A Reusable, Distributed Repository for Configuration Management Policy Programming

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

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

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

University of Colorado
Department of Computer Science
Technical Report CU-CS-864-98 September 1998

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 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 according to which the versions are manipulated) by providing a generic model of a distributed repository and an associated programmatic interface. Specific configuration management policies are programmed as unique extensions to the generic interface, but the underlying distributed repository is reused across different policies. In this paper, we describe the repository model and its interface, discuss their implementation in NUCM, and present how NUCM has been used to implement several, rather different, configuration management policies.

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

Organizations are increasingly involved in software development efforts that are distributed and decentralized in nature. Some organizations are naturally forced into these efforts because their employees are located in multiple geographical locations. Prime examples of such organizations are virtual corporations [11]. These corporations typically do not have a centralized workforce. Instead, employees use the Internet to carry out their day-to-day work.

Other organizations become involved in distributed and decentralized software development when the software to be produced is so large or complex that it requires a group of organizations to jointly solve the problem. In the extreme case, multiple companies in multiple countries form temporary alliances for the sole purpose of producing a specific product. While these companies might be collaborators on one product, simultaneously they may be competitors on another.

In these settings, 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 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 [14]. Structured this way, NUCM allows experimentation, not only with the model and the interface, but also with new CM policies and distribution mechanisms.

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 that are stored in a NUCM repository can be manipulated by a CM system. Section 5 briefly discusses the current implementation of NUCM. We show in Section 6 how we used NUCM to construct a number of CM systems, 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 commercial and research systems, such as CCC/Harvest [42], EPOS [33], JavaSafe [26], NSE [15], Sablime [4], ShapeTools [28], SourceSafe [30], TrueChange [43], and VOODOO [40], do not yet provide any real support for distribution. Those systems 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) [36] is an extension of the popular RCS system [48]. 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) [21] is an extension of the CVS system [5], 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 [13] has been enhanced for distribution through a tool called Mistral [19]. 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 the 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.

Gradient Similar to ClearCase, Gradient [3] is a CM system that is based on replication. But, unlike ClearCase, replication in Gradient is automatic and continuous. Each update that is made to an artifact is broadcasted instantly as a delta to all replicas. Because Gradient only allows incremental modifications to the artifacts it manages, and furthermore assumes that modifications are independent of each other, it permits simultaneous updates to a single artifact at multiple sites. All of the updates are added to each replicated archive, but only one of the updates is included in the latest version of the artifact. The other updates are simply stored as implicit branches. Unfortunately, users are not notified about such incidents, which can cause serious problems with seemingly lost updates. Distributed Source Control System (DSCS) [31] provides a notification mechanism to alleviate this problem, but the task of manually resolving conflicts remains. Moreover, both Gradient and DSCS operate on single files, and it is unclear how either one scales to handle coordinated changes to configurations of files.

WWCM WWCM [22] is one of many CM systems that use the World Wide Web to achieve distribution. An applet that is loaded into a Web browser implements the client interface of WWCM, while a specialized server maintains the repository with artifacts. Although platform independence, ubiquitous access, and ease of use are certainly important benefits of this type of solution, WWCM still does not scale to provide wide-area distribution. Similar to DRCS, the need to ship artifacts back and forth to a single server is a bottleneck that not only can result in a heavy load at the server site, but also creates a dependency on the reachability of the server site.

Other Systems A number of other systems exist that provide distributed capabilities via mechanisms that are similar to the ones employed by the systems listed above. Several of these systems are based on replication across sites. In particular, Continuus has been enhanced with a feature called DCM [9, 10], PVCS has an optional extension named SiteSync [24], and NeumaCM+ incorporates a feature called MultiSite [34]. All of these systems suffer from the same drawbacks as ClearCase MultiSite. Users are responsible for maintaining the distribution, having to explicitly synchronize repositories by merging updates from several sites into a new master copy and subsequently distributing the new master copy to all replicas. If the frequency with which repositories are synchronized is low, merging becomes a real problem, whereas if the frequency is high, the efforts of distributing new master copies and synchronizing the repositories becomes rather large. In either case, users are closely involved in the process of managing the distribution of artifacts among sites.

Besides WWCM, several other CM systems have adopted the World Wide Web as their medium for distribution. In particular, MKS Source Integrity [32] and StarTeam [45] have created Web interfaces to their respective CM systems. In addition, PVCS [25] and NeumaCM+ [34] have created Web interfaces that can be used as an alternative to their replicated repositories. All of these solutions suffer from the same scalability problem as WWCM; the need to ship artifacts back and forth to a centralized server creates a bottleneck.

Three other systems deserve mention. The first, PCMS [44], leverages its workspace synchronization routines to achieve distribution. In essence, this solution is similar to the solution provided by DCVS. The only difference is that PCMS allows for a multi-level hierarchy of distributed workspaces, whereas DCVS allows only one level. The problems remain the same though, since all changes eventually have to be committed to the main workspace. The second system, Perforce [37], employs a client-server distribution mechanism similar to DRCS. Although its communication mechanism has been optimized for wide-area networks through the use of compression and deltas, Perforce still suffers from the same problems as DRCS. Because a single server is used, a dependency is created on the server site—not only on its reachability, but also on its performance. The last system, Vesta [7], employs a hybrid solution. The same replication mechanism as employed by Gradient is used, but, to ensure consistency, certain actions always synchronize with a designated master repository. The advantage of the approach is twofold: no extra merging is required and the distributed version of Vesta behaves exactly like the centralized version. However, the system is dependent on the availability of the master site, which defeats part of the purpose of replication.

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 source files or sections of a document. Collections are used to group atoms into named sets. For example, a collection might represent a program that consists of a set of source files or a document that is composed of a number of sections. Collections can be used recursively; they can be part of larger, higher-level collections. For example, a collection that represents a system release could consist of a collection for the actual system components and a collection for the documentation of the system.

To be able to model shared artifacts, the containment structure of collections is not restricted to be a pure tree. Instead, both atoms and collections can be contained by multiple collections. It is, however, not sensible for collections to be mutually recursive. Therefore, the containment structure of collections forms a directed, acyclic graph.

Rather than employing a global naming scheme, the graph model used by NUCM naturally lends itself to the use of a hierarchical naming scheme for the artifacts that are managed. This provides a rather obvious advantage of scale. However, the use of a hierarchical naming scheme by itself is not sufficient. The ability of a single artifact to exist under different names is rather important in today's configuration management systems. Therefore, NUCM maintains the names

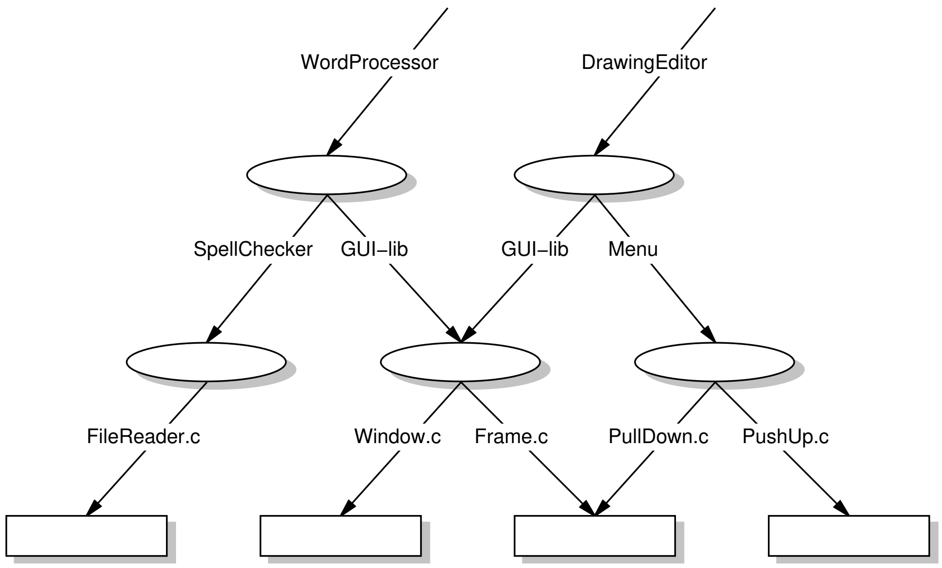

Figure 1: Example Repository Contents Without Versions.

of artifacts in their parent collections. Each collection can thus individually name the artifacts that it contains.

Figure 1 presents an example to illustrate these basic concepts. The figure shows a portion of a NUCM repository for the C source code of two hypothetical software systems, `WordProcessor` and `DrawingEditor`. Collections are shown as ovals, atoms as rectangles, and containment relationships as arrows. We can see that both systems not only contain separate subsystems, as represented by the collections `SpellChecker` and `Menu`, but also share a collection called `GUI-lib`. In turn, these lower-level collections simply contain atoms, which in this example represent the actual source files that implement both systems. It should be noted that one of the atoms is part of two collections, but is named differently in each one, namely `Frame.c` and `PullDown.c`, respectively.

Both atoms and collections can be versioned. However, contrary to other approaches, such as CME [22] and ScmEngine [8], NUCM 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 number with which to identify each version. 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. To build a CM system that explicitly recognizes a version tree with revisions and variants, for example, one would construct a mapping from the version numbers in the version tree to the linear version numbers provided by NUCM (the WebDAV example in Section 6 provides an illustration of such a mapping).

Figure 2 shows how the storage model presented in Figure 1 is enhanced with versions. Stacks of ovals and rectangles represent sets of versions of collections and atoms, respectively. Numbers indicate the relative age of versions: the lower the number, the older the version. Dashed arrows represent containment relationships of older versions. For example, both version 1 and version 2 of the collection `Menu` contain version 1 of atom `PushUp.c`, but both also contain a different atom altogether. In particular, version 1 of `Menu` contains version 2 of the atom `Window.c`, whereas version 2 of `Menu` contains version 2 of the atom `PullDown.c`.

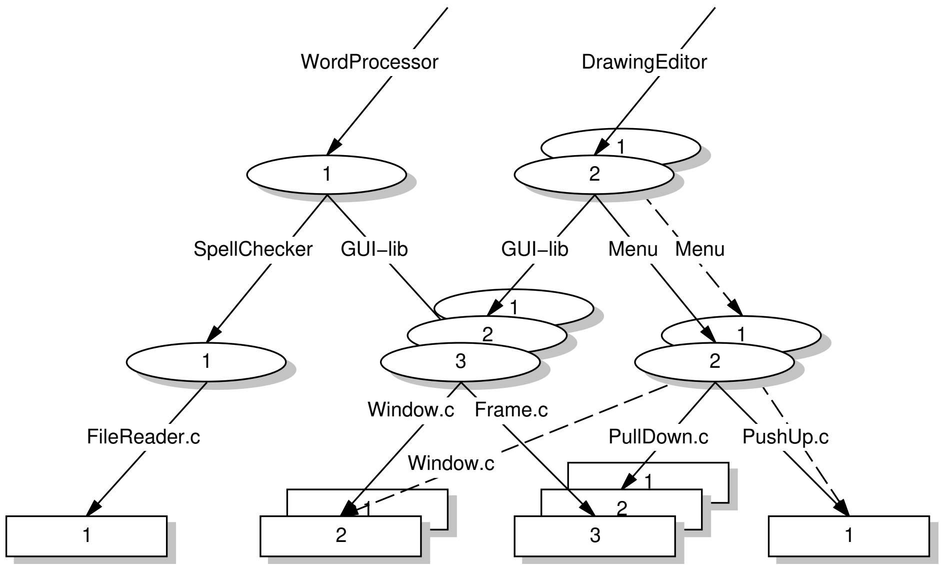

Figure 2: Example Repository Contents With Versions.

The example of Figure 2 illustrates another important aspect of the storage model. Certain versions of artifacts, such as version 1 of atom `PullDown.c`, are not contained at all. To still be able to identify these versions, we have enhanced the naming scheme that is used by NUCM with *version qualifiers*. For example, version 1 of atom `PullDown.c` can be uniquely identified with the following name (although we only use forward slashes in our examples, it should be noted that both forward and backward slashes can be used in artifact names).

DrawingEditor:2/Menu:2/PullDown.c:1

Such explicit use of version qualifiers in artifact names is not required by NUCM. By default, the contained version of an artifact is chosen if no version qualifier is specified. Therefore, version 1 of atom `PullDown.c` could also have been identified as follows.

DrawingEditor:2/Menu/PullDown.c:1

Because version 2 of the collection `DrawingEditor` contains version 2 of the collection `Menu`, no version qualifier is needed in this part of the artifact name.

3.2 Access Model

Configuration management systems are most often used to store artifacts that normally reside in a file system. Therefore, it is desirable to provide access to the artifacts that NUCM stores via the native file system, rather than, say, through a database relation. Such unobtrusive behavior allows existing tools and applications, such as for example Make [16], to continue to work without modification. 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 *workspaces*. A workspace represents a particular version of a 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, version 1 of the collection `WordProcessor` as shown in Figure 2 looks like the following directory structure when materialized in a workspace.

```
.../WordProcessor/SpellChecker/FileReader.c
                 /GUI-lib/Window.c
                         /Frame.c
```

It is important to note that this use of the file system preserves the hierarchical naming scheme as employed by the storage model; a one-to-one correspondence exists between collections and atoms on the one hand, and directories and files on the other.

A workspace is not necessarily restricted to contain just NUCM artifacts. The use of editors, compilers, and other tools that create auxiliary files is facilitated by allowing these transient files to coexist in a workspace. To distinguish between these transient files and the original artifacts, a workspace keeps track of the artifacts that are "owned" by NUCM. Therefore, the NUCM interface functions know at all times the exact nature of each file in a workspace.

Note that the "user" of a workspace does not need to be a human. More likely, a CM system will manipulate the artifacts in the workspace to provide its own style of access. This is illustrated in Figure 3. Three layers, each containing a different representation of the artifacts, can be identified. The bottom layer is the NUCM repository that contains all versions of all artifacts. Through usage of the NUCM interface functions some of these artifacts are materialized in a NUCM workspace. These materialized artifacts, in turn, might be transformed by a CM system to be presented to a user. This transformation could be a simple preprocessing of the artifacts in the workspace before presenting them to a user. Alternatively, a CM system could create its own user directory and copy the artifacts to that directory while using the workspace to communicate with NUCM. In another setting, a client CM system could provide a specialized browser or editor, hiding the details of the NUCM workspace from the user altogether. For example, the CM model described by Lin and Reiss [27] could use NUCM in such a way that its software units map to atoms, its aggregate modules are expressed as collections, and its specialized browser is used to present the contents of workspaces. As another example, the workspaces used by our WebDAV prototype (see Section 6) are never made visible to a user. Instead, the artifacts are transferred to a Web browser that uses a specialized interface to manipulate and version them.

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 a 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 that is part of the logical repository, whether the physical 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, the distribution is controlled at the individual artifact level. To this extent, collections not only maintain the names of the artifacts that they contain, they also track their physical locations. Based on this information, requests for artifacts that are stored at a different repository than that of the parent collection are forwarded.

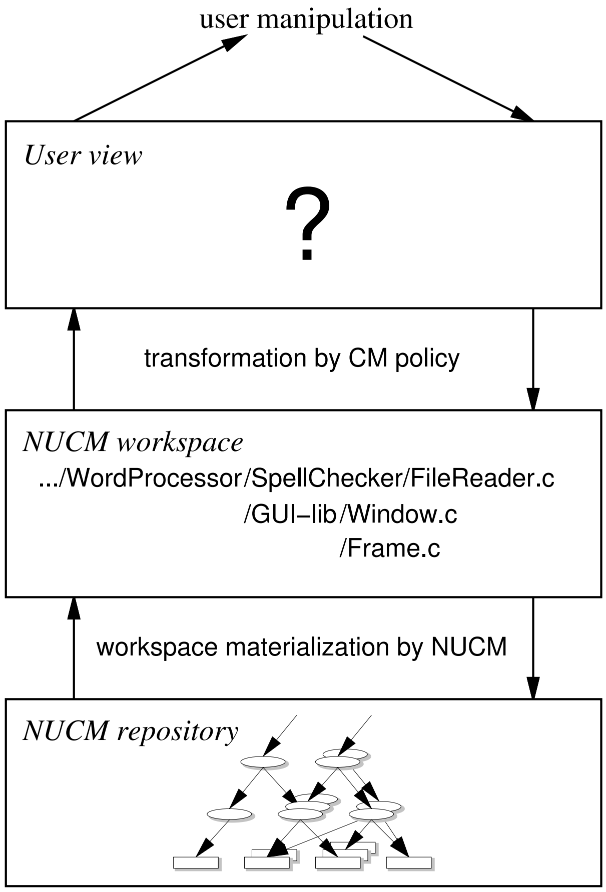

Figure 3: NUCM Access Model.

Thus, repositories act as both clients and servers, requesting services from each other and fulfilling service requests for each other.

Figure 4 presents an example distributed repository. Shown is the repository of Figure 2 as distributed over three different sites, namely Boulder, Milano, and Rotterdam. Each site maintains a physical repository with artifacts. The physical repository located in Boulder maintains the collection `WordProcessor`, the physical repository in Milano maintains the collection `DrawingEditor`, and the physical repository in Rotterdam maintains the collection `GUI-lib`. Because the projects in Boulder and Milano rely on the use of the collection `GUI-lib`, their physical repositories are connected with the physical repository in Rotterdam, thereby forming two logical repositories. In particular, the physical repositories in Boulder and Rotterdam combine into one logical repository to present a complete view of the `WordProcessor` collection and its constituent artifacts, and the combination of the physical repositories in Milano and Rotterdam creates a logical repository that manages the complete `DrawingEditor` system.

The example highlights an important aspect of the distribution model, which is its versatility. Artifacts can be distributed among physical repositories as desired and a single physical repository can be part of multiple logical repositories. Not shown, but supported by NUCM, is the fact that logical repositories, in turn, can be part of other logical repositories. Hence, a CM system built on

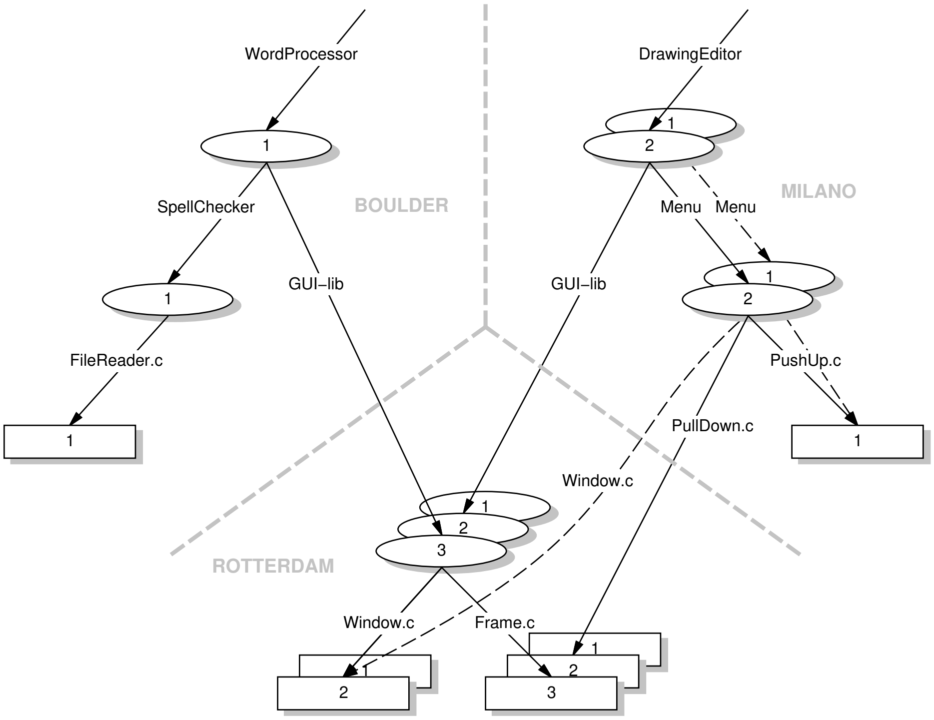

Figure 4: Example Repository Contents of Figure 2 as Distributed over Three Sites.

NUCM can arrange the distribution of artifacts among sites in a wide variety of ways. For example, at one end of the spectrum, the collections and atoms that make up a software system could all be located in a single physical repository. At the other end of the spectrum, the software system could be an assembly of artifacts that are stored in a multitude of geographically distributed repositories.

Access to artifacts that reside at remote repositories is transparent to a user. All the functionality provided by a NUCM repository for local use is available in the remote case. This transparent behavior is attained through collections that manage the physical locations of the artifacts that they contain. A NUCM repository uses these locations to automatically fetch the remote artifacts in case they are requested by a client, without that client having to be involved in managing the distribution. Of course, network delays and servers that are temporarily unreachable always remain visible to a user, but NUCM relieves a user from the responsibility of explicitly managing distribution.

NUCM's peer-to-peer architecture allows for an extremely flexible distribution mechanism that can be used to model many different distribution schemes. For example, one could use a single NUCM repository that is accessed by many CM client systems, as in the case of DRCS [36]. Or, one could construct a fault-tolerant CM system by using two NUCM repositories and one CM client, where the client uses two workspaces, one for each repository, to replicate every versioning action it performs. Furthermore, one could use NUCM repositories to mimic workspaces as in PCMS [44], or use NUCM repositories to provide a setup similar to ClearCase MultiSite [1]. These and other approaches to distributed CM can be built using NUCM's peer-to-peer architecture. Of course, a solution based on NUCM might not be as optimized as a specialized solution for a particular CM policy. However, the flexibility that NUCM provides allows experimentation with new distribution policies that, once settled, can be tailored and optimized for performance.

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 constitute that interface. The interface does not impose any particular CM policy, but rather it provides 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.

Table 1 presents the functions that make up the NUCM programmatic interface. The functions are grouped into seven basic categories. An important characteristic of these categories is their orthogonality; the functions in one class are independent of the functions in the other classes. For example, the distribution functions are the only functions concerned with the distributed nature of a NUCM repository. The other functions are not influenced by the fact that artifacts are stored in different locations. Their behavior is the same, whether the artifacts are managed by a local or a remote repository.

It should be noted that the functionality offered by each individual interface function is rather limited. At first, this seems contradictory to the goal of providing a high-level interface for configuration management policy programming. However, because of the limited functionality, each function can be defined with precise semantics. Not only does that generalize the applicability of the interface functions, it also allows the rapid construction of particular CM policies through the composition of sets of interface functions. In Section 6 we present some of the CM policies that we have constructed this way. Below we introduce, per class, the individual interface functions that constitute the programmatic interface to a NUCM distributed repository.

4.1 Access Functions

Access to a NUCM repository is attained through a workspace. In a workspace, artifacts are materialized upon request. Once the artifacts are materialized, other interface functions become available to manipulate them. In particular, versioning functions can be used to store new instances of artifacts, and collection functions can be used to manipulate the membership of collections. When a client CM system is finished processing, it closes the workspace and access to the artifacts in the workspace is removed.

The access functions in the interface of NUCM are `nc_open` and `nc_close`. The function `nc_open` provides access to a particular version of an artifact by materializing it in a workspace. Atoms are materialized as files, and collections as directories. Each use of the function `nc_open` materializes a single artifact. A workspace, thus, has to be constructed in an incremental fashion. This mechanism allows a client CM system to populate a workspace with only the artifacts that it needs.

The function `nc_close` negates the effects of the function `nc_open` and is used to remove artifacts from a workspace. The function operates in a recursive manner. When a collection is closed, all the artifacts that it contains are removed from the workspace as well.

4.2 Versioning Functions

Once an artifact has been opened in a workspace, the following versioning functions become available to create and store new versions of the artifact: `nc_initiatechange`, `nc_abortchange`, `nc_commitchange`, and `nc_commitchangeandreplace`.

Class	Function	Description
Access	nc_open	Materializes an artifact in a workspace.
	nc_close	Removes an artifact from a workspace.
Versioning	nc_initiatechange	Allows an artifact in a workspace to be modified.
	nc_abortchange	Returns an artifact in a workspace to the state it was in before it was initiated.
	nc_commitchange	Stores a new version of an artifact in a repository.
	nc_commitchangeandreplace	Overwrites the current version of an artifact in a repository.
Collection	nc_add	Adds an artifact to a collection.
	nc_remove	Removes an artifact from a collection.
	nc_rename	Renames an artifact within a collection.
	nc_replaceversion	Replaces the version of an artifact that is contained in a collection.
	nc_copy	Copies the version history of an artifact and adds the new artifact to a collection.
	nc_list	Determines the artifacts contained in a collection.
Attribute	nc_testandsetattribute	Associates an attribute and its value with an artifact (if the attribute does not yet exist).
	nc_getattributevalue	Determines the value of an attribute of an artifact.
	nc_removeattribute	Removes an attribute from an artifact.
Deletion	nc_destroyversion	Physically removes a version of an artifact from a repository.
Distribution	nc_setmyserver	Sets the default physical repository in which new artifacts will be stored.
	nc_getlocation	Determines the physical repository that contains the version history of an artifact.
	nc_move	Moves an artifact and its version history from one physical repository to another.
Query	nc_gettype	Determines the type of an artifact.
	nc_version	Determines the current version of an artifact.
	nc_lastversion	Determines the latest version of an artifact in a repository.
	nc_existsversion	Determines whether a version of an artifact exists in a repository.
	nc_isinitiated	Determines whether an artifact has been initiated in a workspace.
	nc_isopen	Determines whether an artifact has been materialized in a workspace.

Table 1: NUCM Interface Functions.

The function `nc_initiatechange` informs NUCM of a client's *intention* to make a change to an atom or a collection. In response, NUCM gives the client permission to change the artifact in the workspace. If the artifact is a collection, it has to be altered with the collection functions of NUCM (see Section 4.3). An atom, on the other hand, can be manipulated by any user program, since NUCM does not interpret its contents. Note that `nc_initiatechange` does not lock an artifact. Because of the orthogonality of the interface functions, the NUCM attribute functions that are described in Section 4.4 have to be used to properly lock an artifact if so desired.

The function `nc_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 `nc_initiatechange` was invoked. An `nc_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 `nc_commitchange` commits the changes that have been made to an artifact, storing the new version of the artifact in a uniquely named place in the repository and revoking the client's permission to change the artifact in the workspace. The function `nc_commitchangeandreplace` is similar in behavior to the function `nc_commitchange`, but instead of creating a new version of the artifact in the repository, it overwrites the contents of the version that was opened before. Again, versioning and locking are orthogonal, so the functions `nc_commitchange` and `nc_commitchangeandreplace` do 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 its default behavior, NUCM does not automatically create new versions of collections. Under this default behavior, a collection remains unchanged when a new version of one of its contained artifacts is added to the repository. If it is desired that the collection refers to the new version of the artifact, the versioning and collection functions have to be used to update the collection. In particular, the function `nc_initiatechange` has to be used before the collection functions can be used to update the collection, and the function `nc_commitchange` or `nc_commitchangeandreplace` has to be used to store the new contents of the collection.

To illustrate the versioning functions, suppose we have a repository containing the artifacts depicted in Figure 5a. Assume further that, using the function `nc_open`, a workspace has been created that contains these artifacts. To be able to modify the atom `Window.c` in the workspace, `nc_initiatechange` has to be used first. Once the desired changes have been made, the function `nc_commitchange` has to be used to store a new version of `Window.c` in the repository. The result is shown in Figure 5b. The repository now contains three versions of `Window.c`, but note that the collection `GUI-lib` has not changed, since we did not invoke `nc_initiatechange` on that collection.

Had the function `nc_commitchangeandreplace` been used instead of `nc_commitchange`, no new version would have been created of the artifact `Window.c`. For example, suppose the atom `Frame.c` was modified in the workspace after `nc_initiatechange` was used to gain access to it. If the function `nc_commitchangeandreplace` is subsequently used, the repository still looks like the one of Figure 5b. No new version of the atom `Frame.c` is created, but it should be noted that the actual contents of its second version has changed.

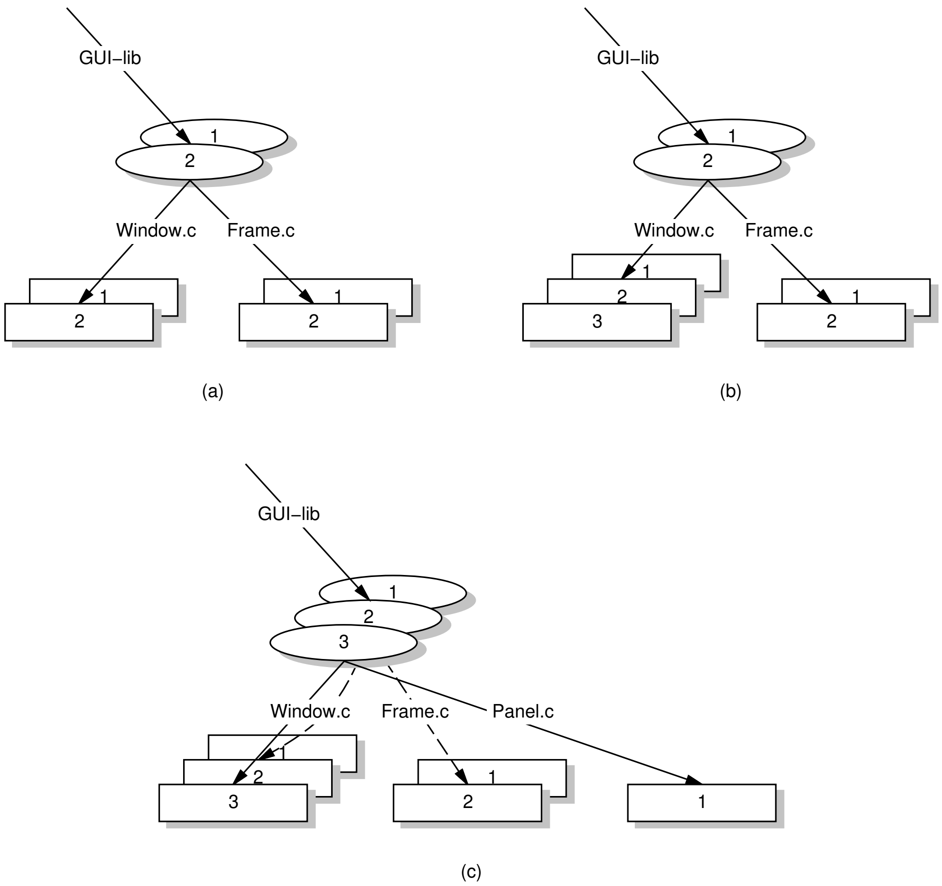

Figure 5: Progressive States of an Example Collection.

4.3 Collection Functions

Similar to the way an editor can be used to change an atom in a workspace, collections need to be changed via some kind of mechanism. But, because collections have special semantics, it would be unwise to allow them to be edited directly. Therefore, NUCM provides a number of interface functions that preserve the semantics of collections while updating their contents. These functions are the following: `nc_add`, `nc_remove`, `nc_rename`, `nc_replaceversion`, `nc_copy`, and `nc_list`.

The functions `nc_add` and `nc_remove` behave as expected, adding and removing an artifact to and from a collection, respectively. The function `nc_add` can add either a new or an existing artifact to a collection. The addition of a new artifact will simply store its contents in a NUCM repository. The addition of an existing artifact, on the other hand, will result in an artifact that is shared by multiple collections and for which a single version history is maintained. If it is so desired that, starting from the moment the artifact is added to the collection, a separate version history is maintained, the function `nc_copy` should be used instead of `nc_add`. As a result of `nc_copy`, a new artifact will be created in a NUCM repository. The new artifact will contain the same version history as the artifact that was copied, but will evolve separately.

A feature that has traditionally been difficult to provide in CM systems is the ability to rename artifacts. NUCM solves this problem by providing, directly in its repository interface, the function nc_rename, which renames an artifact. Because an artifact is only renamed in a single collection at the time, it is very well possible that a single artifact exists under different names in different collections. This is an important feature of the NUCM interface, since it allows an artifact to evolve without compromising its naming history.

The function nc_replaceversion complements the other collection functions because it operates in the version dimension as opposed to the naming dimension. Its behavior is simple: it changes the contained version of an artifact in a collection to another version. In order to support undoing changes, it is of course permissible for older versions of an artifact to replace newer ones.

The function nc_list rounds out the collection functions. It returns a list of the names and versions of the artifacts that are contained in a NUCM collection. Obviously, this functionality is useful in building a CM system that, for example, presents a user with the differences between two versions of a collection, recursively opens a workspace, or simply allows a user to dynamically select which artifacts to lock or check out.

The set of collection functions is complete. If we consider the artifacts that are contained by a collection to be organized in a two-dimensional space defined by name and version, all primitive functionality is provided. A name-version pair can be added, a name-version pair can be removed, a name is allowed to change, and a version is allowed to change. Therefore, despite the rather simple functionality provided by each individual function, the complete set of collection functions allows rapid construction of high-level, more powerful functions. For example, a function that replaces, under the same name, an atom with another one, can be constructed as a sequence of nc_remove, nc_add, and nc_rename.

To illustrate the collection functions, we continue the example of Figure 5b. Assume that once again all artifacts have been opened in a workspace. To manipulate the GUI-lib collection, we first use nc_initiatechange to allow us to modify the collection. To subsequently update Window.c to its latest version, we use nc_replaceversion. In addition, to provide a panel as opposed to a frame in the GUI-lib collection, we use nc_remove to remove the atom Frame.c and nc_add to add the atom Panel.c. If we then conclude by using nc_commitchange, the repository looks as shown in Figure 5c. A new version of the GUI-lib collection has been created that contains the changes we made. In addition, because we used the function nc_commitchange as opposed to nc_commitchangeandreplace, the old version of the collection is still available. This means that if we use the function nc_list on version 2 of the GUI-lib collection, version 2 of the atom Window.c and version 2 of the atom Frame.c are listed as the collection members, whereas if nc_list is used on version 3 of the collection, version 3 of atom Window.c and version 1 of atom Panel.c are returned.

4.4 Attribute Functions

Virtually every configuration management system attaches a certain amount of metadata to the artifacts that it maintains. These metadata usually capture such characteristics as an author, a date of creation, one or more change request identifiers, and a short synopsis of the changes made. To facilitate the association of metadata with the artifacts in a NUCM repository, the programmatic interface contains a number of primitive functions to manipulate attributes. In particular, it is possible to set the value of an attribute with nc_testandsetattribute, to retrieve the value of an attribute with nc_getattributevalue, and to remove an attribute with nc_removeattribute. Although these functions are rather simple, they are sufficient for configuration management purposes since the association of a small set of metadata with an artifact is often the primary usage of attributes in this domain.

The attribute functions were designed to support primitive locking of artifacts. In particular, the function `nc_testandsetattribute` only sets the value of an attribute if it does not yet exist. Therefore, it can be used to lock an artifact by simply setting a unique attribute that represents the lock. Although this results in a rather primitive locking facility, these functions do allow the construction of the actual locking schemes employed in such existing lock-based CM policies as RCS [48] and CCC/Harvest [42]. Because NUCM only focuses on the distributed versioning problem, we do not intend to provide more extensive locking facilities. Instead, it is our belief that if a more complicated locking scheme is needed, a specialized and full-featured lock manager (e.g., Pern [20]) should be used.

It should be noted that locks are not enforced by NUCM. Instead, a CM system has to use the attribute functions appropriately to implement its locking policy. Similarly, a lock on a collection will not cause a request for a lock on an artifact contained by 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 a collection only, or as a lock on the collection and its contents.

4.5 Deletion Function

Although it is an uncommon practice in the domain of configuration management to delete artifacts from a repository, it should still be possible to do so in case of obsolescence or mistakes. Therefore, the function `nc_destroyversion` is provided in the NUCM interface to physically delete a particular version of an artifact from the repository in which it is stored. A version, however, can only be deleted if it is not contained in a collection. For example, even though in Figure 5c a CM policy is permitted to delete the first version of atom `Window.c`, the deletion of the second version is disallowed because it is still contained by the second version of the collection `GUI-lib`.

NUCM also provides an automatic garbage collection mechanism for artifacts that are no longer contained. For example, if in Figure 5c version 2 of the collection `GUI-lib` is modified using the function `nc_remove` to remove the atom `Frame.c`, no version of `Frame.c` is contained in any collection. Because it is thus impossible to address `Frame.c`, the atom is automatically reclaimed by NUCM and physically deleted from the repository.

4.6 Distribution Functions

Users of systems that completely hide distribution often encounter performance difficulties related to the physical placement of data. Therefore, the NUCM interface provides functions that allow a CM client to determine and change the physical location of artifacts within a logical repository. In particular, the functions `nc_setmyserver`, `nc_getlocation`, and `nc_move` are available to manage the physical distribution of artifacts within a logical repository.

The first function, `nc_setmyserver`, specifies the physical repository to which newly created artifacts will be added. The second, `nc_getlocation`, returns the physical repository in which an artifact is actually stored. The last, `nc_move`, moves an artifact and its version history from its current physical repository to a new one. To avoid a repository-wide search for references, NUCM does not update containing collections with the new physical location of an artifact that has moved. Instead, it places a forwarder at the original location of the artifact. When a request is made for the moved artifact, NUCM uses the forwarder to update the old reference. Using a reference-counting scheme, NUCM updates old references as they are made and then eventually removes the forwarder.

We observe that in NUCM all versions of a single artifact are stored in a single physical repository. We have chosen not to support the distribution of individual versions over multiple repositories, because it would incur much more communication across repositories than is currently needed. In particular, the reference counting mechanism used for garbage collection and the forwarder mechanism used to locate artifacts would require the distribution of algorithms that are currently executed within a single physical repository.

4.7 Query Functions

The NUCM programmatic interface would not be complete without the ability to examine the state of artifacts. For example, when multiple CM clients share the same workspace, they should be able to verify whether the version of an artifact in a workspace was changed by another CM client. Similarly, when multiple CM clients share a single NUCM repository, they should be able to check whether new versions of an artifact have been added by other CM clients. The NUCM query functions were designed to provide exactly this type of functionality. Although simple, these functions are essential in the development of CM policies because they provide state information that a CM client does not have to track itself. The query functions that provide information about the artifacts in a workspace are particularly important in this respect.

Although the names of the interface functions speak for themselves, we provide here, for completeness, a one-sentence description and typical use case of each. The function `nc_gettype` returns whether an artifact is a collection or an atom, and is often used to recursively open a collection and all its containing artifacts in a workspace. To manage version trees, the function `nc_version` can be used to determine the version of an artifact before and after the function `nc_commitchange` has been used to store some changes. The function `nc_lastversion` returns the version number of the last version of an artifact, and is used to check for new versions of an artifact that might have been added by other CM clients. If some versions of an artifact have been deleted from a repository, the function `nc_existsversion` can be used to verify whether a particular version is still available or not. Finally, the functions `nc_isopen` and `nc_isinitiated` operate on artifacts in a workspace, and are used to verify whether an artifact has been opened and whether it is allowed to change, respectively.

5 Implementation Considerations

The previous two sections describe a model and an interface for a generic, distributed repository. In this section we discuss how they were implemented. We first introduce the architecture of NUCM and then present some issues we encountered while implementing the architecture.

5.1 NUCM Architecture

Figure 6 illustrates the architecture of NUCM in terms of an example repository structure. Shown is a logical repository that consists of three physical repositories. The artifacts in each physical repository are managed by a NUCM server. Combined, the NUCM servers for the physical repositories provide access to the complete logical repository. In particular, when artifacts are requested that reside in a different physical repository than the one managed by one of the NUCM servers, that NUCM server will communicate with the other ones to provide access to the artifact.

A CM system that uses a NUCM repository consists of two parts: the generic NUCM client and a particular CM policy. The generic NUCM client implements the interface that was discussed in Section 4 and thus serves as the basis upon which particular CM policies are implemented. This is illustrated in Figure 6; two CM policies, namely policy X and policy Y, both use the generic NUCM client to store and version the artifacts that they manage. In general, a single repository

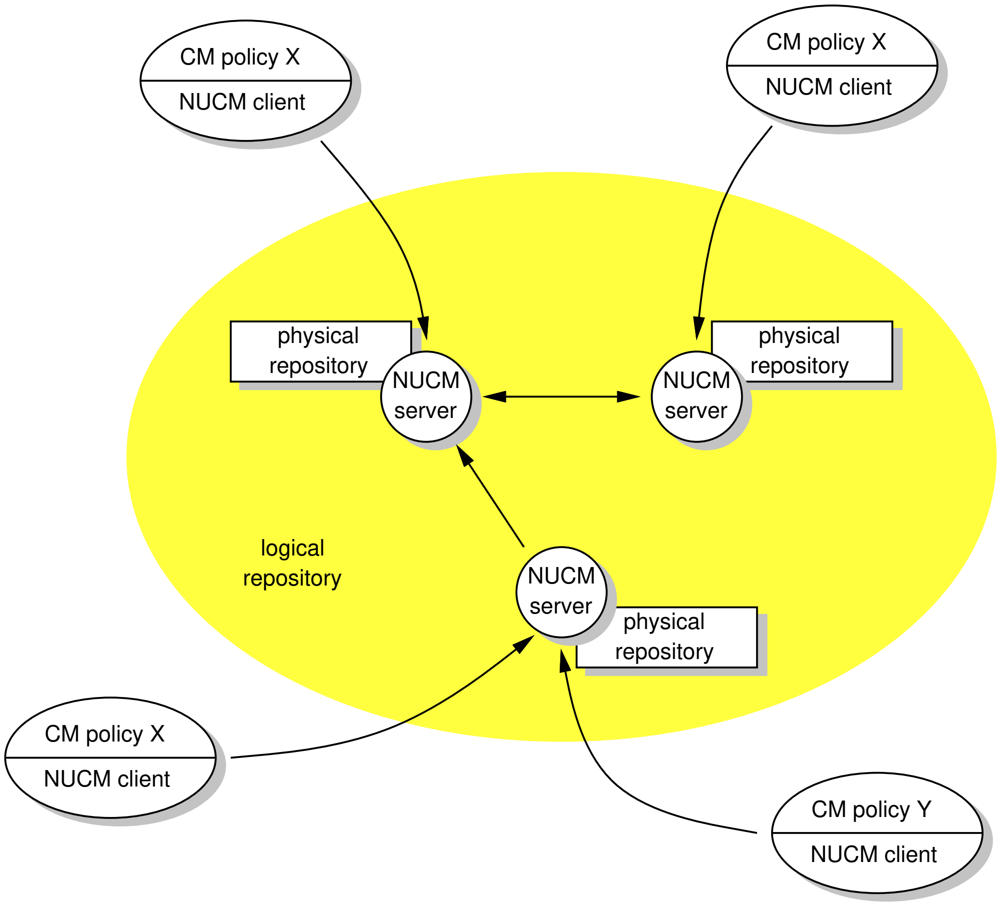

Figure 6: NUCM Architecture.

can store artifacts that are managed by different CM policies, as long as the policies partition the artifacts in separate name spaces within the repository. If different policies operate on the same artifacts, it is the responsibility of the CM policies to resolve conflicts.

It should be noted that although a CM system communicates with a particular NUCM server, this server combines with the other NUCM servers to provide the CM system access to all the artifacts in the logical repository.

5.2 NUCM Implementation

To implement NUCM according to the architecture of Figure 6, some obvious candidates exist, including a distributed file system (e.g., Jade [39] or Prospero [35]), a distributed database [41], or an advanced operating system environment (e.g., PCTE [46]). Our initial approach was to experiment with a rather different alternative, namely the CORBA standard for distributed object programming [12].

For a variety of reasons, we implemented NUCM twice, first with the Orbeline CORBA engine [38] and subsequently with a CORBA implementation called Q [29]. However, in both cases it turned out that CORBA was not the right platform to build a distributed CM repository. The following two reasons are the primary sources of this mismatch.

1. *CORBA is not well suited for bulk data transfer.* The type system of CORBA does not include a type that allows streaming of raw bulk data from one CORBA object to another. However, NUCM requires the transfer of files among servers and clients. In a CORBA-based implementation, this leads to a lot of unnecessary marshaling of data, as well as to a large amount of communication among objects to manage the piece-meal transfer of a file.

2. *NUCM generated too much network traffic.* CORBA hides distribution issues. Therefore, it is enticing to implement a system as a set of objects, irrespective of whether that is appropriate in a wide-area setting. However, NUCM requires careful coordination among its servers. In an implementation that does not reckon with distribution, this need for coordination leads to a large amount of server-to-server communication.

Unfortunately, these reasons were only revealed after our CORBA-based implementation of NUCM was well underway [51]. The important lesson to be learned here is that even though we had carefully analyzed the applicability of CORBA to the implementation of NUCM and had deemed such an implementation to be feasible and desirable, the benefits of CORBA (e.g., an object-oriented implementation combined with transparent distribution) turned out to be significant bottlenecks in the eventual implementation.

Because CORBA proved to be such a bottleneck, we decided to reimplement parts of NUCM by replacing CORBA with a tailor-made distribution mechanism. Unfortunately, this turned out to be a rather difficult exercise. We encountered a phenomenon that we call *CORBA-creep*: once CORBA is introduced in part of a software system, it tends to influence and spread to other parts. Thus, NUCM was much more tied into CORBA than we anticipated and its changeability was low. Therefore, we decided to completely abandon the approach and reimplement NUCM from scratch. We now have a simple yet powerful implementation, based on TCP/IP, that uses the network sparingly and is optimized to transfer files among clients and servers.[1] In particular, the speed of file transfer achieved by NUCM is comparable to FTP. It is with this implementation that we have carried out the experiments that are described in the next section.

Similar to CORBA, the implementation of NUCM hides distribution issues. Therefore, it seems that argument 2 as stated above also holds for NUCM. However, two fundamental differences exist between CORBA and our implementation of NUCM that invalidate this argument. First, whereas distribution in CORBA is completely hidden, it is possible in NUCM to manage distribution through the interface functions. This means that if undesirable effects occur in NUCM, they can be remedied by moving artifacts to closer proximity. Second, whereas operations on remote objects in CORBA always incur network traffic, most operations in NUCM are on artifacts that are cached in local workspaces. This means that far less network traffic is generated by NUCM as compared to CORBA.

Two additional comments are in place about our implementation of NUCM. First, the repository currently does not use a delta mechanism to save space when new versions of artifacts are stored. Such a mechanism can be added with relative ease through the adoption of a delta and compression library like vdelta [18] or bdiff [47] (a comparison of which can be found in [23]). In particular, it is our intention to enhance the function `nc_add` with a parameter that allows a CM client to control whether the storage of an artifact should use compressed deltas or not. Second, the repository does not provide atomicity and recovery capabilities. We consider those outside of the scope of building

[1] http://www.cs.colorado.edu/serl/cm/NUCM.html.

an experimental prototype, but recognize that an industrial-strength implementation of NUCM would certainly require an appropriate transaction mechanism to support such capabilities.

6 Implementing Three CM Systems

The current version of NUCM has been used in the implementation of three different CM systems. At present, two of those systems, namely DVS [6] and SRM [49], are in use, whereas the third system, WebDAV, represents an experimental implementation of an emerging standard in Web versioning [52]. Rudimentary implementations of the "standard" check-in/check-out and change-set policies [14] were also created, but these are based on a previous version of NUCM. We refer to [51] for that discussion.

Below, we discuss each system in more detail and use parts of their implementation to illustrate how NUCM can be used to program particular CM policies that may or may not be distributed. It should be noted that the policies themselves are not the contribution. Instead, the strength of NUCM lies in the ease with which these policies were constructed and the limited amount of work needed to make them suitable for use in a wide-area setting.

6.1 DVS

DVS [6] is a versioning system that is centered around the notion of workspaces.[2] Individual users populate their workspace with the artifacts needed, lock the artifacts they intend to change, modify these artifacts using the appropriate tools, and eventually commit their changes. This policy is similar to the one employed by RCS [48], except for the fact that DVS explicitly recognizes and versions collections and, moreover, operates in a distributed setting.

DVS possesses some characteristics that illustrate the power of NUCM. First, no special code needed to be developed for DVS to operate in a wide-area setting. DVS relies entirely on the mechanism included in the NUCM client to provide its distribution. Consequently, DVS not only can operate in client-server mode, but it is also possible to federate multiple physical repositories into a single logical repository that is used by DVS.

The second advantage of using NUCM in the construction of a CM system shows itself in the number of lines of source code used to develop DVS. Only 2,500 new lines[3] were needed to create the full functionality of DVS. The newly written source code mostly deals with the text-based user interface, the recursive operations on workspaces, the proper locking of artifacts, and the storage of metadata about the artifacts that are versioned. Other functionality, such as distribution, collections, and versioning, is inherited from NUCM.

The third advantage of using NUCM came upon us unexpectedly. On one occasion, DVS was being used by 10 people at 5 different sites to jointly author a document. It turned out that the policy provided by DVS did not completely match the desired process. In response, some of the DVS functionality was changed and new functionality was added. When the second version of DVS was subsequently and incrementally deployed to the various sites, no disruption of their work occurred, since the NUCM repository required no downtime, the artifacts in the repository needed no change, and slightly different policies could be used by multiple authors at the same time. Clearly, the separation of storage and policy proved to be invaluable in this situation.

[2] http://www.cs.colorado.edu/serl/cm/dvs.html.

[3] In this paper, all counts of source code lines are total counts. This includes empty lines and comments.

```
  int synchronize_workspace(const char *pathname, int recursive)
  {
     //
     // Part 1:  Determine the current and latest version of the artifact.
     //
     strip_version_r(pathname, strippedpath);
*    nc_version(strippedpath, "", currentversion);
*    nc_lastversion(strippedpath, "", lastversion);

     //
     // Part 2:  If needed, get the latest version of the artifact, unless
     // it is checked out.
     //
     if (strcmp(lastversion, currentversion) != 0) {
        do_open = 1;
*       if (nc_isopen(strippedpath, "."))
*          if (!nc_isinitiated(strippedpath, "."))
*             nc_close(strippedpath, ".", 0);
           else
              do_open = 0;
        if (do_open) {
           set_version_x(strippedpath, lastversion);
*          nc_open(strippedpath, ".", ".", "");
        }
     }

     //
     // Part 3:  If necessary, recursively synchronize the workspace.
     //
     if (recursive) {
*       atype = nc_gettype(strippedpath, ".");
        if (atype == COLLECTION) {
*          nc_list(strippedpath, "", &members);
           chdir(strippedpath);
           start = members;
           while (members != NULL) {
              synchronize_workspace(members->name, recursive);
              members = members->next;
           }
*          nc_destroy_memberlist(start);
           if (strcmp(strippedpath, ".") != 0)
              chdir("..");
           }
        }
     }
```

Figure 7: DVS Routine to Synchronize a Workspace.

To demonstrate how DVS is built upon the functions in the NUCM interface, Figure 7 presents a simplified[4] part of the DVS source code in which uses of NUCM interface functions are highlighted with a "`*`". Illustrated is a function that synchronizes a workspace with the latest versions of the artifacts in a NUCM repository. The function allows either a single artifact or a recursive set of artifacts to be synchronized, depending on the value of the parameter `recursive`.

The function can be partitioned into three parts. In the first part, the version of the artifact in the workspace and its latest version in the repository are determined through the use of the functions `nc_version` and `nc_lastversion`. If these versions are the same, the artifact is up to date with respect to the repository. If they are not, the second part of the function takes care of synchronizing the two by replacing the version in the workspace with the version in the repository. Additionally, before the latest version of the artifact is opened in the workspace, the current version is closed if it had been opened previously. Also, to avoid the loss of changes that may have been made to an artifact, it is verified whether the current version has been initiated for change. If so, the current version of the artifact is preserved in the workspace and the artifact is not synchronized.

The third and final part of the function deals with the recursive nature of the synchronization of a workspace. If the artifact to be synchronized is a collection, its contained artifacts are obtained and each of these artifacts is in turn synchronized through a recursive call.

6.2 SRM

SRM [49] is a tool that addresses the software release management problem.[5] It supports the release of systems of systems from multiple, geographically distributed sites. In particular, SRM tracks dependency information to automate and optimize the retrieval of components. Developers are supported by a simple release process that hides distribution. Users are supported by a simple retrieval process that allows the retrieval, via the Web, of a system of systems in a single step as a single package.

Although SRM is not a traditional CM system that stores and versions source code, it has many similarities to a CM system: it needs to manage multiple releases, it needs to manage dependencies among these releases, and it needs to store metadata about the releases. Combined with the need for a distributed repository that allows multiple sites to collaborate in the release process, these similarities led to the choice of using NUCM as the platform upon which to build SRM.

Of interest to the discussion in this paper is the flexibility that NUCM provides in the creation of a distributed repository. In particular, we examine the way new participants can join a federated SRM repository. To facilitate this functionality, each participating site maintains a NUCM repository that contains the releases they have created. Additionally, one of the NUCM repositories in the federation maintains a collection that contains all releases from all sites. This is illustrated in Figure 8 by the repositories in Rotterdam and Boulder. Both repositories contain a collection `nucm_root` that contains a collection `my_releases` and a collection `all_releases`. In each repository, the collection `my_releases` contains the releases made by that site. The collection `all_releases`, which is shared by both sites, contains all the releases. Note that the repository in Milano is not part of the SRM federation at this point.

The function `join`, presented in Figure 9, illustrates how a new physical repository can join an existing SRM federation. It operates by creating a new collection for local releases, `my_releases`, and linking to the existing collection that contains all releases, `all_releases`. To do so, it first sets

[4]All error handling has been removed from the example source code shown in this section.

[5]http://www.cs.colorado.edu/serl/cm/SRM.html.

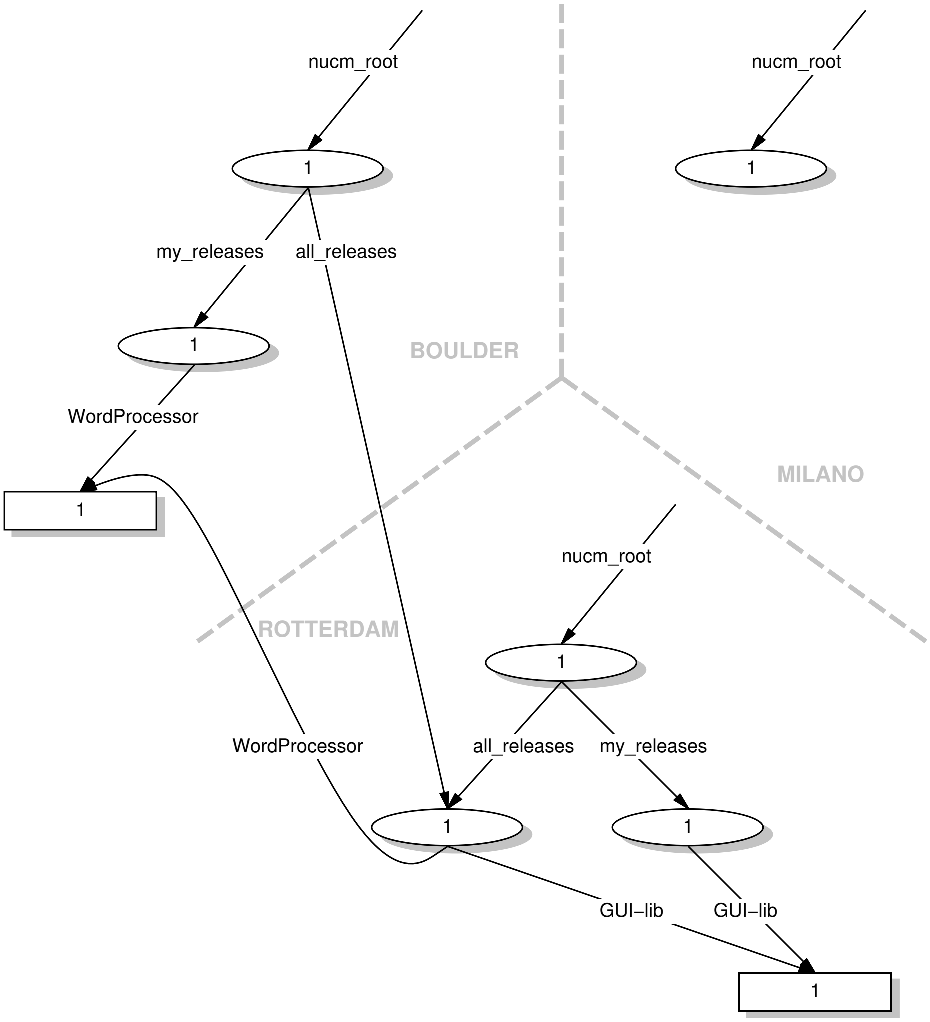

Figure 8: Federated SRM Repository Before Milano Joins the Federation.

up a workspace containing its main collection `nucm_root`, then allows the collection to change by using the function `nc_initiatechange`, and subsequently uses the function `nc_add` to add the newly created collection `my_releases` and the existing collection `all_releases` to the collection `nucm_root`. The collection `nucm_root` is then stored in the repository by using the function `nc_commitchangeandreplace` and the workspace is finally removed by using the function `nc_close`. The result of all these actions is shown in Figure 10. Assuming that the Milano site is being added to the SRM repository of Rotterdam and Boulder, the dashed lines indicate the new artifact and the containment relationships that are created by `join`. Once the artifact and the relationships are created, Milano is a full part of the SRM federation; when it adds new releases to the repository, they can be accessed from all sites.

The main advantage in using NUCM to construct SRM is the fact that distribution, once again, is hidden. Only two lines in the example source code of Figure 9 explicitly deal with distribution: in line 6 and line 7 the remote repository that contains the collection `all_releases` and the

```
  int join(const char* host, const char* port)
  {
     //
     // Part 1:  Set up a workspace.
     //
     sprintf(all_releases, "//%s:%s/nucm_root/all_releases", host, port);
     sprintf(my_nucmroot, "//%s:%s/nucm_root", NUCMHOST, NUCMPORT);
     sprintf(my_releases, "WORKSPACE/my_releases");
     sprintf(collection, "WORKSPACE/nucm_root");
*    nc_open(my_nucmroot, "", WORKSPACE, "");
*    nc_initiatechange(collection);

     //
     // Part 2:  Add a new artifact for my personal releases.
     //
     mkdir(my_releases);
*    nc_add(my_releases, "", collection, "");

     //
     // Part 3:  Import an existing artifact for the list of all releases.
     //
*    nc_add(all_releases, "", collection, "");
*    nc_commitchangeandreplace(collection, "");
*    nc_close(collection, "", 0);
  }
```

Figure 9: SRM Routine to Join a Federation.

local repository that is going to join the SRM repository are explicitly identified. After that, all NUCM programming is transparent with respect to distribution. This particularly exhibits itself in the remainder of the source code of SRM. Adding or removing releases to the SRM repository can simply be programmed as additions and removals to the local collections `my_releases` and `all_releases`, since NUCM hides the physical location of these collections. Similarly, through the collection `all_releases`, any site can retrieve releases from the other sites without having to know where a release is physically located.

6.3 WebDAV

WebDAV [52] is an emerging standard that proposes to add authoring and versioning primitives to the HTTP protocol [17]. In particular, the standard proposes extensions in the following five areas.

- *Metadata.* To be able to describe Web resources, WebDAV proposes the creation of new HTTP methods that add metadata to Web resources, as well as methods to query and retrieve the metadata.
- *Collections.* To be able to structure Web resources into higher-level constructs, WebDAV proposes the creation of new HTTP methods that allow Web resources to be grouped into collections, as well as methods that change the membership of collections.
- *Name space management.* To be able to efficiently move, copy, and delete Web resources, WebDAV proposes the creation of new HTTP methods that manipulate the Web name space.
- *Locking.* To avoid multiple entities updating a single Web resource in parallel and consequently losing changes, WebDAV proposes the creation of new HTTP methods that allow Web resources to be locked and unlocked for exclusive write access.

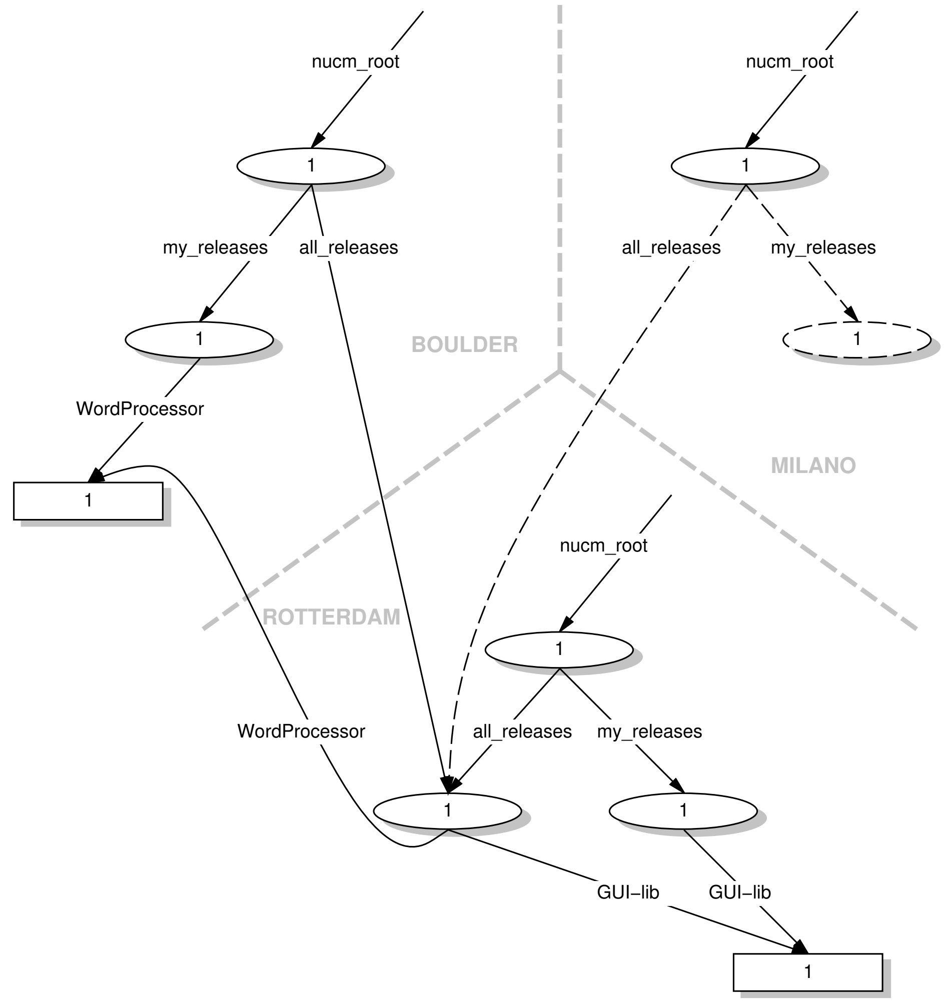

Figure 10: Federated SRM Repository After Milano Joins the Federation.

- *Version management.* To be able to keep a history of Web resources, WebDAV proposes the creation of new HTTP methods that allow Web resources to be versioned.

Our first observation is that, although the objective of WebDAV (providing an infrastructure for distributed authoring and versioning) is slightly different than the objective of NUCM (providing a distributed repository to construct configuration management policies), the interface methods that have been proposed by both are strikingly similar. Only two major differences exist. First, NUCM includes a naming model, whereas WebDAV leaves naming undefined. Second, WebDAV specifies a particular versioning policy, namely the version tree that is modified by checking out and checking in artifacts, whereas NUCM is generic with respect to versioning policies.

Our second observation is that, because of the similarity between NUCM and WebDAV, it seems advantageous to implement WebDAV using NUCM. To this end, we created a simple HTTP server that is also a NUCM client. Most of the new HTTP methods translate into direct calls to the NUCM interface, but some require more work. In particular, the versioning routines of WebDAV prescribe a policy that is based on a version tree. To implement this tree, we have to map the versions in the tree to versions of NUCM artifacts. In our implementation, this mapping is created by storing two NUCM artifacts for each WebDAV artifact, namely the actual artifact and an associated artifact that stores the version tree for that artifact. In addition, the version tree artifact has associated with it attributes that map each version number in the tree to a NUCM version number.

Figure 11 shows how one of the functions in our WebDAV implementation, namely `checkin`, takes advantage of this setup.[6] The function stores a new version of an artifact and updates the version tree accordingly. Its functionality can be divided into five separate parts. In the first part, several parameters used in the remainder of the function are determined. The names of the artifact being checked in and its corresponding version tree artifact are constructed first. Subsequently, the type of the artifact being checked in and its NUCM version number under which it was checked out are obtained. In the second part of the function, the new version of the artifact is read from the WebDAV client and subsequently stored in the repository through the use of the function `nc_commitchange`.

The third part of the function serves an important role; it is the part that updates the version tree. We do not show the actual algorithm that determines the new version number, since it does not involve any usage of NUCM functions and, moreover, would take up too much space. Instead, it is shown how the version tree is obtained from the repository, updated with the new version information, and stored back into the repository. Note the use of the function `nc_commitchangeandreplace` to replace the version tree, since there is no need to store multiple versions of the version tree itself.

The fourth part of the function sets new attributes for some of the artifacts in the repository. In particular, it preserves the type of the artifact that was checked in and relates the version in the version tree with the NUCM version of the new artifact. Note that the type information is attached to the new version of the artifact, for which a new NUCM name is first constructed.

Finally, in the fifth part of the function, the artifact that was previously checked out and locked for modifications is unlocked such that other users can now modify this version.

Once again, the reusability of NUCM proved to be valuable in the implementation of WebDAV. The total number of lines of source code that were developed to create a prototype WebDAV implementation was only 1200, of which approximately 40% accounts for a graphical user interface that can be used to perform WebDAV operations.

Admittedly, our experimental implementation does not cover all the functionality of WebDAV. However, the limited amount of code that needed to be developed and the rapid development time demonstrate an important aspect of NUCM: it can be used to support the rapid development of prototype CM policies. The development of a standard like WebDAV can particularly benefit from such an approach, since the ramifications of specific policy decisions can almost instantly be tried out with an actual implementation.

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 [50], and the issues that arise in supporting multiple engineers at multiple sites appear to be at the forefront.

In this paper we have described our approach to the distributed CM problem. Based on the critical observation that to effectively address this problem it is necessary to separate CM repositories

[6]Because WebDAV is an evolving standard, the example given here is slightly outdated with respect to the current version of the standard.

```
   int checkin(const char* path, const char* oldversion, FILE* client)
   {
      //
      // Part 1:  Determine necessary information.
      //
      sprintf(tree, "//%s:%s/nucm_root/TREE/%s", NUCMHOST, NUCMPORT, path);
      sprintf(treefilename, "%s/%s", WORKSPACE, (rindex(tree, '/')+1));
      sprintf(artifact, "//%s:%s/nucm_root/%s:%s", NUCMHOST, NUCMPORT, path, nucmversion);
      sprintf(filename, "%s/%s", WORKSPACE, (rindex(artifact, '/')+1));
*     nc_getattributevalue(tree, "", oldversion, nucmversion);
*     nc_getattributevalue(artifact, "", "TYPE", type);

      //
      // Part 2:  Store new version of the artifact.
      //
*     nc_open(artifact, "", WORKSPACE, "");
*     nc_initiatechange(filename, "");
      fd = open(filename, O_TRUNC | O_WRONLY);
      while ((n = fread(bytes, 4096, 1, client)) > 0)
         write(fd, bytes, n);
      close(fd);
*     nc_commitchange(filename, "", newnucmversion);
*     nc_close(filename, "", 0);

      //
      // Part 3:  Update the version tree.
      //
*     nc_open(tree, "", WORKSPACE, "");
*     nc_initiatechange(treefilename, WORKSPACE);
      fd = fopen(treefilename, "r+");
      ...
      ... /* Determine new version number */
      ...
      sprintf(line, "%s CHILD OF %s\n", newversion, oldversion);
      fputs(line, fd);
      fclose(fd);
*     nc_commitchangeandreplace(filename, "");
*     nc_close(filename, "", 0);

      //
      // Part 4:  Set new attributes.
      //
      sprintf(newartifact, "//%s:%s/nucm_root/%s:%s", NUCMHOST, NUCMPORT, path, newnucmversion);
*     nc_testandsetattribute(tree, "", newversion, newnucmversion);
*     nc_testandsetattribute(newartifact, "", "TYPE", type);

      //
      // Part 5:  We are done, unlock the artifact.
      //
*     nc_removeattribute(artifact, "", "LOCK");
   }
```

Figure 11: WebDAV Routine to Check In a File.

from CM policies, we have defined and implemented NUCM, a generic and distributed repository that can be reused in the implementation of many different CM policies. Because NUCM was specifically designed for distributed CM, the problems that are described in Section 2 are avoided. In particular, because distribution is handled at the repository level, users are freed from their responsibilities of managing distribution. Moreover, because of the orthogonality of the NUCM interface functions, new CM policies can be created irrespective of distribution. Only when it is needed to explicitly manage the physical location of artifacts, CM policies will have to use the distribution functions that are available in the interface.

NUCM is different from the distributed CM systems described in Section 2. The first major difference is that NUCM is not a complete distributed CM system, but rather a repository that is reusable in the construction of many different distributed CM policies. The second major difference is that the distribution mechanism embedded in NUCM is based on a federation of cooperating servers that manage distribution at the individual artifact level. This results in two concrete advantages in using NUCM. First, a CM policy can ignore most distribution issues, since those are handled by NUCM. Second, if the default distribution mechanism provided by NUCM becomes a problem, it can be flexibly tailored at both the individual artifact and at the repository levels.

The NUCM interface was designed to facilitate rapid construction of, and experimentation with, CM policies. However, NUCM has proven to facilitate more than that. DVS and SRM, two of the systems that were originally constructed to demonstrate the applicability and flexibility of the NUCM interface, have evolved into complete CM systems. Both are now in use in settings that involve multiple parties in multiple geographical locations, and both continue to evolve with respect to the functionality they provide. NUCM, thus, not only supports the construction of new CM policies, but also their gradual evolution into more mature systems.

Our work does not end here. Although we certainly believe that NUCM is a step in the right direction towards solving the distributed CM problem, much work remains to be done. In particular, it is our belief that NUCM facilitates the construction of standard policy libraries, thereby even further reducing the effort of implementing a CM system. Moreover, we expect to be able to use NUCM as a vehicle for exploring other important problems in configuration management. For example, we believe NUCM will be a suitable platform for investigating the problems of CM policy integration.

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*, number 1005 in Lecture Notes in Computer Science, pages 194–214, New York, New York, 1995. Springer-Verlag.

[2] Atria Software, Natick, Massachusetts. *ClearCase Concepts Manual*, 1992.

[3] D. Belanger, D. Korn, and H. Rao. Infrastructure for Wide-Area Software Development. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 154–165, New York, New York, 1996. Springer-Verlag.

[4] Bell Labs, Lucent Technologies, Murray Hill, New Jersey. *Sablime v5.0 User's Reference Manual*, 1997.

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

[6] A. Carzaniga. *DVS 1.2 Manual*. Department of Computer Science, University of Colorado, Boulder, Colorado, June 1998.

[7] S.-Y. Chiu and R. Levin. The Vesta Repository: A File System Extension for Software Development. Technical Report 106, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, June 1993.

[8] J.X. Ci, M. Poonawala, and W.-T. Tsai. ScmEngine: A Distributed Software Configuration Management Environment on X.500. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 108–127, New York, New York, 1997. Springer-Verlag.

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

[10] Continuus Software Corporation, Irvine, California. *Distributed Code Management for Team Engineering*, 1998.

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

[12] 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.

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

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

[15] 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.

[16] S.I. Feldman. MAKE — A Program for Maintaining Computer Programs. *Software—Practice and Experience*, (9):252–265, April 1979.

[17] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, January 1998. Internet Proposed Standard RFC 2068.

[18] G. Fowler, D. Korn, H. Rao, J. Snyder, and K.-P. Vo. Configuration Management. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 3. Wiley, New York, New York, 1995.

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

[20] G.T. Heineman. *A Transaction Manager Component for Cooperative Transaction Models*. PhD thesis, Columbia University, Department of Computer Science, New York, New York, June 1996.

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

[22] J.J. Hunt, F. Lamers, J. Reuter, and W.F. Tichy. Distributed Configuration Management via Java and the World Wide Web. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 161–174, New York, New York, 1997. Springer-Verlag.

[23] J.J. Hunt, K.-P. Vo, and W.F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, April 1998.

[24] INTERSOLV, Rockville, Maryland. *PVCS VM SiteSync and Geographically Distributed Development*, 1998.

[25] INTERSOLV, Rockville, Maryland. *Using PVCS for Enterprise Distributed Development*, 1998.

[26] JavaSoft, Inc., Palo Alto, California. *JavaSafe 1.0 User's Guide*, 1998.

[27] Y.-J. Lin and S.P. Reiss. Configuration Management with Logical Structures. In *Proceedings of the 18th International Conference on Software Engineering*, pages 298–307. Association for Computer Machinery, March 1996.

[28] 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.

[29] M.J. Maybee, D.M. Heimbigner, and L.J. Osterweil. Multilanguage Interoperability in Distributed Systems. In *Proceedings of the 18th International Conference on Software Engineering*, pages 451–463. Association for Computer Machinery, March 1996.

[30] Microsoft Corporation, Redmond, Washington. *Managing Projects with Visual SourceSafe*, 1997.

[31] B. Milewski. Distributed Source Control System. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 98–107, New York, New York, 1997. Springer-Verlag.

[32] Mortice Kern Systems, Inc., Waterloo, Canada. *MKS Source Integrity Reference Manual*, 1995.

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

[34] Neuma Technology Corporation, Ottawa, Canada. *NeumaCM+ FAQ's*, September 1998.

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

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

[37] Perforce Software, Alameda, California. *Networked Software Development: SCM over the Internet and Intranets*, March 1998.

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

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

[40] C. Reichenberger. VOODOO: A Tool for Orthogonal Version Management. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 61–79, New York, New York, 1995. Springer-Verlag.

[41] 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.

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

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

[44] SQL Software, Vienna, Virginia. *The Inside Story: Process Configuration Management with PCMS Dimensions*, 1998.

[45] Starbase Corporation, Irvine, California. *StarTeam Web Connect Users's Guide*, 1996.

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

[47] W.F. Tichy. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.

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

[49] A. van der Hoek, R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Software Release Management. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 159–175, New York, New York, September 1997. Springer-Verlag.

[50] A. van der Hoek, D.M. 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, New York, New York, 1995. Springer-Verlag.

[51] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In *Proceedings of the 18th International Conference on Software Engineering*, pages 308–317. Association for Computer Machinery, March 1996.

[52] E.J. Whitehead, Jr. World Wide Web Distributed Authoring and Versioning (WebDAV): an Introduction. *StandardView*, 5(1):3–8, March 1997.