Envoy from 10,000 feet

Russ Ross <rgr22@cl.cam.ac.uk>

December 2005

1 Introduction

This document describes the basic design of the Envoy storage system. Envoy is designed for clusters of machines managed using virtual machines, where each machine contributes storage to a pool used by the entire cluster. The disk space is managed by a dumb, stateless object storage layer, above which any number of file hierarchies are created. Envoy is designed to distribute control over the hierarchies to the machines most interested in them. This allows for minimal coordination overhead and aggressive caching with virtually no coherence management. The service also supports snapshots and file system forking for administrative convenience.

We take a bottom-up approach, starting with the object storage system and working up to the dynamic namespace management and snapshot/fork control.

We make a few basic assumptions:

- The network is well provisioned, and there is no significant difference in distance between different machines in the cluster.
- Each machine has an administrative virtual machine that is trusted. An instance of the envoy service runs in this VM, and it serves all clients on the same physical machine.
- A catalogue of machines in the cluster is available. We ignore resource discovery.

There are two distinct administrative layers in a complete Envoy system. The object storage layer provides an API for accessing objects with metadata, corresponding roughly to files in a normal file system. The API is simple and dumb. The envoy layer provides an outward-facing file system API, coordinating all file and directory semantics and relying on the object storage layer for stable storage.

2 Object Storage Layer

The object storage layer provides a mapping of numeric IDs to objects. An object is an extent of opaque data with a few metadata fields including size, timestamps, mode, user name, and group name.

2.1 On-disk layout

For simplicity, we storage each object as a file in a normal Linux file system. We use most of the metadata fields provided by the backing file system, and supplement it with a few other fields stored in the file name.

Objects are stored in a hierarchy of directories according to the object ID. Each directory level divides the ID space according to the next few bits in the ID, i.e., the top level directories partition the objects by the most significant bits of the ID, and subdirectories further partition them by progressively lower-order bits from the ID. The number of subdirectories at each level is chosen to fit each directory within a single disk block. All object files are stored in the leaves of this directory tree.

File names are of fixed length, incorporating the lowest-order bits of the ID and a few metadata fields. As before, the number of files in a 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 file system cache with the directory block, so when the object file itself is accessed it will be located through the cached directory. This lets us store metadata in the filenames without incurring any extra disk seeks.

2.2 API and semantics

The object storage layer makes no attempt to coordinate access to objects. It is assumed that all synchronisation is done at a higher level. The API is simple and stateless, with the following basic procedures:

- `Reserve(range)`→`range`: reserve a sequence of unallocated IDs from within the given range
- `Create(ID, stat)`: create a new object and set its metadata
- `Clone(ID, newID)`: create a new object by copying an old one
- `Read(ID, range, atime)`→`data`: read a range of bytes from an object and set the object's `atime` field
- `Write(ID, range, data, mtime)`→`count`: write a range of bytes to an object and set the object's `mtime` field
- `Stat(ID)`→`stat`: get the metadata for an object
- `Wstat(ID, stat)`: set the metadata for an object

Many machines in the cluster offer object storage services, but the individual object servers are unaware of each other. Reserving IDs on a particular object server does not reserve them on any other servers and reservations are not enforced, so this procedure is only useful when combined with higher-level coordination.

An object ID is considered a globally unique name for that object. When replication is introduced, it is assumed that all replicas are given the same object ID. One object server from a replica group is nominated as the master; it is the only one to which `Reserve` requests will be sent, and all servers within that replica group are bound by its allocations (although the servers themselves do not offer any enforcement mechanism). Object servers can participate in multiple replica groups, so the `Reserve` procedure includes a parameter to indicate the ID space from which the allocation should come.

2.3 Open problems

The mapping of object IDs to specific storage servers is not yet defined. For now, a configuration file with a static mapping can be distributed to all participants, and changing that mapping will require a restart of the complete system. No scheme for migrating objects in response to load or failure is currently defined.

Storage servers blindly comply with all requests made to them. They do not restrict object IDs to a certain range of values or attempt any security provisions. Authentication and encryption can be added in an orthogonal way, but for now we merely acknowledge the possibility and make no attempt to implement it.

3 Envoy Layer

The envoy layer provides a file system interface directly to client machines or virtual machine instances, exported through the 9P[1] protocol. It provides access to any number of complete file systems, which can be shared between multiple clients or used privately. Support is provided for forking file systems and taking read-only snapshots, allowing clients to customise from a complete base file system provided by another client or by a system administrator.

The Envoy layer uses the object storage layer to provide a backing store, and builds directory hierarchies over the objects. It coordinates access to ensure that at most a single Envoy instance, called the owner, has direct write access to any object at a given time. Other instances wishing to access that object forward their requests to the owner. Revocable read-only leases allow envoy instances to skip this forwarding process for objects when it is only being accessed by readers, and dynamic reconfiguration in response to demand optimises ownership to minimise forwarding traffic.

The architecture is designed to allow interested envoys (and the clients they represent) to maintain local persistent caches with no coherence overhead whenever possible. Read-only leases allow envoys to communicate directly with the object storage layer for most read requests; assuming ownership extends this ability to situations where an envoy is the only player accessing a writeable object or where contention exists and it is the dominant player. In a steady state, forwarding requests (and losing the benefits of a local persistent cache) is only necessary in situations of genuine contention, and it is limited to the contended regions of the file system.

3.1 Building a file system on objects

A file and its metadata can be represented directly by an object, which takes on an organisational role similar to an `inode` in a a conventional Unix file system. Devices, symbolic links, and other special files can similarly be stored as objects whose contents describe the special file. For this we borrow the syntax used by 9P, which defines a simple set of string representations of special file attributes for use in `stat` and related procedure calls.

Directories are the only file type that must be specifically managed by the envoy layer. The object layer does not provide any special services for directory files, so the envoy layer accesses them like regular files. Just as with regular file systems, clients are prevented from accessing directories except through directory-specific procedure calls.

Directories are stored as a series of blocks, with each block containing the following information:

1. (16 bits) Block version number: this number is incremented each time a file listing in this block is modified, to allow `readdir` requests to easily detect race conditions.
2. (16 bits) End of data offset: directory entries in a given block are always packed to the beginning of the block. This value indicates where new entries can be added, and tells `readdir` requests when they should move from the current block to the next block in the directory.
3. List of directory entries: A list of directory entries follows, each one containing the following:

[1] We use 9P as shorthand for `9P2000.u`, a Unix variant of the `9P2000` protocol defined as part of the Plan 9 project. See `http://v9fs.sourceforge.net/rfc/9p2000.u.html`

- (64 bits) The object ID for this directory entry
- (1 bit) A flag indicating whether or not the object is marked copy-on-write
- (15 bits) The length of the file name
- (variable length) The file name encoded using utf-8

The most noteworthy element here is the copy-on-write flag. If it is set, then the object referred to is considered read-only (note that this mechanism is transparent to clients and is distinct from the file's mode attribute). If a client issues a valid write request (or a metadata change other than `atime`) then the object will be cloned, and the object ID of the old object will be replaced with the new ID and given a clear copy-on-write flag. If the object is itself a directory, the new clone will also be scanned and the copy-on-write flag of all of its entries set.

The copy-on-write flag and its accompanying semantics allows for lightweight snapshot and fork operations, the details of which are discussed in Section 3.3.

3.2 Dividing a namespace across hosts

The 9P protocol is stateful, and files being accessed or queried are represented by 32-bit values called FIDs. A FID is associated with a user and group as well as a file, and most procedures are relative to one or more FIDs. Navigating a directory hierarchy, for example, is done by the `walk` operation, which either moves a given FID to represent another file using a relative path, or creates a new FID attached to the endpoint of the relative path.

For a namespace divided over a set of machines (assuming a static configuration), each host needs to know how to:

- Serve requests for a FID directly (when the envoy owns the file or has an active lease)
- Forward requests to the appropriate envoy
- Walk either up or down the hierarchy from a given FID, either by owning/leasing the appropriate directories or by knowing the envoy that owns them
- Manage leases
- Create new FIDs at the root of the hierarchy

FIDs are closed (or *clunked* in the 9P terminology) immediately when not in use, so an envoy only needs to remember FIDs for active files (in use by its clients or forwarded to it by another envoy) and information about the immediate hierarchical neighbourhood of those FIDs. The directory hierarchy is managed in a top-down fashion; leases and ownership are always granted to an envoy from the owner of the immediate parent directory and can be revoked by that owner.

As long as an envoy has ownership or an active lease for an object, it can cache it. 9P defines a version number for each object that can be used to reliably detect changes, so even after a restart or a lapsed lease, an envoy can verify and reactivate objects in its cache with only a metadata check against the storage layer.

3.3 Snapshots and forks

Envoy is designed to support lightweight snapshot and fork operations. Here we describe both the mechanisms for implementing these operations, and the administrative infrastructure to support a system that may be home to many different file system images.

3.3.1 Mechanisms

Objects in the storage layer do not have a read-only attribute, but a link to an object does have a copy-on-write flag, which—if set—marks that object as read-only. This mechanism is the same as the one we developed in *Parallax*, allowing snapshots without having to mark all affected objects as read-only. Instead, an envoy only considers an object writable if the entire path from the root of the hierarchy to the object enjoyed clear copy-on-write flags.

This suggests the mechanism for snapshots and for forks (which are essentially the same thing). Conceptually, the copy-on-write flag in the link to the root directory is set, marking the entire tree as read-only. In practice, we immediately copy the root directory and mark all of its outgoing links as copy-on-write. Owned objects below the root must either relinquish ownership (losing write privileges), or the path to that object must be copied and any ownerships along the way updated. Since an object owner controls ownership of its descendents, it must already know which ownerships have been granted and can propagate the clone requests in a recursive manner.

It is worth noting that leases do not need to be updated immediately as ownerships do. Leases are only granted for read-only access, and the entire snapshot upon which a fork is based, or—equivalently—the historical version left behind in a snapshot operation, is read-only. Leases can continue to be served using the historical copy. The normal mechanisms for revoking leases and changing ownerships are sufficient at snapshot time just as they are at other times.

3.3.2 Administration

The entire set of file systems maintained by an Envoy cluster is administered as one big tree. For administrative convenience, the top levels of that tree are given special semantics, effectively creating a special file system tree whose leaves are normal file systems. Mount requests from clients are normally only permitted at these leaf nodes, but access is also given to the meta hierarchy for administrative purposes.

Directories at this level contain either other directories (for organisation), or file system roots for a single file system (but not both). A file system can be mounted in two ways. A specially-named directory called `head` gives access to the live version of the file system, and snapshots of the file system are accessed through directories named with increasing integers. In addition, symbolic links are permitted to snapshots to allow friendlier names that can be advertised. An example illustrates:

- `/pub/fc4/1`: the first snapshot of FC4
- `/pub/fc4/2`: the most recent snapshot of FC4
- `/pub/fc4/head`: the active, writable version of FC4
- `/pub/fc4/unpatched`: a symbolic link to `1`
- `/users/rgr22/home/head`: the active version of a user's private file system

Users (or administrative tools) with proper credentials can create directories to start new file systems. The new tree can either be created from scratch through a `mkdir` request with the name `head`, or as a fork of an existing snapshot (but not active head) from another tree through a `symlink` request linking the name `head` to the path of the desired snapshot.

Snapshots are created by writing to a special file called `snapshot`, which signals Envoy to create a snapshot and move `head` (and its currently mounted clients) to a newly cloned copy of the tree.

3.4 Dynamic reconfiguration

Leases are always obtained from object owners, and object ownership can vary on a per-object basis. Ownership of the root of a file system tree is initially granted to the envoy representing the first client that tries to mount it. That envoy is responsible for issuing leases and granting ownership to descendent nodes. A lease for an object implicitly covers all descendents of that object as well.

Leases only cover read access, so when a client makes a write request its envoy must take a series of steps. The owner of the object is the same owner that granted the lease, so the request is forwarded to it. The owner, upon receiving a write request, immediately revokes all leases covering that object. It may then either maintain ownership and serve requests directly, or grant ownership to the envoy representing the writing client.

A few other undeveloped thoughts:

- Leases are granted for an object, with a list of exceptions for descendents of that object. I may grant you a lease for `/home`, but also tell you the envoys that own `/home/nlm24` and `/home/rgr22` (which may be the same as the owner granting the lease; this indicates that requests for those directories must be forwarded to the owner, and leases to descendents of those directories must be administered separately). Leases can be revoked by an owner in their entirety, or new exceptions can be added.
- Envoys don't request leases any more than they request ownership. Requests are forwarded to the owner, and the owner may then choose to issue a lease.
- Ownership can always be reclaimed by the owner higher up the tree. To trigger a reconfiguration, the envoy that owns an object sends a special message to the owner higher up, nominating a new owner. The envoy that has been serving requests has the most information to make these decisions, but it still acts through the owner hierarchy, i.e., there are no direct sibling transfers. Instead, the parent owner reclaims ownership (which has to be propagated to lease holders and descendent owners) and then grants it to the newly nominated owner (which may happen in one step).
- Ownership grants may require a chain of object clones in order to clear all the copy-on-write flags on the path to the object. Ownership is an implicit certification that the copy-on-write flags are clear for all ancestors from the object to the root of the tree. As mentioned above, lease holders can continue serving from the old snapshot, though the lease may need to be modified (which probably means revoked and re-issued) to indicate the new parent object for the owner. Since the caches are managed mainly by object number, many of these metadata changes sort themselves out at the data level.
- An envoy's ownership catalogue may get slightly out-of-date while changes are occurring, causing it to forward a request to an envoy that can no longer service it directly. In this case, the receiving envoy rejects the request, and the originator waits until it receives an update message before restarting the transaction.