies and got married on 4 June 2004[1], adding four new members to my family (my wife Nancy and three in-laws) and expanding my wife's family

[1] `http://www.russross.com/Wedding_pictures.html`

circle by 31 members (me, my parents, my five siblings and their spouses, and my 18 nieces and nephews).

1.3 Teaching

I continued to supervise a variety of undergraduate courses, and in Lent Term 2005 I was asked to be the principal lecturer for the Part II General/Diploma twelve-lecture course *Introduction to Functional Programming*[7].

1.4 Parallax

I collaborated on a paper submitted[8] to HotOS describing a system to provide a cluster of machines driven by Xen[9] (or other VMMs) with disk images that support lightweight snapshot and fork operations. Our principle argument is that adequate storage can be provided by aggregating the disks on the host machines into a shared pool of blocks, then building virtual disk images supporting lightweight fork/snapshot operations and copy-on-write semantics on top of that block pool. Individual virtual machines can request a new disk image that is private to that VM instance (no write sharing) but based on a snapshot of an existing image, such as a standard OS installation. The system was designed to support the scalability requirements imposed by large clusters of machines, each hosting multiple virtual machines that have the option of taking frequent snapshots of the underlying disk image. More details about Parallax are available in the HotOS submission.

1.5 Distributed file system

My thesis proposal[10] for a distributed file system has undergone countless revisions, but has not yet led to an implementation. My principal stumbling block has been a lack of clear research goals, leading to designs and proposals that could have practical value, but which where largely the application of existing ideas to solve an existing problem. I sought a better system design than those already available, but lacked focused research goals and a clear argument for the novelty of my design.

My work on the Parallax project helped me to gather my ideas into a revised proposal that I will describe in greater detail here, one that I think has greater focus, clearer goals, and generally stronger research merit.

2 Revised Proposal

2.1 Background

The ongoing rise of virtual machine monitors as central tools in cluster environments leads to new requirements for cluster storage systems. As clusters move from specialized solutions custom tailored for specific problems and into the realm of general purpose tools, these requirements become even more severe. The conditions that are new to this environment include:

- Physical machines host multiple virtual machines, increasing the effective number of clients demanding storage.
- The ability to make lightweight snapshots of running VMs calls for similar capabilities in the storage system so that the entire state of a running VM can be captured.

- In the utility computing model, a cluster may be required to host many concurrent but independent file systems controlled by distinct clients, not just one large system shared by all participants. A suitable solution must be able to support both models.

In addition to adding these requirements, VM clusters allow us to make some new assumptions:

- Content duplication is expected to be very high. Though independent of each other, most VMs will nevertheless start from common file system images (bandwidth costs make alternatives less attractive, even when such alternatives are available), and a lot of data in the resulting images will be unchanged from the base image. In addition, snapshots over the history of a single file system will share a lot of content.
- Cheap storage is available: the disks attached to hosts won't be required by the individual machines if all needs can be met by the shared file system. This storage can be used locally for persistent caching and can be aggregated to provide a cluster-wide storage pool.
- Though VMs may be untrusted, the controlling VM domain can be trusted and the physical hosts can be centrally controlled and well maintained.

2.2 Parallax—a block-level virtual-disk image service

In the Parallax system, we create a distributed pool of blocks over the cluster with no write sharing. Above that, we construct virtual disk images that can be used by individual VM instances. These images support lightweight fork and snapshot operations, using copy-on-write semantics for data blocks and for the metadata used to assemble images from the block pool.

Redundancy between images due to common ancestry is exploited directly through the copy-on-write mechanism, and because all blocks are immutable after a snapshot, a straightforward extension could coalesce duplicate blocks that are created coincidentally, e.g., through sibling VM instances installing the same package on their respective images.

Parallax has the advantage of being OS agnostic, and it is capable of scaling to large numbers of private images while encouraging shared base images (it is both fast and cheap to fork from an existing image). Content shared between successive snapshots of a single image is stored only once, making it well suited to making frequent snapshots and maintaining an extensive catalog of historical images.

While it answers many of the needs of VM clusters, some applications are a poor fit for the Parallax model. In particular, those that require a shared namespace over multiple hosts must turn elsewhere. Our goal in this proposed system is to retain many of the benefits that Parallax offers at the virtual disk level, and extend them to support a file system model that can scale to many isolated file systems and to large-scale file systems shared by many virtual machines.

2.3 Envoy

With the Envoy file system, we propose a cluster file system built in two layers:

1. **Object service**—The storage layer provides storage and replication of objects. No write sharing is allowed, and lightweight copy-on-write operations are supported directly. Objects are named by unique 64-bit IDs that are location independent.
2. **Namespace service**—A distributed service provides a coherent file system image and manages write access to different parts of the file system. A host may assume control of a subset of the name space; in this case it can write to it directly and acts as a proxy for other hosts requiring write operations. Likewise, that same host must forward write requests for other parts of the image to the respective proxies. The proxy roles and regions of control can move and change in response to demand.

2.3.1 Basic design

A difficult problem in distributed file systems is coordinating shared writes. Our two-level design separates the general problems of storage management from those more specific to our problem space. A guiding principle in the design is that scalability should be limited only by conflicts that actually manifest at runtime.

Our aim is to support large numbers of independent file systems that can be forked and snapshotted cheaply and may share content (through ancestry or coincidence). These file systems can be shared on a large scale, with scalability limited primarily by contention, i.e., concurrent, conflicting writes may require synchronous communication between the participant hosts, but concurrent, independent reads and writes can occur with minimal coordination overhead.

2.3.2 Object service

We disallow write sharing in the storage layer and use copy-on-write for all changes, or at least all changes that are shared. This gives the following advantages:

- Simplicity in the storage layer design. In particular, it never gets involved in decisions regarding file system semantics; it leaves those to a higher level and concerns itself with storing the result.
- Time travel. Objects are stored directly in the storage layer, and file system metadata is also stored as regular objects. Referring to a previous file system state just requires knowing the object ID of the old file system root object.
- Snapshots are easy. Since all objects are modified using copy-on-write, a snapshot is just a historical reference to an earlier file system state. Forking into two divergent file systems can be done with lightweight metadata operations to disallow in-place writes, creating a common read-only image from which the two children will diverge.
- Cache coherency is simple. Objects have unique IDs, which identify not only a particular object, but a particular version of that object. All changes to shared objects are made using copy-on-write, and a writer that wishes to share its changes does so by providing the new object ID. A cache miss by a host will force a lookup in the storage layer, which may result in reading the complete object, or a delta from an object already in cache.

The storage layer itself can be implemented in a variety of ways. Simple replication, chained declustering, erasure codes, and other well-understood techniques are good candidates.

2.3.3 Namespace service

A coherent view of a single file system namespace is distributed over the hosts using that namespace. We seek the following desirable characteristics:

- Most read operations should be able to go directly from the host requesting the data to the storage layer, or be retrieved from a local, persistent cache.
- Most write operations should go directly from the originating host to the storage layer.
- Conflicting writes should be serialized, and should be resolved directly between the relevant hosts, with one consistent result going to the storage layer.

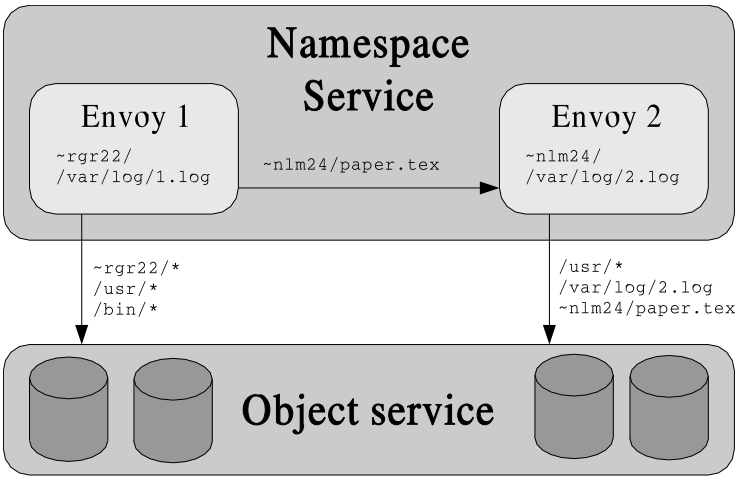

Figure 1: A host can make changes directly to a subset of the namespace. It assumes the role of *envoy* for that subset, called its *territory*; all other hosts make changes through it. Any host can go directly to the object service for objects not in the territory of any envoy.

- Read requests for recently updated data can go to the storage layer or directly to the host that wrote the data. The latter permits immediate access to data that may not have been committed to stable storage yet, and it requests such data from a host that is likely to have it readily available from in-memory cache.
- Readers can optionally ignore the most recent changes. A host can choose not to update its knowledge of the image from which it reads for a fixed interval, e.g., it can refresh its view of an image and then ignore updates for a minute before the next update in order to maximize cache hits and minimize dependence on/network traffic with peer hosts. The host can have a reasonably up-to-date image visible without requiring constant synchronization checks with other active hosts if such semantics are sufficient. Only hosts performing writes or otherwise demanding instant updates need to incur the extra overhead strict synchronization requires.

The file system achieves these goals by dividing control of a single file system image over multiple hosts as shown in Figure 1. Each host acts as a proxy for other hosts performing writes to a particular subset of the file system or demanding reads that are synchronized with the with updates. Control can be subdivided, merged, and moved dynamically in response to actual write activity. We call the host with write control the *envoy* representing a *territory*—a subset of the namespace—which may be as small as a single file.

A host performing many writes to a particular region of the image will negotiate to become the envoy for that territory and provide a proxy service for other hosts making concurrent requests. Realignment of control only occurs in response to sustained activity; sporadic updates go through the existing envoy without changing the topology.

Independent activities by independent hosts divide the file system naturally to give each host control over its region of interest, and sustained proxy/client relationships persist only when sustained updates are actually concurrent and overlapping, in which case consistency makes sustained communication between the hosts in question unavoidable under any scheme.

Updates can be pulled by other hosts directly from the appropriate envoy, or from the object layer by using the updated ID that the envoy provides. Hosts that don't demand completely synchronous reads (consider most OS images, static web content, or other data that can be a minute or an hour old and still be adequate) can ignore the proxy for read operations and only synchronize with it periodically or in preparation for a write operation.

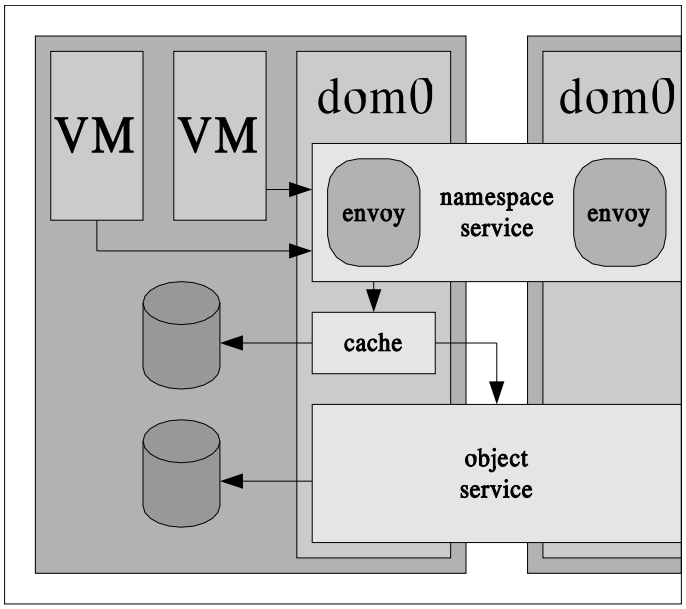

Figure 2: Physical hosts deploy the Envoy services in one VM with control over local storage. This VM exports a file system interface to other VMs resident on the host. The namespace service is distributed over all hosts that share a file system view; each host may assume envoy responsibilities for territories used by VMs on that host. The object service is distributed over the whole cluster, and is relatively independent of the namespace services.

2.3.4 Implementation

In a typical deployment, all physical hosts will share in the duties of all services of the Envoy system, resulting in an implementation depicted in Figure 2. In a hosted VM environment, the host is trusted but the hosted VMs may not be. Direct control over local storage is better reserved for a trusted VM, and running the Envoy services on a single VM per host gives additional opportunities for shared caching. In an environment where heterogeneity is supported but some degree of homogeneity is expected, a shared cache is likely to offer substantial benefits.

The namespace service runs on every host in the controlling VM, and each host can export a coherent file system to multiple VMs. Envoy supports many logical file systems with independent name spaces, and we call each instance a *view*. The namespace service as a whole manages the full set of views in active use, but the namespace service on a particular host only participates in the management of views used by its local client VMs. When a host participates in multiple views, it is effectively running a series of discrete namespace services that interact only through a shared cache. Consequently, a single host may act as an envoy for territories from many distinct views.

The duties of the namespace service on a particular host for a particular view are essentially:

1. Maintain a catalog of envoys
2. Negotiate envoy changes in response to usage patterns
3. Service requests under the control of the local envoy (requests may be from a remote host) as well as reads that are independent of any envoy
4. Forward requests that can't be serviced locally to the appropriate remote envoy

The object service is logically independent of the namespace service, and the physical disks could even be hosted on a different set of hosts or on a storage area network or a network-attached storage device. We expect that most environments will benefit from using cheap storage attached to each host rather than separate, dedicated storage. Every client of the object service will retain a full catalog of the hosts contributing storage so that finding the correct host based on an object ID can be done locally. This catalog of object hosts will be relatively static, changing only when the storage pool changes.

3 Schedule

The Envoy system has a few major components that can be developed and tested in steps. I organize the major phases around deadlines for possible paper submissions.

3.1 Namespace service design and prototype (FAST '05[2], 13 July 2005)

The first phase of development will focus on the namespace service with its protocols and algorithms. Initially, the namespace service will be backed by a normal file system instead of accessing an object service layer. The emphasis will be on defining and implementing the necessary protocols and algorithms, including:

- **Client VM to local namespace service protocol:** This interface presents a coherent file system to client VMs from the namespace service instance hosted on the local host. Existing protocols like 9P2000.u[3] or NFS[11] are promising candidates.
- **Namespace service to namespace service protocols:** This interface manages the catalog of envoys and forwards requests to non-local envoys as appropriate. Two distinct types of traffic will pass over this link, which may be considered as separate protocols:
  - Data traffic forwarded to the appropriate envoy for servicing
  - Metadata traffic to coordinate all hosts that share a file system (including handling failures, arrivals, and departures) and to negotiate changes in envoy territories.
- **Namespace service to object service protocol:** This will start out as a stub interface that communicates with a locally mounted file system (possible over NFS).

This phase of development will result in a prototype implementation of the namespace service that will allow mounting file systems from multiple clients. This will provide a testbed for experimenting with variations on the design choices and validating the overall concept.

3.2 Object service design and prototype (NSDI '06[4], October 2005)

The object service operates independent of the namespace service. In addition to replication and basic persistence, it provides the primitive operations to allow lightweight snapshots and file system creation and forking. The principle components include:

- **Namespace service to object service protocol:** Clients of the object service will cache a catalog of participating object service hosts with enough information to map a location-independent object ID to a set of hosts that hold replicas of the object. This catalog only requires updates when the set of object service hosts changes.

[2] `http://www.usenix.org/events/fast05/`
[3] `http://v9fs.sourceforge.net/rfc/9p2000.u.html`
[4] `http://www.usenix.org/events/futureevents.html`

- **Object service data layout:** Data stored by the object service needs to be quickly accessible and reliably replicated. The object service needs to support the addition and removal/failure of hosts and disks, and ideally some kind of garbage collection. Lightweight snapshots and forks at the namespace level require specific support from the object layer as well.
- **Namespace to object mapping:** In the transition from using the namespace service with a simple file system backing to using the object service, careful consideration for how to store file system metadata in objects will be required.

This phase of development will result in a working prototype of a complete system.

3.3 Evaluation, related work, writing

Once a complete prototype is in place, it will be necessary to evaluate it and refine the design and implementation. I will also spend time researching related work to judge the success of the Envoy system. Envoy introduces a novel way of handling contention that may have application in other kinds of storage systems, and the experience with the prototype should suggest directions for future research.

Around this time I would like to start structuring and writing my dissertation, though speculating about a precise schedule is probably premature. Much will depend on what I learn while building Envoy and while continuing with other projects. I hope to submit during the summer of 2006.

References

[1] Russ Ross, First Year Report, January 2004.

[2] Russ Ross, Copy-on-write NFS Server. `http://www.russross.com/CoWNFS.html`, 2004.

[3] K. Fraser, S. Hand, T. Harris, I. Leslie, and I. Pratt, The Xenoserver Computing Infrastructure. Tech Rep. UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, January 2003.

[4] Sun Microsystems, RPC: Remote Procedure Call. RFC-1057, June 1988.

[5] Gerd Stolpmann, OCaml RPC Library. `http://www.ocaml-programming.de/programming/rpc.html`, 2003.

[6] E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris, Global-scale Service Deployment in the XenoServer Platform. *Proceedings of the 1st Workshop on Real, Large Distributed Systems (WORLDS'04)*, December 2004.

[7] Russ Ross, Introduction to Functional Programming, `http://www.cl.cam.ac.uk/Teaching/2004/IntroFuncProg/`, 2005.

[8] A. Warfield, R. Ross, K. Fraser, C. Limpach, S. Hand, Parallax: Managing Storage for a Million Machines, under consideration for the Tenth Workshop on Hot Topics in Operating Systems, 2005.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugabauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *ACM SOSP*, October 2003.

[10] Russ Ross, Thesis Proposal, January 2004.

[11] Sun Microsystems, NFS Version 3 Protocol Specification. RFC-1813, June 1995.