From the Proceedings of the 18th Inter. Conf. on Software Engineering, Berlin, Germany, March 25–29, 1996

A Generic, Peer-to-Peer Repository for Distributed Configuration Management

André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{andre,dennis,alw}@cs.colorado.edu

Abstract

Distributed configuration management is intended to support the activities of projects that span multiple sites. NUCM is a testbed that we are developing to help us explore the issues of distributed configuration management. NUCM separates configuration management repositories (i.e., the stores for versions of artifacts) from configuration management policies (i.e., the procedures by which the versions are manipulated) by providing a generic model of a distributed repository and an associated programmatic interface. This paper describes the model and the interface, presents an initial repository distribution mechanism, and sketches how NUCM can be used to implement two, rather different, configuration management policies, namely check-in/check-out and change sets.

1 Introduction

A significant segment of today's software industry is moving toward a model of project organization that involves the use of multiple engineers at multiple sites working on a single software system or set of highly interdependent software systems. In the extreme case, multiple companies in multiple countries form temporary alliances, sometimes called *virtual corporations* [6], for the purpose of producing a specific product. And while these companies might be collaborators on one product, simultaneously they may be competitors on another.

In such a setting, configuration management (CM) becomes a serious challenge, and this challenge exhibits itself at several levels. At the lowest levels, there is the issue of distributing large amounts of data in a timely fashion over great distances. At the highest levels, there is the issue of integrating the asynchronous efforts of engineers who may be adhering to different CM procedures and practices. These converge in the middle levels, where lie the issues of providing distributed data management that is specialized to the needs of configuration management in a context that can assume no more than a decentralized federation of cautiously cooperating parties.

The work discussed here begins to address some of these issues. In particular, we are developing a testbed to help explore the middle levels of the problem. The testbed, called NUCM (Network-Unified Configuration Management), embodies an architecture that separates CM *repositories*, which are the stores for versions of software artifacts[1] and information about those artifacts, from CM *policies*, which are the specific procedures for creating, evolving, and assembling versions of artifacts maintained in the repositories. To do this, NUCM defines both a generic model of a distributed CM repository and a programmatic interface for implementing, on top of the repository, specific CM policies, such as check-in/check-out and change sets.[2] Structured this way, NUCM allows experimentation, not only with the model and the interface, but also with new CM policies and distribution mechanisms.

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

[1] We use the term *artifact* to refer to anything that is visible to a CM system as a versionable object. This includes configurations themselves.

[2] Feiler [9] provides a survey of some basic CM policies.

This paper presents our experiences to date in designing, building, and using NUCM. We begin in Section 2 by discussing related work in the area of distributed CM. In Section 3 we describe NUCM's generic repository model and in Section 4 we present the programmatic interface through which the artifacts stored in a NUCM repository can be manipulated by a CM system. Section 5 briefly discusses our initial distribution mechanism, which is based on the CORBA distributed object programming standard [7]. We show in Section 6 how two, rather different, CM policies can be implemented using NUCM and conclude in Section 7 with a brief look at future work.

2 Related Work in Distributed CM

Distribution is a relatively new feature in CM systems. In fact, many of the most widely known commercial and research systems, such as ADC [22], CCC/Harvest [21], NSE [10], EPOS [15], and ShapeTools [14] do not yet provide any real support for distribution. Those that do, appear to suffer from one or more of the following significant problems.

1. *Distribution is grafted onto an existing, non-distributed architecture.* Typically, the CM system is augmented with a simple client/server interface. Such solutions, while straightforward to implement, often exhibit scaling problems, such as a performance bottleneck at the server repository.
2. *Users of the CM system must be aware of the distribution.* In fact, users are typically given primary responsibility for keeping artifacts consistent across sites.
3. *The CM repository and CM policies are tightly coupled.* This inhibits flexibility both in how artifacts are distributed and in what CM policies can be employed.

Below we describe the distribution aspects of several prominent and representative CM systems, illustrating these problems in more detail.

Distributed RCS Distributed RCS (DRCS) [17] is an extension of the popular RCS system [24]. All artifacts, including the individual versions of a file, the version tree, and the descriptive file attributes, are stored in a central repository. Distribution is achieved by establishing a client/server relationship among remote RCS clients and the central repository. Thus, distribution can be hidden from users, since they use the standard RCS interface; this interface is simply reimplemented to work as a remote client that consults the repository each time it performs a CM operation on an artifact. The major drawback to using DRCS is that there is only one server, which can result in a heavy load at the server site, as well as a dependence on the reachability of the site. Clearly, DRCS is best suited for use in a centralized local-area network, not in a setting of wide-area distribution.

Distributed CVS Distributed CVS (DCVS) [12] is an extension of the CVS system [3], a variant of RCS designed to better handle configurations of whole systems. Similar to DRCS, DCVS employs the notion of a central repository to which remote CVS clients connect. As opposed to transporting files, DCVS transports entire configurations to a remote user workspace. A user can then make changes to the artifacts in the workspace. To commit the changes, the configuration is sent back to the central repository. Once there, the modified artifacts are merged with other changes that may have been made concurrently in other user workspaces. This approach suffers from the same performance drawbacks as DRCS, but to a worse degree because of the heavier amount of traffic implicit in transporting configurations. Again, DCVS is best suited for use in a local-area network.

Adele Adele [8] has been enhanced for distribution through a tool called Mistral [11]. Mistral helps a user manage the replication of an Adele database. Using Mistral, an Adele user at one site assembles a database delta, which compactly represents the changes made to artifacts at that site, and ships the delta to users at other sites. A user that receives a delta is responsible for integrating the delta into the database at their site (again using Mistral). Clearly, transporting deltas, rather than artifacts, can significantly reduce communication overhead. But, the Adele/Mistral solution still has problems in serving as a solution to distributed CM. First, users are responsible for assembling, shipping, and merging database deltas to keep replicas synchronized. Without strict procedures, this can quickly lead to inconsistent databases. Secondly, the task of merging artifact relationships and attributes is an error-prone activity. In Mistral, a simple heuristic is employed: add whatever does not exist. This is not always the best choice in a CM setting, and users must be aware of this (implicit) merging behavior.

ClearCase MultiSite The ClearCase system [2] has recently been extended with MultiSite [1], an optional product for distributed CM. Rather than having a single, central repository, MultiSite replicates the repository at remote sites. The replicas are instrumented in such a way that development at each site is performed on a different branch of a global version tree. To represent development at other sites, each site has branches in its repository containing the versions of the artifacts at the other sites. Periodically, updates made at one site are propagated to the surrogate branches at the other sites. Thus, at any given point in time, an individual site will have multiple versions of the same artifact, but only one of those versions can be modified at that site. Therefore, unlike DRCS and DCVS, MultiSite is not restricted to a single administrative site, which makes it better suited for use in a wide-area setting. Nevertheless, there is a conceptual cost to the user, which is the forced creation of extraneous branches in the version tree to represent multiple sites. These branches, which have more to do with project structure than configuration management, must eventually be merged by the users themselves into a single baseline version. This can be a serious burden, especially if the attributes and relationships on the artifacts have changed as well.

Endevor's Workspaces Endevor/WSX [5] is a CM system centered around the notion of user workspaces. A workspace is "spun off" from another workspace, and from then on provides the owner of the child workspace with a private copy of the artifacts in the parent workspace. A reconcile mechanism is used to merge the changes from a child workspace into a parent workspace. This copy/reconcile mechanism can be used to build a hierarchy of distributed workspaces, where each site has its own parent workspace from which local developers inherit their workspaces. One site will have to act as the main site to which all changes eventually have to be reconciled. This approach only works if the various sites operate on separate parts of a product. If that is not the case, then getting changes from one site to another requires a fair number of reconciles and copies, which is not desirable. In addition, the fact that there must be one main workspace can be an organizational, as well as a performance, bottleneck.

Continuus/CM A hybrid of the ClearCase MultiSite and Endevor Workspace approaches to distributed CM is followed in Continuus/CM [4]. Repositories are replicated across sites, and a central repository acts as the main repository to which all changes eventually must be committed. Thus, the repositories act as high-level "workspaces" for the various sites. This approach suffers from the familiar problems of cumbersome merging and performance bottlenecks.

3 A Generic CM Repository Model

The generic repository model used in NUCM consists of three parts: a storage model, an access model, and a distribution model. The storage model defines the primitive versioning and grouping mechanisms, the access model defines how access to stored artifacts is obtained, and the distribution model defines the mechanism for managing how artifacts are arranged among separate sites. In this section we describe each of the models in detail.

3.1 Storage Model

Using NUCM, one can store and version *artifacts*. Artifacts are either *atoms* or *collections*. An atom is a monolithic entity that has no substructure visible to NUCM. Typical atoms include a source file or a section of a document. Collections are used to form a group of individual versions of atoms and can themselves be versioned. For example, a collection might be a program consisting of source files or a document consisting of sections. An atom can be shared among multiple collections. Collections can be used recursively; they can be part of larger, higher-level collections. For example, a collection for a release could consist of both a collection for a program and a collection for a document. Of course, it is not sensible for collections to be mutually recursive, so the containment structure of collections forms a directed, acyclic graph. A repository consists of one or more *top-level* collections. Top-level collections serve as the entry points to the contents of a repository. Thus, a top-level collection is never contained in another collection.

Figure 1 presents an example to illustrate these basic concepts. The figure shows a portion of a repository for the C source code of a hypothetical software system. Collections are shown as ovals, atoms as rectangles, and containment relationships as arrows. Two top-level collections are stored in the repository, one for a collection called `windows` and another for a collection called `text`. We can see that both top-level collections contain separate collections called `source` and a shared collection called `includes`. Both the collections `source` and `includes` simply contain atoms, which in this example are files. Versioning is depicted using shading: the darker the shade, the older the version. Dashed lines show containment relationships for older versions of collections. For example, the older version

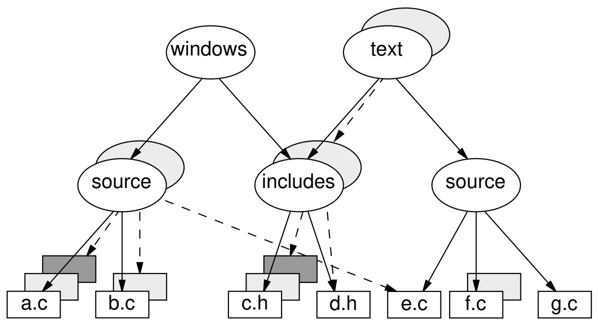

Figure 1: Example Repository Contents.

of collection includes contains an older version of atom c.h, whereas the newer version contains the newer version of atom c.h. Both the older and newer versions contain the same version of atom d.h.

The storage model does not impose any relationship among the various versions of an artifact. In particular, NUCM does not enforce the tree structure of versions found in many—but by no means all—CM systems. NUCM simply provides a unique name with which to identify each version (see Section 4.2). This allows a CM system built using NUCM to enforce its own style of relationship among versions, and hence increases the generality of the repository. For example, as shown in Section 6, CM systems that use a version tree and CM systems that use change sets can both be built using NUCM.

3.2 Access Model

Since CM systems are most often used to store artifacts related to operating system files, it is desirable to provide access to the artifacts through the native file system, rather than, say, through a database relation. This unobtrusive behavior allows existing tools and applications to continue to work without having to make any modifications to them. This is important since, first, few users have access to the source code of the tools and applications they use, and second, few users have the desire to make changes to existing software to adapt to a new CM system. Thus, we use the file system as the basis for NUCM's access model.

Access to artifacts stored in a NUCM repository is handled through what we call *views*. A view represents a particular version of a single top-level collection, and is materialized as a directory in the file system. Lower-level collections are represented as subdirectories, while atoms are represented as files. For example, the new version of the collection text shown in Figure 1 looks like the following directory structure when materialized in a view.

```
.../text/includes/c.h
                 /d.h
        /source/e.c
               /f.c
               /g.c
```

Notice that the use of the file system also provides a hierarchical, rather than global, naming scheme.

In order to maintain control over the materialized artifacts in a view, views remain within the purview of the repository, not within the file space of a user. User access to a view is accomplished by linking a user directory to the directory where the view resides. In addition to providing user access, this setup avoids the limitation of having one top-level collection per view, since multiple views can be part of a single user directory. For example, if one wants to have both the collection windows and the collection text present in one directory, one requests two views and supplies a directory into which the respective version of each collection will be materialized.

Note that the "user" of the view mechanism will typically *not* be a human. More likely, a CM system will manipulate the artifacts in the view to provide its own style of access. Such a CM system might very well preprocess the artifacts in a view before presenting them to a user. It could even create its own user directory and copy the artifacts to that directory while using the view to communicate with NUCM. In another setting, a client CM system could provide a specialized browser or editor, hiding the details of the NUCM view from the user altogether. For example, the CM model described by Lin and Reiss [13] could use NUCM in such a way that its software units map to atoms, its aggregate modules are expressed using collections, and its specialized browser is used to present the contents of views.

3.3 Distribution Model

NUCM provides the concepts of both *physical* and *logical* repositories. A physical repository is the actual store for some set of artifacts at one particular site. A logical repository is a group of one or more (physical and logical) repositories presented to its clients as a single repository. In contrast to the artifacts in a physical repository, the actual location of artifacts in a logical repository is irrelevant. Using the logical repository, CM systems are able to obtain artifacts from any physical repository, whether that repository resides on a local disk, on the local network, or on the other side of the world.

This functionality is obtained in a *peer-to-peer* fashion; there is no global, centralized "master" repository controlling the distribution of artifacts. Instead, each local repository is responsible for maintaining a mapping mechanism through which requests for views of top-level collections not stored in the local repository are mapped to a remote repository. Thus, repositories act as both clients and servers, requesting services from each other and fulfilling service requests for each other. Moreover, the mapping is local, so that new repositories can be easily added and removed where and when relevant. This means that repositories can be joined into federations of cooperating agents, whose degree of coupling can be flexibly determined by the policies of the CM systems involved.

Although any number of top-level collections can reside within a given physical repository, a given top-level collection and its constituent artifacts reside within a single physical repository. This is not a serious limitation, since top-level collections can be arbitrary large or small. Hence, a CM system can arrange the distribution of artifacts among sites in a wide variety of ways. For example, at one end of the spectrum, a directory that is presented to a user by a CM system could correspond to a single top-level collection stored in one physical repository. At the other end of the spectrum, the directory could be formed by the CM system from many smaller collections stored in several geographically distributed repositories.

Access to top-level collections that reside at remote repositories is transparent to the user. All the functionality provided by a NUCM repository for local use is available in the remote case without any difference in behavior. The transparent behavior is attained through the mapping mechanism; each repository has associated with it a map that relates names of top-level collections not found in the local repository to names of top-level collections residing in remote repositories. A NUCM repository can automatically fetch the remote artifacts in case they are requested by a client, without that client having to be involved in the distribution in any way.

Clearly, some responsibility is still left to the user; the user needs to maintain the map. This is not a burden, but rather it is an important feature of the distribution model. Having a mapping mechanism, instead of a fixed distribution mechanism, allows NUCM's distribution model to be used in a variety of ways. In a sense, having a mapping mechanism is analogous to a CM client system directing the NUCM actors in a distributed play.

NUCM's peer-to-peer architecture allows for an extremely flexible distribution mechanism. For example, one could have a single NUCM repository and many CM client systems, as in the case of DRCS. Or, one could provide fault tolerance by having two NUCM repositories and one CM client, where the client uses two views, one from each repository, to replicate every versioning action it performs. Furthermore, one could have NUCM repositories mimic workspaces as in Endevor/WSX, or use NUCM repositories to provide a setup similar to ClearCase MultiSite. These and other approaches to distributed CM can easily be built using NUCM's peer-to-peer architecture.

4 Repository Interface

As mentioned in Section 1, NUCM is intended to provide an interface to a distributed repository that generically supports a variety of CM policies. The functions described in this section form the programmatic interface offered to CM system implementors. They do not impose any particular CM policy, but rather they provide the mechanisms for client CM systems to implement specific policies. Therefore, while the interface functions might seem odd in their semantics from the perspective of a human user, those same semantics are invaluable to a CM system implementor.

The interface functions fall into three basic categories: access, versioning, and locking. In none of these functions is there any mention of distribution or physical location. This is because NUCM hides distribution, taking care to invisibly fetch artifacts from remote repositories as necessary. A fourth category of interface functions, separate from the other three, is used to manipulate the mappings of top-level collections to remote repositories. Due to space limitations, we discuss only the first three categories here.

4.1 Access Functions

Access to a NUCM repository can be obtained by requesting a view on a particular version of a top-level collection. Once access has been granted, versioning and locking functions become available for the client CM system to use on the materialized artifacts in the view. When a client CM system is finished processing, it closes the view, and access to the artifacts is removed. The access functions in the interface of NUCM are OpenView and CloseView.

The function OpenView provides access to a particular version of a top-level collection. If access is granted, OpenView creates a new view directory and materializes the artifacts that are part of the requested version of the top-level collection in that view directory. The view directory is created in the repository, and access to it is established by linking the supplied user directory to the view directory. OpenView returns a *view identifier* to allow multiple clients to work within the same view, as well as to distinguish among multiple views that can be open at the same time.

In order to create a new top-level collection in NUCM, the client CM system must supply a new top-level collection name and an initial version. NUCM then creates an empty top-level collection, and proceeds as if a view is opened on that collection.

The function CloseView removes access to a previously requested top-level collection. It removes the artifacts in the view directory, and removes the view directory and the link from the user directory to the view directory.

4.2 Versioning Functions

Once a view has been opened on a top-level collection, the following versioning functions become available for the materialized artifacts in the view: InitiateChange, AbortChange, and CommitChange.

InitiateChange informs NUCM of a client's *intention* to make a change to an atom or a collection. In response, NUCM preserves the old version of the artifact and gives the client permission to change the artifact in the view. Note that in NUCM, versioning and locking are orthogonal, and therefore InitiateChange does not lock the artifact (see Section 4.3).

The function AbortChange abandons an intended change to an artifact. It reverts the materialized state of the artifact back to the state that it was in before InitiateChange was invoked. An AbortChange performed on a collection can only succeed if no artifacts that are part of the collection are currently in a state that allows them to be changed. This forces the client CM system to either commit any changes or to abandon them. In this way, unintentional loss of changes is avoided.

The function CommitChange commits the changes to an artifact, storing the new version of the artifact in a uniquely named place in the repository and removing the client's permission to change the artifact in the view. Again, versioning and locking are orthogonal, so CommitChange does not release any lock that may be held on the artifact.

In designing the versioning functions, we were faced with the following issue: Does the act of creating a new version of an artifact implicitly create a new version of the collection in which that artifact resides? Clearly, situations arise in which the answer is yes, and situations arise in which the answer is no. Both answers must therefore be supported. But from a pragmatic standpoint, if versions of collections are created as often as versions of the artifacts within them, then there would be a cumbersome proliferation of versions of collections. Thus, as the default behavior, NUCM does not create new versions of collections. Under this default behavior, a new version of an artifact effectively replaces the old version within the collection. Although "replaced", the old version can still be accessed through an explicit request for that version. To obtain a new version of a collection, an explicit action must be taken by the client CM system. In particular, a new version is created only if InitiateChange is invoked on the collection; the new version becomes available upon invocation of CommitChange.

To illustrate the versioning functions, suppose we have a repository containing the artifacts depicted in Figure 2a. Assume further that we have opened a view on the top-level collection source. If we then create a new atom c.c and invoke CommitChange on that artifact, the repository will look as shown in Figure 2b. Notice that the repository still contains only one version of collection source, since we did not invoke InitiateChange on that collection; the atom c.c has simply been added to the current version of source.

Next, suppose we invoke InitiateChange on atom b.c, change it, and commit the change. After the invocation of CommitChange, the repository will look as shown in Figure 2c. The new version of atom b.c has replaced the old version within the collection. The old version of b.c can be accessed by an explicit invocation of InitiateChange on that version.[3]

Suppose we then invoke InitiateChange on collection source. This changes the behavior of CommitChange for artifacts contained in `source`; changes are cached until CommitChange is invoked on the collection. For example, if we invoke InitiateChange on atom a.c, change it, and then commit the change, the repository will contain two versions of atom a.c. The old version of a.c remains linked to the single version of the collection source in the repository, as shown in Figure 2d. If we then invoke CommitChange on source, two versions of the collection will be present in the repository, with the newer version containing the new version of a.c and the older version containing the old version of a.c, as shown in Figure 2e.

4.3 Locking Functions

As stated above, in order to allow both a lock-based CM policy like RCS [24] and a merge-based CM pol-

[3] This current mechanism for accessing an older (or newer) version forces the access to be cast in terms of a change to an artifact. We are currently designing an interface function to explicitly exchange versions within a collection.

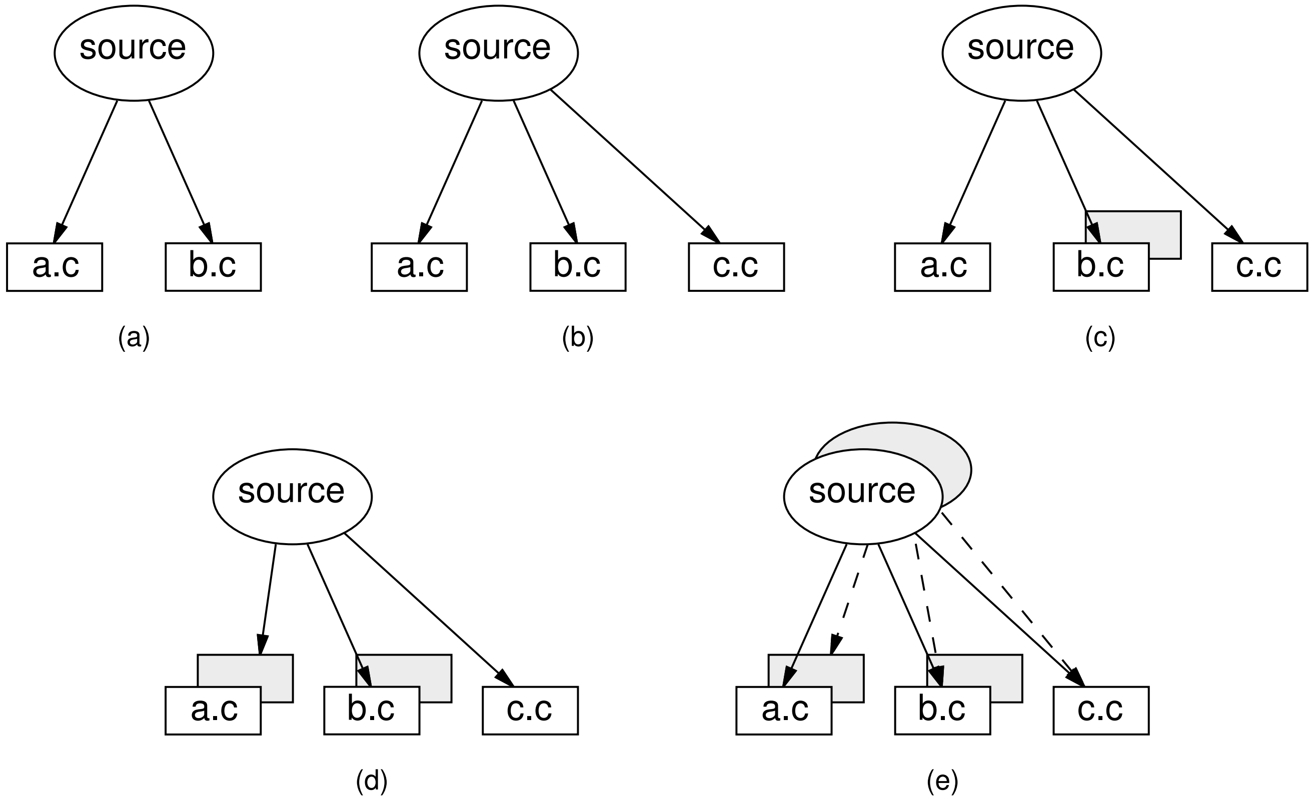

Figure 2: Progressive States of an Example Collection.

icy like CVS [3] to be built on top of NUCM, locking is provided separately from versioning. The following locking functions are provided in NUCM: Lock, LockVersion, Unlock, and UnlockVersion.

The semantics of the locking functions are straightforward. The function Lock locks an artifact, while the function LockVersion locks a particular version of an artifact. If a lock cannot be obtained because it is already held, the functions do not block but return immediately. The functions Unlock and UnlockVersion have the expected behavior; an artifact is unlocked, or a particular version of an artifact is unlocked.

A lock on a collection will not cause a request for a lock on an artifact within that collection to fail—that is, locks run one level deep. It is the responsibility of the client CM system to attach semantics to locks on a collection, choosing to use it as a lock on the collection only, or as a lock on the collection and its contents.

Functions Lock and LockVersion are orthogonal; a lock on a particular version of an artifact will not cause a lock on the artifact itself to fail, and vice versa. In addition, if a client has locked an artifact, other clients can still invoke InitiateChange on the artifact, or even modify the artifact and invoke CommitChange. This is because locking is not enforced by NUCM. Instead, a CM system uses the locking functions provided by NUCM to implement an access protocol.

5 An Initial Distribution Mechanism

The previous two sections describe, respectively, a model and an interface for a generic, distributed CM repository. The next question is, what is an appropriate way to realize the model and interface? There are some obvious candidates, including a distributed file system (e.g., Jade [19] or Prospero [16]), a distributed database [20], or an advanced operating system environment (e.g., PCTE [23]). One could even consider building something from scratch. Our approach was to experiment with a rather different alternative, namely the CORBA standard for distributed object programming [7], using the Orbeline CORBA engine [18].

The resulting architecture of our implementation of NUCM is shown in Figure 3. Within the oval, a group of NUCM access servers provides access to a group

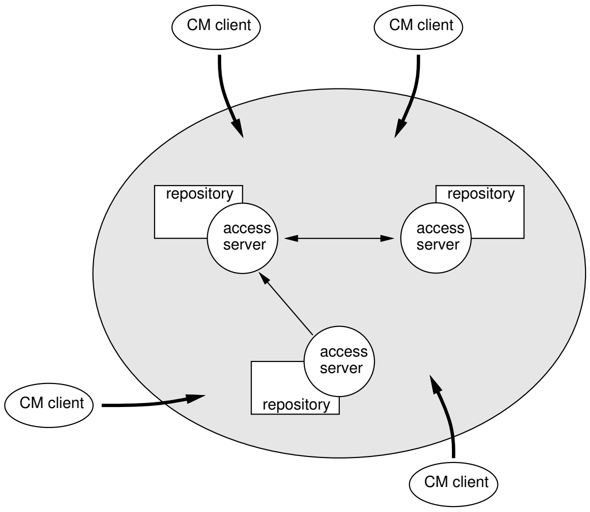

Figure 3: NUCM Distribution Architecture.

of repositories of artifacts. Each access server corresponds to one physical repository, whereas the whole collection of access servers and physical repositories behaves as a single logical repository to the outside CM client systems.

The interface presented in Section 4 has been implemented as a single CORBA object that acts as a repository access server. There are several advantages to this approach, especially in supporting distribution.

1. *The implementation of CORBA we used encompasses a name server.* This makes connecting to remote repository access servers extremely easy. Given a unique naming scheme among repository access servers and the name of a particular remote access server, Orbeline's CORBA implementation takes care of binding to that server.

2. *CORBA provides a remote procedure call mechanism.* Once bound to a remote repository access server, requests can be made in a fashion similar to a function call. No additional parameters are needed, and normal error handling can be employed. CORBA takes care of the transmission of the request and the return value(s) among servers.

3. *It is possible to provide smaller "handle" objects to other objects.* A repository access server can hand out a small supporting object to allow, for example, a remote repository access server to browse its contents, or to have a remote repository access server read the contents of files or directories at the local site.

Within the access server object, several smaller CORBA objects are used to perform certain tasks. For example, objects exist to represent a general attribute value list, to read and write a file or directory, and to navigate around the file system. In addition to the interface functions that are described in Section 4, the CORBA interface for a repository access server contains functions to hand out these objects to remote repositories. The ability to use these objects internally for local access, together with the ability to use a remote object in the same way as a local object, provided for much of the simplicity in implementing NUCM. In many cases, choosing the right objects and then processing a request would leave the code for the request unchanged, while supporting both local and remote access. Less than 10% of the total code written for NUCM is dedicated to handling distribution. Clearly, there is a big advantage in using a CORBA-based implementation.

6 Implementing Two CM Policies

Feiler [9] classifies CM systems into the following four broad categories: check-in/check-out, composition, long transaction, and change set. Each category represents a different CM policy. Thus, a good test of NUCM's flexibility and usability is to implement a CM system from each category. This section sketches the implementations of two CM systems, one using check-in/check-out and the other using change sets. These two categories in some sense represent extremes in existing CM policies. Therefore, although the two policies have not been integrated, these examples demonstrate the flexibility and appropriateness of the abstraction provided by the interface.

6.1 Check-in/Check-out

The check-in/check-out policy is a simple scheme in which versions are managed by locking files during changes. RCS [24] is an example of a CM system that uses check-in/check-out to maintain versions of files. For each file, RCS builds a version tree, which is used to capture revisions (temporal successors) and variants (alternative branches). The two most basic RCS programs are check-in and check-out.

```
ci [-l] [-r rev] filename
co [-l] [-r rev] filename
```

The optional argument "-l" is a request for a lock on a version of a file. A check-out without a lock is a request for a read-only copy of the file, while a check-in with a lock is interpreted as a check-pointing operation. The optional argument "-r" is a request for a specific revision of a file. If no revision number is given, then the latest is assumed.

Using NUCM, we have implemented a check-in/check-out CM policy that mimics the basic functionality of RCS. Two programs are provided, nci and nco, that correspond to the RCS programs ci and co. The two programs are implemented in terms of the NUCM functions InitiateChange and CommitChange, supplemented with facilities for maintaining version trees.

Because NUCM is used as the repository for version information, views are used to communicate between NUCM and the nci and nco programs. When a file is checked out using nco, InitiateChange is invoked and then the file is made available to the user by creating a symbolic link from the user's directory to the file in the view. If the user requests a lock on the file, nco additionally invokes Lock. Conversely, nci removes the symbolic link and then uses CommitChange to place the new version of the file into the repository. It also invokes Unlock, if necessary.

The version tree is kept in a separate NUCM artifact, hidden from the user, that is maintained in the repository along with the file to which it applies. This is necessary to allow multiple instances of nci and nco to be used on the same file; the version tree needs to be shared over time.

Our check-in/check-out CM policy implementation provides easy distribution of the versioned files, since it is built on top of NUCM. No special handling is needed to adapt the nci and nco programs to a distributed setting, since NUCM already provides distribution transparently to client systems.

6.2 Change Sets

As a change-set CM policy, we have implemented something similar to Aide de Camp [22], where first a base version of a configuration is put in NUCM, followed by sets of changes to the whole configuration as the system evolves. No versioning of individual files is performed, only versioning of whole configurations.[4] In theory, a user is able to mix and match change sets at will (assuming the user has verified the compatibility of the changes), building new versions of the software by setting specifications for applying change sets to the base version.

[4] Of course, one could create a new version of a configuration every time a single file has changed to keep track of changes to individual files, but this defeats the purpose of the change-set policy.

Using NUCM, this policy is easily implemented using two directories: NUCM's view directory and a private directory managed by the change-set implementation. Through the view directory, the change set implementation communicates with NUCM to obtain base versions and change sets. In the private directory, the change-set system first assembles the version of the top-level collection requested by applying change sets to the base configuration, and then allows the user to access the assembled version.

In our change-set approach, the base version of a configuration maps to the first version of a top-level collection, whereas each subsequent change set of the configuration maps to a subsequent version of the top-level collection. In this way, the first version contains complete files, whereas the later versions only contain deltas of files and files that have been newly added to the configuration.

As in the check-in/check-out policy implementation, the change-set policy implementation provides easy distribution. Local change-set clients can gain access to remote change sets using the name of the top-level collection, which is the name of the base configuration. NUCM will fetch the requested artifacts from a remote repository transparently.

7 Conclusion

For the past few years the field of configuration management has been in a consolidation phase, with the research results of the 1980s being transferred to the commercial products of the 1990s. New research directions are now beginning to emerge in the area [25], and the issues of supporting multiple engineers at multiple sites appears to be at the forefront.

We are using NUCM to begin an exploration of distributed configuration management. While the design and implementation are certainly in their early stages, they have already allowed us to experiment with a variety of interesting problems. These include:

- the use of CORBA in the construction of a distributed repository;
- the management of peer-to-peer communication and decentralized control among distributed agents;
- the integration of distributed data management with existing tools built for a local-area file system;
- the separation of CM mechanisms from CM policies; and

- the development of a set of primitives for configuration management.

We expect to be able to use NUCM as a vehicle for exploring other important problems in distributed configuration management. For example, we believe NUCM will be a suitable platform for investigating the problems of CM policy integration, not just interoperation.

Acknowledgments

We thank Jonathan Cook and John Doppke for their helpful suggestions on the design and implementation of NUCM.

REFERENCES

[1] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, 1995.

[2] Atria Software, Natick, Massachusetts. *ClearCase Product Summary*, 1994.

[3] B. Berliner. CVS II: Parallelizing Software Development. In *Proceedings of 1990 Winter USENIX Conference*, Washington, D.C., 1990.

[4] Continuus Software Corporation, Irvine, California. *Continuus Task Reference*, 1994.

[5] Legent Corporation. Endevor WSX Product Overview. Available on the world wide web at *http://www.sv.legent.com/Info/Ewsx/Ewsx.html.*

[6] W.H. Davidow and M.S. Malone. *The Virtual Corporation.* Harper Business, 1992.

[7] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft, Inc. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, Framingham, Massachusetts, December 1993. version 1.2.

[8] J. Estublier and R. Casallas. The Adele Configuration Manager. In W. Tichy, editor, *Software Configuration Management*, number 2 in Trends in Software, pages 99–134. Wiley, London, 1994.

[9] P.H. Feiler. Configuration Management Models in Commercial Environments. Technical Report SEI-91-TR-07, Software Engineering Institute, Pittsburgh, Pennsylvania, April 1991.

[10] P.H. Feiler and G. Downey. Transaction-Oriented Configuration Management: A Case Study. Technical Report CMU/SEI-90-TR-23, Software Engineering Institute, Pittsburgh, Pennsylvania, 1990.

[11] C. Gadonna. *MISTRAL User Manual V1.* LGI, May 1995. ESPRIT Project 5327, REBOOT.

[12] T. Hung and P.F. Kunz. UNIX Code Management and Distribution. Technical Report SLAC-PUB-5923, Stanford Linear Accelerator Center, Stanford, California, September 1992.

[13] Y.-J. Lin and S.P. Reiss. Configuration Management with Logical Structures. In *Proceedings of the 18th International Conference on Software Engineering.* Association for Computer Machinery, March 1996. To appear.

[14] A. Mahler and A. Lampen. An Integrated Toolset for Engineering Software Configurations. In *Proceedings of the ACM SOFSOFT/SIGPLAN Software Engineering Symposium on Practical Software Engineering Environments*, pages 191–200, Boston, Massachusetts, November 1988.

[15] B.P. Munch. *Versioning in a Software Engineering Database—the Change Oriented Way.* PhD thesis, DCST, NTH, Trondheim, Norway, August 1993.

[16] B.C. Neuman. The Prospero File System: A Global File System Based on the Virtual System Model. Usenix Association, File Systems Workshop.

[17] B. O'Donovan and J.B. Grimson. A Distributed Version Control System for Wide Area Networks. *Software Engineering Journal*, September 1990.

[18] PostModern Computing Technologies, Inc, Mountain View, California. *Orbeline User's Guide*, 1994.

[19] H. Rao and L.L. Peterson. Accessing Files in an Internet: The Jade File ystem. *IEEE Transactions on Computers*, 19(6):613–624, June 1993.

[20] J.B. Rothnie, P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A Landers, C. Reeve, D.W. Shipman, and E. Wong. Introduction to a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):1–17, March 1980.

[21] Softool Corp., Goleta, California. *CCC/Manager, Managing the Software Life Cycle across the Complete Enterprise*, 1994.

[22] Software Maintenance & Development Systems, Inc, Concord, Massachusetts. *Aide de Camp Product Overview*, September 1994.

[23] I. Thomas. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software*, pages 15–23, November 1989.

[24] W.F. Tichy. RCS, A System for Version Control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[25] A. van der Hoek, D. Heimbigner, and A.L. Wolf. Does Configuration Management Research have a Future? In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 305–309. Springer-Verlag, 1995.