106

The Vesta Repository: A File System Extension for Software Development

Sheng-Yang Chiu
Roy Levin

June 14, 1993

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

The Vesta Repository: A File System Extension for Software Development

Sheng-Yang Chiu
Roy Levin

June 14, 1993

Sheng-Yang Chiu is at the GO Corporation, Foster City, California.

Authors' Abstract

Conventional file systems are increasingly recognized as an unsuitable basis for software configuration management, especially for large systems. While ordinary file systems have many useful properties, their facilities for managing coordinated changes that span many files are weak. To address this problem, the Vesta configuration management system implements a file system extension that tailors the file abstraction to the needs of large-scale software development.

This paper begins by presenting the essential properties required in the storage facility that underlies a successful configuration management system. It then defines a file-system-like abstraction derived from those properties and explains how it can be implemented on top of a conventional file system.

Contents

1 Introduction

Effective software development requires facilities that support the orderly creation and organization of system components and tools that manipulate them. These facilities, roughly speaking, provide *configuration management.* Software developers have long used ordinary file systems or traditional data base systems as the basis for configuration management. By applying careful discipline, they can employ these general-purpose storage systems adequately without any additional facilities. But automation is more reliable than discipline (and scales better), so, in recent years, developers have sought systems that provide configuration management services directly. The notion of a *repository*, a specialized storage system for software development, has therefore received much attention.

What exactly is a repository? There doesn't seem to be a single, widely accepted definition, and the term has been applied to markedly different classes of system. In all cases a repository offers *integration* of the data associated with the design, development, and deployment of a software product. However, different repositories attach rather different meanings to the term "integration". In sections 1–5 of this paper, we present the Vesta repository and show why its particular facilities and approach to integration are well suited to software configuration management. Section 6 briefly compares the Vesta approach with other repositories.

The Vesta repository notion builds directly on the well-understood file system abstraction. By introducing a few carefully chosen modifications to conventional file system semantics, we can create an abstraction that supports effective configuration management, yet remains familiar. That familiarity has considerable value. Users easily adapt to a file-system-based repository. The tools they already know can integrate well with the repository, and they don't need to learn many new concepts.

What is it about software configuration management that leads us to augment the file system abstraction?

First, the creation and arrangement of files used for software development is more stylized than in general-purpose file system use. Files exist in many versions over time, and at any instant, multiple versions may be of interest. Furthermore, particular versions of different files typically "go together"; that is, the user thinks of particular groups of versions of files as corresponding to significant states of a system under development. A generic file system (even one with versioned files) provides limited help in describing and managing these groups of files. A system focused on configuration management can provide grouping and versioning facilities that are well matched to the software development process.

Second, effective configuration management benefits substantially from the knowledge that versions of files, once created, cannot change. If files are immutable, then the construction process can be made reproducible more readily, and reproducibility has great value in software development. In an ordinary file system, mutable files are the norm, which complicates configuration management.[1]

Third, many of the files involved in the software development process are mechanically generated from other files. We can readily distinguish handmade, or *source*, files from mechanically-produced, or *derived*, files. Frequently in software development, most of the files with which the developer has to cope are derived. If the software development system completely understands how to perform the mechanical steps that construct these files, then it can assume responsibility for producing them on demand, choosing either to store or to (re)compute their contents, depending upon the available resources. It can thereby relieve the user of the substantial burden of naming and manually keeping track of mechanically-generated files. An ordinary file system, lacking the source/derived distinction, cannot do this.

As these points indicate, the file-like abstraction best suited for configuration management differs significantly from the one provided by a conventional file system. The remainder of this paper focuses on the nature of that abstraction and the means used to implement it in Vesta. We assume the reader has read section 2 of the companion paper [Levin and McJones], and is therefore familiar with the basic Vesta concepts.

2 A user's view of the Vesta repository

Since Vesta seeks to extend the file system abstraction, it provides access to the files it manages through the existing file system interface. By and large, Vesta files are named in the same way, and have the same semantics, as ordinary files. This has the obvious advantage that conventional applications, which know nothing of Vesta, can readily interoperate with it. In fact, applications that only read files see the contents of Vesta repositories as ordinary Unix files. But some of Vesta's facilities, which extend the file system semantics, cannot be provided so simply. Consequently, users see the Vesta facilities through a combination of the file system interface and discrete tools (applications) that they invoke directly. In this section and the three subsequent ones, we'll examine these facilities and see how Vesta provides them.

The repository abstraction is not particularly dependent upon the details of the underlying file system. As long as the file system supports the notion of hierarchically named files and directories, the Vesta repository can fit in quite naturally. In our implementation, we chose to build on top of Unix, so some detailed choices in the design were made with Unix file system semantics in mind.

[1] Immutable files can generally be simulated to some extent in an ordinary file system, e.g., by using access control mechanisms, but this isn't the customary mode of operation. As a result, this approach is too fragile to depend upon.

Vesta supports an arbitrary number of repositories. Each one is named by a Unix file path. The names of all user-visible files in a repository have this path as a prefix. Thus, the files in a repository form a rooted subgraph of the Unix file-naming space.[2] Vesta retains the Unix syntax for paths within a repository; that is, slash characters ("/") separate path components, and each component except possibly the final one names to a directory. However, the precise semantics of a path are a little different from an ordinary Unix path. These semantics derive from the following three properties that distinguish the contents of a Vesta repository from those of the ordinary file system:

- Directories are versioned.
- All files and directories are immutable.
- Directories contain source files only.

These properties represent fundamental design choices in the Vesta repository, which arise from its emphasis on configuration management. Let's examine them in more detail.

- Directories are versioned.

  The very phrase "configuration management" suggests the notion of grouping objects. File system directories are a familiar mechanism for doing just that. At a particular instant in time, an ordinary file system directory holds a set of related versions of files. At some later instant, the directory may hold a different set, which may have some common membership with the first set.

  Ordinary file systems and source code management systems built on top of them (e.g., VMS [DEC89], SCCS [Rochkind], RCS [Tichy]) keep track of individual file versions, but don't record relationships between versions of different files. Vesta's repository does this by defining versioning at the directory level instead of the file level. So, in Vesta, we refer to two different versions of a file `foo.c` as

  ```
  p.1/foo.c
  p.2/foo.c
  ```

  while a conventional (versioned) file system might instead name the two versions

  ```
  p/foo.c.1
  p/foo.c.2
  ```

[2]Readers experienced in the ways of Unix will recognize that each repository appears in the name space at a mount point, just as an NFS file system does.

The distinction is uninteresting if only a single file is involved. However, we normally have multiple files in a directory, and the Vesta naming scheme clearly groups related versions of these files. Thus, the three directories `p.1`, `p.2`, and `p.3` with contents

```
p.1/foo.c   p.2/foo.c   p.3/foo.c
            p.2/baz.c   p.3/baz.c
```

naturally define three configurations of the files `foo.c` and `baz.c`. In contrast, a single directory `p` containing

```
foo.c.1  foo.c.2  foo.c.3
baz.c.1  baz.c.2
```

doesn't give us any clue about which versions of `foo.c` and `baz.c` go together.

This naming scheme has a disadvantage, however. While we can tell that `foo.c.1` and `foo.c.2` are (almost surely) different by examining only their names, we can't tell so easily whether `p.1/foo.c` and `p.2/foo.c` differ. `p.1` and `p.2` are (almost surely) different, but the difference may be in some other file, say `baz.c`, while `foo.c` remains unchanged. So, versioning at the file level makes it easy to spot (potential) differences between file versions, while versioning at the directory level makes those differences less evident. To be certain that two files differ, we must actually compare their contents[3]. Nevertheless, this inconvenience is out-weighed by the advantage of clearly identifiable configurations.

- All files and directories are immutable.

  A Vesta repository changes only by growing. Once a new version of a directory comes into existence, it cannot subsequently be changed or deleted. Clearly, Vesta departs in this regard from conventional file systems, which may support the notion of making a directory read-only, but which certainly expect most directories to change over time. A Vesta repository is embedded in an ordinary file system, in which ordinary files and directories can change over time. However, in a Vesta repository, only the top-level directory can change, and only by the addition of new subdirectories. So, for example, if `/vesta/proj` is the name of a repository, `/vesta/proj/p.3` is the name of an immutable directory, and `/vesta/proj/p.3/foo.c` is the name of an immutable file within it. Over time, additional subdirectories of `/vesta/proj` may come into existence, such as `/vesta/proj/p.4`, but, once created, these directories never change.

[3] Or compare something other than the file names, like Unix i-node numbers or Vesta unique identifiers (see section 5.1).

Because directories are immutable, only one version number is necessary in any path to a Vesta file. As the examples above illustrate, we attach this version number to the subdirectories of the top-level directory of the repository. We call these versioned directories *packages* (or *package versions*), and we refer to the collection of versioned directories whose names agree up to the "." as a *package family*. So, for example, `/vesta/proj/p.3` and `/vesta/proj/p.4` name two package versions, and the set of directories `/vesta/proj/p.*` is a package family. (Henceforth, we will omit the repository name when it is irrelevant or evident from context.) Files in a package version are named in the obvious way, so `p.3/foo.c` is a file in package `p.3`. Package versions may have subdirectories as well; for example, `p.3/tests/script.run` names the file script.run in the subdirectory tests of the package version `p.3`.

Vesta's package versioning scheme permits tree-structured names, which provide a natural way to identify branched and parallel development. So far, all our examples have used simple versions like `p.3`. However, as with `RCS` [Tichy] and `SCCS` [Rochkind], Vesta allows path names like `p.3.bug_fix.1` or `p.2.fred.1.new_approach.4`. Notice the alternating names and numbers. The names represent lines of descent (or *branches*); the numbers identify different versions along each line.

We should also note that, while directories provide a particular hierarchical organization for files, it is necessary to have other connections that cut across the hierarchy. For example, Unix systems provide "hard" links that represent the normal naming hierarchy and "soft" (or "symbolic") links that allow arbitrary cross-connections. In a Vesta repository, a cross-connection is called an *import*, and allows one package to refer to another. An import is a link to a package version, that is, a top-level directory within a repository. An import must refer to an existing directory (unlike a Unix symbolic link). Hence, because Vesta package versions are immutable, an import cannot cause a cycle among directories. (For an example of an import, refer to section 2.1 of the companion paper [Levin and McJones].)

- Directories contain source files only.

  Vesta makes a crisp distinction between source and derived files. A source file is one that cannot be mechanically created by Vesta; it is essentially handmade by a human. A derived file is just the opposite; it can be mechanically produced by Vesta using other files.

  Vesta adopts the principle that derived files don't appear in the repository name space. That is, the Vesta repository assigns human-sensible names only to source files. Of course, the repository must store derived files somewhere; we'll discuss this in section 5.2.

On the face of it, this approach may seem surprising. How can a configuration management system hope to help the user when it makes the derived files invisible? In our view, Vesta simplifies the user's life by allowing him to concentrate exclusively on the source files of a configuration. Vesta assumes responsibility for creating derived files on demand, giving them (machine-oriented) names, storing them for later use, and deleting them when they become obsolete. The creation occurs as a result of interpreting Vesta's complete building descriptions (that is, applying the Evaluate operation to a system model) as explained in the companion paper [Hanna and Levin]. The other operations are handled by the repository.

Obviously, if Vesta can deliver on this promise, the user can cope with many fewer files and is freed of the bookkeeping responsibility that often overwhelms large software development projects. The benefits of managing derived files automatically become especially evident when we consider multi-targeted systems, for which multiple sets of derived files typically are produced from only slightly differing source configurations. In this situation, derived files can dominate the name space, and the bookkeeping task looms large.

These three properties that distinguish a Vesta repository from an ordinary file system also have two secondary consequences that affect the user's view.

Atomic creation of configurations. Because every package version is immutable, it first appears in a repository as a "fully populated" directory. Unlike a conventional, mutable Unix directory in which files are normally inserted one-at-a-time, a package version is created *in toto*, atomically. Thus, a user cannot see the directory in an inconsistent, intermediate state; either it's there or it isn't. This reduces the opportunity for confusion when users are sharing configurations; the mere appearance of the directory name in the repository name space provides sufficient synchronization. While the same effect can be achieved in an ordinary file system by using auxiliary tools or willpower[4], it is a direct consequence of the Vesta repository's immutability semantics, and eliminates a situation in which users can make subtle, time-dependent errors. To ensure the atomic-creation semantics, Vesta requires that new configurations be produced with special, Vesta-specific operations rather than with ordinary file system operations. These operations are summarized at the end of this section.

[4] A Unix program that wants to achieve this effect creates a directory off in a corner, populates it, then renames it, causing it to appear atomically where the user can see it. However, this method requires care to do properly and to recover correctly in the case of failure at an intermediate point. Perhaps as a result, it is not systematically used by existing Unix tools.

Easy distributed development. When software development occurs at two or more geographically separated sites, developers need to share access to a common base of files. Because of the inherent bandwidth limitations of wide-area communications, one or more sites typically pay a substantial performance penalty in accessing the common files. To alleviate this problem, files are often replicated at the various sites. However, if an ordinary file system is used, this approach risks introducing transient inconsistencies in the replicas, since all files are, in principle, mutable. By contrast, because a Vesta repository contains only immutable files, and changes only by the addition of new directories, replication becomes much easier. Only newly created directories and their contents need to be shipped between sites; there's no need to worry about changes in previously replicated directories.

We can summarize the preceding discussion of the properties of a Vesta repository as follows:

- Each repository is a subtree of the Unix file name space.
- A repository consists of a (flat) name space of versioned packages, each of which is an immutable directory of immutable subdirectories and source files.
- Derived files are created as a side-effect of the Evaluate operation on a system model. They are invisible to the user.

While these properties follow from the three fundamental design choices, there are other aspects of the repository design that, although not essential, significantly affect the user's mental model of the repository. Let's look at these properties briefly; they are discussed in more detail in the companion paper [Levin and McJones].

- A Vesta repository is either *public* or *private*. Shared files are stored in public repositories, and individual development work occurs in private repositories. Both kinds of repository provide a versioned name space for packages, with each package version being immutable. Thus, every package version that a user deals with, including all the intermediate ones produced in the course of a development session, is immutable. This is quite different from the more customary approach of providing a versioning facility only for shared files and leaving the individual user to keep track of what happens while files are checked out.
- The creation of a new package version in a public repository begins with a *Checkout* operation and ends with a *Checkin* operation. Checkout reserves a package version name in a public repository and establishes an initial version in the user's private repository to serve as the starting point for

development of a new public version. That development produces a sequence of versions in the private repository. Eventually, the user invokes Checkin, which binds the reserved name in the public repository to a package version from the private repository (generally, but not necessarily, the final one in the sequence of development versions). Checkin thus creates a new package version in the public repository, that is, it atomically creates a new immutable directory.

- During the period between Checkout and Checkin, a user creates a series of new package versions in the private repository. This works as follows. The user actually creates new source files in an ordinary (Unix) working directory. A snapshot of that directory is made whenever the user invokes the *Advance* operation, which creates a new package version in the private repository using the files it finds in the working directory. Because the user creates and edits source files in an ordinary directory, she can use ordinary tools (e.g., unmodified text editors) that understand nothing about Vesta.[5]

The three operations Checkout, Advance, and Checkin provide the only means for a user to modify a repository. Their semantics are too different from the ordinary file system semantics to be hidden under the file system interface. Accordingly, Vesta must provide them separately and include tools so that the user can invoke these operations. The next section discusses some important considerations in the design of the repository programming interface (that is, its API) and the tools that use it.

3 Policy implementation

In the preceding section, we presented the repository facilities from a user's point of view. However, different organizations can arrange for their users to see somewhat different facilities; in particular, the detailed behavior of major operations (like Checkout and Checkin) can vary with the organization. The variation results from the different development methodologies and processes employed by different groups. To support a variety of development styles, the Vesta repository API presents a number of mechanisms that can be used to implement various policies. In this section, we look into this separation of mechanism and policy.

[5] This approach is simple, but doesn't scale well for large working directories, since Advance must check every file to determine if it has changed. To eliminate this problem, Vesta can implement the working directory itself, at a mount point, thereby intercepting the operations on files in the directory. Vesta then arranges for this directory to look to the user just like an ordinary one, but it keeps some additional hidden information. Specifically, whenever a file in the directory is created, deleted, or altered, Vesta records the change in an associated log. The Advance operation then uses the information in the log to create a new package version without having to scan all the files in the directory.

We can identify three strategies for implementing policy at the repository API:

> *Encapsulation* is perhaps the most obvious way of implementing policy and requires no explicit facilities in the repository API. A tool that implements a selected policy using encapsulation does so by wrapping up one or more repository operations in custom code. An example of this technique is the numbering of package versions, which is discussed in section 3.1, below.
>
> *Triggering* permits an arbitrary tool to be invoked as a consequence of some specific operation on the repository. The repository API provides the mechanism that connects the operation to the tool invocation. Triggering is asynchronous with the operation, and the invoker of the operation is unaware of it. Section 3.2 describes several uses of the triggering mechanism.
>
> *Parameterization* is used in certain repository operations to permit a selection of choices from a restricted set. The repository API provides a form of access control that illustrates this technique, as described in section 3.3, below.

Although the repository API is designed to facilitate implementation of policy in tools, we have exploited this facility in only limited ways to date. Consequently, the examples below mostly illustrate expected uses of the available mechanisms rather than policies that we have actually implemented.

3.1 Version numbering

In the repository API, the Checkout procedure accepts a parameter that specifies the package version to be checked out. It requires only that this version be new; that is, it must not already exist or be checked out. However, the user doesn't invoke this procedure directly; instead, he uses a Checkout operation provided by a tool that encapsulates the API's Checkout procedure. This tool customarily imposes two additional requirements: the package version to be checked out must be the next one in sequence, and its immediate predecessor must be checked in. For example, if the user asks that `/vesta/proj/p.4` be checked out, the tool requires that `/vesta/proj/p.3` already exist and be checked in.

This encapsulation of the Checkout procedure might seem unnecessary. At first glance, the additional requirements imposed by the tool appear obviously desirable, so one might be tempted to implement them beneath the repository API. However, this turns out to be unwise, for not every software development organization that uses Vesta wants sequentially numbered versions and sequential checkouts. For example, some organizations want the version numbers in a particular package family to correspond to independently chosen release numbers, which don't necessarily advance sequentially. Furthermore, two of these releases may be independent; whether or not one is checked out has no bearing on the other. Consequently, there is no compelling reason to limit flexibility on this point at the repository API. Accordingly, Vesta leaves the policy around the assignment of version numbers up to tools that can be altered conveniently to support different development styles.

3.2 The log

Each Vesta repository maintains a log of actions that alter its user-visible contents, chiefly Checkout, Checkin, and Advance. The repository API provides operations to read this log, and to discover when new entries have been made. These primitives enable a variety of services to be provided by tools above the repository API. For example, it is a simple matter to generate reports on repository activity by scanning the log. A daemon process that sends mail to interested users when selected packages are checked in is equally straightforward.

More generally, the log, monitored and filtered by a daemon process, serves as a source of fairly arbitrary triggers for tools that assist developers in complying with the development process rules imposed by their organization. For example, if a developer is required to report the disposition of a bug report, the tool used to do so could be automatically invoked by a daemon that watches for Checkin operations. As another example, a checkin trigger can alert a quality assurance group of a new version that requires testing, or can serve as input to a critical-path tool used by a project manager to track the progress of a large system toward release. Many other cases arise in the software development process that fit the trigger-on-new-version paradigm.

3.3 Access control

Because the Vesta repository semantics differ significantly from those of an ordinary file system, the facilities for controlling access need to be different as well. Modification of a repository is limited to the creation of new package versions, and the repository API includes an operation to set an access control list that specifies which users can do so. There is one other access control list for the administrative functions, which are described in section 4.

At SRC we had only limited local need for access controls and therefore didn't actually design this part of the repository API very carefully. (For example, all our repositories extend read privileges to all users.) It is evident that finer control of access is necessary in many organizations. Some groups want to limit the ability to perform Checkout operations on a per-package-family basis, and to allow only certain users to read the files in particular package families. We believe that the repository API could be readily extended with the mechanisms to support policies like these, but we haven't yet done so.

4 The administrator's view of the repository

We now turn our attention to more specialized facilities of the repository that are of interest to administrators.

4.1 Derived file management

Recall that, because Vesta generates all derived files as necessary, ordinary users do not concern themselves with the location, or even the names, of these files. Consequently, the task of keeping the consumption of disk storage under control falls to the repository administrator. We want this task to be easy for the administrator to carry out, and for the results to be "natural" for the users. That is, we want to find an invariant for the retention of derived files that both satisfies the desire of the users to have useful files retained and enables the administrator to reclaim adequate disk space when necessary.[6]

What can a user reasonably expect? When developing a Vesta package, the user expects that recently produced derived files won't disappear. That is, if a Vesta evaluation causes a compilation of module `X` to occur, the resulting derived file should be available for at least a while, so that `X` won't have to be recompiled on every subsequent evaluation. But the user can't reasonably expect *every* derived file to be retained indefinitely. So when does the result of compiling `X` disappear?

We adopt the following strategy. Derived files accumulate until the administrator of a repository announces to the users that the repository is to be *weeded* (our term for getting rid of unnecessary derived files). The administrator specifies a list of package versions whose associated derived files will be retained.[7] (We'll see in a minute what "associated" means.) All derived files not associated with those packages will be deleted from the repository. After giving the users sufficient time to suggest alterations to the set of packages whose derived files are to be retained, the administrator runs a tool (the *weeder*) that implements the necessary deletions.

To understand what weeding means, we must delve a little deeper into the machinery used to keep track of derived files. When the Vesta evaluator interprets the system model of a particular package version, it records the set of all derived files that are produced.[8] This set includes not just the files produced

[6] The reader might wonder whether the administrator needs to be concerned with source files as a significant consumer of disk space. The short answer is no; we'll see why in section 7.

[7] Actually, the administrator gives a predicate on package versions, rather than an explicit list. This makes some common situations easier to express, e.g., "keep all derived files associated with checked-out package versions."

[8] More precisely, it records all the derived files that would have been produced if the model were evaluated from scratch. Because of the extensive caching implemented by the evaluator [Hanna and Levin], an evaluation might find some derived objects that were produced by previous evaluations. However, the list recorded for the purposes of weeding does not reflect this; it includes everything that would have been produced in the absence of caching.

by the package model itself, but also those resulting from the interpretation of any system models imported by that model. An association between this set and the package version is recorded for use by the weeder.

Operation of the weeder is conceptually straightforward, and similar to that of a mark-and-sweep garbage collector. The administrator provides a set of roots, that is, package names whose associated derived files are to be retained. The weeder reads the sets of derived files recorded by the evaluator, then scans the repository deleting every derived object not contained in these sets.[9]

If this were the whole story, the invariant preserved by the weeder would be easy to understand: a file is retained if and only if it is involved in the construction of at least one of the specified roots. While this invariant is certainly simple, it is unfortunately too strong to be practical. That is, it causes too many derived objects to be retained, and therefore doesn't solve the administrator's problem of limited disk space. Let's see why.

Recall (from [Levin and McJones]) that a Vesta system model expresses the construction of software (derived files) entirely in terms of source. That is, in principle, every Vesta evaluation constructs every derived file it uses, including the "standard environment" files (compilers, libraries, and the like). So, the list of derived files produced by an evaluation (and recorded for the weeder) includes the executables of all the compilers, which, in turn, are constructed from other derived files using (other) compilers, and so on. Thus, the list includes all the derived files used to produce all the generations of tools, back to the original source.[10] So, if the weeder naively retained *all* derived files produced by an evaluation, it would potentially include many versions of the standard environment, most of which are almost certainly uninteresting.

A natural solution presents itself. We modify the weeder's marking algorithm to "cut off" when it reaches sufficiently old derived objects that are part of (a version of) the standard environment. The precise definition of the cut-off is a little messy, but the basic notion is simple enough. The importing relationship among versions of the environment forms a directed acyclic graph. The representation of the lists produced by the Vesta evaluator retains this graph structure, so it is a simple matter for the weeder to cease marking when it reaches a specified depth in the graph. The choice of depth depends on the structure of the system models that define the standard environment; for the

[9]Weeding can take hours to complete, and it would be intolerable to prohibit Vesta evaluations (which produce new derived files) while collection is in progress. Consequently, the weeder must operate concurrently with evaluation, and the two actions must be properly synchronized.

[10]In many programming shops, this doesn't pose a problem, since the tools are not constructed locally; they are externally produced and treated as source. However, some shops, including ours, develop their programming environment tools and libraries in Vesta as well, and these facilities undergo continuous change. In such a shop, even a package that builds a simple application program imports a Vesta system model that constructs (a version of) the standard environment. That model, in turn, imports an earlier version of the environment to obtain the tools required for its construction, and so on.

environment described in the companion paper [Levin and McJones] using the `building-env` model, we found a nesting depth of two gave quite satisfactory results.

So, to summarize, the invariant for derived file retention is: a derived file is retained if and only if it is produced by evaluation of a system model reachable (by importation) from a specified root, unless it is produced only by evaluation of standard environment models that are more than an administrator-specified importation depth from all roots. This invariant is certainly less intuitive than the simple one mentioned earlier, but, in our experience, it is possible to set the cut-off depth large enough that users are essentially unaware of it, yet small enough that weeding is effective in controlling the use of disk storage.

4.2 Replicas

Vesta repositories are intended to be used by software development organizations that span a wide-area network. But if a repository were stored at a single site, users of the repository at other sites would likely see significantly poorer performance. Vesta avoids this problem by permitting public repositories to be replicated. With replication, however, comes the potential problem of inconsistency. Fortunately, the repository semantics (especially immutability of package versions) largely eliminate this problem, as we'll see shortly.

The only purpose of replication in Vesta repositories is to improve performance for users in a network that lacks ubiquitous fast communication. We do not use replication within a site to increase local availability. Instead, we regard highly available file storage in the local network as a lower-level service to be provided by the underlying file system (as indeed it was in our environment [Mann *et al.*]). So, all users at a site share a replica, and if the replica becomes unavailable, the users do not attempt to use a remote one instead.

Given this assumption about the use of replicas, we can tolerate some inconsistency between the contents of two replicas of the same repository. All users at the same site see a consistent repository, since they all use the same replica. However, updates do not propagate instantaneously between replicas, so users at different sites see somewhat different contents.[11] Even so, the nature of the difference is constrained. Two sites never see different files within the same package version; at worst, a package version is defined at some sites but not at others. This is because the replicated repository still maintains the fundamental property that files and directories, once created, are immutable. Immutability implies that new package versions appear atomically in a repository (replica); hence, if the same package name is defined in two or more replicas, its contents are identical.[12]

[11] Of course, users are not likely to be aware of the difference, since each user can see only the local replica. Only if they communicate outside the Vesta system, say by telephone, can they discover the difference.

[12] The copying algorithm also follows all import links (within the same repository).

Our replication technique is simple, and is based on experience with the siphon mechanism [Prusker and Wobber]. Whenever an operation occurs that changes (that is, adds to) the name space of a repository, that operation is propagated to all other replicas. Since only public repositories are replicated, this boils down to propagation of Checkout and Checkin operations. Recall that Checkout reserves a new package name; thus some sort of synchronization among replicas is required to ensure that conflicting Checkout operations do not occur simultaneously. There are many ways to do this; we chose a simple one—a designated replica must be consulted synchronously for all Checkout operations on its repository. If the Checkout operation is successful, the newly-reserved name is visible in both the "master" replica and the user's "local" replica. Often these are the same one.

Note that no such synchronization is required during Checkin. Since Checkin binds a previously reserved name to a system model, and can be performed only by the user who made the reservation, there is no possibility of race conditions. (A user can be at only one site at a time!) Thus, it suffices to perform the Checkin operation on the local replica only, and propagate its effects to the other replicas asynchronously. Since, in general, Checkin binds an entire directory tree of files, the replication entails the transmission of that tree across a wide-area network, which can be time-consuming. So, making it asynchronous is attractive. (The machinery that actually implements the replication is discussed in section 5.5.) By contrast, Checkout performs a small, fixed amount of work (recording a reserved name). Doing this synchronously across the wide-area, while slower than a strictly local action, is still fast enough.

All the replication occurs automatically, with little work required by the repository administrator. The administrator designates the master replica at which all Checkouts are synchronized. This choice is made when the repository is created and naturally is the site at which most updates to the repository occur. The administrator also adds and removes replicas as necessary. It is possible to change a repository's master replica, if required, but this is an infrequent operation.

We should note that, although most of the repository replication machinery is implemented, it has not actually been put into service. The replication scheme is straightforward, and we expect it will be adequate for the traffic rate of Checkout/Checkin operations we observe (a steady-state average of 3–4 per hour in a community of 25 full-time users). More elaborate algorithms that assign "mastership" on a finer grain (for example, the package family) or more dynamically (say, by holding elections among the replicas) are certainly feasible and could be added if experience shows them to be necessary.

4.3 Load-balancing in the file system

As with any file system, the Vesta repository administrator faces the problem of arranging the files and directories on physical disks to balance the load on the file servers. Typically, there are two problems the administrator must cope with: computational load and available disk space.

As we will see later (section 5.6), the Vesta repository implementation does not use central servers. Rather, a dedicated server runs on each user's workstation. These servers communicate with the underlying file system through the usual operating system interfaces. We do not observe excessive load on the Vesta servers, but inappropriate arrangement of the repository files can cause load imbalances for the underlying file system in which they are stored.

Limitations on available disk space arise because the Vesta repository stores the files associated with a package family together. Furthermore, as we will see in section 5.6, the repository implementation exploits atomicity properties that the file system can guarantee only within certain limits. This means that a newly created file cannot be stored on whatever disk has adequate space; it can go only on the disk associated with its family. Consequently, an administrator must occasionally move families between disks in order to free adequate disk space in the appropriate places.

We should note that, while this arrangement obviously isn't ideal, it is familiar to any administrator of a Unix installation of significant size. For example, the arrangement of users' home directories frequently must be altered due to changing space requirements. If future file systems solve this problem adequately for ordinary Unix files, the Vesta repository will benefit equally.

5 Implementation

We now consider the issues and techniques involved in implementing the Vesta repository API. In the main, the implementation techniques are simple ones; we use the most straightforward data structures except when performance or robustness considerations demand more complex ones. Because the repository semantics generally follow those of an ordinary Unix file system, the implementation can take considerable advantage of the underlying file system.

In this section, we focus on the aspects of the repository implementation whose implementation is not immediately obvious.

5.1 UIDs

In Vesta repositories, as in most file systems, there is a low-level, machine-oriented mechanism for identifying files. In Unix, this mechanism is the "i-node number"; in Vesta, it is the unique identifier, or UID.

A Vesta repository UID unambiguously and immutably identifies a sequence of bytes. Thus, UIDs are never reused (unlike Unix i-node numbers). Multiple path names may lead to the same UID, just as multiple Unix file names may lead to the same i-node number. Thus, if we want `p.2/foo.c` and `p.3/foo.c` to refer to the same file, then we arrange for them to be bound to the same

UID. This occurs very often, since successive versions of a package (e.g., `p.2` and `p.3`) nearly always have files in common.

A UID is an absolute name; that is, it incorporates the name of the repository in which it is defined. Both directories and files have UIDs.

UIDs are used to identify both source and derived files. As we have noted several times, only source files have human-sensible names. But UIDs can be used through the repository API to access derived files as well; in fact, they are the only way to name derived files. The UID of a derived file gives no information about the file's provenance. That is, neither a program nor a human can look at a derived file's UID and determine, say, that it is an object file produced by compiling `p.3/foo.c`.

5.2 Storing and locating files

Now that we understand the properties of UIDs, let's see how user-sensible names are interpreted to yield UIDs and how the byte sequences that UIDs immutably identify are actually stored.

Vesta files (named by UIDs) are stored in ordinary Unix directories. In principle, all files in a repository could be stored in a single directory, since their names are, by definition, unique. Two problems with typical Unix implementations make this impractical. First, directory search is typically linear, which yields poor performance for even a moderate-sized repository. Second, the files named in a Unix directory (by hard links) must all reside on a single disk[13], which uncomfortably limits the size of a repository. To eliminate these problems, the Vesta repository implementation uses a separate Unix directory for each package family, which is further subdivided into separate directories for source and derived files. Each of these subdirectories is then assigned to a disk.

In practice, this works out well enough, although the directories do sometimes contain hundreds of files. We designed the repository to permit a further subdivision of the directories for performance purposes (by a simple hashing scheme on the UIDs), but we never found it necessary to implement this refinement.

Now we can see how source files are actually read. Recall that the repository provides read-only access to source files through the ordinary file system interface. Suppose, for example, that a program asks to open (for reading) the file `/vesta/proj/p.3.bug_fix.1/test.c`. Let's see how Vesta implements this request. Refer to figure 1.

1. The prefix `/vesta/proj` is interpreted by the Unix file system, which discovers that it corresponds to a Vesta repository "mount point". The Unix `open` system call invokes the repository implementation, passing it

[13] Strictly speaking, we should say "disk partition" rather than "disk". Actually, since Vesta was implemented on Echo [Mann *et al.*], the proper notion is an Echo volume, which in many ways is a more flexible kind of partition. See section 5.6.

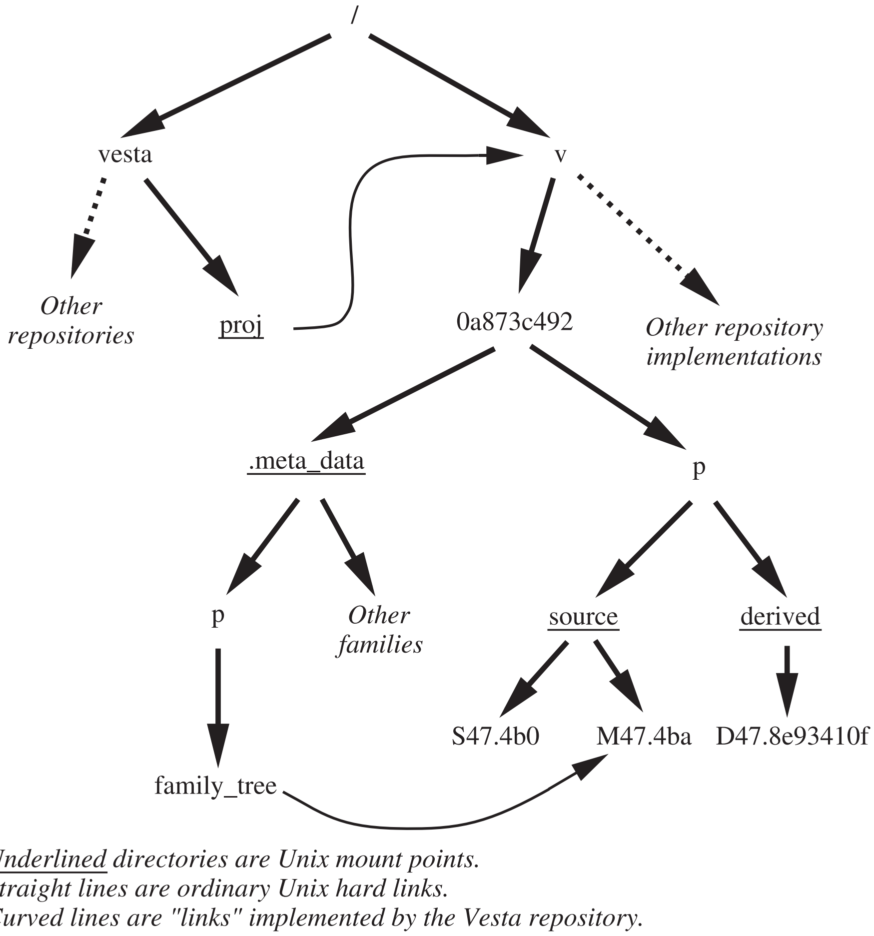

Figure 1: Locating files in a Vesta repository

the remainder of the path (`p.3.bug_fix.1/test.c`) and an identifier for the mounted volume. The latter is used to construct the name of the ordinary Unix directory that is the "root" of the repository, which in this case is `/v/0a873c492`.

2. The repository implementation extracts the family name (`p`) from the first arc of the path. It then reads `/v/0a873c492/.meta_data/p/family_tree`, a Unix file that contains a mapping from package version names within the family to the UIDs of the system models to which they are bound. In this case, let's suppose that the package version `3.bug_fix.1` is translated

to the (source) UID `/v/0a873c492/M47.4ba.p`.

3. This UID is rearranged to form `/v/0a873c492/p/source/M47.4ba`; this file is then opened. Since it is a system model, it begins with a `DIRECTORY` clause [Levin and McJones], which is parsed and searched for the next element of the path (`test.c`). The `DIRECTORY` associates a UID with the name `test.c`, say `/v/0a873c492/S47.4b0.p`.

4. The repository implementation rearranges this UID into the Unix file name `/v/0a873c492/p/source/S47.4b0`. All arcs of the original path have been consumed, so this name is returned to the Unix `open` implementation as the file to be accessed.

Two performance-related issues are immediately evident. First, to map the package version name to a system model UID requires reading a `family_tree` file (step 2). We eliminate most of the cost here by caching that mapping in main memory. This is effective because the data structure is relatively small, is typically used several times following the first use, and needs to be updated only when a Checkout or Checkin occurs in the package family. Second, the `DIRECTORY` clause of each system model along the path must be parsed before it can be searched (step 3). An alternative would be to store the directory in a non-textual form with better lookup performance. We found that the simple approach had adequate performance (roughly comparable with Unix name lookup). We considered various potential improvements, such as storing a hash table in an auxiliary file with a name derived from the system model's UID, but never found the need to implement them.

Reading derived files works similarly, but is quite a bit simpler, since derived files do not have human-sensible names. A derived file's UID is already quite close to a Unix file name. So, the repository implementation simply takes the UID, say `/v/0a873c492/D47.8e93410f.p`, and rearranges it slightly to form the Unix file name `/v/0a873c492/p/derived/D47.8e93410f`, which it presents to the Unix `open` system call.

Next, let's consider how new source files are created. Creation is more complicated than reading, because of Vesta's immutability requirement and the atomic directory creation it implies. Fundamentally, new files are created by the Advance operation. Recall from section 2 that Advance copies an ordinary Unix directory tree rooted in a working directory into a (private) repository. Advance performs this replication "bottom up" (i.e., it processes subdirectories before the directory that contains them) as follows:

1. Copy each file in the subdirectory by obtaining a new UID and copying the contents of the file in the working directory into the file in the repository named by the UID. (The implementation actually copies the data into a temporary file with a related name, then renames the file as implied by

the UID. This makes file creation atomic with respect to crashes.[14])

2. For each copied file, update the `DIRECTORY` portion of the system model for the subdirectory to include the newly-created UID.

3. When all files have been copied, copy the system model itself, obtaining a UID for it in the same manner.

4. When the entire tree has been copied, update the `family_tree` file for the package to bind the new human-sensible version to the UID of the system model. This update is performed atomically by the same write-then-rename approach that is used for individual source files. Thus, from the point of view of a user of the repository, the entire directory tree associated with the new package version becomes available at the instant that its version name is bound.

We should note that the updating of the `family_tree` file in step 4 represents the commit point of the Advance operation. Any failure before then causes the entire sequence of file and directory copying to occur again when the Advance operation is subsequently retried by the user. This simple approach makes Advance robust, which is certainly what the user wants, but orphans any files in the repository that were created by the failed Advance. These files are eventually garbage-collected as a side-effect of weeding.

Creating derived files is much simpler than creating source files. Atomic creation of individual derived files is implemented using the write-then-rename technique of step 1 above. No other atomicity requirements exist, since derived files are not directly visible to users.

5.3 The log

Unfortunately, not all changes to a repository can be implemented by an atomic change to a `family_tree` file. More complex operations—Checkout and Checkin—require atomic changes to multiple repositories, that is, to multiple directory trees. In general, the Unix file system provides no atomicity guarantees that are useful in this case, so the repository implementation must provide its own.

We accomplish this with the familiar device of a write-ahead *log*, to which the data required to perform an operation is first atomically appended, then acted upon in a sequence of steps. This sequence has the property (sometimes called *log idempotence*) that arbitrary prefixes can be arbitrarily replayed without altering the final result. Thus, if a crash occurs, the recovery procedure is simple: process the tail of the log and redo the operation from the beginning.

[14] Here the repository implementation depends on the detailed semantics of the Unix `rename` operation. Actually, Echo provides a particularly strong version of that operation, but ordinary Unix `rename` semantics are sufficient to ensure the necessary atomicity.

Some initial sequence of suboperations will have been completed previously, but log idempotence ensures that redoing them is harmless.[15]

Building the implementation around a log has a second significant benefit beyond simplying crash recovery—it also makes recovery from internal inconsistencies simpler. For example, should a `family_tree` file become corrupt, it can be rebuilt by scanning the log and processing each relevant entry. We found this quite useful on more than one occasion. Also, because we could perform this reconstruction easily, we had the freedom to change our minds about the internal representation of the `family_tree` file, and did so on more than one occasion. In each case we simply installed a new version of the repository implementation and told it to rebuild the `family_tree` file from the log.

This illustrates a principle that was exploited in several places in the repository implementation: record the truth in as simple a form as possible (the log) and build derived structures that give optimized access to the data. If something goes wrong with the optimized structure (corruption, or a bug in the repository implementation), it is simply discarded and rebuilt from the truth. We even went to the trouble of writing the log as a human-readable ASCII file, so that if the log becomes corrupt, the damage can be repaired with an ordinary text editor. Happily, we never found ourselves in this predicament.

5.4 Source file representation

Systems that store many versions of source files customarily provide a mechanism to store only the version-to-version changes (or "deltas") rather than complete copies of each version. Experience with conventional source code control systems [Tichy] [Rochkind] [LeBlang and Chase] has shown that the reduction in storage can be quite substantial. We therefore augmented the simple source storage scheme described in section 5.2 with what is generally called a "delta engine".

Recall that in Vesta repositories derived files may be deleted when space becomes limited, but source files persist forever. This suggests that the disk space required to store them may become a concern, and some sort of space-reduction scheme should be employed. Vesta performs line-oriented differencing of source files transparently. That is, the UID of a source file may lead either to a *clear-text* version directly or to differencing information. In the latter case, the deltas are applied transparently as the client reads the file. Vesta can change between the two representations on the basis of available space, or access frequency, or whatever, without the user being explicitly involved or even aware of the representation choice.

The implementation of this facility is straightforward. When the file associated with a UID is looked up, the repository implementation first tries to locate

[15] The individual suboperations are usually set-insertion or set-deletion, which makes log idempotence straightforward to implement in this case.

a clear-text version, as described in section 5.2. If none exists, a per-family table is consulted, which gives the mapping between a UID and the file used to hold the deltas from which the clear-text version is constructed. The repository implementation then reconstructs the clear-text version, using the same sort of algorithm as a conventional delta engine [Tichy] does.

Converting from the clear-text form to the delta form is also straightforward. The deltas are computed and added to the appropriate file (using atomic replacement, as described earlier), then a new entry is added to the per-family mapping table for the UID under consideration (again using atomic replacement). The clear-text version can be deleted at the same time or subsequently, depending upon available disk space.

5.5 Replication

The implementation of repository replication exploits the log mentioned in section 5.3. There is a separate log for each repository replica. A daemon process watches the tail of the log and notices new Checkout and Checkin entries. It forwards these entries to other replicas where they are appended to the log if they don't duplicate a pre-existing entry. Once in the log, they are acted upon in the usual way (e.g., to update the `family_tree` files of the replica), just as if they had been locally generated. In the case of Checkout, this simply involves updating the family tree information to record the newly reserved name. For Checkin, the source files must be copied from another replica before the package version name can be bound.

The file transmission employs the same technique as was used in the Siphon [Prusker and Wobber]. However, the Siphon must deal with the complexities induced by an ordinary, unversioned, Unix file system. Vesta's immutable, versioned file system simplifies the synchronization requirements of the repository siphon. Vesta's siphon works much like the Advance operation, copying the source tree bottom-up, exploiting the atomic creation of directories at each level. Relevant derived files may be moved as well, using the same lists that the weeder uses (see section 4.1).

We must note that the replication machinery just described has not been fully implemented. However, the log-processing machinery exists, and the implementation of the Siphon that replicates ordinary (non-Vesta) directories receives extensive use daily. We haven't connected the two together yet; the delay results from a lack of need rather than any technical impediment.

5.6 The underlying file system

We've made repeated reference to the Unix file system upon which the Vesta repository implementation is layered. We designed the repository to depend on as few properties of the Unix file system as we could. However, when we actually implemented it, we took advantage of some attractive properties of our locally-available file system to improve the performance of key operations.

Specifically, we implemented the Vesta repository on top of the Echo file system [Mann *et al.*], which provides a Unix-compatible API with several important additional facilities not found in ordinary Unix file system implementations. The ones we exploited were:

- the ability to control the order of write operations on a collection of files and directories, subject to certain limitations, and
- the ability to use advisory locks on files and to receive a properly synchronized notification when a lock is broken.

Let's consider these in more detail.

The Echo file system allows a client program to specify a partial ordering of operations that modify files and directories. The file system guarantees that the modifications occur according to the partial order. In the case of a crash, some modifications may be lost, but the ones that actually occur reflect the partial order.

This property eliminates many of the force-to-disk (`fsync`) operations that are otherwise required. For example, recall the bottom-up copying algorithm of the Advance operation. With ordinary Unix semantics, Advance has two choices, neither of which performs well: it can issue either an `fsync` operation for each file it copies, or a "flush all writes" (`sync`) operation before writing each system model. Without these operations, a system model could be written to disk before one or more of the files it names in its `DIRECTORY`, and an inopportune crash would leave a dangling reference. However, exploiting Echo semantics, Advance simply registers the requirement that the modification of the system model follow the modification of the files it refers to. No `fsync` or `sync` is needed.

Practicalities in the Echo implementation impose some limitations on the orderings that are permitted. Echo provides a concept called a volume, which is an abstraction of a disk connected to a file server. (Actually, a volume may span several physical disks, but all of them must be connected to a single file server.) Echo supports ordering guarantees only among files on the same volume. Thus, to support the "`fsync`-less" implementation of Advance, the Vesta repository implementation puts all the source UIDs in a package family on a single volume, as described in section 5.2.

Echo's convenient write-ordering semantics within a volume simplify the implementation of a fast, reliable, recoverable Advance operation. However, other operations, notably Checkout and Checkin, need to make atomic modifications to data structures on multiple volumes. Specifically, these operations modify two repositories at the same time, and a more elaborate mechanism is therefore required to achieve atomicity.

The mechanism has two parts. We've already seen one of them—the log. By writing a log record at the start of the operation, then performing a log-idempotent sequence of steps to modify the repository data structures, we ensure that the repository modification is recoverable. However, Checkout and Checkin modify two repositories; hence two log records must be written, one to each repository's log. To make these writes atomic, we bundle the two log records into a single *intentions record*, which we use as follows:

1. Write the intentions record stably to a file in the private repository.
2. Test that the proposed operation (Checkout or Checkin) is valid. If so, proceed to step 3; otherwise, delete the intentions record and report an error to the user.
3. Append the appropriate record to the public repository's log.
4. Append the appropriate record to the private repository's log.
5. Update the repository data structures as dictated by the log records.
6. Delete the intentions record.

The sequence of steps 2–6 possesses the log idempotence property described in section 5.3, so if a crash occurs after step 1 and before the completion of step 6, the intentions record will exist at recovery time and the log-idempotent sequence will be replayed.

This sequence must be atomic with respect to other Checkout operations; otherwise, nearly simultaneous, conflicting Checkouts may not be properly detected. To ensure atomicity, we rely on one additional property of Echo—dependable file locks. Like several Unix variants, Echo provides advisory locks, which enable cooperating clients to synchronize their access to a shared file. Echo provides a strong guarantee: a write operation performed to a file by the holder of the file's lock succeeds only if the lock remains held at the conclusion of the operation. That is, if the lock holder loses its lock involuntarily—say, because of a lock server crash or because the lock server lost network connectivity with the lock holder and declared it down—the lock holder simultaneously loses access to the locked file as well. Let's see how this property enables us to make atomic the sequence of steps described above.

Before step 1 of the sequence above, the repository implementation acquires a lock on the public repository log. The commit point is step 3, which appends an entry to that log. If the lock is lost sooner, step 3 fails and the Checkout operation is aborted. However, if step 3 succeeds, then a subsequent crash and recovery can complete the Checkout operation from information in the intention record, secure in the knowledge that the Checkout is still legitimate; that is, nothing can have happened to violate the checks performed in step 2.[16] Thus, the lock provides atomicity of the crucial pair of steps, 2 and 3, which test that the Checkout operation is legitimate and record a stable record of it in the public repository log.

Echo's reliable file locks provide a natural, efficient mechanism that achieves the necessary atomicity. By comparison, we were unable to find a way to achieve equivalent synchronization at reasonable cost on an ordinary (NFS) file system. Neither of the advertised mechanisms—atomic file creation and an auxiliary lock server—offered sufficiently strong guarantees. It may be that these facilities, or others, can be combined to produce the desired semantics, but it seems likely that such a combination will be significantly more cumbersome than Echo's rather straightforward mechanism.

Echo's convenient locking semantics also permit Vesta servers to be *stateless*, in the sense that no individual server is a site for synchronizing access to persistent state. Each server caches data from repository files in its own main memory. To validate the cache data, a server locks the file, verifies that the file's last-written time matches the cached value of that time, extracts the desired data from the main memory cache, and unlocks the file. To modify the file, a server locks it, tests to ensure the modification is consistent with the cached data, updates the main-memory cache, writes it to the file (using atomic replacement or atomic append), and unlocks the file. These simple protocols, together with the Echo locking semantics, ensure that updates to a file are atomic and that cached data is a copy of data in the file on disk.

To summarize: Because Echo makes guarantees stronger than normal for file locking and write-ordering, it is feasible to implement the Vesta repository as a one-per-user server with no permanent local state. All the state is shared in the file system, and the locking machinery permits local caching for good performance. The write-ordering eliminates the delay that is otherwise required for synchronous force-write-to-disk operations. Thus, unshared (i.e., one-per-user) servers can be built with correct synchronization and adequate performance. By contrast, a shared (i.e., multi-user) server requires considerably more elaborate administration to ensure adequate availability and performance.

6 Related work

We've now surveyed the facilities provided by the Vesta repository. Others have adopted different approaches for storing software configuration management data, sometimes also under the name "repository". In this section, we briefly compare Vesta's approach with some other notable ones.

[16] The details of recovery are beyond the scope of this paper. It relies on the assumption that a private repository is not shared by multiple users, and that the intention record is stored in the private repository and acted upon (i.e., recovered) before any subsequent action on that repository.

6.1 File-oriented repositories

Repositories in this class build directly on the file system abstraction, but embellish it very little. Most of these systems evolved from simple file-versioning systems like SCCS and RCS, which also provide a checkout/checkin mechanism. Examples include CVS II [Berliner], ODE II [Open Software Foundation], and the packagetool [Prusker and Wobber]. These systems provide some mechanisms for grouping related files into configurations and may support parallel development to a limited extent. However, none of them actually guarantee immutability or provide any support for consistent removal of groups of derived files (weeding). Because these systems largely retain the file system abstraction, they are relatively easy to adopt, although sometimes the versioning mechanism is introduced in a way that prevents unmodified tools from using it.

6.2 Object-oriented databases

At the other extreme are the most ambitious repositories, which build on object-oriented databases ([Boudier *et al.*], [DEC91]). These systems store typed objects, and have many object types that are specialized for software development. Individual object types may or may not be versioned, and a mechanism is usually provided for grouping objects into configurations. These repositories may provide a simple checkout/checkin paradigm on objects, or may leave this for add-on tools to implement. They generally define their own naming mechanism, which is not integrated with the file system, so existing tools must be modified to access repository objects. The concept of immutability and the source/derived distinction have no special status; they are simply properties attached to certain object types.

6.3 Hybrids

Between these two extremes lie systems that combine aspects of each. These systems (such as DSEE [LeBlang and Chase], ClearCase [Atria Software], and TeamNet [TeamOne]) begin with a file-based approach and tend to retain the file naming regime of the platform under them. Sometimes the naming is introduced seamlessly, so that unmodified tools can access repository objects; in other cases, special tools must be used. However, the systems add a database on top of the file system, in which each file is represented by an object to which a collection of additional attributes is attached. Search paths and/or special objects in the database group file-like objects into configurations, but these groupings are rarely immutable. The database also is used to support checkout/checkin of file objects, but typically not configurations.

6.4 Discussion

Where does Vesta fit into this taxonomy?

The Vesta repository is most similar in spirit to the file-oriented repositories, but provides crucial features—immutability and a clear separation of source and derived files—that the simpler repositories lack. Indeed, no other repositories, not even the most elaborate ones, support these facilities well. As a result, while these systems are all reasonably competent at storing files and organizing meta-data associated with them, they don't support incremental, reproducible system construction on a large scale. Vesta does so because the repository was explicitly designed to support the builder. In other systems, the builder is "just another tool".

Of course, Vesta's repository can't do everything that the object-oriented repositories can do. These repositories emphasize the organization of data and meta-data in a type system intended to facilitate tool interoperation. They aim to make it easy for tools to exchange (typed) data through the repository rather than through private agreements. This is certainly an admirable goal. However, these systems require a critical mass of tools that work with the repository, and, at present, no commercially available system provides such a collection. Demonstration systems exist, but exhibit poor performance, and there are formidable technical difficulties in delivering an OODB-based repository that supports the development of large software systems with acceptable performance.

Although the Vesta repository doesn't include a database for storing meta-data, as the hybrid repositories do, there is a natural way to couple such a database with a Vesta repository. We view the Vesta repository as the appropriate place to store the essential information that constitutes a software system, while an auxiliary database is an attractive place to store relationships derived from that essential core. Many tools that support the software development process can extract information from the repository, process it, and enter the results in the database. For example, the builder can record inter-module dependency relationships in the database in a form that facilitates subsequent queries. Since this information is mechanically derived from the contents of the repository, it can be reconstructed if necessary, which simplifies the reliability demands on the database system.

7 Evaluation

The preceding sections illustrate the essential properties of the Vesta repository:

1. It integrates closely with the file-system abstraction, making existing file-based tools able, without modification, to access files in Vesta repositories.

2. It implements additional properties—notably immutability, versioned directories, and a source/derived distinction—that support effective configuration management.

3. It exhibits adequate performance.
4. It has a robust yet economical implementation.
5. It provides for administration that is only slightly more elaborate than that of an ordinary file system.
6. It scales to accommodate geographically dispersed organizations that share data while shielding users from the unpleasant realities of wide-area networking.
7. It permits organizations to introduce policies for repository use that fit their software development methodologies and management practices.

We built a prototype implementation of Vesta and used it extensively for over a year. Based on that experience, we can evaluate how several of these properties have worked out in practice.

1. *File-system integration.* Embedding Vesta's notions of package families and versions in a familiar and pervasive hierarchical naming scheme proved extremely effective. Many existing facilities were immediately reused without modification. Our software developers depended upon a rich set of tools for reading, parsing, and comparing files (e.g., `grep`, `awk`, `sed`, `diff`), which, because of Vesta's close integration with the file system, could continue to be used without modification. The users exploited this interoperability in two ways, both of which are important. First, they invoked standard Unix tools from a Vesta system model as part of the construction or testing of a package, just as they would with a conventional builder like `make` [Feldman]. Second, the users invoked these tools from the Unix shell, applying them directly to files in the repository. (Indeed, many of the numbers reported later in this section were collected using such tools to read files stored in the repository.) Because of this close integration, the Vesta system didn't have to provide analogues of these familiar tools, and the users required no additional training to apply them to Vesta files.

   Although we understood how important it was for the Vesta repository to integrate well with the file system, we didn't always achieve ideal integration. A particular example is illustrative. Recall that package versions are essentially tree-structured names whose components are delimited by periods. Unix file names are also tree-structured, but are delimited by slashes. Smooth integration would suggest using the Unix approach uniformly, so that the package version `/vesta/proj/p.3.bug_fix.1` would instead be written `/vesta/proj/p/3/bug_fix/1`. We considered adopting this approach, but it doesn't permit easy identification of the repository, package, and file name components of a path, which we felt would cause problems. So, we chose instead to use periods to separate components of a package name. As a consequence, we had to replicate some of the standard browsing facilities (like "wildcarding", the Unix `find` command, and some forms of the Unix `ls` command) that would otherwise have come for free and been better integrated.

2. *Immutability and the source/derived distinction.* When users can truly depend on immutability and reproducible construction, they can afford to adopt a flexible development approach. The Vesta prototype system was used by about 25 users on a daily basis, collectively evolving a code base of about 1.4 million source lines. They used the repository's mechanisms that support branched and parallel development quite extensively. In particular:

   - Branched development was common. Even with a relatively small group of developers, the need to record variants from a "main line" of development arises frequently; 23% (1372 of 5864) of the package versions in the public repository are on branches.
   - Parallel development (i.e., concurrent work on more than one version within a single package family) was common. 19% of the package families (93 of 486) exhibited parallel development. 16% of all development sessions were conducted in parallel with at least one other session in the same package family. Of these sessions, 72% involved two or more distinct users; 28% involved a single user changing multiple versions within the package in parallel. So, about 1 of every 9 development sessions represented work within a package family carried out in parallel with one or more other users.

   The source/derived distinction definitely simplified the users' view and the administrators' job. This was particularly true for our environment, in which a significant number of apparent source files—that is, files presented to a compiler—were actually derived files (such as the output of an RPC stub generator). Since Vesta knew for certain which files were source and which were not, it could ensure that only the derived files were deleted and that apparent sources were not needlessly retained.

   Nevertheless, the source/derived distinction was sometimes inconvenient. Occasionally, a user needs to look at the output of a preprocessor or compiler. This often happens because the translator has done something unexpected, which may actually turn out to be a bug. In our environment, the tools themselves were under active development, and bugs of this sort did indeed occur. Vesta's strict source/derived dichotomy hides the derived files from the user, which complicates the task of tracking down these problems. Although it is possible to extract derived files from the Vesta repository and look at them with conventional tools, the process is clumsy. In the future we intend to make derived files visible to the user without compromising their clear distinction from source.

3. *Adequate performance.* We devoted only modest effort to performance tuning of the prototype implementation. In several cases we made an implementation choice on the basis of simplicity, with the intention of doing something more elaborate later if experience indicated a need. Usually the simple solution was quite adequate. For example, the prototype implementation serialized all Checkout and Checkin operations, each of which takes 30 seconds or less. The long-term average rate of these actions was two per hour; even assuming they all occurred during a single 8-hour shift (actually, only 65% occurred during the busiest shift), we would see only one every ten minutes.

   Actually, we didn't apply this principle—build only what is necessary first—as ruthlessly as we should have. The source differencing machinery (see section 5.4) seemed like a necessary facility to reduce the consumption of secondary storage. (We based it on a related mechanism reported in DSEE [LeBlang and Chase].) But though we built all the machinery to perform the differencing and to change source file representations dynamically, we never found the need to turn it on. This is because the total space required for source files, even in clear-text form, was dwarfed by the total amount of space required for a "working set" of derived files. We found that, in our public repositories, only about 20% of the disk space was consumed by source files. Applying aggressive differencing would perhaps have cut this in half, giving a modest increase in the amount of available disk space and thus in the interval between runs of the weeder. However, the rate of source creation is roughly constant (programmers can type only so fast), and the asymptotic size of source files is also constant, while the amount of disk space one can purchase for a fixed amount of money is growing exponentially. We believe the cross-over point has been reached, and that source-differencing machinery, while important for this sort of application a decade ago, is no longer necessary.

   Performance of the repository implementation on common operations, such as opening a file or listing a directory, is also a concern. This is especially true because the repository implements portions of the file name space, and users are dismayed if the performance of the file system appears to vary substantially in different parts of the name space. Access to Vesta files incurred a small but noticeable penalty (variable, but around 20%) as a result of going through the repository layer on top of the Echo file system. We did very little performance tuning of that layer in the prototype. Moreover, Echo itself was a new and experimental system, and had not received much performance tuning either. However, some analysis after the fact suggests that the whole system could have been made to perform competitively with standard file servers. (Making the repository perform well without Echo is somewhat more work—see below.)

4. *Robust and economical implementation.* The repository implementation in the Vesta prototype is approximately 30,000 lines of source code. It required less than a person-year to build and has never lost a file. Most of the complexity lies in Checkout and Checkin, arranging that each of these operations is implemented by a log-idempotent sequence of sub-operations. Log-idempotence enables simple recovery, which in turn contributes to the robustness of the system as seen by the user.

   Having a log simplifies disaster recovery as well. We used the log to recover repository meta-data on several occasions. We also used it as a simple and convenient way to change the internal representation of the meta-data without having to write elaborate conversion code. If we weren't already convinced of the advantages of a log-driven implementation before we started, these positive experiences certainly eliminated any residual doubts.

   The implementation benefited from the special facilities of the Echo file system (see section 5.6), but sacrificed portability as a consequence. To build the Vesta repository on a standard, widely available file system like NFS requires more machinery, chiefly to cope with the absence of orderly locking and write-behind. Implementing that machinery without sacrificing performance presents some significant challenges.

   Overall, the implementation achieves adequate performance and excellent reliability with rather low-tech machinery. It requires neither a DBMS nor a transactional substrate, although it employs some related techniques (like write-ahead logging). No elaborate persistent data structures are used unless there is a simple mechanism (like log replay) for reconstructing them in case of corruption. This approach requires a little extra implementation care but pays handsome dividends in system robustness.

   We believe that robustness is of particular importance in a repository. When a user is directly manipulating files in a file system, there is some chance that she will be able to correct blunders by hand. However, the repository adds a substantial layer of implementation complexity over the basic file system, in exchange for some more useful and powerful invariants. The burden of restoring those invariants by hand when the implementation fails is simply too great, therefore, a robust implementation is essential.

5. *Modest administration demands.* As noted in section 4, administering the Vesta repository is much like administering a normal file system, with the additional need to perform weeding. Over about 14 months, the weeder was run 33 times (roughly bi-weekly) to delete unnecessary derived files, recovering an average of about 1.1GB of disk space each time. The total disk space available to the repository during this time was about 9GB. While the weeder itself wasn't particularly speedy (taking 6–13 hours on each run), the time spent by the administrator to set up the work was

perhaps a couple of hours each time, or about 2% of the administrator's time. Note that these numbers reflect the administrative cost for the public repository. Private repositories were largely weeded by a mostly automatic procedure; the setup time required by their administrators was negligible.

However, we did have problems with the weeder, despite the rather small burden it imposed on the administrator. In the prototype, we didn't pay sufficient attention to the design of the data structures shared by the weeder and the evaluator. As a result, the weeder didn't really have enough information to do its job properly. Some heuristics enabled us to get around the worst of the problems, but some difficulties remained, showing up when the evaluator thought a particular derived file existed even though it had actually been weeded. These problems occurred rather unpredictably, and recovery required assistance from a Vesta implementor. This was probably the most serious design error in the Vesta prototype. To correct it, we have substantially redesigned the shared data structures, but have not yet implemented the result.

Two properties of the Vesta design were not fully realized in the prototype system, so our evaluation of them is necessarily incomplete.

6. *Geographical distribution.* The Vesta prototype system did not fully implement replication across a wide area. However, the design for wide area replication builds on the highly successful Siphon [Prusker and Wobber]. We feel confident that when the siphon approach is coupled with Vesta's immutability guarantees, the result will be a system that is even more useful.

7. *Flexible policies.* We implemented only a few rather simplistic policies in the Vesta prototype system. Access control and limitations on certain forms of checkout were mentioned in section 3. We have some additional experience using the mechanisms intended to support flexible policy, most notably the log (section 3.2). We haven't yet encountered problems in this area, but more experience is needed before their value can be reasonably assessed.

In summary, our experience indicates that a simple, file-system-based repository with additional semantics and support keyed to configuration management provides attractive functionality and implementation economy. The prototype implementation on which our experience is based proved quite serviceable for a non-trivial workload. Its deficiencies didn't seriously detract from its usability, and its functionality does not appear to be available either in other file-based repositories or in more elaborate database-oriented ones.

8 Acknowledgements

The authors are particularly grateful to Butler Lampson and Mark Brown, whose insights and experience greatly improved the Vesta repository design and implementation. The members of the Echo team, especially Tim Mann and Garret Swart, helped us to integrate the repository with the Echo file system and provided valuable advice on proper use of the Echo facilities. Bob Ayers built an early Vesta prototype that exposed several flaws in our early repository designs.

References

[Atria Software] Atria Software, Inc. *ClearCase Concepts Manual.* 1992.

[Berliner] Brian Berliner. *CVS II: Parallelizing Software Development.* Prisma, Inc., Colorado Springs, CO. 1992.

[Boudier *et al.*] Gerald Boudier, Ferdinando Gallo, Regis Minot, and Ian M. Thomas. "An Overview of PCTE and PCTE+." *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* November 1988.

[DEC91] Digital Equipment Corporation, *CDD/Repository Architecture Manual.* Order #AA-PJ1JA-TE. Maynard, MA., 1991.

[DEC89] Digital Equipment Corporation, *VMS User's Manual*, Order #AA-LA98B-TE. Maynard, MA., 1989.

[Feldman] S. I. Feldman, "Make—A Program for Maintaining Computer Programs", *Software—Practice and Experience.* 9(4), April 1979.

[Hanna and Levin] Christine B. Hanna and Roy Levin. "The Vesta Language for Configuration Management." Research Report 107, Digital Equipment Corporation Systems Research Center, June 1993.

[LeBlang and Chase] David B. LeBlang and Robert P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. *SIGPLAN Notices* 19, 5, May 1984.

[Levin and McJones] Roy Levin and Paul McJones. "The Vesta Approach to Precise Configuration of Large Software Systems." Research Report 105, Digital Equipment Corporation Systems Research Center, June 1993.

[Mann *et al.*] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. "A Coherent Distributed File Cache with Directory Write-behind." Research Report 103, Digital Equipment Corporation Systems Research Center, 1993.

[Open Software Foundation] Open Software Foundation, Inc. *ODE-II Developer's User Guide.* Version 1.0, June 1991.

[Prusker and Wobber] Francis J. Prusker and Edward P. Wobber. "The Siphon: Managing Distant Replicated Repositories." Research Report 7, Digital Equipment Corporation Paris Research Laboratory, May 1991.

[Rochkind] Marc J. Rochkind. "The Source Code Control System." *IEEE Transactions on Software Engineering.* SE-1, Issue 4, December 1975.

[Tichy] Walter F. Tichy. "RCS—A system for version control." *Software—Practice and Experience*, 15,7, (July 1985).

[TeamOne] TeamOne Systems Inc. *TeamNet Quick Start Tutorial.* Sunnyvale, CA, 1993.