August 17, 2001

SRC Research Report 172

Partial Replication in the Vesta Software Repository

Timothy Mann

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables 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 approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed 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, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Partial Replication in the Vesta Software Repository

Timothy Mann

August 17, 2001

Timothy Mann is no longer with Compaq. He can be reached by electronic mail at the address tim@tim-mann.org.

The Vesta project's Web site is at http://www.vestasys.org/.

Abstract

The Vesta repository is a special-purpose replicated file system, developed as part of the Vesta software configuration management system. One of the major goals of Vesta is to make all software builds reproducible. To this end, the repository provides an *append-only* name space; new names can be inserted, but once a name exists, its meaning cannot change. More concretely, all files and some designated directories are immutable, while the remaining directories are appendable, allowing new names to be defined but not allowing existing names to be redefined.

The data stored in a repository can be replicated, to support distributed software development. The append-only nature of the repository greatly simplifies the problem of maintaining consistency among replicas. Conceptually, all files and directories stored in all Vesta repositories are named in one single, global name space. Each repository stores some subtree of the complete name space. Replication is present when the subtrees stored by two different repositories overlap; that is, some of the same names and data occur in both. We call this concept *partial replication*, because each repository can choose to replicate all, part, or none of the data stored in any other repository.

In this paper we outline the main features of the repository, give a definition for the consistency of partial replicas, describe how our replication tools maintain consistency, and briefly relate our experience in using the system for distributed software development between groups on opposite coasts of the United States.

1 Introduction

The Vesta repository is a special-purpose replicated file system, developed as part of the Vesta software configuration management system.[1] Some of our major goals in developing Vesta were to make all software builds reproducible, to support a flexible form of distributed software development, and to make the system as fast, simple, and understandable as we could. An overview of the full system [15] and papers on its executable modeling language [13] and incremental builder [16] are available. A book-length report is in preparation [14].

To make the repository understandable and convenient for users, we designed it as an extended file system. All versions of all source code stored in the repository are visible in the file system name space, so users can browse and compare them using existing, familiar tools. We do not provide mechanisms such as views [2] or viewpaths [3] that can make the meanings of names appear to change depending on the user, the environment, or what versions currently exist.

The repository plugs into the file system name space by acting as an NFS [19] server. It also exports a separate remote procedure call (RPC) interface, for access to operations that do not map naturally onto a standard file system interface. The repository runs as an ordinary user process, not in the operating system kernel. It implements directories as in-memory data structures that are kept stable using a simple logging and checkpointing technique [5]; it implements files by storing their contents in a hidden native file system on the host machine.

To support reproducible builds, we extended the file system abstraction by allowing files to be designated as *mutable* or *immutable*, and directories to be designated as *mutable*, *appendable*, or *immutable*. In an appendable directory, new names can be created, but existing names cannot be deleted and their meanings cannot be changed. Children of an appendable directory must be appendable or immutable; children of an immutable directory must be immutable. Vesta's builder reads source code and build instructions from a directory tree (conventionally named /vesta) consisting only of appendable and immutable objects, so the meanings of names cannot change from one build to another.[2]

Versioning is not built into the repository itself; instead, a set of replaceable *repository tools* handle versioning by defining a naming convention. Immutable files and directories are used for particular versions of source files or source trees.

[1]Vesta as discussed in this paper, also known as *Vesta-2*, is a redesign of the Vesta-1 system developed in the early 1990's [7, 8, 12, 17]. The Vesta-1 and Vesta-2 repositories have recognizable similarities, but most of the key ideas presented in this paper are new in Vesta-2.

[2]The Vesta modeling language [13] is designed so that a successful build cannot depend on the nonexistence of a name in an appendable directory; thus adding a new name to an appendable directory cannot change the result of a build, though it can make a build that formerly failed succeed.

Appendable directories are used both as directories of versions, where their children are immutable objects named by version numbers; and at all higher levels in the tree, where they form a user-defined hierarchical name space of versioned software packages. This approach contrasts with that of software repositories that have built-in versioning [18, 20], which does not map straightforwardly onto a Unix-style unversioned name space.

Besides the appendable /vesta tree, the Vesta repository also provides a tree of mutable files and directories, conventionally named /vesta-work. The repository can create and initialize a mutable directory subtree with the contents of a given immutable directory, and can create an immutable directory subtree as a snapshot of a given mutable directory. Both these operations are implemented using copy-on-write for files and delta encoding for successive versions of directories, making them very fast and avoiding wasted space.[3] The repository tools use these operations to implement a checkout/checkin style of source control.

The repository also supports the Vesta builder by providing a means for dynamically constructing arbitrary name spaces in which build tools (such as compilers and linkers) can be run, called *volatile* directories. Volatile directories enable the builder to run build tools in an encapsulated environment, in which only the desired versions of sources and derived files are available. (Thus they give at least the same power as viewpaths [3], but under direct control by the builder.) Volatile directories also let the builder detect the precise dependencies and results of each build tool invocation: the repository tells the builder exactly which of the available files and directories the build tool accessed and what new files and directories it created. We do not discuss the volatile directory mechanism or derived file management further in this paper.

To support distributed software development, sources stored in the repository's appendable tree can be *replicated*. Our concept of replication is broad, encompassing everything from source distributions issued on CD-ROM to cooperative development of one source pool across many geographically separated sites. Conceptually, all files and directories stored in all Vesta repositories are named in one single, global name space. Each repository stores some subtree of the complete name space. Replication is present when the subtrees stored by two different repositories overlap; that is, when some of the same names and data occur in both.

We call this concept *partial replication*, because each repository can choose to replicate all, part, or none of the data stored in other repositories. Typically, each Vesta site runs one repository and replicates in it all the sources that need to be built

[3]We do not currently delta-encode files, because they are stored on disk and normally contain source code that is typed by a human. Considering the rapidly falling price of disk space, delta encoding or otherwise compressing such files has little benefit. We do delta-encode directories, because we store them in memory.

locally, since the Vesta builder works on only one repository at a time. Sources that are needed at many sites can be replicated at each one, while other sources need not be.

The append-only nature of the repository greatly simplifies the problem of defining and maintaining consistency among replicas. Roughly speaking, we say that two repositories *agree* (are consistent) if there are no cases where the same name means something different in two different repositories. We make this definition precise in Section 2. To allow names to be added to appendable directories without risking disagreement or requiring multiple repositories to coordinate, we adopt the simple expedient of designating (at most) one repository to be the *master* of each directory. The master keeps a complete list of names that have been defined in the directory and can freely add new names, while nonmaster replicas may have incomplete lists and can add a new name only by copying it from another replica (not necessarily the master). A newly created object initially has the same master as its parent directory, but we allow mastership for any object to be transferred to another repository. To free a directory's master from the storage burden of having to keep a complete copy of all objects below it, we also provide *stub* objects—placeholders for names that exist but whose content is not stored locally. Stubs can be children only of appendable directories.

Within our concept of replication, many different algorithms might be used to update one replica from another. We have written a simple replication tool that updates a destination repository from a source repository. The tool walks over the source repository, finds all names that match a set of patterns provided to it and that are not already in the destination, and copies them.

As additional support for the repository tools, the repository provides *mutable attributes* on the files and directories it stores. An object's attributes are a map from arbitrary character string names to sets of character string values. The tools use attributes for various purposes, such as keeping track of relationships between versions and recording checkin comments. The repository itself uses attributes to hold access control lists. Attributes are not visible to the Vesta builder, so their mutability does not interfere with build reproducibility.

Since attributes are arbitrarily mutable, their replication has to be handled differently from files and directories. We use a replication scheme inspired by Grapevine [4]. We define the current value of an attribute by a history list of timestamped update tuples. All tuples are valid and meaningful regardless of the past history; for example, deleting a value from a name is valid even if the name did not previously have that value. An object's attributes at one replica can be updated from a second replica by copying in any tuples from the second that do not exist at the first and merging them into the list in timestamp order (with ties broken by sorting on the other fields of the tuple). There is no notion of inconsistency; any

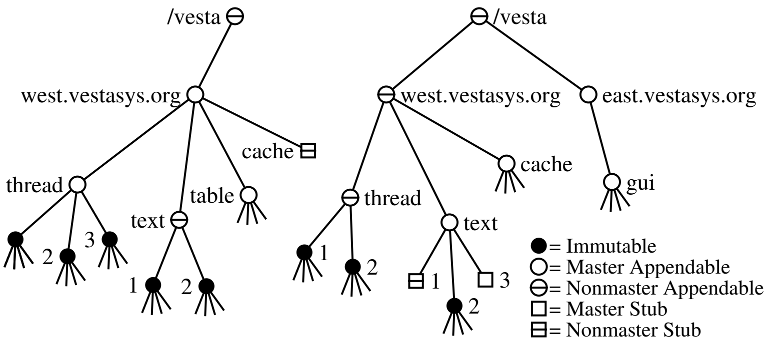

Figure 1: Two repositories (western and eastern) that are in agreement.

history can be merged with any other to produce a well-defined result. Although this form of replication is much looser than the agreement condition we use for files and directories, it is adequate for our purposes.

The remainder of this paper discusses the above issues in more detail. Section 2 gives the full definition of agreement between replicas and shows how replicas can be created and changed in a decentralized fashion without the risk of introducing disagreement. Section 3 describes our replica update tool, and Section 4 describes how the standard repository tools operate across multiple repositories. Access control and security are difficult problems in a system that can cross administrative boundaries; Section 5 explains how we have addressed these issues in our design. We compare the Vesta repository with related work in Section 6. Section 7 summarizes our experience with the system and our conclusions.

2 Replica Agreement

2.1 Example

Figure 1 shows an example of two repositories that partially replicate each other and are in agreement (consistent). The figure illustrates several common patterns that occur in real Vesta usage, but the pathnames have been shortened slightly to reduce clutter. On the left is the west coast repository of the imaginary Vesta Systems Organization; on the right is its east coast repository.

First, note that the root directory /vesta is not mastered at either repository, and the names directly under it look like Internet domain names. In order to avoid getting into the business of running a global name registry for /vesta, we treat this directory as a special case; any individual or organization may create a name in it using an Internet domain name that it owns.[4] In the example, the western repository has created a subdirectory named west.vestasys.org and holds its master copy, and the eastern repository has done the same for east.vestasys.org.

Partial replication occurs at several levels of the tree. At the top, part of west .vestasys.org is replicated in the eastern repository. The western copy has a complete list of names (thread, text, table, and cache), while the eastern copy is lacking table. The western copy does not have the contents of the cache directory, but it does have a stub with that name as a placeholder; thus no one can create a different cache directory that would clash with the copy in the eastern repository.

One level down, in the thread package, the western copy is master and has all three versions that currently exist, while the eastern copy does not currently have version 3. The text package illustrates the point that a directory need not have the same master as its parent; it is mastered at the eastern repository. Perhaps when first created it was mastered at the western repository and later moved to the eastern repository, since version 1 is present in the west but not in the east. Since the eastern copy is master, it must have a complete list of names, so it has a stub for version 1, perhaps inserted at the time it received mastership. In addition, the eastern copy has a master stub for version 3. A master stub is a placeholder for an object whose content has not yet been supplied; the master repository is free to replace it later with a different type of object, but thereafter it cannot be changed back to a master stub. The repository tools use master stubs to implement a locking style of checkout (see Section 4).[5]

2.2 Definition

We are now ready to give the precise definition of agreement. Let A and B be Vesta source objects. Then $A \simeq B$ (read "A agrees with B") if the following recursively defined conditions hold. Two repositories agree when their replicas of the root directory /vesta agree. Let $A.master$ denote the master flag of A, let $A.repos$ denote the repository where A is stored, and if A is a directory, let $A.names$ denote the list of names that are bound in it.

1. $A.master \wedge B.master \Rightarrow A.repos = B.repos$ and
2. At least one of the following holds:
   (a) A and B are files with identical contents.
   (b) A and B are immutable directories where

[4]This rule is not perfectly safe, since the Vesta system cannot check that users are following it, and moreover domain names can be deregistered and reregistered by different owners, but it is a good practical compromise.

[5]For historical reasons, the current repository implementation also provides *ghosts*, which are essentially a variant type of nonmaster stub; we do not discuss them in this paper.

$A.names = B.names$, and
$\forall n : n \in A.names \Rightarrow A/n \simeq B/n$,

(c) A and B are appendable directories where
$\forall n : n \in A.names \land n \in B.names \Rightarrow A/n \simeq B/n$,
$A.master \Rightarrow B.names \subseteq A.names$, and
$B.master \Rightarrow A.names \subseteq B.names$.

(d) A and B are both master stubs.

(e) A or B is a nonmaster stub.

Condition 1 effectively says that the same source is not mastered in more than one repository. It is stated in an odd-sounding way (and condition 2(d) is included) so that agreement can be reflexive. Conditions 2(a) and 2(b) require replicas of immutable objects to be identical. Conditions 2(c) and 2(e) make partial replication possible. By 2(c), two appendable directories can agree even if one or both have only a subset of the complete set of names defined in the directory. But the master replica has a complete list of names; thus, the master can coordinate the creation of new names, assuring that the same name is never bound in different replicas to sources that do not agree. By 2(e), two appendable directories can agree even if one has a nonmaster stub where the other has some other object.

Notice that the agreement relation is not transitive. Pairwise agreement between A and B and between B and C is not sufficient to guarantee agreement between A and C. This nontransitivity is an unavoidable property of partial replication. Replicas are considered to agree when their overlapping portions do not clash; but A and C may overlap and clash in a portion that does not overlap with B. For example, /vesta/foo/bar might be an immutable directory in repository A, absent in repository B, and an immutable file in repository C. Then B agrees with A and with C (assuming /vesta/foo is not mastered at B), but the directory at A clashes with the file at C.

2.3 Preservation

It is easy to establish initial agreement among repositories, since a new repository that contains only an empty copy of the root directory /vesta agrees with every other repository. Thereafter, we need to know how to make changes to agreeing repositories in a safe way, one that is guaranteed not to break the agreement. We want repositories to operate mostly autonomously, so it is important that most operations can be performed without consulting another repository, and that the rest require consulting only one other. Our definition of agreement is designed to make this fairly easy. The following operations are safe, and the repository server provides each one as a primitive. All are atomic except primitive 7.

1. Create a new master appendable directory in /vesta, using a locally owned Internet

domain name.

2. Create a new child object of any immutable or appendable type in a master appendable directory.
3. Replace a master stub with a new immutable object.
4. Replace any child of an appendable directory with a nonmaster stub.
5. Copy any child into an appendable directory from another repository, possibly replacing an existing nonmaster stub. If the original is an appendable directory, the copy is an empty nonmaster appendable directory; if desired, its children can be copied by further applications of this primitive. If the original is immutable, however, it is copied in full, including all its descendants.
6. Create a nonmaster stub in an appendable directory, if another repository has that name defined.
7. Transfer mastership on an object from one repository to another, at the same time adding stubs to the new master for any missing children of the old master.

Internally, the repository builds the more complex primitives on top of a set of simpler primitives for adding and replacing single objects, using a built-in feature that allows for short atomic transactions within a single repository.

Primitives 1–4 run entirely at a single repository.

Primitives 5 and 6 require consulting another repository, but a multi-site atomic transaction is not required; it is sufficient to read the data from the source repository, then atomically insert a copy into the destination. No lock on the source repository is needed while reading the original, since it cannot change; at worst, it can be replaced with a stub while the read is in progress, but this simply causes the primitive to return an error without changing the destination. The destination repository optimizes the copying process to avoid making redundant copies of objects (such as multiple objects that have the same content but different names) by keeping a table in which each locally stored immutable file and immutable directory tree can be looked up by its fingerprint [6]. When an object is to be copied, the repository first looks up its fingerprint in the table to find whether a copy is already present; if so, the repository links the existing copy into its name space instead of making another. In addition, if a directory being copied is delta-encoded in the source repository, and the destination repository already has a copy of the directory that the delta is relative to, then the destination delta-encodes the copy as well.

Primitive 7, mastership transfer, is the most complex. Our goals in choosing an implementation were to guarantee that agreement could not be violated, to avoid blocking either repository during the transfer protocol, to minimize the likelihood of a failure resulting in neither repository being master, and to keep a hint on each nonmaster object as to where its master repository is located. In outline, our implementation consists of two separate atomic operations. First, the repository ceding mastership on an object makes a complete list of its children and turns off its master

Replicate	Size	vrepl		rcp	
		Nearby	Distant	Nearby	Distant
+repos/124 to empty repository	1.2 MB	1.7 s	45 s	17 s	57 s
+repos/125 with 124 present	582 KB	0.6 s	14 s	1.6 s	14 s
@repos/124 to empty repository	127 MB	140 s	3200 s	620 s	2600 s
@repos/125 with 124 present	640 KB	9.6 s	120 s	1.4 s	13 s

Table 1: Vesta replicator performance.

flag; second, the repository acquiring mastership inserts any missing children into its copy as nonmaster stubs and turns on the master flag. To meet our goals, the full implementation also takes care of updating the master location hints, and it keeps a stable record of in-progress transfers at both repositories, persistently retrying them until they are complete. With this implementation, agreement cannot be violated, and the object can be left without a master only if one repository crashes permanently or the network link between them is permanently severed while a transfer is in progress.

3 Replicator

The primitives listed in the previous section give us a safe way to copy data and transfer mastership between repositories, but they are quite low-level. In this section we briefly describe a higher-level replicator. It is available both as a standalone tool (**vrepl**) that can be invoked from the command line and as a library that can be called by other tools, such as the checkout and checkin tools described in the next section.

Our replicator takes as input a set of pathname patterns and the network addresses of two repositories, a source and a destination. The replicator walks the directory tree of the source to find all names that match the patterns and copies those that are not already present at the destination. It also updates the mutable attributes of every name that matches by merging update tuples from the source into the destination. As a trivial example, the command "vrepl -d east.vestasys.org -e+ /vesta/west.vestasys.org/vesta/repos/LAST" would replicate the highest-numbered version of the Vesta repository source code from the local repository into the repository at east.vestasys.org. The full pattern language is similar to Unix shell **glob** patterns with some extensions. Prefixing a pattern with "+" adds the objects that match it to the set to be copied; prefixing it with "-" removes them.

The replicator also has a feature that replicates everything needed to do a particular Vesta build. For example, the command "vrepl -s west.vestasys.org -e@ /vesta/west.vestasys.org/vesta/repos/124/.main.ves" would replicate everything needed to rebuild version 124 of the Vesta repository from the west.vestasys.org repository into the local repository, including the entire programming environment (libraries, compilers, etc.) that was available to the build. This feature works by first parsing the build description (written in the Vesta modeling language [13]), walking its import tree, and emitting a + pattern for every package found; then passing these patterns on to the basic replication algorithm. In practice, it has turned out that @ is used far more often than + and -.

To give an idea of the replicator's performance, Table 1 gives a few simple measurements. Each repository was running on a 500 to 600 MHz Alpha 21164A processor. All times are averaged over three trials and rounded to two significant figures. In the *Nearby* cases, the two repository host machines were directly connected via gigabit ethernet. In the *Distant* cases, the two machines were on opposite coasts, connected via 10 hops through a corporate intranet. As a rough point of comparison, the *rcp* columns give the time to copy the same files directly out of the native file system that the repository is built on top of.

4 Cross-repository Checkout

When two sites running separate repositories are closely cooperating, users at one site may want to check out packages whose master copies are in the other site's repository. In this section we outline how our checkout tools support this. We first briefly describe the single-repository case, then explain the extension to the cross-repository case.

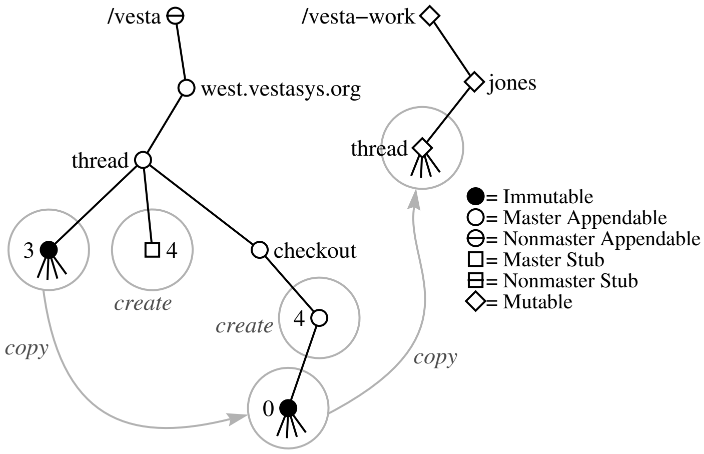

Figure 2: Single-repository checkout.

The **vcheckout** tool, illustrated in Figure 2, makes a copy of the latest version of a software package (here /vesta/west.vestasys.org/thread/3) in a mutable working directory that the user can edit (/vesta-work/jones/thread). As mentioned earlier, making this copy is very fast, because the mutable directory is itself stored in the repository and is implemented using delta encoding, with copy-on-write for the files stored in it. The tool also makes an appendable directory called the *session directory* (thread/checkout/4), where the user can store intermediate, private versions of the package that he creates during development but that are not ready to be checked in for use by others. The session directory is initialized to contain a version 0 that is identical to the initial contents of the working directory. Finally, the next version number for the package is reserved by creating a master stub with that number as its name (thread/4).

The **vadvance** tool advances to the next private version by taking an immutable snapshot of the working directory and inserting it in the session directory. In the example of Figure 2, repeatedly editing files in /vesta-work/jones/thread and invoking vadvance would create private versions named /vesta/west.vestasys.org/thread/checkout/4/1, 4/2, 4/3, etc. Our tools support such private versions for two reasons: first, they are a convenient feature, offering the developer the ability to look back at his recent work and undo mistakes; second, in order to guarantee that *all* builds are reproducible, even a developer's private test builds, the Vesta builder operates only on immutable sources. Thus an immutable version must be created each time the builder is to be run on modified sources. (We generally run the builder from a short shell script that automatically invokes vadvance before starting the build.)

Finally, the **vcheckin** tool makes a new public version. It replaces the master reservation stub created by vcheckout (thread/4) with a copy of the newest private version in the session directory (in our example, perhaps thread/checkout/4/3), then deletes the working directory to end the session. The session directory is not automatically deleted, since it may be of use later.

How do the tools change for the cross-repository case? Almost all of the changes are in vcheckout, as shown in Figure 3. The source repository, where the package being checked out is mastered, is on the left; the destination repository, which the user doing the checkout wants to work in, is on the right. Notice that the actions in the destination repository are similar to the single-repository case in Figure 2. The steps in cross-repository checkout are: (1) Examine the master replica in the source repository to find the highest version number. (2) If this version does not exist in the destination, call the replicator to copy it in. (3) Create the reservation stub and empty session directory in the source repository. (4) Call the replicator to copy them to the destination repository. (5) Transfer mastership on them from the source to the destination. (6) Insert version 0 in the session and create the working directory at the destination.

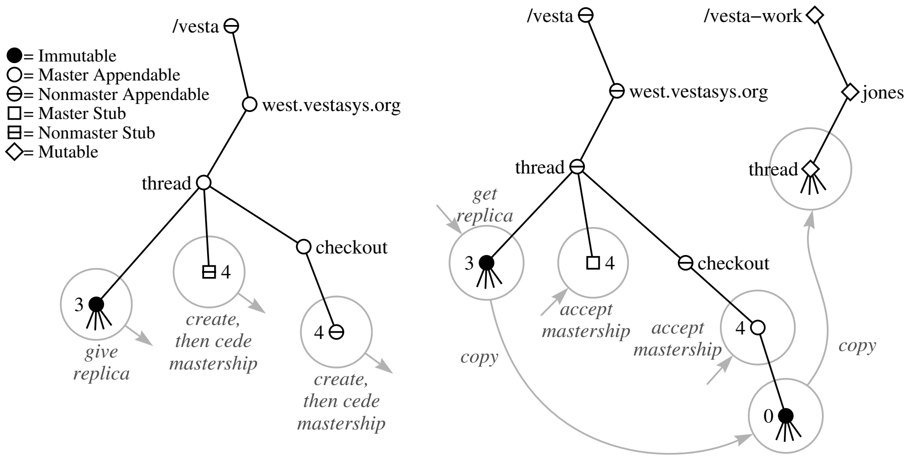

Figure 3: Cross-repository checkout.

No changes are needed to vadvance, since both the working and session directories are in the destination repository.

It would not be strictly necessary to change vcheckin either, since vcheckout moves mastership of the reservation stub to the destination repository. However, it is likely that the source repository will want a copy of the new version soon, so we modified vcheckin to call the replicator and copy the new version there after checking it in locally.

There are also Vesta tools for creating new packages, branching the version sequence, finding the latest version, and finding who has packages checked out. Each of these tools required minor modifications to be cross-repository aware, similar to what was done to vcheckout but considerably simpler.

One problem currently remains with the cross-repository tools. In the single-repository case, each of our tools uses the repository's short atomic transaction support to make its complete action atomic. This support does not work across multiple repositories, so the tools become nonatomic in this case. With vcheckout, steps (3) and (6) are individually atomic, but if there is a failure between them, the checkout is left in an incomplete state. We have not yet automated the recovery from this state, but it is not hard to recover manually.

Table 2 shows the results of a performance benchmark on the Vesta tools. The steps listed were run in order, 50 times each on 50 separate packages. The table gives the average time for each step, rounded to two significant figures. The *Local* column is the single repository case, *Nearby* is the cross-repository case where the local and remote repositories are connected by a single hop of gigabit ethernet, and

Step	Description	Local	Nearby	Distant
1	create an empty package	50 ms	250 ms	6400 ms
2	check out the new package	64 ms	590 ms	5700 ms
3	copy in 1204 KB of source code	5300 ms	5400 ms	5200 ms
4	advance the package	2500 ms	2600 ms	2500 ms
5	advance again (no changes)	170 ms	170 ms	180 ms
6	touch a 108 KB file	34 ms	32 ms	30 ms
7	advance the package	160 ms	180 ms	180 ms
8	touch all files in the package	3200 ms	3300 ms	3200 ms
9	advance the package	2500 ms	2500 ms	2600 ms
10	check in the package	110 ms	780 ms	18000 ms
11	check out the package	49 ms	730 ms	5800 ms
12	touch a 108 KB file	49 ms	73 ms	59 ms
13	advance the package	150 ms	160 ms	160 ms
14	check in the package	64 ms	170 ms	4700 ms

Table 2: Vesta repository tool performance.

Distant is the cross-repository case where the repositories are on opposite coasts connected by ten hops through a corporate intranet. Note that copying, touching, and advancing are always local operations. In each case, the tools were run on a client workstation connected to the local server by 100 Mb ethernet. As the table shows, the tools are very fast in the local and nearby cases, and fast enough to be usable even in the distant case.

5 Cross-realm Access Control

Access control for replicated repositories presents some challenges. We expect replication and even cross-repository checkout to sometimes be needed between repositories that are in different *realms*; that is, repositories that are under separate administration, have different spaces of user names, and perhaps do not entirely trust one another. Thus we need a practical way of authenticating and access checking cross-realm requests. In addition, for repositories that are cooperating closely, we would like it to be meaningful to replicate access control lists, so that each repository does not have to separately administer them. This section outlines how we have addressed these issues.

All access control lists and access checking in the repository are done in terms of global principal names, with the syntax `user@realm` or `^group@realm`. The realm is an arbitrary name chosen by a system administrator, typically an Internet domain name that makes the realm's user names valid email addresses.

We have kept the repository's access control lists close to the Unix style so that they can be manifested fairly accurately through the NFS interface. Each object has an owner ACL and a group ACL, plus a set of nine flags that indicate whether the owner, group, and others are granted read, write, and/or directory search access. Unlike the Unix model, the owner and group are sets of global names, not singletons; we have provided this feature mainly so that an object can be given different owners in different realms if desired. Therefore, when choosing which owner and group to manifest through the NFS interface, the repository searches first for one in the local realm. The repository maps between global principal names and the numeric ids used through the NFS interface by examining the local operating system's user and group registries (on Unix, /etc/passwd and /etc/group) and building up a translation table.

Objects also have a few special ACLs that refer to repositories rather than users: one lists the repositories that mastership on the object can be ceded to, one lists the repositories that mastership can be accepted from, and one lists the repositories that replicas can be taken from. No special ACL is needed to control giving replicas; read access by the requesting user is sufficient for that.

Access control lists and flags are stored in mutable attributes and can be replicated if desired. To save space, we use a form of inheritance; if an object does not have a particular access control attribute, it inherits the value from its parent directory. The names of all access control attributes begin with an identifying character ("#"), and the replicator can be instructed not to copy them even if ordinary attributes are being copied.

Each incoming request to the repository must be authenticated as coming from some particular user. Several authentication methods are supported. The repository administrator fills in a table (similar in style to an NFS export table) that specifies which user names to accept, from which hosts, using which authentication methods, and whether to grant normal or read-only access. Currently we have implemented only two rather insecure authentication methods: the NFS AUTH_UNIX style, where the user provides his numeric Unix user id and is believed if he comes from a trusted host (needed to support most current NFS clients), and a similar style where the user supplies a global principal name and is believed if he comes from a host that is trusted for names from that realm. We have a design sketch for adding Kerberos authentication and hope to implement it in the future.

Our general model for group access is that the user need only authenticate his user name; the repository determines what groups he is in. This model works well for intra-realm access, because the repository uses the local operating system's facilities to determine the group membership of local users. A user accessing a remote repository, however, by default will not be recognized as a member of any groups, even groups from his own realm. To address this problem, we allow repository administrators to augment the group membership table with additional entries, but this is a bit labor-intensive. A useful future addition might be to allow group membership tables to be replicated.

As an additional feature, one user name or group name can be designated as an alias for another; this is useful when the same person has a login in two different realms or when two cooperating realms each have a group that is working on the same project.

6 Related Work

The term *repository* is common in the software configuration management literature; it has been used to mean anything from a simple collection of versioned source files to a full-fledged relational or object-oriented database containing detailed information to support the software development process. Van der Hoek [20] provides a useful overview of such systems. Our repository lies between these extremes, storing source files and mutable attributes in a hierarchical name space, and also providing unstructured storage (not described in this paper) for derived files managed by the Vesta builder and cache.

Replication of mutable data has engendered many complex algorithms and systems. Strict consistency tends to imply low availability, and is not suitable for our application: reads and writes become atomic operations on replica quorums, with the requirement that every write quorum intersect every read or write quorum, forcing writes (and perhaps also reads) to be performed as multi-site atomic transactions [11]. Loosening consistency requirements tends to open a Pandora's box of complexity; the many algorithms that have been used for propagating updates and reconciling inconsistent changes made to different replicas are complicated, and it is difficult to give a precise definition or make guarantees on what sort of consistency they provide [9]. We have cut through this knot of complexity by restricting the mutability of the data to be replicated, allowing us to give a simple, flexible, and comprehensible definition of consistency for partial replicas and to preserve this consistency with simple algorithms.

Our approach to replication differs sharply from that of ClearCase MultiSite [1], which is perhaps the most closely related software configuration management system to Vesta. In ClearCase, the choice of what to replicate is made at a much coarser grain than in Vesta—an entire Versioned Object Base, of which there are typically only one or a few per site. ClearCase replicas exhibit eventual consistency; that is, an update algorithm is used that would eventually make the replicas identical if they all were to stop changing for sufficiently long, but there are no clear guarantees on what differences can exist between replicas when changes have been made recently. ClearCase's update algorithm is operation-based and requires knowledge of the full set of replicas; that is, each ClearCase replica must keep a history of recent operations that have changed it and keep track of which changes have not yet been propagated to each other replica. Vesta's algorithm is state-based and works when the set of replicas is unknown and changing; the replication tool simply compares the states of two replicas, copying any data from the first that is missing and desired in the second. The ClearCase approach has some advantages; in particular, an operation-based approach should scale better when replicas share a great deal of data but very little is changing. However, the Vesta approach is much simpler, provides a clearly defined level of consistency, supports usage patterns where the replicas are more loosely coupled, and has performed adequately in our experience.

The Vesta repository is implemented as a user-space NFS server. Other approaches to plugging into the file system name space include intercepting system calls by replacing a shared library (as in n-DFS [10]) or connecting to the kernel's file system switch (as in ClearCASE [2]). We chose the NFS server approach for ease of debugging, portability, and compatibility with statically linked applications. Our approach probably sacrifices some performance; although we do not have measurements of the systems just cited, the repository is certainly slower than a kernel resident file system. (In comparisons on the same hardware, the repository shows a write data rate of about 96% of a standard in-kernel NFS server, a read data rate of about 55%, and comparable performance on other operations. Vesta builds are still roughly as fast or faster than builds using *make* in a conventional file system, however, owing to efficiencies in the builder [16] and to some steps of a typical build being CPU-bound.)

7 Experience and Conclusions

For the past two years, Vesta was in daily use by a group of about 150 developers working on a large microprocessor design. Both the chip design itself and the group's custom design software were stored in the repository and processed using the builder; the current code size is about 500,000 lines.

Initially only two repositories were in production use, one belonging to the chip designers (in New England) and the other to the Vesta developers (in California). We ran the replicator in both directions: distributing updates to Vesta by copying the source code from west to east, and reproducing build problems for debugging by copying parts of the chip design from east to west.

More recently the replication features have been used much more intensively, as some of the members of the New England group began working on Vesta maintenance and porting, and a group of developers in California joined the chip design team using a third repository. The three groups have made extensive use of cross-repository checkout and cross-realm access control.

We found this experience highly valuable in validating the Vesta design, shaking out bugs in the implementation, and exposing the need for features that were not initially anticipated. The chip design team in turn was pleased with Vesta; they have stated that its strong support for parallel source development and reproducible builds saved them considerable time (3 to 6 months in the architectural design phase alone), and that the cross-repository features provided answers to some extremely difficult problems in bicoastal software and design database management.

Our ongoing work on Vesta includes porting the implementation to Linux, making it publicly available under an open source license, and completing a book-length report on the entire system.

Acknowledgements The Vesta project has extended over many years, and there have been many contributors. For suggestions, support, and feedback on the Vesta-2 repository design and implementation, special thanks are due to Roy Levin, Allan Heydon, Yuan Yu, Butler Lampson, Neil Stratford, Matt Reilly, and Ken Schalk. I am also indebted to Sheng-Yang Chiu for his work on the Vesta-1 repository.

References

[1] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, pages 194–214, 1995. Available as volume 1005 in *Lecture Notes in Computer Science*, Springer-Verlag.

[2] Atria Software, Inc., 24 Prime Park Way, Natick, MA 01760. *ClearCase Concepts Manual*, 1992.

[3] Dave Belanger, David Korn, and Herman Rao. Infrastructure for wide-area software development. In *Proceedings of the 6th International Workshop on Software Configuration Management*, pages 154–165, 1996. Available as volume 1167 in *Lecture Notes in Computer Science*, Springer-Verlag.

[4] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[5] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 149–154. The Association for Computing Machinery, November 1987.

[6] Andrei Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[7] Mark R. Brown and John R. Ellis. Bridges: Tools to extend the Vesta configuration management system. SRC Research Report 108, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.dec.com/pub/DEC/SRC/research–reports/abstracts/src–rr–108.html.

[8] Sheng-Yang Chiu and Roy Levin. The Vesta repository: A file system extension for software development. SRC Research Report 106, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.dec.com/pub/DEC/SRC/research–reports/abstracts/src–rr–106.html.

[9] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[10] Glenn Fowler, David Korn, and Herman C. Rao. n-DFS: Multiple dimensional file system. In Walter Tichy, editor, *Configuration Management*, volume 2 of *Trends in Software*, pages 135–154. John Wiley and Sons Ltd., 1994.

[11] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 150–159. ACM SIGOPS, December 1979.

[12] Christine B. Hanna and Roy Levin. The Vesta language for configuration management. SRC Research Report 107, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.dec.com/pub/DEC/SRC/research–reports/abstracts/src–rr–107.html.

[13] Allan Heydon, Jim Horning, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta-2 software description language. SRC Technical Note 1997–005c, Digital Equipment Corporation, Systems Research Center, June 1998. http://gatekeeper.dec.com/pub/DEC/SRC/technical–notes/abstracts/src–tn–1997–005c.html.

[14] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. Vesta: A system for software configuration management. SRC Research Report, Compaq Computer Corporation, Systems Research Center. To appear.

[15] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta approach to software configuration management. SRC Research Report 168, Compaq Computer Corporation, Systems Research Center, March 2001. http://gatekeeper.dec.com/pub/DEC/SRC/research–reports/abstracts/src–rr–168.html.

[16] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[17] Roy Levin and Paul R. McJones. The Vesta approach to precise configuration of large software systems. SRC Research Report 105, Digital Equipment Corporation, Systems Research Center, June 1993. http://gatekeeper.dec.com/pub/DEC/SRC/research–reports/abstracts/src–rr–105.html.

[18] Boris Magnusson and Ulf Asklund. Fine grained version control of configurations in COOP/Orm. In *Proceedings of the 6th International Workshop on Software Configuration Management*, pages 31–48, 1996. Available as volume 1167 in *Lecture Notes in Computer Science*, Springer-Verlag.

[19] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.

[20] André van der Hoek, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf. A testbed for configuration management policy programming. *IEEE Transactions on Software Engineering*. To appear. Available from http://www.ics.uci.edu/ãndre/.