CPCMS: A Configuration Management System Based on Cryptographic Names

Jonathan S. Shapiro John Vanderburgh
shap@cs.jhu.edu vandy@srl.cs.jhu.edu
Systems Research Laboratory
Johns Hopkins University

Abstract

CPCMS, the Cryptographically Protected Configuration Management System is a new configuration management system that provides scalability, disconnected commits, and fine-grain access controls. It addresses the novel problems raised by modern open-source development practices, in which projects routinely span traditional organizational boundaries and can involve thousands of participants. CPCMS provides for simultaneous public and private lines of development, with post hoc "publication" of private branches.

This paper describes the repository architecture of CPCMS, and in particular its use (and abuse) of cryptographic naming mechanisms to achieve collision-free disconnected operation.

1 Introduction

CPCMS, the Cryptographically Protected Configuration Management System, is a new configuration management system that provides scalable, distributed, disconnected, access-controlled configuration management across multiple administrative domains. All of these features are enabled by the pervasive and consistent exploitation of cryptographic names and authentication.

Software configuration management (CM) systems provide multiple developers with a consistent, shared view of a project that is changing and evolving. When development occurs in geographically dispersed locations and is split across multiple organizations, issues of access control, integrity management, update distribution, and high-volume change integration become significant concerns in choosing a CM system. This is especially true in open source projects. In addition to the "core" team, which may consist of developers from several companies, an open source project may have small patches submitted by thousands of users that need to be integrated. As open source projects become more successful, the demands placed on the supporting CM tools grow with them.

In the most immediate sense, CPCMS was brought about by the expansion of the EROS team and the impending release of the EROS system to outside development groups. EROS [SSF99] is an open source, secure operating system that is currently gaining mainstream interest. From 1991 to 2000, the EROS project relied on the CVS system [Ber90] for its configuration management. As the project reached the point where outside developers would be making substantive changes, it became clear that CVS did not meet several of our requirements:

- First-class support for configurations, and in particular for *rename* operations.
- Support for disconnected commit and distributed repositories.
- Scalable, consistent replication. While tools such as `rdist`, `rsync`, and `mirror` provide replication, a user "updating" at the wrong moment can easily end up with an incomplete or inconsistent result. There is no simple way for a user to detect this.
- Per-file access controls. The CVS-ACLS system does not protect RCS logs.
- Support for development groups that span multiple companies or administrative domains. While uncommon in proprietary projects, such groups are nearly universal in open source efforts. CVS provides no authentication mechanism suitable for multi-organizational development.

When only a limited number of developers worked on the EROS code base, these shortcomings were relatively manageable annoyances. As we contemplated the prospect of hundreds or (optimistically) thousands of potential contributors and "lurkers," these requirements suddenly loomed as serious concerns.

Subversion, a successor to CVS currently under development by Tigris.Org [SVN], addresses many of these issues, but is notably lacking in disconnected commit support or multi-organizational authentication. Other CM systems similarly did not address our requirements.

Ignoring the problem of delta management in the repository, a configuration management system is not conceptually difficult to design and build. The principle architectural challenges in distribution lie in naming and consistency. Cryptographic naming seemed to provide a solution for both problems that we wished to explore. Cryptographic names uniquely identify the content they represent. They are collision-free, but can be generated without shared access name binding agent. They are also pragmatically impossible to forge. A properly constructed cryptographic name therefore provides an inherent integrity check on the content it names.

Finally, we felt that a CM system based on cryptographic names might provide an interesting basis for future research. In particular, different projects impose different policies on their repository content and often use local customizations (e.g. triggers). We were intrigued by the possibility that a safe programming language might be integrated with the CM client to enable this in a safe, platform-independent way. This is a "future," but the current base will clearly support it.

This paper describes the architecture and the implementation of CPCMS, our early experiences and problems with the first design, and the results obtained by its reformulated successor: OpenCM.

2 Comparison to CVS

CPCMS is ultimately intended to replace CVS, and the command line CPCMS client has a very similar "feel" to the CVS client. Like CVS, the basic CPCMS usage model is to check out some branch of a product, edit it locally, perform integration updates along the way, and ultimately commit the result back into the repository.

CPCMS differs from CVS in several ways:

- To support rename operations, CPCMS maintains a set of bindings from workspace object names (e.g. file names) to objects.
- Versioning is on configurations, not on files. Every commit, whether a single file or a complete replacement of the code base, creates a new configuration object in the repository.
- In a laptop configuration, CPCMS replication can be used to cache the originally checked out version in a laptop-based repository. Status checks and "undo" can be done without connecting to the network.
- CPCMS is workspace-neutral, in the sense that it can support workspaces other than files and directories. For example, the information architecture could readily support an object workspace such as those used by several integrated development environments for Java.
- CPCMS uses cryptographic rather than password-based authentication.
- Commits need not be made to the originating repository. This allows work to be committed – and potentially undone – while disconnected.

The last point is important, as it is the basis for disconnected development. A user can check out a project, make some changes, and can realize after the fact that they are disconnected or for some reason lack the authority to check in those changes. Instead of losing their work, they can instead create a new branch in a local repository, allowing development to proceed. At a later time, after the impediment has been removed, this local branch can be synchronized to the central repository and a merge can be performed.

3 System Architecture

In this section we describe the CPCMS architecture and how this architecture facilitates replication and disconnected development.

3.1 A Client/Server System

CPCMS is a client/server system. The client implements all of the semantics of the configuration management system. The server provides an archival object repository with specialized support for objects that are frequently revised.

Clients use an RPC-based protocol to make requests on the server over an encrypted, mutually authenticated connection provided by the Secure Sockets Layer (SSL). While remote procedure calls do not achieve the best possible bandwidth utilization, they provide a reasonable compromise between client simplicity and effective network utilization. Most of the possible asynchronous exchanges between a CPCMS client and server can be simulated in the remote procedure call interface by providing an interface that supports bulk batch transfer of managed objects. A more asynchronous interface and protocol is under consideration for a future version of CPCMS.

3.2 Naming in the Object Store

Objects in a CPCMS repository are divided into two categories: frozen and mutable.

A **frozen** object is immutable. From the perspective of the repository, each version of a file, each configuration, and each access control list is a frozen object. Frozen objects cannot be modified. Changes to (e.g.) file content in the repository are recorded by introducing new frozen objects corresponding to the new state, and updating some mutable object (see below) to point to the new objects rather than the old ones. This is because a CM repository is an archival store. While a new version of a file may be introduced, the old version must continue to exist as an independent object for reasons of historical traceability.

A **mutable** object is one whose identity must remain the same across modifying operations. A branch, for example, is a sequence of configurations. As new configurations are added, the branch grows, but the branch itself retains the same identity after the commit as before. When a new configuration has been appended to a branch, we say that the branch has been **revised**.

Frozen objects are named by the cryptographic hash of their content. A cryptographic hash provides a short, unforgeable bit-string that uniquely identifies the original content. Because they are unforgeable, cryptographic hashes are also collision resistant. CPCMS uses the SHA-1 cryptographic hash function for frozen object names [FIP94]. Mutables are named using server-generated **swiss numbers**, which are identifiers generated using a strong random number generator. Mutable names also contain the server's unique identifier. The mutable itself contains the signature verification key of the originating server. The mutable content plus its name are signed by the originating server, and the signature is distributed as part of the mutable.

Both naming mechanisms ensure unforgeability of content. For this reason, we refer to these two types of names collectively as **true names**. Server-generated true names are a pair consisting of the server's unique identity followed by the randomly generated server-relative object ID. The server checks for collision of mutable true names.

CPCMS has no means to recover from a collision of cryptographic hashes. We are aware of no mechanism that (in principle) can provide such recovery without central coordination. The SHA-1 algorithm is a 160-bit hash. In such a hash, the expected number of objects required to generate a hash collision is 2^{80}. As an empirical test, we have obtained copies of several large CVS-based repositories and extracted every version of every file in these repositories. To date, we have not observed a collision except where content was actually identical.

Identical content is the one case in which cryptographic name collision is guaranteed. From the CPCMS perspective, this is not only desirable, it is essential. Name collision on identical objects is the basis for eliminating unnecessary communication. As a practical example, if two otherwise empty files are independently checked in with the same copyright notice, only one frozen object will be stored and both configurations will name the same frozen object. Given that the bits are identical, it does not matter which entity is returned on checkout. Because the CPCMS repository does not interpret the content of its stored objects (other than for garbage collection), the semantics of objects in the repository is completely defined by their content.

Having acknowledged the "Achilles heel" in the CPCMS design, it should also be said that the use of cryptographically generated names has several desirable properties:

- Given two objects with different content, we can generate non-colliding names without appeal to a centralized naming mechanism. This is a fundamental requirement for successful disconnected commits.
- The hash serves as an integrity check that lets us verify the integrity of the object. By recomputing the hash, the client can determine whether the content of the object has been improperly altered.
- Because hashes are universally unique, the hash can be used to avoid network transmission of objects that are already present at the destination.
- Because hashes are sparsely allocated from a very large space, they are pragmatically impossible to guess. This is a crucial underpinning for the CPCMS access control architecture, which is described below.

Using cryptographic object names eliminates the need to deal with case sensitivity, file name length, or path length issues in choosing server-side names. In fact, a file-independent naming strategy divorces the repository implementation from underlying platform dependencies altogether. Repositories can be constructed to use storage strategies ranging from use of an object database to storage within the server file system itself via some transformation on the true name. Further, it is straightforward to construct a "union" repository by which a new repository implementation can be run in parallel with an old one for testing purposes.

3.3 Naming Managed Content

The CPCMS server makes minimal assumptions about the client-side semantics of managed content. Ignoring issues of garbage collection, the CPCMS server provides configuration-based versioning of uninterpreted "blobs" (binary large objects) and records bindings between client-side names (*C-Names*) and internal object names (*true names*). To the server, these c-names are uninterpreted strings. It is entirely up to the client what names to use, and what organizational semantics should be associated with these names and their bound content in the client-side workspace. Examples of possibly valid c-names include:

`sys/kerninc/Process.h` (a file)
`org.apache.xalan.xsl.Process` (a class)
`<Object 0x0040824>` (an object pointer)

For example, CPCMS does not assume that the objects managed consist of files, nor does it depend in any way on file semantics. While the current command line client is designed to manage workspaces of files, an alternative client might use the repository to manage an object workspace just as easily. As a result, the CPCMS repository (and most of the logic of the file-based client) could be used for workspaces that are not file oriented, such as a Java or Smalltalk class repository. One could also imagine directly binding a CPCMS branch as a namespace that can be exported directly through a web server (we are in the process of building this).

Similarly, the CPCMS server has no responsibility for client side serialization and deserialization. CPCMS delivers the requested blob. Further interpretation is left to the client. Correct client interpretation is facilitated by recording the "type" of the object along with its name binding. For example, the client may record that the object content is ASCII at check-in time in order to know that newline conversion may be needed on checkout. The server knows nothing of client side canonicalization.

The file-oriented CPCMS client stores configurations as a set of (c-name, type, truename) triples. When the CPCMS client processes the triple:

`(kerninc/process.h, ASCII,` *truename*`)`

it interprets this to mean that the intervening directories must be created if they do not already exist. That is, the client views the existence of directories as an emergent consequence of the need to bind c-names. Directories typically have no first-class existence in the repository. This means that directories are renamed in the workspace as a side-effect of renaming the files that live in them. The client facilitates this by simulating the behavior of the UNIX `mv` command when it is passed arguments that imply that a "directory" is being renamed.

Where it is necessary to ensure that an empty directory appears in the client working tree, a name binding with type DIR can be created. This name binding is handled in the same way as the file name binding shown above, and renaming is handled similarly.

3.4 Naming for Users: Take 1

While cryptographic names solve the repository's object naming problem, they are not terribly useful to human beings. As a user, I want to work on "the main branch of the EROS project", not some strange ascii-encoded random number.

To facilitate human association, the original CPCMS architecture provided a per-server namespace of "pet names." Each project or branch carries with it both a human readable description and a "nickname." When a project or branch is first replicated to a server, the server generates a petname by applying a collision avoiding transformation on the nickname. This provides both a means of human association and a means by which two users on distinct servers can arrive at closely related names for the projects they share in common.

Experience shows that this design does not work. The CPCMS architecture encourages users to make many branches, and the results quickly create a pet name space too crowded to be practically useful. One usability conclusion from this is that users need to be actively involved in name selection for the naming mechanism to be useful. The CPCMS architecture also makes it difficult to supply an equivalent to the CVS "tag" mechanism. In the course of cleaning up this and other problems in the original CPCMS design we arrived at a different and more workable approach, presented in Section 7.

3.5 Replication

CPCMS's scalability is built on replication. Because frozen objects are named by their cryptographic hash, these objects are non-colliding, and mutual replication of such objects is straightforward. For mutable objects differences are resolved by taking the copy with the higher sequence number. This works because of two constraints that are imposed on mutable objects:

- Every mutable object has a single server that is said to "originate" that object. Valid modifications to the object can only be performed by the originating server.
- Each change to a mutable object is signed by the originating server. The signature verification key is itself embedded in the mutable's state. Accuracy of an alleged version can therefore be determined by checking the object signature.

Ironically, the problem in the resulting replication architecture isn't so much deciding what to keep as deciding what to throw away. Client side repositories can rapidly come to hold many object versions that are no longer of interest. At present, a simple garbage collection mechanism that preserves the top N configurations of all unowned branches is used. Other options are being considered, and this is an open area for future work in CPCMS. One reviewer of this paper suggested "last N days" as a potentially useful metric.

A second reviewer encouraged us to re-examine some of the approaches used in the Elephant file system [SFH+99], which must similarly decide when to forget. The Elephant design works in part because each content version is logically independent. Any object pointers implicated by forgetting a file version are owned by the file system, which is free to update them. This approach does not translate straightforwardly into an archival object system, because objects in the archival store embed pointers to predecessor versions.

CPCMS content is encoded in ASCII XML form, allowing replication across heterogenous servers without regard to issues of word order or host platform restrictions.

4 The CM Schema

The CPCMS configuration management schema can be divided into two parts: the content schema and the configuration schema. Each content object, such as a file, is implemented as two repository objects (Figure 1):

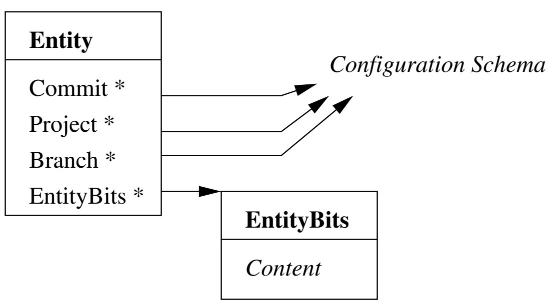

Figure 1: Content schema

Entity An Entity is a single version of a managed object (a file, the text of a function, a document, etc.) Each entity records the identity of the commit description record under which it was created, the names of its containing project, branch, and version, and a list of its immediate predecessors (it can have two – the second due to a merge). It also records the *c-name* of the object, which is the name by which it is referenced in the client environment (e.g. "`docs/dcms.xml`").

EntityBits An EntityBits object contains the actual content of an entity version (along with its length). In RCS and SCCS, entity metadata and content are stored in a single file. By separating these in CPCMS, we allow the merge algorithm to trace the evolution of objects without actually fetching the large volume of data associated with the content of each version.

Further, the cryptographic hash by which the EntityBits object is named allows the merge algorithm to trivially check whether the file version that is being merged is identical with the one already in the workspace – which is the majority of cases. The merge algorithm simply constructs an EntityBits object in memory from the file already present, and checks if the true name of this object is the same as the true name of the EntityBits to be merged.

The schema supporting configurations is more complicated (Figure 2). Every CPCMS commit generates a new CommitInfo object, a new Change object, and an append to an existing Branch object. The CommitInfo record is separated from the Change object so that each Entity can carry the history of its changes and ancestry. The configuration schema must also track the responsible party for each change.

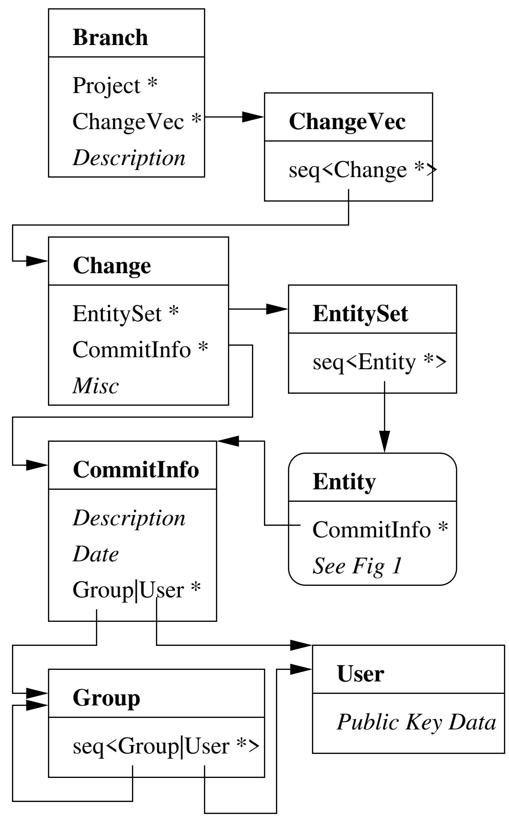

Figure 2: Configuration schema

Project and Branch objects are unusual in that these are mutable objects. Every mutable object carries a "read group" and a "write group" identifying who can read or modify the object. Each Project and Branch object provides a human-readable description of the corresponding project or branch.

The repository storage format for objects includes their type in the first word, allowing tools such as integrity checkers or browsers to be constructed, and supporting occasional garbage collection to reclaim storage from uncompleted transactions.

5 Relationship to Objectives

CPCMS is primarily focused on three problems: scaling, disconnected commits, and access control. Each of these is a source of difficulty in current CM systems. In this section, we discuss how the CPCMS information architecture addresses these issues.

5.1 Scaling

By "scaling," we mean that we wish to enable hundreds of thousands or millions of users to track updates and revisions to one or more aggregate works. This is clearly beyond the number of users that can be handled by a single server, and doing so is not an objective. Rather, we wish to ensure that the CM schema allows simple incremental replication so that multiple, satellite repositories can be used to serve a large user base.

Connections between machines are sometimes lost, and machines occasionally crash, yielding incomplete transfers. Both clients and servers are able to validate their current content without consulting other agents. Using purely local information, this examination can determine:

- Which objects in the repository, if any, have been incompletely transferred or (equivalently) have been locally corrupted. These objects must somehow be re-acquired.
- Which objects, while uncorrupted, are not referenced by locally visible configurations. These should (eventually, but not too eagerly) be garbage collected.
- Which configurations are complete, in the sense that all constituent entities are locally available.
- Which entities in the client working arena have "gone missing" and should be recovered by re-acquiring them from the server. Because CPCMS (by default) allows file modifications in the working arena without locking, it is usually unable to distinguish between corruption and modification. The rule here is "notify where possible, but don't make an uncertain fix."

By combining checks of the true names against the actual content and traversing the reachable objects, each of these concerns can be validated by a client application.

5.2 Disconnected Operation

Under the heading of "disconnected operation" we include two types of usage: revisions committed by disconnected developers and revisions made in private lines of development.

As the storage and processing capacity of laptop machines has increased, their use as development machines has become ubiquitous. The bulk of this paper was first written on a laptop while sitting in front of a fireplace or on various airplanes. The greater part of CPCMS itself was written on various airplanes between New York and Seattle. Neither of these locations is networked.[1] While dial-up connections from hotels are usually possible, they are hardly ubiquitous and are usually quite expensive.

There are two consequences of this type of migratory development:

- Development mistakes are increasingly made in disconnected locations. The ability to "check in" locally in order to have a means of recovery is increasingly important.
- The configurations resulting from these check-ins will later need to be integrated back into the main development repository. This integration should preserve the entire history trail of the change sequence, not just commentary on the final resulting delta.

This problem can be solved by introducing a hierarchy of repositories, as in NUCM [dHHW96], but semi-connected hierarchies require use of a "lock before modify" approach. Locking requires connection, which largely defeats our goal of disconnected commit. Also, we have found in practice that locking is disruptive for mid- to large-sized projects. Header files, in particular, become sources of lock contention. Post-integration, as provided by CVS, is (subjectively) more productive in properly structured projects. In any case, an unrelated aspect of open source development argues against hierarchical repositories: the need to operate under multiple administrative domains.

Open source developers are frequently part time. In the context of their employer they work on one set of projects. In the context of the open source community they work on another. If we imagine that their laptop holds a repository that serves as a local cache, then this repository has two parents: one associated with their employer and one (or more) associated with open source efforts. The hierarchy, if any, is a function of the *project*, not of the *repository*.

The CPCMS solution to this problem is to allow private branches. The laptop user creates a laptop-local repository that mirrors the desired project from the primary server for that project. To develop remotely, a new branch is created in the laptop repository and commits and changes are applied locally. At a later time, when a connection is available, the laptop branch can be re-integrated (via merge) into the main line of development.

[1] The fireplace is now served by wireless, but it wasn't at the time.

This approach incidentally solves the private line of development problem. The "private branch" can serve equally well as the start of a private, in-house line of development.

5.3 Access Control

The access control design objectives for CPCMS are relatively straightforward. Wherever access control is applied, we wish to distinguish between who can read, who can write, and who can alter the access control rules. The mechanism for access checking should make use of cryptographic techniques so that no local "account" is required on a server to read or write the repository. In part, this is necessary to ensure that mirrors of a repository can enforce access rules without prior knowledge of particular developers. Our current authentication model is provided by SSL using self-signed certificates. The server holds a certificate for each authorized user.

It is assumed that downstream replicate repositories are trusted in a limited sense. Because the replication mechanism does not use a privileged API, replicating servers make copies under the same access constraints as end users. We are not unduly concerned with malicious modification of entities, because the truename system provides a self-checking means of detection. For mutable objects, each change must be signed by the originating repository, whose private key is undisclosed.

CPCMS does not prevent people from making modifications to projects in their own repositories, so long as these modifications cannot be used to corrupt some other line of development. Given that a user can read a project branch, there is clearly no way to prevent them from checking this image into a local repository as though it were a new project of their own. Because of this, there is no strong reason to prevent the creation of private, mutable branches in privately controlled repositories. There are, however, two reasons to encourage this in preference to checking in the code again:

- In some cases, it may become desirable to publish (and possibly to re-integrate) these private lines of development at a later time. By preserving their relationships to the public portions of the project, an orderly merge is greatly simplified.
- If the private effort is in fact a private line of development, the developer can use CPCMS to continue integrating changes from the main line of development into their private version, reducing version drift.

The original design of CPCMS called for access control only at the project and branch level. We quickly concluded that this was unworkable. The main reason for this is that different portions of a code base may require different degrees of expertise to modify them successfully. Requiring people to work in separate branches to perform restricted modifications places a cumbersome burden on the system integrator.

To avoid this burden, CPCMS includes as part of each branch or project description a text object that specifies for each user or group a set of patterns describing the objects they can modify. By associating this object with the branch, access control propagations propagate as quickly as changes to the branch. For branches and projects, read access changes are rare, typically *increase* access (that is: you don't delete people from the read list often), and reasonable propagation delays seem acceptable. Write access for branches is centralized by the existence of a single originating server in any case, so there is no propagation delay for write controls. For frozen objects, access control is applied not so much on the object as on the c-name namespace. In effect, the access controls determine which portion of the c-name space a given user is permitted to rebind.

Oddly, CPCMS does *not* implement read access checks on entities. Entities in the CPCMS repository are named by cryptographically secure true names. As a result, they are unguessable. If the user does not already have in hand the object describing the branch, it is impossible for them to successfully guess the true names of entities within that branch.

There is only one case in which this is untrue, which is the case where a legitimate user exposes the entity's true name to the unauthorized user. We note that in all cases where this might occur the legitimate user could also have passed the entity itself; at modern bandwidths there is no meaningful difference in transmission cost or time. There is a small marginal disclosure in transmitting the entity true name, which is that each entity knows its predecessors. Disclosing the entity (as opposed to its content) therefore discloses the sequence of changes that arrived at the current entity version. Here again, however, the legitimate user could simply transmit the entire sequence.

See the future work section for a proposed solution to this problem.

6 Initial Implementation

Our initial server implementation used a simple file tree structure to store objects:

```
./frozen
./frozen/[hash-1]
./frozen/[hash-2]
./frozen/...
```

```
./petnames/
./projects/
./projects/[rand-3]/
./projects/[rand-3]/branches/
./projects/[rand-3]/branches/[rand-4]
./projects/[rand-3]/branches/[rand-5]
./projects/[rand-3]/petnames/
./projects/[rand-6]/
./projects/...
```

This repository essentially ignores space efficiency issues. Its one merit is that it was very quick to implement. Using a simple structure reduced the entire effort to something that we thought might be tractable to test with bounded effort.

While we were aware that flat files were not the way to go in the long term, this design seemed initially credible. One of the most widely used file version management tools today is RCS [Tic85]. Measurement using the EROS source repository revealed that RCS storage is only 20% more efficient than Lempel-Ziv compression of individual file versions. Since users seem to find the storage cost of RCS acceptable, and the cost of disk space continues to fall, we decided *not* to implement any clever storage techniques in the our first repository implementation.

A second factor motivating our initial decision was XDFS [Mac00], a filesystem-like repository built on XDELTA. XDFS provides most of the repository storage optimizations we want, and already supports cryptographic naming and delta-based compression. Discussions between Shapiro and Josh MacDonald (the author of XDFS) early in the CPCMS design cycle suggested strongly that the two pieces of work would converge at about the right time. As a result of these discusions, it was decided jointly to integrate XDFS into CPCMS in a later release of CPCMS and focus our initial attention in the CPCMS project on schema and usability issues.

As events turned out, our plausibility argument about RCS file sizes was entirely and horribly wrong, for reasons that should have been obvious in hindsight. The issues, and the steps we have taken to solve the problem are described in Section 7.

Fortune sometimes favors the paranoid. Though we chose an initially simple implementation, we were aware that a better implementation would someday be necessary. The repository interface therefore provides a hinting mechanism to support delta storage in the repository. While frozen objects are immutable, the fundamental operation by which a new entity is introduced into the repository is the `revise()` operation, which specifies a (possibly null) hint about the predecessor from which the new entity is allegedly derived. This hint can be used as a basis for delta calculation in storing the new entity. For configuration management purposes, entities internally encode their true predecessors where appropriate. The predecessor specified via the `revise()` interface is considered advisory, and the repository implementation is free to ignore the hint in favor of other compression strategies.

7 From CPCMS to OpenCM

CPCMS became fully operational as we were revising this paper for final publication. In particular, our completion of a CVS repository conversion tool made it possible to perform the first storage overhead measurements for the system design presented in the preceding section. Test cases that run nicely on 10 or 15 operations do not reveal problems that arise when thousands of operations are performed.

Several changes were needed to overcome storage and usability issues revealed by these larger tests. The revised system is now being called OpenCM.

7.1 RCS Storage Revisited

Our first test case was to check in the entire change history of the "build" subdirectory of the EROS tree. The CVS tree for this subsystem occupies 164 blocks (96 if gzipped).[2]. CPCMS, using an uncompressed flat file repository, required 5,504 blocks. The rude surprise was that compression did not help substantially. Compressing the files yielded a CPCMS repository of 4,236 blocks – far from the 20% increase over RCS that we had hoped for.

On examination, we discovered that the objects in the repository corresponding to the original RCS files (the Entity and EntityBits records) accounted only for 40% (2,212 blocks) of the uncompressed total. While the CommitInfo records also record state from the RCS files, they are not stored in a directly comparable way. If CommitInfo records are included as "content" then the content portion of the space rises to 48.8% (2,688 blocks).

This was much higher than expected, but the real surprise was that 60% of the space was going to the configuration schema portion of the store. While we expected to see a proportionally large amount of configuration data for this test case – the build subtree is characterized by few new file introductions and many small changes – this was unexpectedly high.

Our first goal was to reduce the space occupied by the Entity and EntityBits portion of the data. To do so, we built a repository using RCS as the underlying storage layer. Each commit introduced two new cryptographic names in the content schema (one for the EntityBits, the other for the CommitInfo). In the RCS implementation, we tag

[2] All block numbers reported here are 1 kilobyte blocks

each version using its cryptographic name, which adds still more overhead to each version (there are now three uncompressable names rather than two). Without compression, the resulting repository occupies 1,104 blocks, but the portion of the repository state corresponding to the content schema now occupies only 300 blocks. Given that the Entity vs. EntityBits split has visible impact on merge performance, we are reluctant to unify these two objects. We expect that the XDFS-based system, when implemented, should recover most of the balance of content storage overhead.

7.2 On the Perils of Cryptography

While the content schema using the RCS storage repository occupies only 300 blocks, the total repository size remained excessive. The marginal 804 blocks are recording information that is not stored by CVS, but they seem like a steep price to pay for the configuration schema part of the store. The culprit in the story initially appeared to be cryptographic naming.

Part of the problem is the schema design. There are four objects in the original configuration schema (Branch, Change, ChangeVec, EntitySet) where only one is really needed. These objects are connected by cryptographic names, each of which adds (after encoding) 32 bytes of uncompressable state to every commit. We were able to quickly eliminate the ChangeVec and EntitySet structures by merging them inline into their containing objects. We also made changes to the management of mutables. Where CPCMS implemented multiple mutable object types, OpenCM implements a single mutable type. Mutables name revision records and revision records name content. Together, these reduced the total overhead to 1060 blocks.

We then noticed a bug in the client code – the client was failing to specify a "predecessor" when performing entity revision on the CommitInfo records. Fixing this bug brought the total RCS repository size for the build tree down to 704 blocks. If gzip compression were incorporated, the total size of this repository would be 348 blocks. This suggests that while a delta-based encoding scheme in the repository is worthwhile, the RCS encoding is relatively inefficient. We would prefer to incorporate an encoding scheme directly into OpenCM in any case, as calling RCS has performance consequences.[3]

Fortunately, the EROS build subtree proves to be a pessimistic test case. Just the thing to prompt a mad scramble to improve a CM system, but not representative of typical usage. Changes in the build subtree are small, so the relative cost of the configuration data overhead is quite high.

[3] One possibility would be to integrate the RCS library implementation from CVS and also use zlib as the file I/O interface. We have not yet had an opportunity to investigate this.

When the revised system is run against the "base" subtree of the EROS repository (the tree containing our kernel source and key applications), a somewhat different picture emerges. The CVS repository for this tree is 28,208 blocks, the OpenCM content schema objects take 47,304 blocks, and the total OpenCM storage is 65,692 blocks. If compressed via gzip, the OpenCM numbers reduce to 17,848 and 23,260 blocks respectively. For comparison, the EROS build tree has seen 119 changes, mostly minor, in the last 5 years. The EROS base tree has seen 3199 changes over the same period.

	build	**base**
Content Only		
CVS	164 (100%)	28,208 (100%)
CPCMS-flat	2,212 (1,348%)	360,376 (1,277%)
OpenCM	300 (182%)	47,304 (167%)
Content Only, gzipped		
CVS+gz	96 (58%)	12,660 (44%)
CPCMS-flat+gz	1,668 (1,017%)	221,692 (785%)
OpenCM+gz	136 (82%)	17,848 (63%)
Total Storage		
CVS	164 (100%)	28,208 (100%)
CPCMS-flat	5,504 (3,356%)	600,698 (2,129%)
OpenCM	704 (429%)	65,692 (232%)
Total Storage, gzipped		
CVS+gz	96 (58%)	12,660 (44%)
CPCMS-flat+gz	4,236 (2,582%)	336,060 (1,191%)
OpenCM+gz	348 (212%)	23,260 (82%)

Table 1: Summary of storage use. CPCMS figures show flat file repository sizes. OpenCM figures show sizes after RCS and Schema repairs are applied. All percentages are relative to uncompressed gzip.

The space performance results are summarized in Table 1. The current OpenCM repository stops short of re-encoding objects for reasons of robustness – re-encoding would impose a need for transaction support that the current server does not require. We are considering additional schema modifications to further reduce storage consumption.

While there is clearly more work to be done, we consider the degree of compressability on the current RCS-based OpenCM repository extremely promising. For the EROS base tree, the reduction from compression in the original RCS repository was 56%, while the reduction from compression (total storage) for OpenCM was 75%. Since cryptographic names are inherently uncompressable, the compression ratio must arise from two factors:

- The fact that OpenCM uses XML as its storage format.

- The fact that the OpenCM schema still contains considerable redundancy.

As a result of these numbers we plan to integrate compression into the OpenCM storage layer, and use a more efficient delta encoding system. We are more cautious about reducing schema redundancy. One of the (currently untested) potential strengths of the current OpenCM shema is the degree to which information can be reconstructed if an object is lost or damaged. We are concerned that reducing this redundancy would reduce the likelihood of successful recovery.

7.3 Naming for Users Redux

As mentioned in Section 3.4, the original pet name design was not viable. The essential lesson from this design was that human naming of objects in OpenCM needed real attention and a basic redesign. To address these issues, we introduced a directory system for human-management object names.

To facilitate human association, each OpenCM server provides a private directory space for each user. Users can "bind" cryptographic names to human-readable names within this space. For example, the command

```
opencm create project eros
```

proceeds as follows:

1. It creates a new mutable object that will represent the identity of this project.

2. It creates a new, frozen project object carrying the initial project description, and sets the "value" slot of the project mutable to point to this object.

3. It creates a new mutable object to serve as the identity of the "eros" directory.

4. It creates a new frozen directory object containing the pair (`project`, *pm-name*), where *pm-name* is the true name of the project mutable object. This directory object is made the "value" of the directory mutable.

5. It locates the mutable that represents the user's "home directory", and revises the content (a directory object) of that mutable to include a new binding (`eros`, *em-name*), where *em-name* is the mutable name of the previously created eros directory.

6. It revises the user's top-level directory mutable to point to the new directory object containing the combination of the previous state and the new directory binding for "eros."

The net effect of all this is that the user can now type:

```
opencm ls
opencm ls eros
opencm show eros/project
```

and obtain a "directory" listing showing respectively that `eros` is a directory, that this directory contains the name `project`, and showing the content of the `eros/project` object.

The new naming system carries an additional benefit: the `cvs tag` command can now be achieved by creating a duplicate mutable naming the content of some existing branch. Changes to the original mutable can proceed, but the duplicate's state is unchanged. Mutables can be marked "frozen," in which case the tagged version can be deleted but not inadvertently altered. Conversely, an unfrozen mutable of this type provides all of the mechanism needed to perform a "branch" operation. The `pcms branch` command operates in just this way.

In the command line OpenCM client, it proves that this naming scheme extends naturally into the managed object trees. Branch objects are displayed by the `opencm ls` command as a directory of versions. A given version is in turn displayed as a directory containing the top level names of the managed content. In effect, an entire source tree can be browsed as a single hierarchical namespace. We plan to use this to provide a simple CGI script that displays OpenCM managed content via a web browser.

8 Early Experience

OpenCM is now self-hosting, and we have started using OpenCM internally to support the EROS operating system project. While the development groups in question are small, these groups make heavy use of laptop environments, and therefore routinely test the disconnected operation features of OpenCM.

In the EROS and OpenCM projects, we have a steady supply of captive early adopters (students) who are initially unfamiliar with either OpenCM or EROS. While many of their modifications are ultimately accepted into the main tree, there is some desire for quality control in the acceptance process. This is enforced by encouraging the students to create private branches. While they lack the authority to modify the public development trees, this does not deprive them of the ability to modify, save, undo, branch, or evolve their own line of work. This work can later be (repeatedly) integrated, preserving not just the changes but the history trail and commentary that accompanied them. No comparable mechanism exists in other CM systems we are aware of.

Three differences relative to CVS have been immediately noticeable. The first is that *rename* works without loss of evolution history. This is rather like having a new tooth implanted: one has grown accustomed to the hole and is surprised on obtaining the implant to realize how much one missed the tooth after all.

The second difference is the relative ease of mobile development. The standard practice is to set up a caching repository on a laptop, create a branch originated by this repository, and check this branch out. As a side effect, the "top" version of the parent branch is copied to the laptop repository. Commit, undo, and history of changes made while on the road is ready to hand. On return from travel, the local repository is synchronized to the main repository and the final configuration (which includes its complete change history) is merged into the parent branch. With only slight modifications to the current design, it will be possible to cache the entire change history text without caching all of the object versions involved.

The third difference is the relative ease of change integration. Instead of mailing a patch, the merge hand-off in OpenCM is accomplished by synchronizing repositories and mailing the name of the branch containing the desired changes. Because the integrator has an entire, connected change graph to work with, the merge is frequently automated. When conflicts arise, the change history of both lines of development is available for examination to decide what to do.

9 Related Work

A number of related efforts exist or are currently underway.

9.1 RCS and SCCS

RCS and SCCS provide file versioning and branching for individual files. The two differ slightly in feature set, and significantly in their storage strategies. Older SCCS implementations are dramatically slower than RCS. The GNU implementation (CSSC) is considerably faster. GNU CSSC uses a storage strategy that can extract arbitrary versions in linear time, where RCS must internally reconstruct various intermediate versions in branching cases.

Both SCCS and RCS provide operation on a single file. Neither provides configuration management.

While either SCCS/CSSC or RCS could be used as an underlying storage implementation for OpenCM, some form of name translation strategy would be required to map true names into SCCS or RCS version names. In the quick and dirty OpenCM RCS repository, we accomplished this by symbolic linking each RCS file under all truenames and tagging each RCS version with its associated truename. The RCS file is treated as a flat, unbranching sequence of changes. A more deliberate server implementation would accomplish the same mapping using a DB file. One advantage to our choice of encoding robustness: the symbolic links can, if necessary, be recovered from the tag names in the RCS file.

9.2 NUCM

NUCM (University of Toronto) [dHHW96] uses a server information architecture that is similar to that of OpenCM. NUCM "atoms" correspond roughly to OpenCM frozen objects, but atoms cannot reference other objects within the NUCM store. NUCM collections play a similar role to OpenCM mutables, but the analogy is not exact: all NUCM collections are mutable objects. Further, the NUCM information architecture includes a notion of "attributes" that can be associated with atoms or collections. These attributes can be modified independent of their associated object, which effectively renders every object in the repository mutable.

The distinction in their respective handling of collections and mutables is a significant architectural gap between the two systems. The NUCM repository does not have a simple means of providing integrity controls, and the "everything is mutable" design imposes a hierarchical and strongly connected structure on the repositories.

Finally, NUCM versions atoms rather than mutable collections. This is unfortunate, as it conflates the workspace name of the object with its content, and requires that the repository impose a canonicalization policy on names.

While we were intially attracted to the NUCM server as a possible base for OpenCM, we discovered on reading the repository code that it has not been written with robustness in mind. System call return codes were not checked in the version we examined (in fairness, this may since have been fixed). CM systems are integrity-critical systems. We cannot say from experience that NUCM is unreliable, but given the lack of error checking in the code we are unwilling to commit a large enough work base to NUCM to find out.

9.3 BitKeeper

BitKeeper [McV] is similar in feature set to OpenCM, in that it provides multiple satellite repositories and change replication. BitKeeper does not provide cryptographic integrity controls, and its authentication model is not based on cryptographic authentication. Provenance tracking in BitKeeper across multiple organizations is unreliable in the absence of a universal authentication system. In the absence of integrity protection, hostile replicates can inject modified copies of code in such a way that clients cannot detect the substitution.

A secondary concern about BitKeeper is licensing. While the "free if you use our log server" license is appealing, a license that straddles the free/pay boundary is difficult to manage in projects that span commercial and public project members. Subjectively, it seems clear that BitKeeper has failed the test of user acceptance in the open source community. For many users this is unimportant. Given that the core of our own work is open source we feel that this is a significant concern for our applications.

9.4 CM Systems

A brief examination of existing tools is what convinced us that a new CM system was needed. Commercial CM tools did not provide distribution. `Subversion`, the successor to CVS, had elected to defer the question of distribution and adopted (in our view) an information architecture that was unlikely to distribute easily. It also requires long transactions that are relatively easily disrupted by loss of a phone line.

Given the statements of Section 6, it is hopefully clear that PRCS [MHS98] significantly influenced our design. The OpenCM merge strategy is directly borrowed from PRCS, and we are very greatful to Josh MacDonald for the time he put in discussing replication options with us. As with OpenCM, future versions of PRCS are expected to be built on XDFS. In contrast to OpenCM, PRCS provides client/server connection but not distribution. Discussions with Josh MacDonald at the time the OpenCM project was started suggested that distribution would not be available in our timeframe, which encouraged us to persue OpenCM quasi-independently.

9.5 WebDAV

The "Web Documents and Versioning" [WG] initiative is intended to provide integrated document versioning to the web. It provides branching, versioning, and integration of multiple versions of a single file. When the OpenCM project started, WebDAV provided no mechanism for managing configurations, though several proposals were being evaluated. Given the current function of OpenCM, OpenCM could be used as an implementation vehicle for WebDAV.

9.6 Other

Both Microsoft's "Globally Unique Identifiers" and Lotus Notes object identifiers are generated using strong random number generators. Mark Miller's capability-secure scripting language `E` [MMF00, Mil] uses strong random numbers as the basis for secure object capabilities. The `Droplets` system by Tyler Close has adapted this idea to cryptographic capabilities encoded in URLs.

The Xanadu project was probably the first system to make a strong distinction between mutable and frozen objects (they referred to them respectively as "works" and "editions") and leverage this distinction as a basis for replication [SMTH91]. In hindsight, the information architecture of OpenCM draws much more heavily from Xanadu ideas than was initially apparent.

10 Acknowledgments

Mark Miller first introduced us to cryptographic hashes, and assisted us in brainstorming about their use as a distributed naming system. Josh MacDonald took time for a lengthy discussion of OpenCM and PRCS in which the strengths and weaknesses of both were exposed. It can fairly be said that this conversation provided the last conceptual validation of the idea and prompted us to go ahead with the project. Various discussions on the Subversion mailing list pointed out issues we might otherwise have failed to consider. Paul Karger first pointed out to us the requirement for traceability in building certifiable software systems, and the fact that given then-current tools this requirement was incompatible with open-source development practices.

Within the Hopkins Systems Research Laboratory, Raphael Schweber-Koren split the original monolithic implementation into client/server form, implemented the server connection and dispatch loop and the client side session cache.

Niels Provos served as the shepard for this paper. His suggestions and comments have significantly improved the paper as you see it. While insisting on clarity and accuracy, Niels simultaneously maintained a constructive and supportive tone throughout our exchange. This balance, in our experience, is difficult to achieve and maintain. We appreciate his efforts on both counts.

11 Future Work

The OpenCM command line client was an obvious first step, but we are considering two possible directions for enhancement on the client side. The first is to build a GUI-based client, enabling the user to get a continuously monitored overview of their project status. Comparable front-ends have been built for CVS, but for OpenCM, we feel that a complete, separate client is potentially appropriate. Indeed, there is no fundamental reason why the visual and command line clients cannot be used at the same time by the same user if appropriate locking disciplines are observed in the workspace. The second is IDE integration, allowing OpenCM to be directly used in various development environments.

A second direction for further work is repository storage.

As previously mentioned, the XDFS versioned file system is a nearly ideal substrate for use as a OpenCM store. Integrating this will significantly reduce the server-side storage overhead of OpenCM.

There is a problem in the current design concerning scope of potential theft. The client-side file that records the workspace state contains the true name of the configuration object from which the workspace was constructed. This object in turn has predecessor names that (transitively) name the entire history of the project. Frozen objects are not uniquely associated with projects or branches, and frozen object fetches therefore are not individually access checked by the repository in the current design. Any user who can authenticate to the server and present a valid true name – even if they can only authenticate as the anonymous user – can obtain a frozen object.

Ryan Goltry, a student at Hopkins, has proposed a modification to OpenCM in which every user of OpenCM has a unique encryption key that is used to secure and validate their entity names. In this design, the entity names stored on the client would be encrypted in a user-specific way. If stolen, they are useless without the user's pass phrase, and the user's encryption key can be invalidated on the server without significant loss if necessary. One beauty of public key cryptography is that the client-side workspace file in this scenario can be re-encrypted without losing any changes that may be in progress.

12 Conclusion

OpenCM provides first-class support for configurations, support for disconnected commit and distributed repositories, per-file access controls, and support for development groups that span multiple companies or administrative domains.

A recent discussion on the `linux-kernel` mailing list [Bro02] generating the following (edited) list of desirable CM system features. We have annotated each to indicate which ones are addressed by OpenCM:

1. Working merges [*yes*]

2. Atomic checkins of entire patches, fast tags [*yes*]

3. Graphical 2-way merging tool. [*no*]

   This is a very important aspect of a successful CM client that we have not yet addressed. A graphical merge mechanism could very easily be integrated into OpenCM, and we would be happy to adopt a reasonable revision from the community to support it.

4. Distributed repositories [*yes*]

5. Ability to exchange changesets by email [*yes**]

   OpenCM goes one better – email an OpenCM URI that directly references the new configuration on the developer's server. This preserves not just the changes, but the *history* of the changes.

6. Ability to rename files [*yes*]

7. Ability to do archival and renaming of directories. [*yes*]

8. Remote branch repository support [*yes*]

9. Support for archiving symlinks, device special files, fifos, etc. [*no*]

   Version 0.1 of the OpenCM client incorporates type tags in entities, but does not currently know how to interpret any type other than file. Addition of new entity types is straightforward, and we would be happy to adopt a reasonable revision from the community.

While the linux kernel effort is an extreme test of any CM system, we suspect that these features will also be useful to other projects.

OpenCM is built on a small set of simple ideas that are pervasively applied. While there are many interdependencies in the design, there are no clever or excessively complicated algorithms or techniques in the system. The fundamental insight, such as it is, is that successful distribution and configuration management can be built on only two primitive concepts – naming and identity – and that cryptographic hashes provide an elegant means to unify these concepts.

The core OpenCM system, including command line client, local flat file repository, RCS repository, and remoting SSL support, consists of 18,896 lines of code. In contrast, the corresponding CVS core is over 62,000 lines (both sets of numbers omit the diff/merge library). In spite of this simplicity, OpenCM works reliably, efficiently, and effectively. It also provides greater functionality and performance than its predecessor. One of the significant surprises in this effort has been the degree to which a straightforward, naïve implementation has proven to be reasonably efficient.

OpenCM is available for download from the EROS project web site (`http://www.eros-os.org`) or the OpenCM site (`http://www.opencm.org`). A conversion tool for existing CVS repositories is part of the distribution.

References

[Ber90] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[Bro02] Zack Brown. Kernel traffic #158 for 18 mar, 2002. http://kt.zork.net/kernel-traffic/kt20020318_158.html.

[dHHW96] A. Van der Hoek, D. Heimbigner, and A. Wolf. A generic peer-to-peer repository for distributed configuration management. In *Proc. 18th International Conference on Software Engineering*, Berlin, Germiny, March 1996.

[FIP94] Security requirements for cryptographic modules. In *Federal Information Processing Standards Publication 140-1*, 1994.

[Mac00] J. MacDonald. File system support for delta compression, 2000.

[McV] Larry McVoy. BitKeeper web site. www.bitkeeper.com.

[MHS98] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. In *System Configuration Management*, pages 33–45, 1998.

[Mil] Mark S. Miller. The E web site. www.erights.org.

[MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proc. Financial Cryptography 2000*, Anguila, BWI, 2000. Springer-Verlag.

[SFH+99] 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 *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[SMTH91] Jonathan S. Shapiro, Mark Miller, Dean Tribble, and Chris Hibbert. *The Xanadu Developer's Guide*. Palo Alto, CA, USA, 1991.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[SVN] Tigris.Org. *Subversion Web Site*. http://subversion.tigris.org.

[Tic85] Walter F. Tichy. RCS: A system for version control. *Software – Practice and Experience*, 15(7):637–654, 1985.

[WG] E. James Whitehead, Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. of the Sixth European Conf. on Computer Supported Cooperative Work (ECSCW'99), Copenhagen, Denmark, September 12-16, 1999*, pages 291–310.