OpenCM: Early Experiences and Lessons Learned

Jonathan S. Shapiro John Vanderburgh Jack Lloyd
shap@cs.jhu.edu vandy@jhu.edu lloyd@randombit.net
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University

Abstract

OpenCM is a configuration managment system that supports inter-organizational collaboration, strong content integrity checks, and fine-grain access controls through the pervasive use of cryptographic naming and signing. Released as an alpha in June 2002, OpenCM is slowly displacing CVS among projects that have high integrity requirements for their repositories. The most recent alpha version is downloaded 45 times per day (on average) in spite of several flaws in the OpenCM's schema and implementation that create serious performance difficulties.

Since the core architecture of OpenCM is potentially suited to other archival applications, it seemed worthwhile to document our experiences from the first year, both good and bad. In particular, we review the original OpenCM schema, identify several flaws that are being repaired, and discuss the ways in which cryptographic integrity has influenced the evolution of the design. Similar design issues seem likely to arise in other cryptographically checked archival storage applications.

1 Introduction

OpenCM [SV02a, SV02b] is a cryptographically protected 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. Originally prompted by the needs of the EROS secure operating system project [SSF99], OpenCM is slowly displacing CVS among projects that have high integrity requirements on their repositories.

The essential goals of the OpenCM project are to provide proper support for configurations, incorporate cryptographic authentication and access controls, retain the "feel" of CVS [Ber90] to ease transition, provide exceptionally strong repository integrity guarantees, and lay the groundwork for later introduction of replication support. Performance is not a primary design objective, though we hope to keep the user-perceived performance of OpenCM comparable to that of CVS.

Security and storage architects use the term "integrity" to mean slightly different things. A storage architect is concerned with recovering from failures of media or transmission. A security architect is additionally concerned with detecting modification in the face of active, knowledgeable efforts to compromise the content. Because OpenCM is used to manage trusted content (such as trusted operating systems and applications), one of our concerns is that a replicate server controlled by an adversary might be used as a vehicle to inject trojan code. A key design objective has been to provide "end to end" authentication of change sets even when the replicating host is hostile [SV02a]. This prompted us to adopt a secure schema based on cryptographic (non-colliding, noninvertable) hashes rather than cyclic redundancy checks as our integrity checking mechanism.

One result is that an OpenCM branch can be compromised only at its originating repository. When truly sensitive development is underway, a suitable combination of host security measures and repository audit practices should be sufficient to detect and correct compromised branches. We do not suggest that such auditing practices are pleasant, nor are they necessary to run an OpenCM repository for most purposes. The point is that they are *possible*, and for life-critical or high assurance applications the ability to perform such audits is essential. We are not aware of any other source configuration management system with this capability.

Though the techniques have long been known, delta-based storage, cryptographic naming, and asynchronous wire protocols are not widely used in current applications. The importance of asynchrony in latency hiding is similarly well known, but it is difficult to use in practice because it does not follow conventional procedure call semantics. Practical experience with the *combination* of these techniques is limited. OpenCM has exceeded most of our goals during its first year of operation, but it has also suffered its share of design errors and object lessons, many of which are described here.

While we anticipated most of the causes of our mistakes, we failed in several cases to fully understand how various design features would interact. As a result, some of our expectations about how to solve these problems proved misleading. One year later, the basic archival object storage model behind OpenCM appears to be validated, but we are now able to identify some issues (and solutions) that are likely to impact other applications built on top of this type of store. Most of the issues we have found can be viewed as factoring errors in our schema design. In each case, the need to plan for delta storage and cryptographic naming was the unforseen source of these mistakes. Taken individually, none of these schema mistakes are particularly unique or surprising. Taken collectively, we believe they paint a somewhat unusual picture of the constraints encountered by a cryptographically checked store.

While there are significant benefits to the use of cryptographic names, there are also costs. Hashes are significantly larger than conventional integer identifiers and do not compress. They therefore interact with delta-based storage mechanisms, impose potentially large size penalties in network object transmission, and can create unforseen linkages between client and repository. Cryptographic hashes provide strong defenses against hostile attempts to modify content. A corollary to this strength is the difficulty of *legitimate* modification – for example when schemas need to be updated. We will discuss each of these issues in detail.

Online delta compression strategies rely on identifying common substrings between two objects for the purpose of computing a derivation of one from the other. In addition to problems with the particular delta storage strategy we adopted (an "improved" version of Xdelta [Mac00]), there are some unfortunate interactions between delta compression and cryptographic names. We will discuss these and their implications for schema design.

Though asynchronous messaging is a well-known solution for avoiding network round trips and hiding network latency, OpenCM does not lend itself readily to this approach. Some of the issues are inherent in cryptographic naming, while others are inherent in the schema of OpenCM (and probably *any* configuration management system).

While each of the areas we discuss has been a source of inconvenience for OpenCM users, we emphasize that each creates problems of performance (storage size, computation, or wire latency) rather than problems of integrity. Over the last year, we have had occasion to uncleanly terminate our repository server on many occasions. During that time, we have observed only two storage-related errors in the OpenCM alpha releases. One was a serializer error, while the other was the result of an internal computation generating incorrect values in an object. We got lucky on the second error in that it didn't actually get triggered in the field; repair might have proven very difficult *because* of the integrity properties of the store. We will discuss this at length and the provisions that we made in the schema to make repair of this form possible when necessary.

The balance of this paper proceeds as follows. We first review the design of OpenCM, providing a basis against which to measure our experiences. We then identify those areas in which OpenCM has been an unqualified success – most notably in the area of integrity preservation. We then describe in sequence the issues that have arisen from cryptographic names, mistakes in the original schema, delta storage, and communication. We also describe the changes that are being made as a result of what we have learned.

2 Design of OpenCM

Before turning to a discussion of successes and mistakes, it is useful to provide a sense of how the OpenCM application is structured. The schema described here has evolved from that described in earlier OpenCM papers [SV02a, SV02b]. Earlier descriptions characterized the application as having a single unified schema, but noted our belief that the repository schema could be cleanly distinguished from the schema of the OpenCM application. The description that follows reflects that separation.

2.1 The RPC Layer

OpenCM is a client/server application. The client and server communicate via a wire protocol implementing an extended form of remote procedure call (RPC) [Nel81, BN84]. The low-level OpenCM RPC protocol extends conventional RPC by allowing the server to respond with an arbitrary sequence of optional "infograms" before sending a response to the original request. The final response may consist of either an exception or a response:

$\longrightarrow$ request
$\longleftarrow$ infogram*
$\longleftarrow$ reply-or-exception

The ability to transmit exceptions is a reflection of the design of the OpenCM runtime system. Though the application is written in C, we have extended the language using macros to provide an exception handling system. In support of this, OpenCM application is built using the Boehm-Weiser conservative garbage collector [BW88] in place of a conventional memory allocator.

Lesson: Given our past experience with the SGI VIEW debugger and other distributed applications, we went to some care to design infograms into the protocol from the beginning. The idea was to allow clients in a future protocol version to send a series of asynchronous "send" requests terminated by a normal RPC:

$\longrightarrow$ SendEntity *name*
$\longrightarrow$ ...
$\longrightarrow$ SendEntity *name*
$\longrightarrow$ Flush
$\longleftarrow$ EntityGram (Infogram)
$\longleftarrow$ EntityGram (Infogram)
$\longleftarrow$ OK (Response)

Using an asynchronous protocol, trip delays can be reduced where the application's control flow is not inherently data dependent. In hindsight, OpenCM's control flow is much more data dependent than we realized, and this feature will have limited value. Fortunately, this functionality carries no performance penalty.

2.2 Core Repository Schema

The OpenCM application is constructed on top of a repository schema that is largely application-neutral. The repository layer handles versioning, permissions, authentication and (eventually) replication.

2.2.1 Mutables, Revisions

The OpenCM store may be imagined as a versioning file system whose content model is a graph of immutable connected objects. The primitive schema of the repository consists of *mutables*, *revisions* and *buffers* (Figure 1). The

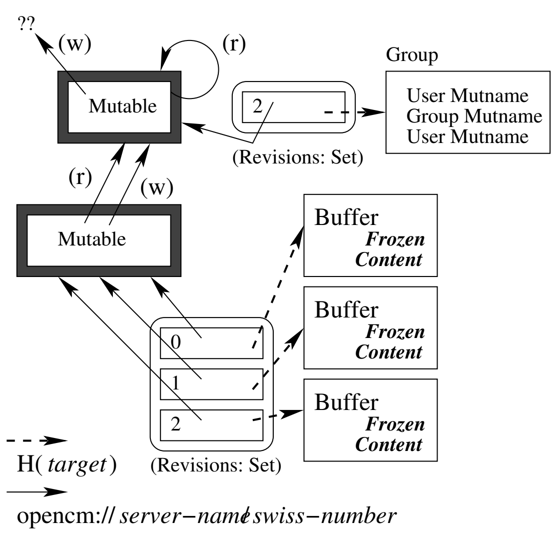

Figure 1: Core schema.

Buffer object serves as the univesal "envelope" for serialization of all content objects. The `Mutable` corresponds approximately to a "file" or "document" in a conventional file system. Each Mutable has a `readGroup` and `writeGroup` fields identifying the users and groups that can perform the respective operations. Since the basic mutating operation on a `Mutable` is "revise," members of the `writeGroup` are also allowed to read the object.

Revisions are made by first uploading a new content graph and then performing a `ReviseMutable` operation on some `Mutable` object to create a new `Revision` record. There is no direct equivalent to a file system "root" directory; the repository keeps a record of the set of authorized users. Each user has a corresponding Mutable that describes their authentication information and names their top-level directory object in the repository. The live state of the repository is defined by the set of transitively reachable objects beginning from the set of current users.

Figure 1 uses two different types of arrows to indicate two different types of names. Content objects are named by the cryptographic hash of their canonically serialized form (written as **H(***target***)**). This hash can be recomputed by the repository without deserializing the object. Mutable objects are named by a URI of the form:

opencm://*server-name*/*swiss-number*

Where *server-name* is the cryptographic hash of the server's initial signature verification key, and *swiss-number* is a cryptographically strong random number chosen by the server at the time of object creation. The swiss-number has meaning only with respect to objects from the same server – there is no risk of collision across servers. As a shorthand, we will use the notation **URI(**`User`**)** to refer to a mutable whose content is an object of type `User` in the rest of this paper.

Revision records are named by appending their revision number to the mutable URI.

Integrity and Security The cryptographic hashes inherently provide an unforgeable check on their content. Since all pointers in the OpenCM content graph are cryptographic hashes, the top-level hash stored in the `Revision` record provides a transitive check on the entire content graph. Content-based naming also allows the server to store repeated objects once.

For mutables and revisions, integrity and security are provided by digital signatures. The mutable or revision is first serialized in canonical form and an RSA signature is computed using the server's signing key (written **S(***target***)**). Both objects have a field reserved to contain this signature. The signature is computed with this field set to NULL. Provided that the OpenCM client has access to the server's signature verification key (the server's public key), compromises of mutable or revision content can be detected. Until a repository changes its signing key, the key's validity can be checked by computing the cryptographic hash of the server public key and comparing it to the server name. The client stores this association with an

IP address on initial connection in much the same way that OpenSSH does [Ylo96]. A technique for validating subsequent changes to the repository's signing key has been described elsewhere [SV02a].

The net result is that the signature checks on the initial mutable and revision records provide a complete end to end integrity check of the content graph associated with that mutable. All of this is handled transparently by the OpenCM client application.

Lesson: In principle, the repository layer has no need for specific knowledge of the application semantics. In practice, we did not attempt to separate these layers fully in the first implementation. Our thought was that we would achieve a better separation after building at least one real application. In hindsight, this was a good decision, because before building the OpenCM application we did not fully understand the layering interactions between the repository and the application.

Lesson: There is no forward linkage (pointer) in the schema from mutable objects to their revisions. In the earliest versions of the primitive schema, revision records contained a predecessor field, and a linked-list traversal was required in order to locate earlier revisions. Each step in the traversal required a round trip in the wire protocol, and it is frequently true that the client knows exactly which revision they require. The revision record was altered to facilitate faster retrieval of specific versions, and to permit selective replication of versions.

2.2.2 Authentication and Permissions

The repository is responsible for connect-time authentication of users. Our current authentication mechanism is built on SSL3/TLS [DA99]. The client presents the server with the public key of the connecting user. The server serializes this key as a `PubKey` object, computes **H**(PubKey(*user-public-key*)), and uses the result as an index into its user database. This index contains a set of (**H**(PubKey), **URI**(`User`)) pairs, from which the identity of the per-user mutable for that user is obtained.

The per-user mutable has the same permissions structure as any other mutable. When adding a new user, the repository sets up their mutable to have self-modify access, and to be readable by the distinguished group `Everyone`. This allows users to alter their own read group. Note that the user's home directory is *not* initially readable by `Everyone`. At the time users are created, other users can see their existence, but not their work.

The `User` object is in turn a mapping from the user's public key to their "home directory." Every repository maintains a directory hierarchy for each user in which objects can be bound with human-readable names.

`User:`	`PubKey` key **URI**(`Directory`) directory
`Group:`	**URI**(`User` *or* `Group`) members[]

Table 1: Authentication-related schema

Groups are simply a set of URIs for other users or groups. Group membership is transitive: if a user is a member of group A, and group A is a member of group B, then the user is also a member of group B. In practice, this transitivity is not often used. It is based on an earlier permission system designed for the Xanadu global hypertext system [SMTH91], and is designed to allow a limited form of delegation for purposes of project administration.

2.2.3 Directories

Every user has a home directory that serves as the root of their accessable state. OpenCM directories provide a shared namespace mechanism that is controlled by the OpenCM permissions scheme rather than the host permissions scheme. Cryptographic monikers are not especially readable; directories serve a critical role as a human-readable name space. For collaborative groups that straddle many administrative organizations, this can greatly simplify day to day development, and it is necessary that such namespaces exist within the repository (as opposed to in the host namespace) for successful replication.

`Directory:`	`DirEnt`[*] entries
`DirEnt:`	*string* key **URI**(*target*) value

Table 2: Directory schema

It is common for project working groups to wish to maintain a common directory namespace that several members can modify – for example as a place to publish new branches for inspection. In the EROS project, we keep all of the publicly accessable branches in a common directory, and bind this directory into the directories of all of the project team members.

Lesson: Directory objects did not need to be part of the OpenCM core schema. They became part of the core repository to resolve a bootstrapping problem: when a User object is initially created, what content should its initial `Revision` object point to? In hindsight, the correct answer is "allow NULL to be a well-defined value for the content pointer in a Revision record." If this is done, the responsibility for creating, maintaining, and binding directories can be taken entirely out of the repository code. Once created, their handling is no different from any other mutable. The "tip off" is that only the user creation code manipulates Directory objects.

Lesson: Readers familiar with remote file system designs such as 9fs [PPT+92] or NFS [SGK+85] might be tempted to think that directory traversal should be performed on the server for reasons of protocol efficiency. OpenCM uses a distributed namespace in which the client may be able to resolve URIs that are not visible to the server. For this reason, directory traversal must be performed on the client.

2.3 Application Schema

OpenCM is a configuration management application layered on top of the OpenCM core repository schema. Its content schema consists of a very small number of object types (Figure 2).

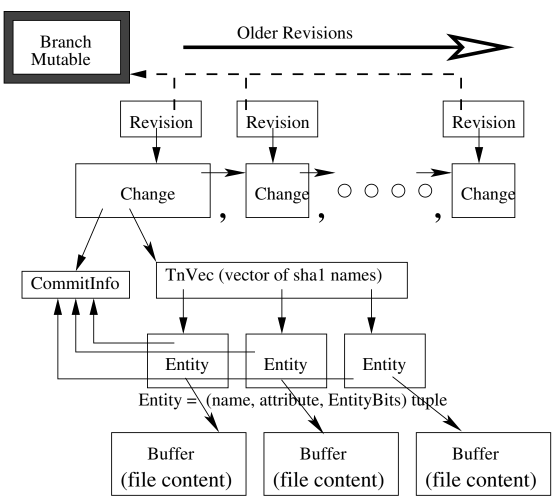

Figure 2: Application schema.

No explicit `Branch` object is needed in the OpenCM application, because the underlying repository keeps a history trail for every mutable object. Every revision results in a new `Change`, and each `Change` object records a new configuration for this line of development. `Entity` objects and `Change` objects both record predecessor pointers, but the former are omitted from the figure for clarity. The history of a branch is therefore encoded in two ways: through the predecessor hashes in the `Entity` and `Change` objects, and through the revision records of the per-branch mutables.

Lesson: Figure 2 shows the *current* object diagram for OpenCM. In the original schema, the vector of `Entity` objects was inlined into the `Change` object. This was a source of tremendous performance loss, and is discussed in Section 5.2.

3 Successes

With the exception of performance, the initial implementation of OpenCM has met its overall goals extremely well. Using SSL3/TLS for authentication [DA99] makes it easy to allow "guest" users write access to our repository. Our hope that this would ease integration by replacing the patch application with merges has been validated. One of the OpenBSD team members has maintained the OpenBSD port in a branch that we periodically re-integrate. The ability to successfully hand this task off and smoothly re-integrate the results has greatly simplified maintenance. This is particularly important given that we do not run OpenBSD and are therefore unable to test these results. We have had similar assistance with ports for MacOS X and IRIX. We have likewise used the merge mechanism within the EROS project group, allowing each developer to play in their own branch before merging working code back into the tree.

Subjectively, our attempts to preserve the feel of CVS have been successful. The OpenCM command interface is not a clone of CVS. We preserved the most common commands in similar form, but regularized options across the board and favored a sensibly structured command set over compatibility. Users of CVS quickly find that they are comfortable with OpenCM. Subjectively, it is somewhat painful to revert to the less regular command structure of CVS. This result meets our goals on all fronts: we want it to be easy for developers to adopt OpenCM and we want them to notice when they are forced to revert to less functional tools. Several users report that they have simply switched their new projects over to OpenCM.

The integrity objectives for the OpenCM repository have also been achieved, and the results have exceeded our expectations. Because our naming and signing mechanism provides an end to end integrity check on the content of all objects, it is extremely difficult for data to be corrupted by the store or the network transport layer without detection.

We knew abstractly that cryptographic hashes provided a simple way to validate that an object is correctly retrieved. Only in actual use have we come to appreciate how useful this is. It is possible in OpenCM to garble the wire protocol, damage (by hand) objects in the store, or accidentally upgrade or modify objects whose content was not supposed to be modified. The validation ability inherent in the use of cryptographic hashes has caught all of these errors early in development. The principle sources of bad repository state in OpenCM have been a buggy serialization package (an escaping error in string encoding) and errors in the merge algorithm that caused objects to be incorrectly created (but stored with high integrity). In spite of crashes both deliberate and otherwise, we are not aware of any instance in which an OpenCM server or transport has undetectably corrupted the state of an object.

Our research group has been using OpenCM since before the day of its release. While there were a few early struggles as bugs emerged in the first few alphas, the tool has proven remarkably stable overall. Generally we are aware of flaws before they are reported by outside users. It is somewhat encouraging that a large percentage of the requests we receive are feature enhancement requests rather than bug reports. In many cases the desired function exists already, but is provided in a form that is not apparent to the requestor.

Perhaps the best indicator of success has been the reactions from other members of the open source community. We have been contacted informally by developers associated with nearly every major open source operating system to ask whether OpenCM could be incorporated into their release. For reasons that we are about to describe, our answer has uniformally been "yes, but doing so now would be premature." The OpenCM alpha has intentionally been a long process so that we could determine what problems the design would encounter. Once the tool is widely dispersed, repairs to the kinds of problems we describe here will become quite difficult. Our goal in the long alpha cycle was to minimize the later pain to others that might be caused by early mistakes such as the ones reported here.

4 Issues with Cryptographic Names

OpenCM uses cryptographic names because they provide a universally non-colliding namespace that can be computed without global agreement or connectivity. The goal is to have an object naming system for frozen objects that can be replicated without collision even though the individual systems may have been disconnected at the time objects were created. By using a content-derived name, the replication engine process is able to efficiently determine by comparing object names which objects must be replicated.

While cryptographic hashing and signing provide exceptionally strong integrity, they also create some challenges.

4.1 Space Issues

Given the total number of objects that a given repository can be expected to store, a 64 bit (8 byte) object identifier would be more than large enough to name all of the objects in the repository. By comparison, a 27 byte cryptographic name (after base64 encoding) carries a large size penalty, and does not compress at all. Examination of the OpenCM application schema will reveal that the majority of bytes in the objects comprise these cryptographic names. As there is no hope of compressing these monikers, any space recovery must come from other mechanisms.

To achieve better space utilization, OpenCM uses delta-based storage. Each object is stored as a delta relative to some existing object, and is reconstructed on demand (Section 6). In order for delta-based space recovery to be effective, fields that are unlikely to change from one object version to the next should be stored adjacent to each other. This creates a maximal length identical byte sequence for the delta generation system to exploit for compaction. To date, we have approximated this by hand-ordering the fields in the object and preserving that order in the (de)serializer routines. In a larger scale application, the order of fields in the object is best determined by what makes sense to the programmer. In consequence, we suggest that an object definition language of some form should be applied and annotations should be used to indicate the preferred order of serialization rather than maintaining them by hand.

In abstract, it would be possible to maintain *two* preferred serializations simultaneously: the canonical serialization used for purposes of computing object hashes and the one used to maximize exploitable repetition across object versions. This proves to be counterproductive. In order for the server to perform integrity validation on objects it must either (a) know how to serialize them or (b) be able to recompute their hash without knowing anything about the object type. The latter is preferred, as the former destroys the separation between the application schema and the repository schema. The consequences are discussed in Section 7.3.

4.2 Name Resolution Issues

A previous paper [SV02a] on OpenCM identified the need for a secure name resolution mechanism to support server authentication. The resolution mechanism must provide a robust mapping from the server's name **H**(**S**($OrigVerifyKey_{server}$)) to its currrent IP address and signature verifcation key. This mapping is needed because the verification key and/or location of a repository may change over its lifespan. We have considered a number of designs for this secure name resolver, and remain partial to the one described in [SV02a].

To date, we have not found that the absence of this registry is a significant issue. In practice, users connect to repositories that are known to them *a priori*. The client records the server name and signature verification key of that server on first connect. One of the delays in the implementation of distributed OpenCM is that this mechanism will break down completely when distribution occurs. Once server A is permitted to store a reference to an object on server B, it will become possible (indeed commonplace) for users to transparently dereference object names whose servers they have never contacted before.

A concern here that we did not adequately credit in our initial design is that the necessary scale of the name resolver is potentially comparable to that of DNS. Whoever runs the target name resolver is likely to be quickly overwhelmed. While client-side result caching is likely to be more effective in OpenCM than in DNS – each user contacts only a small number of servers and remembers them once contacted – we are reluctant to undertake the maintenance of such a large, availability critical data set. A question that arises is: how can we distribute the data set in such a way that the the server containing the *reference* to the object is usually able to supply an up to date name resolution for the server that is actually *hosting* the object. That is, we would like on reflection to establish a two tier hierarchy in which OpenCM servers act as proxies for their clients in the name resolution process.

The scenario that really concerns us is as follows. Imagine that a new release of EROS is announced on Slashdot. Millions of people want to obtain the release (if we weren't optimists, we wouldn't have built a new configuration management system either). Most of these users have never seen EROS before, so their first step is to attempt secure name resolution. The problem is to prevent the name resolver from being overwhelmed – should this occur, then the run on EROS downloads would effectively preclude access to *other* repositories as well. DNS can be used for the IP-resolution portion of the problem, but does not provide a standardized means of distributing per-service keys. This problem awaits satisfactory resolution (sic).

4.3 Versioning Issues

For reasons discussed below (Section 5.2), certain types of schema changes cannot be made in a backwards-compatible way. Incompatible schema changes are rare, but they do not arise capriciously. Each has been introduced to solve a compelling problem with the OpenCM application. This has two unintended consequences:

1. OpenCM will tend to resist "forks" in the development process. As the body of useful repositories grows, there is considerable pressure to avoid competing rewrites of the object schema, and therefore to stick with a centrally coordinated OpenCM application.
2. Incompatible upgrades will tend to be "all at once." If repository A replicates content from repository B, a schema upgrade on A will tend to force clients of B to upgrade their copies of the OpenCM software to understand the new schema.

The replication itself can proceed without upgrading the B repository server, and lines of development originating on B can still be accessed with older OpenCM clients. The current repository garbage collection strategy would need to be replaced with a conservative approach in order for "blind" replication (i.e. replication where the replicating server doesn't know the application schema associated with the content) to be practical. This is one of the reasons that we chose a relatively self-describing format for our encoding of hashes and mutable names.

5 Mistakes in the Schema

While cryptographic hashes guarantee integrity, they also prevent *intentional* changes to objects. If a field is added to an object, existing objects cannot be rewritten to incorporate the new field. Instead, object types must be versioned and the application must know how to forward-convert older objects into the current format. As long as new fields have a reasonable default value, this is straightforward, but object refactorings will generally require a rewrite of the repository to implement an incompatible schema change. In the history of OpenCM to date we have encountered both issues.

5.1 Backward Version Compatibility

The original `Entity` schema was missing a modification time field; we believed initially that storing the creation time of the entity (i.e. the checkin time) was sufficient. In practice this proved mistaken, because some commonly used build procedures rely on correct restoration of modification times on checkout. For example, it is common practice for tools using `autoconf` to include makefile dependencies designed to regenerate configuration files on demand. To eliminate autoconf version dependencies, these same projects ship the *output* of the autoconf script in their distributions. If modification times are preserved, autoconf will not be reinvoked.

Unfortunately, if modification times are *not* preserved, autoconf is invoked and versioning conflicts between the `configure.in` file and the locally installed copy of autoconf can arise. The issue is particularly acute with the `autoconf` package itself, which uses `autoconf` to perform its own configuration and includes an automatic regeneration target in its makefile.

We were able to compatibly add modification times to the existing `Entity` schema by choosing the creation time of older Entities as the default modification time in the absence of better information. This resolved the problem with `autoconf`, but it left us with an unintended consequence: upward compatibility is insufficient; at least partial downward compatibility is also required.

The issue arises because OpenCM stores copies of the `Entity` objects in the client-side workspace file for later reference when performing merges. To avoid identity mismatches, the `Entity` written to the workspace must be byte-identical to the one originally obtained from the repository. If forward conversion occurs by inserting default values in new fields, these fields must *not* be rewritten when the object is reserialized. The original object serialization format included both an object type and a version number of that type, but our original in-memory object structure did not preserve the version number. Fortunately, we were able to modify the in-memory form to preserve the version number without altering the serialized form of our objects.

5.2 Refactoring Change

Before the work described in this paper, the `Change` object contained a vector of `Entity` objects:

`Change:`	`Entity` entities[*]
	H(predecessor `Change`) preds[2]
	H(`CommitInfo`) commitInfo

The earliest schema contained a vector of Entity *hashes*, but a very common operation in OpenCM is to request a `Change` object and immediately request all of its member `Entity` objects. Each frozen object request requires a round trip on the wire, which made this sequence prohibitively expensive. Having initially failed to adequately consider the role of asynchronous messaging requests in our protocol design, We inlined the `Entity` objects before the first alpha release of OpenCM in order to reduce the total number of round trips. This proves to have been a mistake for three reasons:

1. We later realized that by recording additional data in the `Entity` object we could frequently eliminate the `Change` object traversals altogether by walking the `Entity` predecessor chain. Inlining the `Entity` objects deprived us of the ability to exploit this.

2. The resulting `Change` objects are difficult to delta-encode. Where the hashes of the `Entity` objects can be sorted before writing to maximize exploitable repetition, the objects themselves have just enough differences to defeat delta compaction.

3. As the project grows, the `Change` objects become measured in megabytes. As both merge and update algorithms need to traverse the history of `Change` objects, this results in a serious performance problem – even at broadband connection speeds.

We have therefore revised the `Change` objects to place even the vector of names in a separate object, modifying the OpenCM client to fetch individual `Entity` objects only as they are needed.

Unfortunately, this change requires a grand rewrite of the repository, because it is not a problem that can be straightforwardly handled by version-sensitive serialization logic. The deserializer could clearly replace the `Entity` vector by a vector of corresponding hashes, but subsequent attempts to fetch those objects would fail because they were never stored as individual objects in the repository.

If we had to do so, we could "repair" this problem by reading the existing `Change` objects, extracting their `Entity` members and rewriting those separately, but given that we need to do a rewrite of the repository anyway, we have decided to do an in-place conversion. This is better than carrying legacy support for the wrong schema in the post-alpha OpenCM client, and allows us to test the server rewriting process while the experience is still survivable.

The original decision to inline seemed plausible for several reasons:

- In the absence of a subsequent enhancement to the `Entity` schema, the application reference pattern suggested that this inlining decision was a good call.

- In the absence of protocol asynchrony, adding a round trip delay for each `Entity` fetch had been a source of performance issues.

- We had briefly tried a vectorizing fetch operation, but concluded (wrongly) that this imposed undesirable memory overheads and potential resource denial of service issues on the server. We failed to recognize early that asynchrony would enable request flow control, which would in turn resolve the problem of resource denial.

- We wished to avoid introducing into the wire protocol a request of the form "send all `Entity` objects referenced by this `Change` object," because we wanted to avoid building application specific knowledge (in this case, of the `Change` object) into the wire protocol.

In actual fact, the real problem is none of the above. A properly designed asynchronous request, possibly combined with a length-limited vectorizing fetch operator, would have addressed both the round trip problem and the server memory overhead problem. The underlying issue is that the `Entity` object is only four or five times as large as its cryptographic name, so the number of bytes needed to *request* the desired `Entity` objects is considerable. If fetching all of the `Entity` objects in a `Change` were indeed as common as it initially appeared, the wire overhead of the additional fetch requests would not be not justified.

Lesson: The lesson here is that object schema designs in a cryptographically named object store must take into account both the delta encoding effects and the actual usage patterns of the application. This was a case of premature optimization. It would have been relatively easy to inline the vector later. Inlining it early is going to cause us to rewrite all existing repositories.

6 Issues in the Store

OpenCM repositories can be based on flat files or an XDelta variant called SXD. Where XDelta uses backward-encoded deltas, the SXD implementation chose to use forward encoding. The idea behind forward deltas was to allow new object encodings to include references to fragments from all previous objects in the same delta container. We were forced in any case to re-implement the basic XDelta algorithm because the existing implementation had an I/O streams design incompatible with the one we used elsewhere in OpenCM and also because it was crafted in C++. More than 17 calendar years of experience with C++ leads us to conclude that code written in C++ is nearly impossible for ordinary programmers to maintain and presents runtime compatibility issues on some platforms.

Lesson: With the benefit of hindsight, this decision was simply foolish, and the lesson is that we should have read the literature. The analysis of alternatives performed by MacDonald in the design of XDelta [Mac99], as well as the measurements performed in connection with VDelta [HVT96], clearly indicate that reverse deltas are more efficient than forward deltas in nearly all cases. Our own rough experiments confirm that this decision alone added 40% to overall storage consumption. To add to our embarassment, it is plain in hindsight that the issue of self-referencing copy operations in the delta encoding is completely orthogonal to the question of forward versus backward deltas.

Fortunately, the output of the delta encoding strategy is independent of the generation strategy. It has been possible for us to revise this decision incrementally without externally visible impact outside the repository.

7 Issues in Communication

In the course of implementing the cryptographic hash computation, the OpenCM runtime introduced the notion of typed I/O streams. These encapsulate an interface to the object I/O layer. Just above this sits the serialized data representation (SDR) library, which can read/write data in a number of encodings ranging from binary to a human-readable text encoding. There are three types of output streams: streams to files, streams to buffers, and streams that append their content to a pending hash computation but do not record the actual content.

Since OpenCM is a data motion intensive application, the stream implementation is performance-critical. Our original implementation was created in haste, and had approximately a factor of 20 in unnecessary overhead.

Lesson: Code in haste, repent forever.

7.1 Compression

Because of the nature of the OpenCM request stream, it is likely that there is re-referenceable content from one request to the next. In implementing the OpenCM wire protocol, we made the mistake of compressing on a per-request basis rather than simply using a compressed stream. In hindsight this was a mistake. The connection, rather than the messages, should be compressed as a stream. We suspect that the loss of compression opportunities resulting from this decision is substantial, but have no numbers to demonstrate this.

7.2 Inefficiencies in Hashing

Given the existing SDR library, it became natural to specify the definitive method for computing the hash of an OpenCM frozen object as follows:

1. Serialize the object to a `Buffer` stream using the SDR layer.
2. Compute the cryptographic hash on the resulting byte string.

Further, it is natural to perform inbound object I/O by reading the object into a `Buffer`, because the buffer implementation takes care to avoid unnecessary memory reallocations.

This specification for hash computation has a desirable side effect: given a buffer containing the raw bytes of an object, the naively computed cryptographic hash of the buffer content is the cryptographic hash of the object.

Ironically, we failed to notice this for several months. Instead of specifying the `GetEntity` wire invocation to return a Buffer object, we specified it to return a pointer to an object of the expected type. Because this object is what the server needs to return, we actually reload the byte representation of the object from its SXD container into a Buffer, deserialize this Buffer back into an object, and then *reserialize* the object to a hashing stream to check its integrity. A new RPC call returning a buffer delta has been introduced, and the old call will shortly be retired.

A second (and arguably more significant) benefit to manipulating objects purely at the Buffer level is improved ability to stream data motion through the server. There is no reason for the server to verify the object hash before returning the object to the client. Since the client must check the hash in any case, a more efficient design would simply pass the Buffer through without examination. The current implementation does so. Note that this facilitates streaming of large objects. Since the server *should* handle objects larger than the available address space (but currently does not), the ability to stream objects in this way is a potentially critical change in the design.

7.3 Client/Server Version Lockup

The mistake in the specification of the GetEntity wire invocation has a second consequence: it forces the schema version numbers of client and server to remain identical. Because OpenCM clients in practice talk to multiple servers and servers talk to multiple clients, this effectively demands that all installations of OpenCM upgrade when a new version of OpenCM is released.

After some initial settling, the wire protocol itself has not proven to be a source of versioning issues. All integers traversing the wire as part of messages (distinct from object content) use only strings, and unsigned integer values. We encode unsigned values as length-independent strings, avoiding possible compatibility issues resulting from integer length. The OpenCM client and server negotiate a wire protocol version at the start of each connection. This has allowed us to extend existing wire operations compatibly by doing version-sensitive encoding and decoding of the operations.

The source of the problem lies in the present need to deserialize and reserialize each object on its way through the server. The current ServerRequest and ServerReply messages directly serialize the object instead of simply passing along the containing Buffer (the envelope) as an opaque object. In order to perform the deserialization and reserialization, the server must know the type of the object being transmitted, which exposes the server to schema changes in the application.

In practice, the only frozen objects that the server *needs* to know about are the Mutable, Revision, Buffer, User, and Group objects. For reasons explained in Section 2.2.3, the server currently knows know about Directory and Dirent objects as well, but use of these objects is restricted to the "create user" function of the repository. The server is responsible for creating and initially populating the new user's home directory. After creation, the server performs no further interpretation of Directory or Dirent objects.

By changing the wire protocol specification to return Buffers rather than the objects they contain, a hierarchical layering of schema dependencies emerges:

1. If the underlying wire schema is upgraded in a way that cannot be made backwards compatible, then everybody must upgrade. To date we have successfully avoided this in most cases.

2. If the server schema is upgraded, then all clients connecting to that server must upgrade. We have not made significant changes to the server schema since it was originally deployed.

3. If the application schema is upgraded, older servers remain able to act as "carriers" for newer application-layer payload. There remain possible compatibility issues between two copies of the OpenCM client; older clients may be unable to read objects placed on the server by newer clients.

7.4 Failure of Asynchrony

In the original OpenCM design, we included infograms in the wire protocol specification but did not use them. One of the authors had built several heavily asynchronous applications in the past, and we were very much aware of the benefit of asynchrony.

We were also aware of the fact that *useful* asynchrony is limited by the application semantics. If the nature of the algorithms in the application create significant data-driven control dependencies, blocking is dictated by the application-level semantics and cannot be eliminated at the wire level. In OpenCM, it has proven that the majority of requests made by the client are dependent on the results of the immediately preceding request. So far, there are only two exceptions:

- When building a merge plan, the client retrieves the top two objects to be merged (these can be either Change objects or Entity objects, depending on the merge phase) and performs a breadth-first ancestor walk in order to find the nearest common ancestor of the two entities. The search expansion step could be performed using asynchronous requests.

- When performing a checkout or update, the client retrieves the current `Change` object and immediately retrieves all of the associated `Entity` objects in that `Change`. This can be vectorized, as can the later fetch of the `Buffer` object containing the file content for those files that are needed.

The combined impact of these two changes is potentially significant, but it is not immediately clear how they will interact with client-side caching. In merge operations this optimization is essential, but in other operations the dominant cost in using OpenCM is the SSL cryptography setup cost.

8 Measurements

The changes described here have impacted both the repository size and the operational speed of OpenCM. The size improvements come primarily from the introduction of the SXD2 repository format. The speed improvements come primarily from the stream logic changes and the `Change` schema modifications.

All of the numbers that follow are shown for the *local* (direct file access) repositories. Given the reduction in `Change` size, the new schema should show network performance improvements as well, but the major improvement in network performance will not be measurable until we introduce the wire delta transmission and asynchronous requests described in Section 7.

Space OpenCM now supports three storage formats: flat files, gzipped files, and SXD2 archive files. The flat file format is used primarily for testing purposes. A typical store has a bimodal distribution of object sizes. Table 3 shows object size distributions for a test repository holding the complete checkin history of OpenCM itself and for the our main repository (which contains EROS, OpenCM, and various internal projects). The only objects in the store larger than 1 block are the file content and the `TnVec` objects of Figure 2 (the EXT3 file system uses 1 kilobyte blocks). The `TnVec` objects are not compressable at all, and the majority of objects are already one block or smaller, so the achieved compression of 37.5% is actually quite good.

	objects $\leq$ **1 block**	**objects** $>$ **1 block**	**disk use 1k blocks**
Test flat	5,121	2,968 (18,944)	94,932
Test gz	5,230	2,859 (6,992)	59,332
Main flat	115,802	55,265 (22,907)	1,889,560
Main gz	132,007	39,060 (13,980)	1,207,416

Table 3: Object size distributions and total disk use under the new schema. Parenthesized numbers show average object size among objects larger than 1 kilobyte.

The schema change alone reduced the size of our main repository significantly. While surprising, this number is correct. The EROS project has more files in each configuration than the OpenCM project. In OpenCM, the reduction in stored metadata was offset by the changes to content. In the EROS project, the reduction in stored metadata dominates the total repository size. The EROS project dominates the main repository.

The SXD2 format combines multiple frozen objects into delta-based archive files, thereby reducing internal fragmentation within the file system. Disk utilization for SXD2 is shown in Table 4. The SXD2 storage format remains highly compressable. Initial estimates collected while gathering these numbers suggest that another 42% reduction in overall storage is possible.

Schema:	**old**	**new**	**new**	**new**
Store:	**gz**	**gz**	**SXD2**	**mut+rev**
Test	48,076	59,332	24,736	2,104
Main	1,745,632	1,207,416	874,832	32,572

Table 4: Repository storage in disk blocks under various schemas and storage schemes. The "mut+rev" column shows the subset of disk blocks occupied by mutables and revisions, which are stored as gzipped files in all formats shown.

A key factor in reducing total SXD2 storage size is to store indexing structures using a DBM (or similar) file-based format to gain efficient space utilization. If (e.g.) symbolic links are used, total SXD2 storage is *higher* than storage using gzip, because each symbolic link requires a full disk block. Unfortunately, the GDBM library performs updates in place on the existing file, and this introduces risk of index corruption if operations are interrupted. Fortunately, the SXD2 archiver *never* performs in-place modifications, and the DBM index can be reconstructed from the archive files. We are considering storing revision records and mutables in a similar fashion (using SXD2 and DBM). While we expect no delta compression, the improvement in file system utilization from reduced internal fragmentation is probably worthwhile.

Performance To measure performance, we checked out revisions of OpenCM that were 100 revisions old, and then timed three versions of OpenCM performing the update. The first version of OpenCM uses the old schema. The second incorporates only the serialization library improvements. The third includes serialization improvements and uses the new schema. The principle cost of the update is walking the `Change` hierarchy to locate the "nearest common ancestor" in order to compute the necessary update. The results are shown in Table 5.

Version	User	Supervisor
Old	5.11s	0.39s
New Serializer	1.12s	0.33s
New Schema	0.97s	0.41s

Table 5: Update performance

The "new schema" improvements come from two sources: the size reduction of `Change` objects and an optimization in the merge code that eliminates unchanged `Entity` objects from further consideration without fetching them. This optimization significantly reduces the number of objects that must be fetched in the update process, and consequently will reduce the total number of network round trips.

9 Related Work

There is a great deal of related prior work on configuration management in general. We focus here only on architecturally related systems or systems that are being used by open source projects. Interested readers may wish to examine the more detailed treatment in the original OpenCM paper [SV02b] or various other surveys on this subject.

9.1 CM Systems

RCS and SCCS provide file versioning and branching for individual files [Tic85, Roc75]. Both provide locking mechanisms and a limited form of access control on locks (compromisable by modifying the file). Neither provides either configuration management or substantive archival access control features. Further, each ties the client name of the object to its content, making them an unsuitable substrate for configuration management.

CVS is a concurrent versioning system built on top of RCS. It is the current workhorse of the open source community, but provides neither configurations nor integrity checks. A curious aspect of CVS is that it has been adapted for use as a software distribution vehicle via CVSup [Pol96]. This directly motivated our attention to replication in OpenCM.

Subversion is a successor to CVS currently under development by Tigris.org [CS02]. Unlike CVS, Subversion provides first-class support for configurations. Like CVS, Subversion does not directly support replication. Subversion's access control model is based on usernames, and is therefore unlikely to scale gracefully across multi-organizational projects without centralized administration.

NUCM uses an information architecture that is superficially similar to that of OpenCM [dHHW96]. 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. 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. NUCM does not provide significant support for archival access controls or replication.

WebDAV The "Web Documents and Versioning" [WG99] initiative is intended to provide integrated document versioning to the web. It is one of the interfaces used by Subversion, which allows strong web integration. WebDAV 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.

BitKeeper incorporates a fairly elegant design for repository replication and delta compression. To our knowledge, it does not incorporate adequate (i.e. cryptographic) provenance controls for high-assurance development. Further, it does not address the trusted path problem introduced by the presence of untrusted intermediaries in the software distribution chain. In contrast to current implementations of all of the previously mentioned technologies, BitKeeper's license does not facilitate community involvement in improving the tool, and has been a source of controversy in the open source community.

9.2 Other

Various object repositories, most notably Objectivity and ObjectStore, would be suitable as supporting systems for the OpenCM repository design. This is especially true in cases where an originating repository is to be run as a distributed, single-image repository federation. Neither directly provides an access control mechanism similar to OpenCM.

Both Microsoft's "Globally Unique Identifiers" and Lotus Notes object identifiers are generated using strong random number generators. Miller *et al.*'s capability-secure scripting language `E` [MMF00] uses strong random numbers as the basis for secure object capabilities.

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. The OpenCM access control design is closely derived from the Xanadu Clubs architecture [SMTH91], originally conceived by Mark Miller.

OpenCM’s use of cryptographic names was most directly influenced by Waterken, Inc’s *Droplets* system [Clo98]. Related naming schemes are used in Lotus Notes and in the GUID generation scheme of DCE.

10 Acknowledgments

Mark Miller first introduced us to cryptographic hashes, and assisted us in brainstorming about their use as a distributed naming system. The directory system in OpenCM is indirectly based on the “pet names” idea that he invented while at Electric Communities, Inc. 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.

At the risk of forgetting others who may deserves mention, Todd Fries and Jeroen van Gelderen stand out as particularly active among the outside contributors to OpenCM. Each has made significant contributions to OpenCM’s improvement, and has assisted us in hunting down some obscure, irritating, and difficult to track bugs. Interaction on the `opencm-dev` mailing list has been a significant source of improvements and ideas.

11 Conclusion

Delta-based storage, cryptographic naming, and asynchronous wire protocols are not widely used in general purpose applications. Though the importance of asynchrony in latency hiding is well known, asynchronous wire protocols do not interoperate easily with conventional procedure call semantics, and are therefore difficult to use in pratice. Practical experience with the *combination* of these techniques is limited. In this paper, we have tried to report on our experiences building a highly robust application combining these techniques.

OpenCM has suffered its share of design errors in the first year. In several cases the decisions we made in the design interacted with the use of cryptographic naming and delta compression in unforseen ways, and these have taken time to sort out. These interactions proved particularly tricky in the layering of the OpenCM schema and in their performance consequences. At this point, we have identified most of the problems, and are in the process of deploying their solutions.

On the whole, the underlying design approach of the OpenCM repository seems sound. Cryptographic naming is a significant but manageable contributor to storage, communications, and runtime overhead in schemas involving small objects. In repositories storing larger objects such as documents, this issue would be significantly less important and the robustness and modification controls supplied by OpenCM would be significant.

OpenCM is built on a small set of simple ideas that are pervasively applied. While there are many interdependencies in the design, the only complex algorithm or technique is the merge algorithm. The key insights are 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.

Taken overall, OpenCM has met or exceeded nearly all of our expectations. The pain of correcting the errors described here has more to do with the sheer scale of the application than with the complexity of the repairs. We have lived with the tool extensively for a long time now, and none of us would go back to other open source tools given a choice.

The core OpenCM system, including command line client, local flat file repository, gzipped file repository, SXD2 repository, and remote SSL support, consists of 25,794 lines of code – a 27% growth over the last year. Much of this growth has occurred to complete existing function and clarify the merge algorithm. 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.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, pages 39–59, 1984.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, 1988.

[Clo98] Tyler Close. Droplets, 1998.

[CS02] Ben Collins-Sussman. The subversion project: Building a better cvs. *The Linux Journal*, February 2002.

[DA99] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. Internet RFC 2246.

[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.

[HVT96] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An empirical study of delta algorithms. In Ian Sommerville, editor, *Software configuration management: ICSE 96 SCM-6 Workshop*, pages 49–66. Springer, 1996.

[Mac99] J. MacDonald. Versioned file archiving, compression, and distribution, 1999.

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

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

[Nel81] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.

[Pol96] J. Polstra. Program source for cvsup, 1996.

[PPT+92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.

[Roc75] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proc. 1985 USENIX Annual Technical Conference*, pages 119–130, Portland, Ore., June 1985.

[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.

[SV02a] Jonathan S. Shapiro and John Vanderburgh. Access and integrity control in a public-access, high-assurance configuration management system. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, August 2002. USENIX Association.

[SV02b] Jonathan S. Shapiro and John Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *Proc. FREENIX Track of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002. USENIX Association.

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

[WG99] 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, 1999.

[Ylo96] Tatu Ylonen. SSH — secure login connections over the Internet. In *Proc. 6th USENIX Security Symposium*, pages 37–42, 1996.