

ClearCase MultiSite: Supporting Geographically-Distributed Software Development

Larry Allen, Gary Fernandez, Kenneth Kane,
David Leblang, Debra Minard, John Posner

Atria Software, Inc.
24 Prime Park Way
Natick, Massachusetts
U.S.A.

Abstract

For a software configuration management system to support large-scale development efforts, it must address the difficult problem of geographically-distributed development. This paper describes the rationale and design of Atria Software Inc.'s *ClearCase MultiSite*TM software, which extends the ClearCase[®] configuration management system to support geographically-distributed development through *replication* of the development repositories. This paper considers alternatives to replication and discusses the algorithms used by ClearCase MultiSite to ensure replica consistency.

1 Introduction: Parallel Development and ClearCase

The size and complexity of software projects has increased greatly over the years. It is common to find a single software system with many million lines of code under development by several hundred software engineers. Large projects need to have several independent "lines of development" active at the same time. The process of creating and maintaining multiple variants of a software system is termed *parallel development*. A particular variant might be a major project (porting an application to a new platform), or a minor detour (fixing a bug; creating a "special release" for an important customer). Good support for parallel development is a key requirement for any version control and configuration management system targeted at large development environments.

Atria Software's *ClearCase* product provides configuration management and software process control for parallel development in a local area network. *ClearCase MultiSite* extends the single-site parallel development model of ClearCase to provide geographically-distributed parallel development. Accordingly, we first describe the fundamental concepts of ClearCase, and then describe the MultiSite approach.

1.1 ClearCase Basics

ClearCase is a comprehensive software configuration management system. It manages multiple variants of evolving software systems, tracks which versions were used in software builds, performs builds of individual programs or entire releases according to user-defined version specifications, and enforces site-specific development policies and processes.

A ClearCase *versioned object base* (VOB) is a permanent, secure data repository. It contains data that is shared by all developers: this includes current and historical versions of source objects (*elements*), along with *derived objects* built from the sources by compilers, linkers, and so on. In addition, the repository stores detailed “accounting” data on the development process itself: who created a particular version (and when, and why), what versions of sources went into a particular build, and other relevant information. In addition to source, derived, and historical data, a VOB stores user-defined *meta-data*, such as mnemonic version labels, inter-object relationships, and object attributes.

VOBs are globally accessible resources, which can be distributed throughout a network. They act as a federated database: they are independent but cooperative, and can be linked into one or more logical trees. A project may have some private VOBs, and also use shared VOBs that hold common interfaces or reusable components. The VOB is the unit of data that is replicated with MultiSite.

There are many versions of each file in a VOB, and there may be many names and directory structures for the files (reflecting reorganizations of the source tree over time). Rather than copying versions into a physical workspace, ClearCase uses virtual file system technology to create a virtual workspace called a *view*. A view makes a VOB look like an ordinary file system source tree to users and their off-the-shelf tools (Figure 1). A set of user-specified rules determines which version of each file and directory is visible through a view.

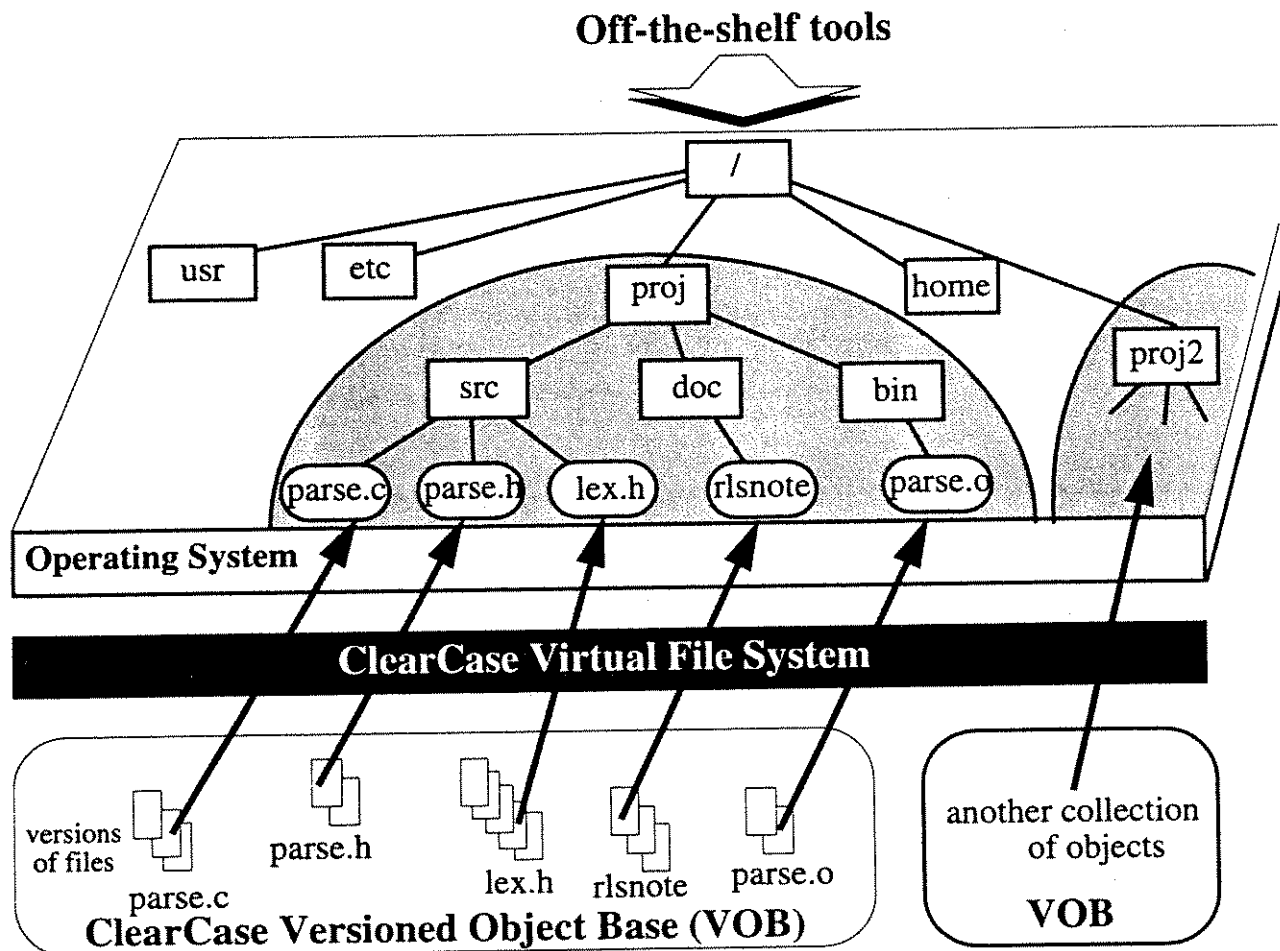


Fig. 1. ClearCase Virtual File System

1.2 Parallel Development with Branches

ClearCase allows multiple developers to modify a single source file simultaneously, without contention or loss of changes. This parallel development capability is accomplished through *branching* — the maintenance of multiple independent lines of descent in the version tree, each of which evolve independently (Figure 2).

For example, a product may require ongoing bug fixes to Release 1.0, while development of Release 2.0 continues in parallel. ClearCase supports this scenario by maintaining separate branches for new development (“main” branch) and for bug fixes (subbranch). This ensures that bug fixes do not accidentally pick up new development work, which has not yet been fully tested. Each branch can be independently checked out and checked in.

ClearCase also has the notion of *branch types* to provide central administration of branches. A branch in an element is really an instance of a particular branch type — that is, it points back to the branch type for common information such as the branch name, access control rights, and comments describing the purpose of the branch.

Eventually, changes made on multiple branches should be reconciled or *merged*. ClearCase provides powerful tools for finding branches that need to be merged, for performing the merges (automatically where possible, and obtaining human input if required), and for tracking merges that have been performed, both for reporting purposes and to optimize subsequent merge operations.

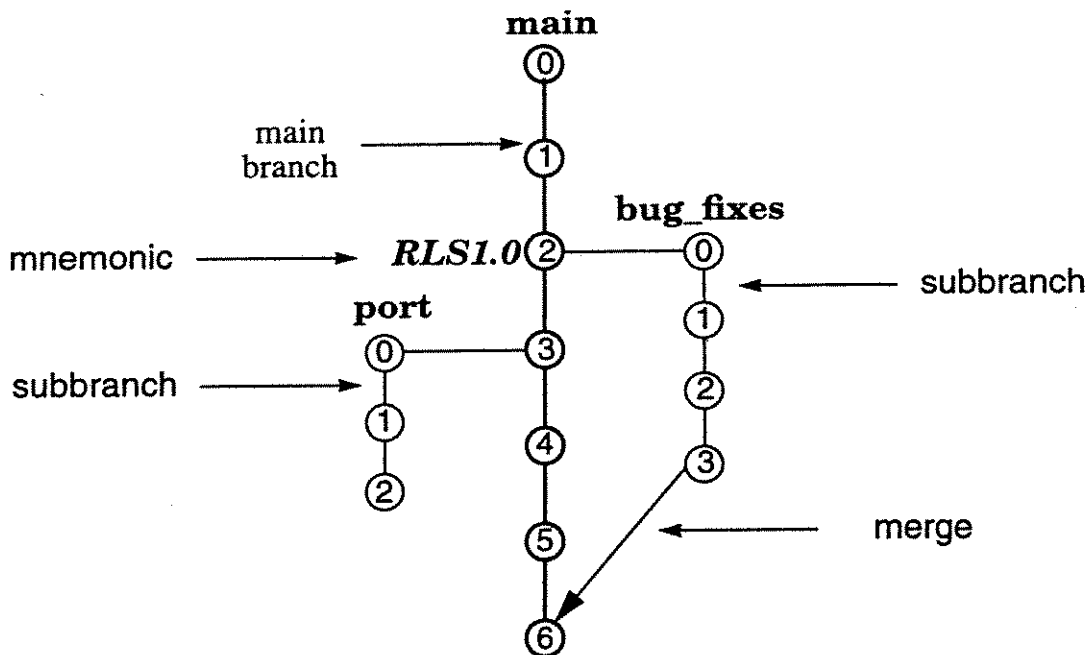


Fig. 2. ClearCase Branching Model for Parallel Development

1.3 ClearCase Meta-Data

In addition to source files and change histories, a VOB holds several of other types of important information, called *meta-data*:

- *Version labels* are mnemonic names for particular versions. For example, *foo.c* version 21 might be tagged with the label “RLS2” in order to indicate that the version was used in the build of the second release.

- *Attributes* are name/value pairs, which can be attached to individual versions, to entire branches, or to entire elements. Attributes are often used to represent state information about a version, for purposes of integrating process-control mechanisms with the version control system. For example, an attribute may be attached to a source file version in order to indicate the bug(s) fixed by that version.
- *Hyperlinks* enable users to define structured or ad hoc relationships between pairs of objects. For example, *foo.c* can point to *foo.doc* via a “design_for” hyperlink. Hyperlinks are useful for requirements tracing and, with a graphic display of the relationship network, navigating between related objects.

2 The Challenge of Geographically-Distributed Development

In a large development organization, software developers typically are located at several physical sites; each site develops one or more subcomponents of a large software system. Sites may be physically near each other and connected by a high-speed network, or they may be on different continents with poor network connectivity. Sites may have private sources but they also may need to share sources, libraries, and header files with other sites. (This is particularly true of header files and libraries, which act as component interfaces.)

Parallel development is more difficult in a geographically-distributed environment; time-zone differences, language barriers, and other problems complicate communication and coordination among team members. “Parallel” doesn’t always mean just two development paths; in large corporations development may be underway at three, four, five, or more sites simultaneously. Coordinating changes becomes more complex as the number of sites increase.

Companies typically take a snapshot of the sources at one “master” site and ship them to other remote sites. If changes are made at a remote site, they must be carefully merged back into the sources at the master site. The process is largely manual and error prone, especially if several sites are making changes to the same sources. With different sites in charge of different subcomponents, it may be difficult to know the current location of the “original” copy of a source.

A scheme to manage geographically-distributed development of ClearCase VOBs must handle meta-data (labels, attributes, and hyperlinks) as well as source file data.

3 Alternative Designs

We considered a number of alternative approaches to supporting geographically-distributed software development. This section describes several of these alternatives.

3.1 Global Access to a Centralized Repository

The simplest approach is to extend the tools used for local software development, providing all users at all sites with access to a centralized shared repository across a wide-area network. In fact, several ClearCase customers used this approach to permit, for example, developers working in California to access and update a repository located in Massachusetts. This approach has significant usability problems, however:

- The need to access the central repository frequently makes the system vulnerable to network problems such as partitions.
- Frequent accesses to the central repository have an unacceptable effect on ClearCase performance when they occur over a relatively low bandwidth wide-area network.
- Remote access to a central repository presents problems with scaling the system to very large numbers of users, since the load on the central server increases with the number of users in the entire network.

3.2 Locally-Caching File Systems

The problems of using a global repository might be alleviated by caching information locally at each development site, perhaps by making use of a caching remote file system such as AFS [5]. Unfortunately, such a solution does not go far enough in addressing the problems of sophisticated configuration management system. A ClearCase VOB includes both version data (stored in standard files) and *meta-data* (stored in a database). At best, a caching file system would allow a development site's version data to be cached locally; it does not (and cannot) help with the meta-data.

In fact, databases such as those embedded in ClearCase VOBs represent a worst case for this kind of caching file system: they are frequently written from multiple machines, and their pages are accessed randomly, thereby defeating both the caching and the pipelining of the file system. So, the introduction of a caching file system does not solve the problems of robustness, performance, and scalability caused by use of a central repository; at best, it can only reduce the file I/O load imposed by remote access to version data on the central server.

3.3 Repository Replication

These considerations led us to conclude that an adequate solution to the problem of distributed software development would have to be based upon replication of the entire repository, including both the database and the version data storage, to each local site. Replication carries with it the possibility that the sites may change their replicas independently, with the potential for conflicting changes and subsequent inconsistency of replicas.

The problem of replicating a version-data repository has much in common with the general problems of file and database replication [1,2]. In particular, the key problem to be solved is how to allow multiple replicas to be updated independently without losing changes or allowing inconsistent changes (that is, changes that violate data structure invariants).

3.4 Serially-Consistent Replicas

There are algorithms for updating replicated data at multiple sites, keeping all the replicas continuously synchronized and avoiding the possibility of lost or conflicting changes [3,4]. Replicas synchronized by such an algorithm are termed *serially consistent*. The serial consistency constraint, however, imposes a significant penalty on the availability of data in each replica: either reading or writing of data at any replica requires that at least a majority of all replicas be accessible. It is possible to trade off the number of sites that must be contacted when reading data against the number of sites that

must be contacted when writing data, but a majority of all the extant replicas must be available for either reading or writing operations to occur.¹

This majority-consensus requirement also means that the serially-consistent replication approach has even worse scaling characteristics than the approach using a central repository. With the central repository, both the load on the central server and the aggregate network traffic are proportional to the number of users. With serially-consistent replicas, the load on *each* replica is proportional to the number of users (since each user must contact at least half of the replicas for each read or write operation), while the aggregate network load increases as the square of the number of users.

3.5 Weakly-Consistent Replicas

Relaxing the requirement of serial consistency allows the contents of individual replicas to temporarily diverge, with no guarantee that a change made at one replica is immediately visible at the other replicas. The presumption is that eventually (perhaps on a periodic basis) the replicas will be resynchronized. A number of approaches have been taken to the problem of resolving inconsistencies that may be detected during the resynchronization, but none seemed directly applicable to the problem of distributed software development.

The Locus system, for example, demanded manual intervention upon detecting that conflicting changes had been made to a particular file.² This approach may be adequate in a system in which individual files are infrequently modified at multiple sites, and in which only a small number of replicas exist (so that the person who originated the changes is readily available to resolve such conflicts). But it is not suitable for the complex database of a software repository, which is modified continuously at all active development sites. With a large number of replicas and a complex pattern of replica updates, it is easy to imagine that a conflict may be detected at a “third-party” replica, far from those at which the conflicting changes were originally made, and not readily able to resolve the conflict. So, a system that depends on manual intervention at replica synchronization time to resolve conflicting changes to the repository is not acceptable.

An alternative approach to weakly-consistent replication of a database is that taken by Grapevine. In Grapevine, each modification made to a particular data item in the database is assigned a modification time, and modification times are totally ordered (if necessary, by including the replica identifier as part of the modification time to break ties). When two potentially conflicting changes to the same data item are detected, the most recent change (the change with the later modification time) wins. This rule ensures that all replicas will eventually reach the same state, but allows some changes to be lost without notification – adequate for a mailing-list registration database such as Grapevine, but not for a software configuration management system.

-
1. Gifford's weighted-voting approach actually allows the implementor to trade off network load against availability, by weighting some sites more heavily than others. A majority of *votes* is still required for either reading or writing, however.
 2. The description of Locus' support for replication in this paper is necessarily incomplete. Locus supported serial consistency of replicas within a single network *partition*; only when two previously-disconnected partitions reconnected was manual intervention as described here required.

4 The MultiSite Solution

To make the weakly-consistent-replicas approach usable for distributed SCM, it is necessary to partition the objects being replicated into disjoint sets, each of which can only be modified at a single replica. The branch and merge development model employed by ClearCase provides a natural way to partition the development activities occurring at multiple replicas in way that is practical and does not unduly restrict the utility of the system, allowing development work to proceed in parallel at each replica while still avoiding conflicting changes.

With MultiSite, geographically distributed development is structured in the same way as parallel development at a single ClearCase site: different development projects proceed concurrently and independently, each project using a different branch of an element's version tree. The only difference between local and multiple-site parallel development is that MultiSite *enforces* the rule that different sites work on different branches, by assigning mastership to individual branches. This ensures *automatic resynchronization*, eliminating the need for manual intervention to resolve conflicts. Only one site can extend a particular branch; thus, all those changes can be trivially grafted onto the corresponding branch at other sites. When a remote effort is complete, or at a convenient integration point, it can be merged into an integration branch; integration of parallel work, testing, and release, can happen at any site.

For example, a multinational company has groups developing compilers in different parts of the world (Figure 3). All the compilers use a common code generator, and all the groups want to share and modify the same code generator sources.

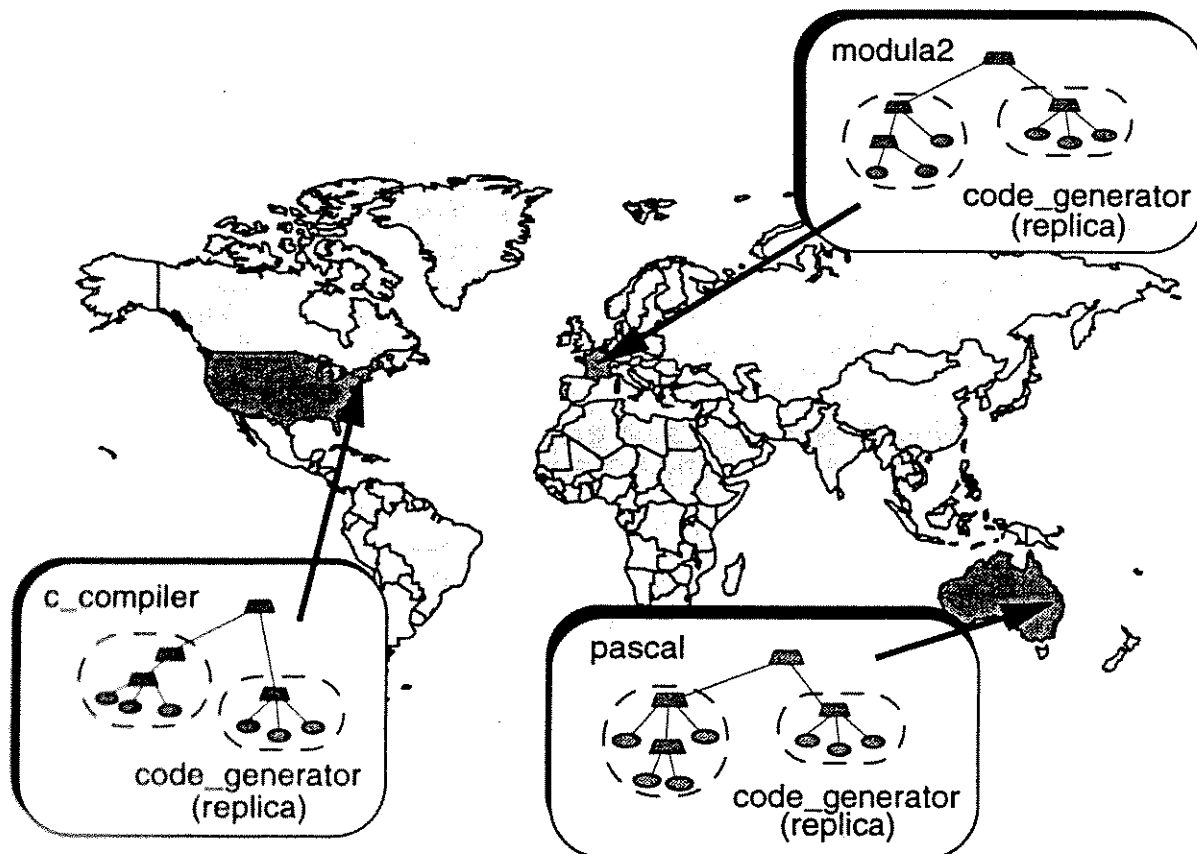


Fig. 3. Multinational Project Sharing Sources

Each site has a copy (*replica*) of the VOB that contains the code generator source files. The Pascal group in Australia makes changes to the source files on a "pascal" branch in

its local copy; similarly, the Modula2 group in France makes its changes to the same source files on a “modula2” branch in its local copy (Figure 4).

To support this strategy for geographically distributed development, MultiSite provides these services:

- Maintenance of a local *replica* of a VOB at each site. Each site can “see” the entire VOB through its local replica (i.e. all versions of all files are present).
- Enforcement of the rule that different sites work on different branches of an element.
- Synchronization of the multiple VOB replicas, communicating the changes among the sites via network connections or magnetic tape.

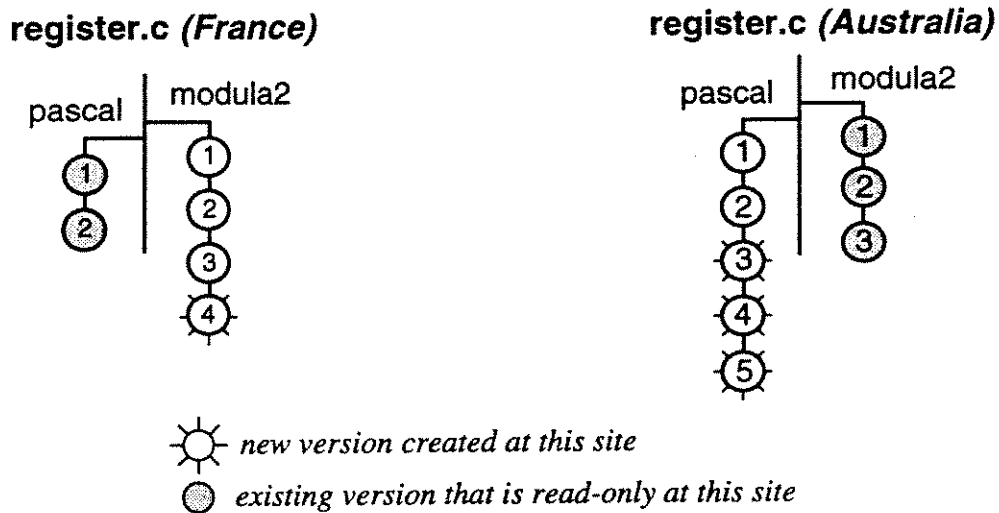


Fig. 4. Replicas Evolve Independently

Segregation of changes onto different branches makes the synchronization procedure completely automatic; since only one site can extend a particular branch, all the changes on that branch can be trivially grafted onto the corresponding branch at other sites (Figure 5).

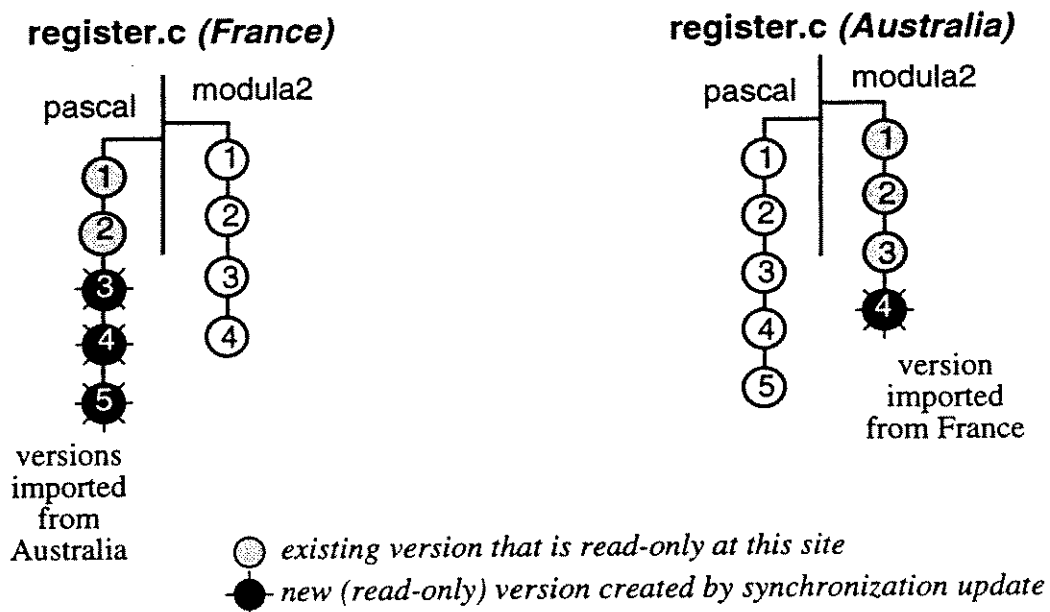


Fig. 5. After the next Periodic Update

After synchronization, independent changes to the same source file can be merged together, using the standard ClearCase merge tools. (Typically, merges are deferred to meet relatively infrequent “integration” milestones, even if synchronization takes place frequently.)

4.1 VOB Replicas

For each ClearCase VOB, MultiSite can maintain any number of *VOB replicas*, distributed at different sites. The original VOB and its replicas form a *VOB family*. All of the replicas are peers — any one can be modified, any one can spawn a new replica, any one can be deleted when no longer needed, and any one can send updates to any other.

The replicas in a VOB family are loosely consistent. Local updates to replicas (new versions checked in, files renamed, labels and attributes added, etc.) make their contents diverge; synchronization makes their contents converge again.

The VOB is the MultiSite “unit of replication” — users cannot restrict replication to a particular element or directory tree.

4.2 Branch Mastership

Enforcement of MultiSite’s branch-development strategy is accomplished by assigning *mastership* to individual branches — new versions can be created on a branch only in the replica that masters that branch.

Branch mastership provides the right level of granularity for parallel development. Coarser-grained mastership, such as at the element (file) level, precludes parallel development because people at two sites can’t work on the same file at the same time. Without any mastership, if any site can change anything, automatic resynchronization is not possible.

The concept of mastership extends to other objects, such as symbolic version labels. In order to prevent conflicting operations from being performed at different replicas, MultiSite assigns each object a mastering replica that is the only replica permitted to modify the object. For example, each element in a VOB is assigned a mastering replica, only that replica is permitted to delete the element or change its type. Mastership can be transferred between replicas if needed.

4.3 Synchronization

Synchronization updates circulate changes among the replicas in a VOB family. The replica update topology can be a star, a multi-hop chain, or any graph that enables updates to eventually flow from one replica to all others.

Sites that are not connected by TCP/IP can use a file-transport mechanism (e.g. one based on electronic mail) or magnetic tape. All updates are idempotent, so import of an update can be restarted or repeated without difficulty.

5 Implementation of MultiSite

Synchronization of VOB replicas in MultiSite is implemented using a mechanism similar to the multiple-part timestamp schemes found in other replication systems [6,7,8]. As changes are made to a replica, a record of each change is stored as an entry in an *operations log* in the VOB database. A replica exports its changes to other replicas by

generating a *synchronization packet*. This packet (file) contains all of the operations log entries made in the replica since the last generated synchronization packet. This includes changes that originated at the replica, as well as changes received from other replicas. The data is stored in XDR¹ format so it can be processed by any target architecture. At the destination replica or replicas, the mechanism *imports* the changes contained in the packet by replaying them in order. Any such changes previously seen (imported) by the replica are ignored.

MultiSite ensures that operations are imported and performed in a consistent order. The dependencies between operations originating at the same and/or at different replicas form a partial order that reflects the potential flow of information between the operations [9]. Any operation performed on a VOB replica may depend on any preceding operation performed on that replica, including both operations that originated at the replica as well as operations imported from other replicas. The MultiSite synchronization mechanism is designed to ensure that no operation is imported into a replica before any of the operations on which it potentially depends (i.e. the mechanism guarantees that the state of each replica reflects a consistent cut in the dependency graph of operations).

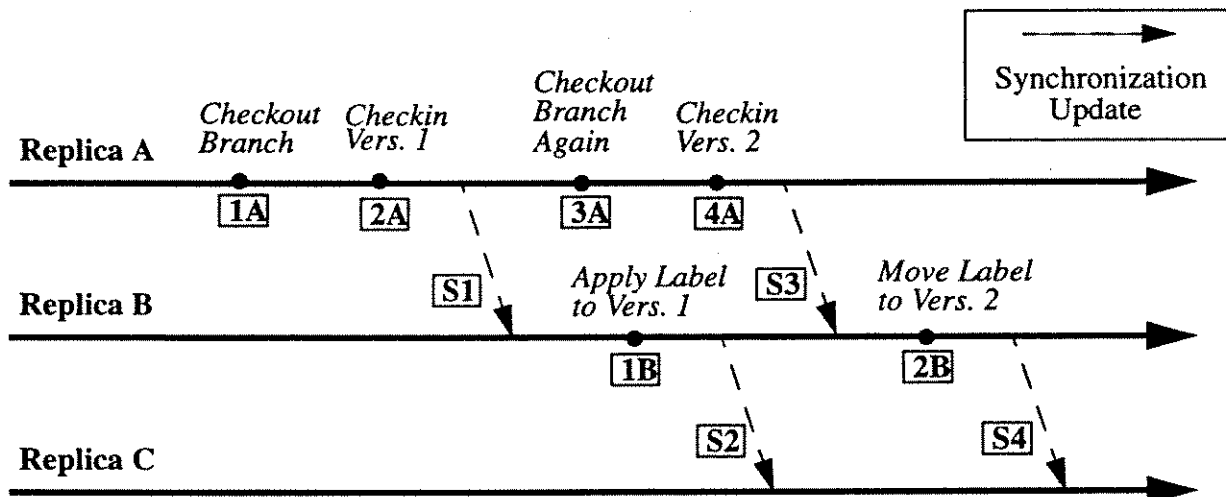


Fig. 6. Operation Dependencies

In Figure 6, creating a symbolic label (e.g. "Beta-Release") on version 1 of the file at replica B clearly requires that version 1 has already migrated to replica B. In other words, the label creation operation (1B) depends on the first checkout/checkin operation pair (1A/2A) made by replica A. These operations were later imported by replica B, creating version 1 (synchronization S1).

The label moving operation (2B) at replica B moves the symbolic label from version 1 of the file to version 2. It depends upon the earlier label application (1B) at that replica, as well as on the second checkout/checkin pair of operations (3A/4A) made at replica A. MultiSite guarantees that all replicas will import these operations in an order consistent with their dependencies. A replica might import both checkout/checkin pairs before either labelling operation, or each labelling operation might immediately be imported after the corresponding checkout/checkin pair. Either order of operations is consistent with the operations' dependencies. However, no replica will import either of the labelling operations without previously having imported the corresponding checkout/

1. eXternal Data Representation (used by NFS and other tools)

checkin pair. And, no replica will import the label movement without previously having imported the original label application.

5.1 Logging and Virtual Timestamps

Each VOB replica records in its operation log every operation performed on that replica, including both operations that originated at the replica as well as operations imported from other replicas. The operation log for a replica is maintained as part of the VOB database itself, and operations are logged immediately as part of the transaction of the operation. As a result, the log always reflects exactly the set of operations performed on the replica, and the order in which the operations were performed.

In order to track the potential dependencies between operations, each operation logged by MultiSite is tagged with both the identity of the VOB replica at which the operation originated, and a virtual timestamp reflecting the order of the operation with respect to others that originated at the same replica. The virtual timestamp is implemented as a counter of the number of operations that originated at the replica, and is termed an “epoch number”.¹

Each replica also attempts to track the state of each other replica, using a table of virtual timestamps maintained in the VOB replica’s database. The table contains one row and one column for each replica in the VOB family. The virtual timestamp in row A of column B of the table reflects the last known operation originating from replica B that was imported by replica A. Each row of the table therefore represents a multiple-part timestamp (i.e. a cut on the operations dependency graph) reflecting the last known state of the corresponding replica. This is not necessarily the actual current state of that replica, but a conservative estimate of it.

Figure 7, based on Figure 6, shows the state of replica B’s table immediately after the first update generated to replica C (S2). Note that replica B is not aware of the second pair of checkout/checkin operations (3A/4A) being performed concurrently at replica A. It is only aware of the operations sent to it by replica A in the earlier update (1A/2A), as well as its own operations.

		FROM		
		A	B	C
TO	A	2	0	0
	B	2	1	0
	C	2	1	0

Fig. 7. Table of Virtual Timestamps at Replica B

Note that a replica’s table also contains a row for the replica itself. This row represents the highest numbered operations performed or imported at the replica, and is always kept

1. In the original MultiSite design, groups of operations sent to other replicas in a single update were assigned the same virtual timestamp and were considered to constitute an “epoch” of changes. Their order within the operations log determined the potential dependencies between them. The implementation of the product was later simplified by assigning each operation a unique virtual timestamp. However, the term “epoch” has remained with the product.

up-to-date with the actual state of the replica (i.e. it always reflects the VOB replica's actual state).

5.2 Generating Updates

In order to generate an update from one replica to another, MultiSite scans the log of the sending replica looking for operations that are not known to have already been imported by the destination replica. It does this by scanning for entries with timestamps (epoch numbers) higher than those reflected in the table row for the destination replica. A record of each such operation found is XDR encoded and recorded in the update packet, along with the identity of the replica at which the operation originated and the operation's virtual timestamp. Operations are recorded in the packet in the same order in which they occur in the log. Note that because the destination replica's row in the table reflects a conservative estimate of its actual state, the update packet may contain operations already performed or imported by the destination replica. These are discarded later by the destination replica when importing the packet.

In the earlier example, the second update generated from replica B to replica C would contain the second checkout/checkin pair of operations followed by the label movement operation. These are the only operations in the log of replica B with virtual timestamps larger than the corresponding entries in its table row for replica C.

MultiSite takes an optimistic approach to maintaining virtual timestamp tables. Immediately after generating an update packet, the sending replica increments the entries in the table row for the destination replica in order to reflect the operations sent in the packet. No acknowledgment of receipt is required from the destination replica. The next update generated from the replica will pick up where the last update left off, without duplicating any of the operations contained in the previous update. Although this is optimal during normal operation, it requires special actions to detect and handle lost updates when they occur. This issue is discussed in more detail below.

Each update packet contains the virtual timestamp table row (for the destination replica) that was used to determine which operations to include in the packet. This row represents the starting state of the packet (i.e. the set of operations already expected to be possessed by the destination replica before it imports the operations contained in the update packet). This row is also useful for determining the order in which the destination replica should process update packets. This issue is discussed further in the following section.

Each update packet also contains the sending replica's virtual timestamp table row from its own table. This allows the destination to track the actual state of the sending replica without substantial overhead.

5.3 Importing Updates

Before allowing a replica to import the operations contained in an update packet, MultiSite first checks to determine if doing so would create an inconsistency in the importing replica. MultiSite compares the "starting state" virtual timestamp row contained in the packet to the importing replica's own table row for itself. If any entry in the packet row is larger than the corresponding entry in the replica's table row, then the importing replica is missing operations that the sending replica expected the receiver to have already imported. These may be operations contained in other packets that have not yet been imported, or they may be operations contained in a packet that was lost before

being imported. In either case, importing the packet could create an inconsistency, and its processing is deferred until the missing operations are imported.

Note that because successive packets from the sending replica will use successively larger “starting state” virtual timestamp rows (i.e. the rows will contain larger entries on a component-by-component basis), the destination replica can determine the order in which the packets were created by the sending replica and the order in which they should be processed.

In the earlier example, if the first update from replica B to replica C were lost, then this situation would be detected by replica C when it attempted to import the second update from replica B. The “starting state” timestamp row in the second packet would be the last row from the table on page 10, and would indicate that replica C was expected to have already imported the first two operations originated from replica A, and the first operation originated from replica B.

If the importing replica has already performed or imported all of the operations identified by the “starting state” row in the packet, then the operations contained in the packet are also imported into it. All operations are imported in the order in which they appear in the packet, and the importing replica’s own table row is updated to reflect each operation as it is imported. If the packet contains an operation previously imported into the replica from some other packet (i.e. an operation with a virtual timestamp smaller than the appropriate component of the importing replica’s table row), then the operation is ignored. Note that this implies that the importation of update packets is idempotent. If the receiving host fails part way through the importation of a packet, then it is safe to restart the importation from the beginning once the host has recovered.

In proof that this mechanism preserves consistency of VOB replicas, observe that if one replica imports an operation from another replica, then the importing replica must previously have performed or imported every operation preceding the imported operation in the sending replica’s log. This follows because each such operation must either have originated at the importing replica (and is therefore already possessed by that replica), or be contained in the importing packet at some location preceding the operation being imported, or be tagged with a virtual timestamp that is no larger than the corresponding component of the importing packet’s “starting state” timestamp row. In the last case, the fact that the packet is being imported implies that all such operations must previously have been imported from some other update packet. By induction, if the sending replica’s log reflects a consistent ordering of operations, then the importing replica’s log (and VOB state) will also be consistent.

It follows as a corollary that if a replica imports an operation that originated at some replica, then the importing replica must previously have imported *all* other operations with lower virtual timestamps that also originated at the same replica. This follows because each such operation with a lower virtual timestamp is a potential dependent of the operation with a higher virtual timestamp.

5.4 Purging Operations Logs

In order to prevent operation logs from growing without bound, entries must eventually be purged from the logs. MultiSite uses an age-based mechanism to decide when an entry should be removed from a log. By default, an entry is deleted after it has been in a log for 180 days (this value can be configured by the site administrator to reflect their

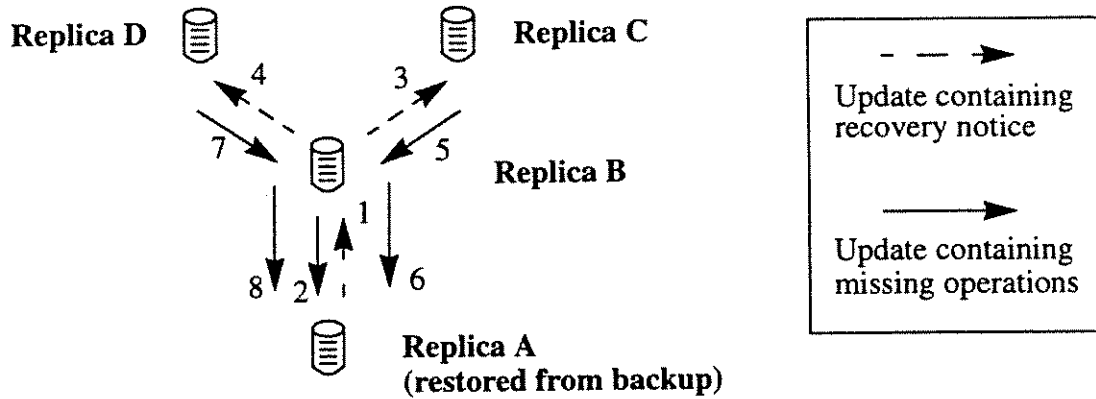


Fig. 8. Recovery from Backup

5.6 Recovery Details

When a replica is restored from backup tape, it is immediately locked to prevent users from making any new changes that would generate operations log entries. A special entry is added to the operations log to indicate that the replica was restored from backup tape, and the restored replica's virtual timestamp table row is added to that entry to indicate its restored state. The entry is assigned a special virtual timestamp (the highest possible timestamp) and treated specially by the synchronization mechanism. Note that the special entry cannot be assigned a normal virtual timestamp because any such timestamp may conflict with one previously used by the restored replica before the failure. The normal process of updating and importing packets then resumes (although the importing of most packets will be deferred by the restored replica due to the "starting state" check).

When other replicas import the special operations log entry, they reset their virtual timestamp table row for the restored replica to the value indicated by the entry. The importing replica also adds the special entry to its operations log so that it can be propagated to other replicas that may not be being directly updated by the restored replica. The importing replica also adds a special acknowledgment entry to its operations log to indicate that it has seen the entry from the restored replica.

When the restored replica imports an acknowledgment entry from every other replica, the VOB replica is unlocked and new operations are permitted to be performed on it.

In proof that this algorithm restores all of the missing operations to the restored replica, consider the operation possessed by any replica with the highest virtual timestamp that originated from the restored replica. The restored replica is required to import an acknowledgment from this replica before unlocking the restored replica. Such an acknowledgment is only created by the other replica in response to importing the special entry from the restored replica. The acknowledgment is added to the end of the log (i.e. after the latest operation from the restored replica), and is therefore considered by MultiSite to be potentially dependent on the last operation. According to the earlier results, the acknowledgment cannot be imported by any replica unless the replica has already imported the last operation from the restored replica. According to the corollary, the last operation from the restored replica cannot be imported unless all operations (that originated at the restored replica) with lower virtual timestamps have also been imported.

update pattern and rate). No check is performed to determine if the entry might still need to be sent to another replica.

More sophisticated algorithms exist for determining when an entry can safely be deleted from a log (i.e. when it will never need to be sent to another replica) [10]. However, such algorithms are based on knowledge of the replica backups that are performed. Such information is not available to MultiSite because VOB replica databases are just regular files that are backed-up as part of the normal filesystem. As a result, MultiSite cannot use these algorithms.

MultiSite *does* detect when needed entries have been purged from a log, and prevents potentially inconsistent updates from being made. Because successive operations originating at a replica are assigned virtual timestamps that differ by one, MultiSite can detect operation gaps between the last known state of a replica, and the actual operations written to an update packet for that replica. MultiSite verifies that the operations written to an update packet have virtual timestamps beginning with values exactly one larger than the timestamps in the table row for the destination replica. If this is not the case, then the destination replica must import the missing changes from some other replica or the replica must be recreated from a more up-to-date replica.

5.5 Recovery from Backup

When a VOB replica is restored from backup tape, it is not possible to immediately permit users to access the replica without risking the consistency of the VOB family. Other replicas in the VOB family may have imported operations that originated at the restored replica before its failure, but that were made after the backup (and not therefore recovered by the restored replica).¹ Any new operations originated at the restored replica therefore risk reusing the same virtual timestamps as those used by other operations already imported by other replicas, thereby breaking the synchronization mechanism.

MultiSite takes a roll-forward approach to dealing with this problem by requiring the restored replica to re-import any operations it originated (and that are possessed by other replicas) before permitting users to perform new operations on the replica. The recovery algorithm piggybacks on the normal synchronization mechanism and is similar in principle to an echo algorithm [11]. The basic idea behind the recovery algorithm is to cause all replicas to reset their virtual timestamp table row for the restored replica to reflect the restored state of that replica. Subsequent updates generated from the replicas will therefore carry any missing operations — that is, any operations with higher virtual timestamps — to the restored replica (Figure 8). Once all of these operations are imported by the restored replica, new operations can safely be performed without reusing any virtual timestamps already possessed by VOB family members.

Actually, it is only necessary that the restored replica import a recovery update from the last replica that it updated prior to the failure. Because updates always consist of an entire suffix of the sender's log, this other replica is guaranteed to have imported all of the desired operations that have been imported by any other replica. Unfortunately, there is not any way for MultiSite to determine from the restored VOB replica the identity of this last replica updated prior to the failure, so recovery updates are required to be imported (either directly or indirectly) from all other replicas.

1. In this sense, VOB replicas act as incremental backups of one another.

5.7 Re-recoveries

A real timestamp (clock time) is also added to the special log entry created by the restored replica to handle the case where the restored replica is re-restored from backup tape before completing its recovery. It is only necessary for the other replicas to acknowledge the latest recovery phase of the restored replica. The other replicas therefore only reset their virtual timestamp tables in response to a special log entry with a real timestamp later than any previously seen.

6 A MultiSite Usage Example

This section attempts to give a fairly realistic example of how MultiSite might be used to support distributed software development in a moderate-size organization. In this example, an organization does its primary development of a software product at a plant in Evanston, Illinois in the USA; it is just about to start work on a new release, V3.0. The new release will include functional enhancements, bug fixes, and will also be ported to a new workstation and localized to the Japanese language. The company decides that the porting work should be done by a subsidiary in Paris, France, and that the localization should be done by a subcontractor in Osaka, Japan.

The company uses a common ClearCase software development methodology:

- Individual subprojects, and often individual developers, work on separate subbranches.
- The *main* branch is reserved for integration of subprojects. To prepare for an internal baselevel or an external release, engineers first merge selected development subbranches back into the *main* branch.
- The ClearCase “views” used by developers ordinarily isolate them from changes occurring on the integration mainline by selecting versions from the most recent baselevel. When necessary, developers “merge out” changes from the main branch to their subbranches, in order to bring themselves up to date with changes occurring on the integration mainline.
- Periodically, a baselevel of the product is built on the main branch, is tested, and is made available for internal use. The versions used in the baselevel are labelled, both for ease of identification and so that developers can instruct their “views” to select those versions.

With MultiSite, the organization can continue to use this accustomed methodology. The Evanston office will be the master of the *main* branch — the only site allowed to make modifications on the integration branch. A porting integration branch, named *paris_port*, will be mastered at the Paris office, as will any additional subbranches of *paris_port* that may be needed to organize the work there. And, a localization integration branch, named *osaka_locale*, will be mastered at the Osaka subcontractor’s office, as well as any subbranches of *osaka_locale* that may be needed there. The Evanston office will periodically merge changes from the *paris_port* branch back into the *main* branch, to keep the two sites from diverging; however, localization changes from Osaka will only be merged back into the main branch after V3.0 of the product ships, because they are expected to be too disruptive.

To support the porting and localization efforts, the project management decides that both the Paris and Osaka offices need to see changes from Evanston on a daily basis; and, for

convenience, that it is useful for the Osaka and Evanston offices to see Paris' changes on a daily basis also. However, they decide that it is only useful for Osaka to update Evanston once a month, and that Osaka never needs to update Paris directly (MultiSite automatically ensures that Paris will receive Osaka's updates indirectly via Evanston). A wide-area network with the topology shown in Figure 9 is already in place among the three sites.

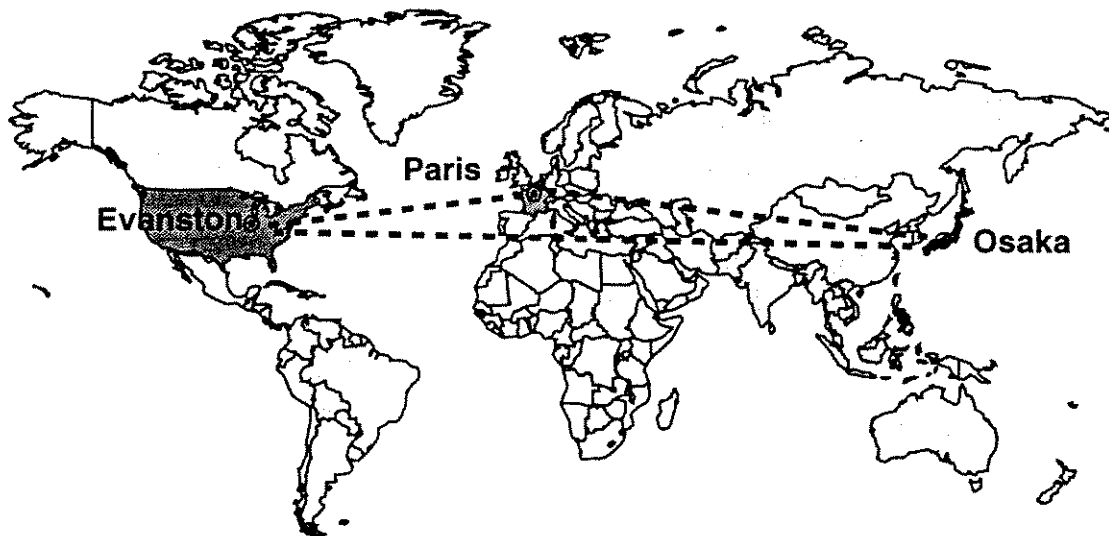


Fig. 9. Parallel Development in a Wide-Area Network

Figure 10 summarizes the direction and frequency of updates among the three replicas:

from/to	Evanston	Paris	Osaka
Evanston	—	Daily	Daily
Paris	Daily	—	Daily
Osaka	Monthly	Never	—

Fig. 10. Replica Update Frequency

6.1 Replica Creation and Configuration

Assume that the VOB to be replicated originally resides in Evanston. Then, the initial replica setup is a two-step process:

1. The administrator at Evanston runs a MultiSite command which dumps (*exports*) the contents of the VOB, in a machine-independent format, into one or more *data packets* for transmission to the remote sites. During this export step only, the VOB must be locked against changes. The administrator then arranges for these packets to be sent, via network or magnetic tape, to both Paris and Osaka.
2. The administrators at Paris and Osaka execute a MultiSite command to *import* the replica-creation packets. Once the import commands are complete, the VOB replicas are available for use by developers at the Paris and Osaka sites.

Once the replicas have been created, the administrators at each site must configure the periodic updates among the sites. For controlling periodic operations, MultiSite uses the UNIX *cron* facility. So, this configuration consists of creating entries in *cron*'s scheduler table to:

- run the MultiSite commands that package up the changes to each replica's VOB
- transport the change packets to the other replicas
- replay the changes received from the other replicas into the recipient's VOB

6.2 Concurrent Development

Development can now occur concurrently at all three sites. Developers at Evanston can continue working the way they always have. In Paris, project administrators can create the local integration branch type (*paris_port*). The Paris replica will master this branch type, having created it; thus, all development activity on branches of this type is restricted to the Paris replica. Similarly, Osaka can create the *osaka_locale* integration branch, mastered by the Osaka replica.

Each site can now make any number of changes to its respective branches, using normal ClearCase development methodologies: developers check out and check in files and directories, merge changes to and from integration branches, and build and test software using the versions selected by their views. Periodically, according to the update schedule configured using *cron*, the replicas automatically exchange update packets containing the sequence of changes made to each replica; these update packets are automatically applied to each replica in the proper order by MultiSite, allowing the developers at each site to see their colleagues' work from other sites.

6.3 Building a Release

The organization can integrate branch development and produce a release using almost the same procedure it used in the pre-MultiSite era. The only significant change is that the merges of the subprojects' branches are interspersed with synchronization updates where necessary. Rather than waiting for the periodic updates run by *cron*, at times it may be more convenient for the project administrators to manually force an update to be sent; this is easily accomplished by running the relevant MultiSite update commands by hand. A typical baselevel build procedure might include steps like these:

1. When development on the *main* branch reaches a stable point, Evanston labels the latest versions on the *main* branch, establishing a baselevel.
2. The *main* branch is locked, so that a consistent set of versions can be sent to Paris.
3. Evanston generates a synchronization update and sends it to Paris, to ensure that the *main* branch is up-to-date at Paris. The commands are the same as those used in the periodic update procedure set up using *cron*.
4. Paris developers merge changes on the *main* branch out to the *paris_port* branch. They build and test the application, then *checkin* a stable set of versions on the *paris_port* branch. As in Step #1, another baselevel is established by attaching labels to this set of versions. Then, the *paris_port* branch is locked in advance of generating an update.
5. Paris sends a synchronization update to Evanston, propagating the revisions to the *paris_port* branches.

6. Evanston developers merge the *paris_port* changes back into the *main* branch. (This should be a trivial merge.) Then, they build, test, and wrap up the release, labeling its versions appropriately.

This merge strategy puts the potentially-difficult work of resolving merge conflicts in the hands of the Paris developers, who know more about their own changes.

6.4 Using Two Replicas of the Same VOB at One Site

As the organization grows, the load on a single VOB replica may become too large to be handled by a single server machine. One possibility might be to upgrade the hardware, but MultiSite offers another alternative: the possibility of splitting the load by creating another replica of the VOB at the same site, and moving some of the developers over to the newly-created replica. In the example described here, management might decide to split the Evanston replica into two — one for new development and bug fixing, and the other for build-intensive release-time integration work.

This task is easily accomplished, again by using the MultiSite command to create a new replica, dumping the existing replica's contents and then importing the replica-creation packets to form the new replica. Knowledge of the new replica's existence is automatically propagated by MultiSite to the remaining replicas. Then, the administrators will want to reconsider the patterns and frequency of updates among the replicas. For replicas at a single site connected by a high-bandwidth local area network, it may be appropriate to exchange updates more frequently — perhaps three or four times per day. This is practical because each update only includes the *changes* made since the previous update, so that adding extra updates only imposes a small additional fixed cost per update (of computing the changes to be sent, and of setting up and tearing down the network connection).

7 Experiences from Actual Usage

As a way of measuring the impact of MultiSite on our usage of ClearCase, we examined the main VOB in use by our development organization. It contains the main source for ClearCase, comprising approximately 190Mb of source files and 470Mb of database. After five months of use of MultiSite on our VOB, we accumulated approximately 13Mb of log information data. This is the main space overhead associated with the presence of MultiSite. When compared with the total size of the database, this is a small percentage overhead (2.7%).

Currently, we're using two replicas for development. Another measure of activity is the amount of data shipped from replica to replica as part of day-to-day operation. At the time of writing, there is approximately 400-1200Kb of data shipped from each replica to the other. The day-to-day amount varies substantially, depending on details of work in progress. For example, on weekends there is usually no data to transfer. The time to read out log information and generate a packet for shipment is short, taking 1-2 minutes of elapsed time and 10-20 seconds of CPU time. The time to replay log information against a receiving replica is also usually 1-2 minutes elapsed with 10-40 seconds of CPU time. Occasionally the elapsed time is longer, due to the fact that some operations require proportionally more updating of the database. For example, one particular transfer took

17 minutes elapsed; this was due to the fact that version labelling on replay has not yet been optimized (as it has been done for ClearCase's original "make label" operation).

8 Future Directions

A new way to use MultiSite is for online backup. ClearCase requires that a VOB be locked for the duration of a backup, so that the database and filesystem objects are captured consistently. After creating a second replica in a local area network, it is possible to perform backups against the second replica, thereby avoiding locking the VOB. This is increasingly important as the developer work and nightly rebuilding combine to increase the busy time of a VOB to around-the-clock, making it inconvenient to lock VOBs.

Another possible use of MultiSite would be for software distribution. New software would be made available by adding it to a replicated VOB. As part of the normal process of updating other replicas, those replicas would receive the new software, which could then be used at the other locations.

One area where MultiSite could be improved would be the addition of facilities to allow portions of VOB information to be eliminated from the information being exchanged between replicas. One variation would be to mark elements as being private to a particular replica. This would have the effect of disabling operation logging for all operations performed on the element. Another variation would be "filtering", whereby a filter would be defined that would limit replication, probably by specifying an inclusive list of objects to be replicated. A number of problems appear when adding filtering. For example, what happens when only partial information is propagated and later the filter is widened, allowing previously excluded information? If portions of a version tree are propagated, what rules need to be added to deal with operations on non-propagated versions? What operations need to work even when complete information is not present? For these reasons, the area of filtering will require further investigation.

9 Conclusions

This paper has described an extension to the ClearCase software configuration management system that supports geographically-distributed development by replication of Versioned Object Bases. A combination of loosely-synchronized replicas with fine-grained object mastership prevents conflicting changes from occurring at different replicas. This enables automatic synchronization, with no need for manual intervention to resolve conflicts.

Our experience so far has validated our design choices. The MultiSite product is currently in use by about 1800 developers at 35 sites around the world, with remarkably few problems reported to date.

One of MultiSite's biggest design strengths is its unobtrusiveness. In most respects, developers in a distributed organization see little or no change in their development environment or development policies when MultiSite is introduced. Even for administrators, MultiSite imposes few additional administrative tasks, once the initial setup and configuration of replicas have been completed.

Software configuration management demands planning; geographically-distributed software development requires even more careful planning on the part of project leaders and development teams. ClearCase and MultiSite provide a framework for organizing

the software development process, and a rich set of tools for enforcing and monitoring it across many sites. They cannot, however, eliminate the need for a development policy that spans the organization.

10 References

1. B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. "The LOCUS Distributed Operating System" *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* ACM, October 1983.
2. A. Birrell, R. Levin, R. Needham, and M. Schroeder. "Grapevine: An Exercise in Distributed Computing" *Communications of the ACM* vol. 25, No. 4 (April 1982).
3. D. Gifford. "Weighted Voting for Replicated Data" *Proceedings of the Seventh Symposium on Operating Systems Principles* ACM, December 1979.
4. J. Gray. "Notes on Database Operating Systems" *Lecture Notes in Computer Science* vol. 60, Springer Verlag, Berlin, 1978
5. J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System" *ACM Transactions on Computer Systems* 6(1):51-81, February 1988
6. B. Liskov and R. Ladin "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection" *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing* (August 1986).
7. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. "Providing High Availability Using Lazy Replication" *ACM Transactions on Computer Systems* vol. 10, no. 4 (November 1992).
8. K. Birman, A. Schiper, and P. Stephenson. "Lightweight Causal and Atomic Group Multicast" *ACM Transactions on Computer Systems*, vol. 9, no. 3 (August 1991).
9. L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System" *Communications of the ACM*, vol. 21, no. 7 (July 1978).
10. R. Strom and S. Yemini. "Optimistic Recovery in Distributed Systems" *ACM Transactions on Computer Systems*, vol. 3, no. 3 (August 1985).
11. E. Chang. "Echo Algorithms: Depth Parallel Operations on General Graphs" *IEEE Transactions on Software Engineering*, vol. 8, no. 4 (July 1982).