Design and Implementation of a Distributed Versioning System

Antonio Carzaniga

October 13, 1998

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy

Technical Report n. 98-88

DVS is a simple versioning system that adopts a check-in/check-out policy with exclusive locks much like the one implemented by RCS. In addition to the basic functionalities of RCS, DVS provides extensions in two main directions: *distribution* of artifacts and versioning of *collections* of artifacts.

DVS has been implemented on top of NUCM 2, a generic distributed platform aimed at providing a policy-neutral programmable interface for realizing configuration management systems. NUCM provides support for storing artifacts and collections of artifacts and their attributes in a set of distributed servers. DVS implements the locking policy and all the related higher level services including check-in and check-out of single artifacts as well as collections, managing locks, change logs, and recursive check-in and check-out.

This paper describes the main design principles underlying DVS and NUCM together with the basics issues regarding their implementation. DVS has been used and is currently being used for collaborative authoring involving several authors distributed over five different sites on the Internet. We also discuss this first experience and the feedback and validation for both DVS and NUCM.

1 Introduction

DVS (pronounced *devious*) is a simple *distributed versioning system* that allows to store and retrieve versioned artifacts as well as collections of artifacts. Changes to artifacts or to collections are controlled with a strict locking policy. DVS is inspired to RCS [32] in that it is meant to have the same straightforward set of operations and the same change management policy (with locking). But at the same time, DVS has been designed to cover two major shortcomings of RCS.

The first contribution is an extension of the data model to support the versioning of *collections* of artifacts. In DVS, a collection is simply a set of versions of other artifacts, i.e., files or other collections. A collection can represent a particular *view* or *configuration* of a set of documents, for example, a program that is composed of many source files can be represented by a collection. Each version of the collection could correspond to a different version of the system by referring to the appropriate set of versions of the component files. Also, since a collection can recursively contain other collections, any kind of break-down structure of documents can be represented. In general, collections can be used to denote any kind of relations between artifacts. DVS has been designed so that the basic versioning operations can be applied exactly in the same way to collections as well as files.

The second extension provided by DVS is the ability to distribute the storage of artifact over the network. Much like RCS, DVS is based on a concept of *repository* (the `RCS` directory in RCS) and *workspace* (the current working directory). The repository is the *logical* storage area for all the managed artifacts while the workspace is a per-user working area. Users that want to collaborate and share data, can access the repository with the check-out and check-in operations. As a consequence of these operations, artifacts are copied from the repository to the user's workspace and vice-versa. In DVS, each artifact can be stored in a different physical repository, each one located anywhere on the Internet. Accessing the different repositories is transparent to the user. A clear advantage of this approach is that users don't need to share a file system to cooperate with DVS. Also, distant users that intend to collaborate by sharing artifacts can set up the repositories so that artifacts are stored in a local repository that is close to authors that access them more frequently.

DVS has been implemented on top of NUCM 2, a generic distributed platform aimed at providing a policy-neutral programmable interface for realizing configuration management (CM) systems. NUCM provides support for storing artifacts and collections and their attributes in a set of distributed servers. DVS inherits from NUCM its data model, i.e., the artifacts and collections structure, the linear versioning schema, and the naming schema. Using these basic building blocks, DVS implements the locking policy and all the related higher level services including check-in and check-out of single artifacts as well as collections, locks management, change logs, recursive check-in and check-out, and sync. NUCM and DVS are strongly related, thus, this paper also presents a description of the most important design and implementation features of NUCM, in particular, a NUCM generic CM file system abstraction is presented together with the NUCM application programming interface.

Another important motivation for designing and developing DVS has been the need for a validation of the suitability and usability of the NUCM interface. In the same direction of experimenting and validating, it is worth mentioning that DVS has been effectively used in some real collaborative authoring efforts involving several users distributed among various sites across the United States. Both the design of DVS and its use, have provided useful comments, usage patterns, desired features, and critiques for the NUCM infrastructure. Here we discuss the lessons we learned in this process.

This paper is laid out as follows: Section 2 surveys a number of related systems that attack the problem of distributed configuration management, Section 3 describes the data model and functionalities of DVS, Section 4 presents the NUCM file system abstraction and gives an overview of the NUCM architecture and programmatic interface, Section 5 presents in short the functionalities of DVS, Section 6 then evaluates and critiques DVS and NUCM providing some guidelines for future developments, and Section 7 summarizes this work and draws some conclusions. In Appendix A, each DVS functionality is presented in details in the form of a user manual.

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 [28], EPOS [23], JavaSafe [18], NSE [10], Sablime [4], ShapeTools [19], SourceSafe [20], TrueChange [29], and VOODOO [27], 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. *Coarse granularity of distributed artifacts.* This means that only big pieces of the repository can be distributed. In the extreme case of a centralized architecture, only the repository as a whole can be distributed. 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) [25] is an extension of the popular RCS system [32]. 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 re-implemented to work as a remote client that consults the repository each time it performs a CM operation on an artifact.

Distributed CVS Distributed CVS (DCVS) [14] 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.

Adele Adele [9] has been enhanced for distribution through a tool called Mistral [12]. 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) [21] 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 [15] 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.

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 [7, 8], PVCS has an optional extension named SiteSync [16], and NeumaCM+ incorporates a feature called MultiSite [24]. 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 [22] and StarTeam [31] have created Web interfaces to their respective CM systems. In addition, PVCS [17] and NeumaCM+ [24] have created Web interfaces that can be used as an alternative to their replicated repositories.

Two other systems deserve mention. The first, PCMS [30], 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 [26], 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.

3 Data Model and Basic Functionalities

3.1 DVS/NUCM Data Model

DVS manages *distributed versioned artifacts*. An artifact is either an *atom*, or a *collection*. Atoms are arbitrary blocks of data, they are black-box objects in the sense that they have no visible structure for DVS. Typical atoms include source files or sections of a document as well as libraries or executables. A collection is a set of *references* to other atoms or collections. Each reference points to a specific version of an artifact and has a name that is unique within the collection. Typically, collections are used to model projects, components of a project, entire documents or entire systems.

As suggested here, an artifact, either an atom or a collection, can have multiple *versions*. Within an artifact, each version is identified by a unique version tag.

References in collections are a very flexible structuring mechanism that allow to express several different kinds of relations among artifacts. One particular version of an artifact can be referenced in more than one collection, possibly with different names. Any two not necessarily distinct versions of the same artifact can be referenced in the same collection, provided the two references have two distinct names.

In general, we can use a simple graphical notation to model atoms, collections, and references. The data model is represented as a directed graph in which references are depicted as arcs starting from the collection they belong to and pointing to the referenced artifact. Arcs are labelled with the name of the reference. Collections are nodes that accept both incoming and outgoing arcs while atoms are leaf nodes, i.e., nodes that have no outgoing arcs. In this notation, any acyclic graph represents a valid DVS data model.

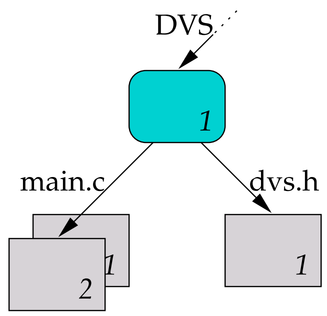

Figure 1: Example: a very simple DVS/NUCM data model

Figure 1 and Figure 2 displays a couple of simple examples of DVS data model. In the diagrams, collections and atoms are rectangles with round and sharp angles respectively. In Figure 1, a collection referenced as `DVS` contains two atoms referenced as `main.c` and `dvs.h` respectively. In particular, version 1 of `DVS` refers to version 2 of `main.c` and version 1 of `dvs.h`. This

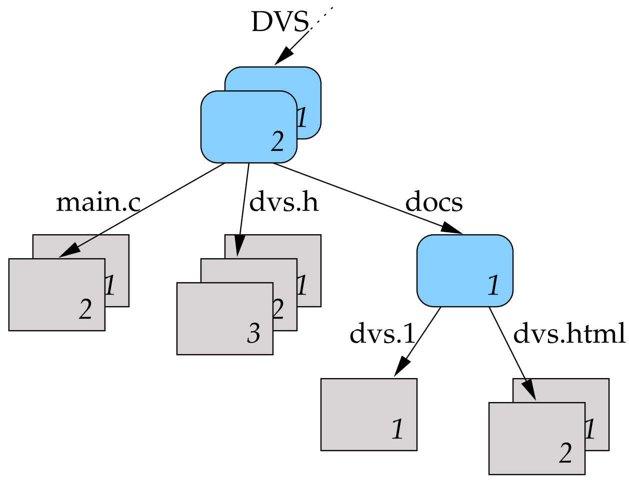

Figure 2: Example: a simple DVS/NUCM data model with sub-collections

data model might evolve into the one depicted in Figure 2 where version `2` of the `DVS` collection contains the same reference to `main.c` version `2` and an updated reference to version `2` of `dvs.h`. In addition to these two, the collection also refers to another collection named `docs` containing two other atoms named `dvs.1` and `dvs.html` respectively.

As we have seen, references in collections form breakdown structures of artifacts in which atoms are the leaves. The starting points for these structures are NUCM *root* collections that can be accessed directly by means of their location.

We also pointed out that an artifact can be referenced by more than one collection. The examples rendered in Figure 3 show some of these situations. In sub-figure (a), a collection, corresponding to the DVS source tree refers to an artifact named `nucm.h` that is also contained in another collection corresponding to the NUCM source tree. In sub-figure (b) a similar scenario shows the documentation collections of DVS and NUCM. These collections share an artifact that is referenced as `nucm.html` in DVS documentation and as `index.html` in NUCM documentation. Finally, sub-figure (c) shows an artifact that is reachable from two collections under the same sub-tree.

Note that in some of the models in figures 1 and 1 there might be other references that are not explicitly depicted. For example, the model in Figure 2, being an evolution of the one in Figure 1, should also show the references of version 1 of the `DVS` collection.

3.2 Repository and Workspace

DVS operates according to a classical client-server schema. It defines two logical environments: the *repository* and the *workspace* (see Figure 4). The repository is the storage area for artifacts while the workspace is the working area in which artifacts can be manipulated by users. The repository is composed of a set of NUCM *physical* repositories (servers), possibly distributed over the Internet. Workspaces represent the client side and are usually created on a per-user

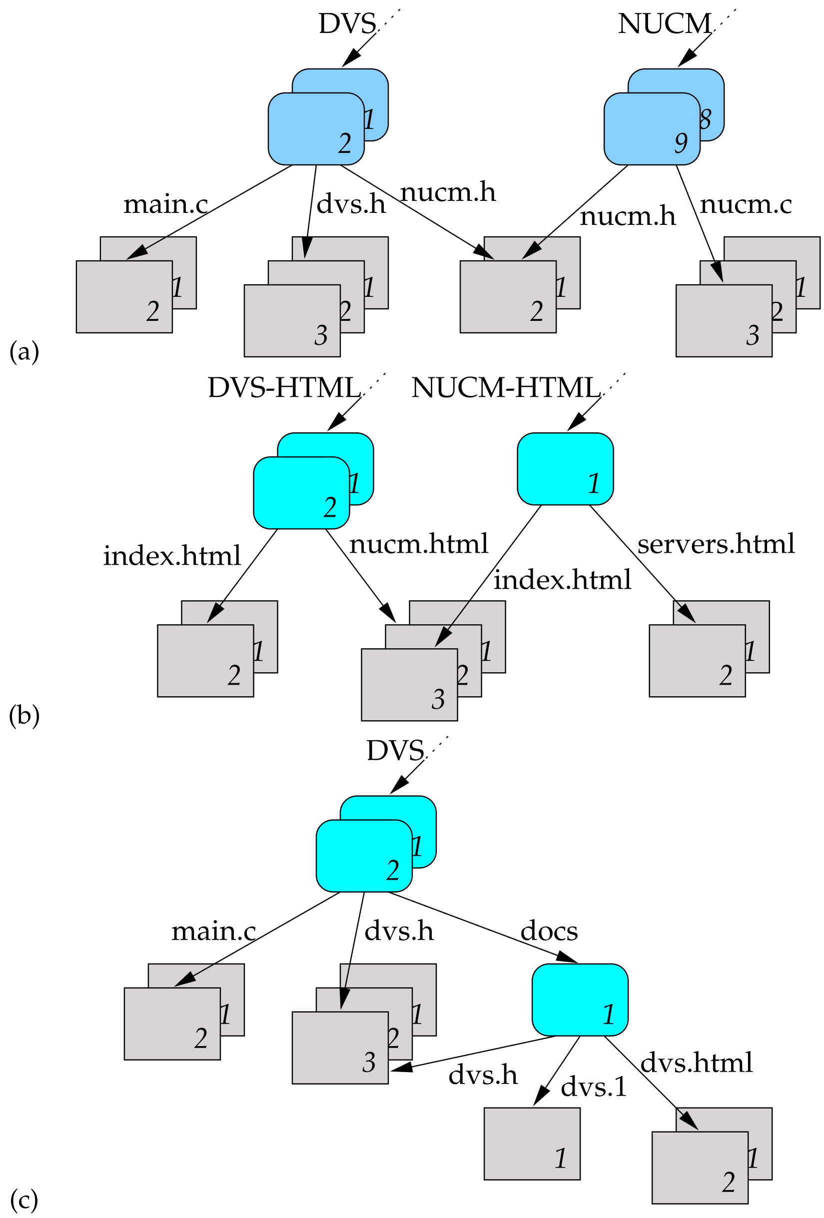

Figure 3: Example: DVS/NUCM data models that share artifacts

basis.

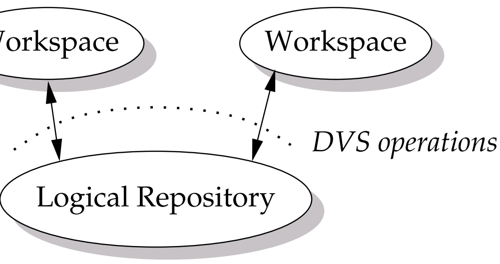

Figure 4: DVS client-server architecture

All the DVS operations involve some interaction with the repository (e.g., printing the log information), with the workspace (e.g., listing the members of an open collection), or between the workspace and the repository (e.g., locking or checking in or checking out an artifact). The workspace is also the environment in which artifacts are *used* and possibly changed by other tools.

DVS, like most configuration management systems, is most often used to store artifacts that normally reside in a file system and that are accessed by several "external" tools that are not aware of the CM system. Therefore, rather than exporting some kind of programmatic interface for the workspace, DVS adopts an unobtrusive access model by making artifacts available via the native file system. The workspace is thus a directory in the file system of the user. In the workspace, collections are materialized as directories, while atoms are represented as files. Artifacts are materialized with their reference name, i.e., the name of the reference used in the collection through which they are referenced. For example, Figure 5 shows a data model and its materialization in the workspace.

As you can see in the example of Figure 5, version tags are not explicitly reflected in the workspace. This is because in general, it is not desirable to have different file names for different versions of an artifacts, but also because version tags are redundant since DVS allows only one version of an artifact to be present in the workspace at the same time.

A workspace is not necessarily restricted to contain just DVS artifacts. The use of typesetting programs, compilers, and other tools that create auxiliary files is facilitated by allowing these transient files to coexist in a workspace.

3.3 DVS/NUCM Distribution Model

A NUCM repository is a set of NUCM servers possibly running on different sites over the Internet. By distribution model, we identify the mapping between artifacts in the logical data model and the artifacts as they are stored in the physical repository.

DVS provides for distribution mechanisms at the level of one single artifacts. This means that every DVS artifact is stored in one of these distributed servers. Once an artifact has been created and referenced within one or more collections, it can be accessed transparently by any user irrespectively of the user's location or the physical location of the collection that refers to it. To

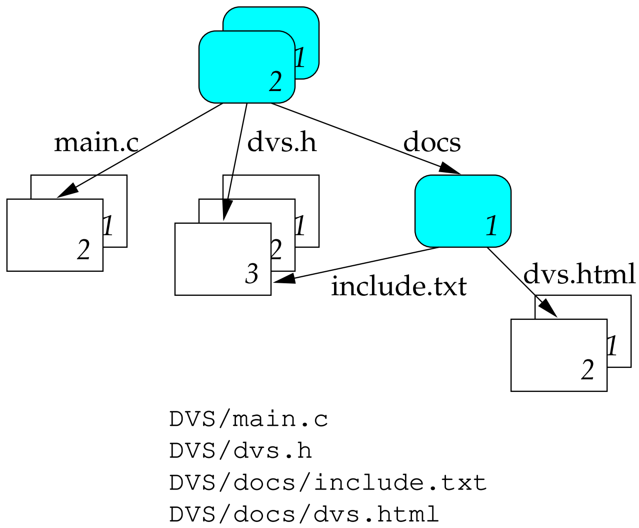

Figure 5: Materialization of the data model in the workspace

this extent, collections not only maintain the names of the artifacts that they contain, they also track their physical locations.

Access to artifacts that reside at remote repositories is transparent to a user. All the functionality provided by DVS for local use is available in the remote case. Whenever necessary, 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. More details on the processes implemented by NUCM and DVS in accessing remote artifacts are given in Section 4.4. Currently, neither NUCM nor DVS do not implement any caching or pre-fetching policy, thus, network delays and unreachability of servers always remain visible to a user.

4 A Network Unified CM File System

NUCM [35] (Network Unified Configuration Management) is a distributed repository that implements primitive services for configuration management systems. NUCM provides a file repository especially geared towards the implementation of configuration management systems. The main idea of NUCM is to provide a *distributed, versioned file system with links and attributes*. With its data model and its repository facilities, NUCM constitutes the platform that supports DVS.

Note that, although conceptually NUCM provides a file system abstraction, its implementation as it currently stands does not have a real operating system (virtual file system) interface, but instead it exports a user-level API. The bulk of this section illustrates the features of NUCM in the framework of a file system while Section 4.6 presents the actual API functions. In the following the NUCM/DVS workspace can be thought of as the NUCM file system.

4.1 Paths

A DVS artifact can be identified by means of a NUCM *path*. A path is an extension of usual file system paths that can also incorporate:

- a starting NUCM root collection with a NUCM server address;
- version tags for artifacts; and
- an attribute for the target artifact

If we consider the NUCM services from the viewpoint of its "file system" abstraction, then all its functionalities can be accessed by some form of encoding of paths.

$$\begin{aligned}
\langle \textit{path} \rangle &:= [\langle \textit{server} \rangle]\langle \textit{artifact-path} \rangle \\
\langle \textit{server} \rangle &:= \texttt{//}\langle \textit{hostname} \rangle[\texttt{:}\langle \textit{portnumber} \rangle]\texttt{/} \\
\langle \textit{artifact-path} \rangle &:= \langle \textit{artifact} \rangle(\texttt{/}\langle \textit{artifact} \rangle)^* \\
\langle \textit{artifact} \rangle &:= (\langle \textit{id} \rangle[\langle \textit{version} \rangle][\langle \textit{attribute} \rangle])+ \\
\langle \textit{id} \rangle &:= \langle \textit{ascii-chars} \rangle - \{\texttt{/}\ \texttt{\^{}}\ \texttt{:}\ \} \\
\langle \textit{version} \rangle &:= \texttt{:}[\langle \textit{id} \rangle] \\
\langle \textit{attribute} \rangle &:= \texttt{\^{}}[\langle \textit{id} \rangle]
\end{aligned}$$

Table 1: Form of DVS/NUCM paths

Table 1 shows the syntax of NUCM paths (with the usual convention that symbols enclosed in square brackets [] are optional). Table 2 instead shows some examples. In the following sections we will comment these examples and we will explain the process that NUCM uses to interpret a path to fetch the appropriate resource. In doing this, we will highlight the main features of the NUCM file system: distribution, versioning, and attributes.

1.foo.c
2.foo.c:3.1
3.inc/bar.h:1.0^author
4.inc:1.2/bar.h
5.test.c:
6.inc:4b/new.h:^lockedby
7.//host.xyz.edu:1234/nucm˙root
8.//machine.hjk.com/nucm˙root:4/main.tex

Table 2: Examples of DVS/NUCM paths

Paths that do not refer to a NUCM server explicitly, e.g., `foo.c`, correspond to files in the local file system that are workspace images of NUCM artifacts. In NUCM the starting point in the local file system must be explicitly included in the access functions. DVS instead implicitly uses the current working directory. Thus, the path `foo.c` will refer to the file `foo.c` in the current working directory and to the NUCM artifact to which that file is associated.

As we pointed out, NUCM collections are mapped onto directories in the NUCM file system. Every directory contains a reference to itself named '`.`'

similarly to usual file systems. However, unlike other file systems for which the *parent* directory is univocally determines, in NUCM there is no '`..`' reference for the parent directory simply because such a unique mapping doesn't exist.

4.2 Versioning

NUCM artifact can exist in several different versions. Each version can be selected by appending a colon character followed by the version tag to the artifact name. So, for example, a path of the form `foo.c:3.1` refers to version '`3.1`' of artifact '`foo.c`'. It should be noted that NUCM doesn't impose any versioning schema, therefore version tags are generic identifiers that have no special semantics. This implies that NUCM would by no means relate artifact `foo.c:3.1` to, say, artifact `foo.c:3.2`. For the same reason, a version tag must be explicitly provided by the user in opening new versions. Version tags are not at all limited to numeric expressions, for example, `foo.c:2.8b` and `foo.c:XY` are paths containing valid version tags.

If no version tag is specified for an artifact, then NUCM selects the version that is referenced by the collection used to refer to that artifact. This collection is either explicitly derived from the path or it is provided to NUCM as a "context" parameter. DVS uses by default the collection corresponding to the current working directory. So, for example, `inc:1.2/bar.h` refers to the version of `bar.h` that is referenced in version `1.2` of collection `inc`.

An empty version identifier can be used to access the set of all the versions of an artifact in the form of a directory. Thus, for example, the path `bar.h:` refers to a directory containing one entry for each version of the artifact `bar.h`. Note that, as a consequence of this syntax, in the NUCM file system, the path `foo.c:/3.1` is a valid path referring to version `3.1` of artifact `foo.c`.

4.3 Attributes

An attribute is an arbitrary block of data associated with an artifact. Attributes have names just like artifacts and versions. The NUCM file system provides both *version attributes* that are associated with specific versions of artifacts, and *artifact attributes* associated to the entire artifact.[1] The file system interface grants access to an attribute in the form of a file. The path for an attribute is obtained by appending a caret sign ('`^`') followed by the attribute identifier to the artifact path. These rules for the identification of attributes are orthogonal to the ones that determine versions, described in the previous section. In particular, if no version tag is specified, the attribute will always be a *version attribute* of the "current" version. Instead, accessing an *artifact attribute* is equivalent to accessing an attribute of the directory representing the whole artifact, thus, the path `foo.c:^designer` refers to the a file containing the artifact attribute `designer` of `foo.c`.

Similarly to what we do for the encoding of versions, we can use a null attribute id to access a list of all the existing attributes in the form of a directory. Every entry of this directory is a file representing an attribute and named with the attribute id. So, for example, if the current version of `foo.c` has the two attributes `author`, and `accesslist`, then `foo.c^` is a directory containing

[1]The current implementation of NUCM has only *version attributes*.

two files named `author`, and `accesslist` respectively. This works for version attributes, but similarly, `foo.c:^` denotes another directory containing one file entry for each artifact attribute.

4.4 Distribution

The procedure used by NUCM to interpret paths is conceptually very similar to the one implemented by UNIX operating systems in interpreting usual file system paths. Clearly, in addition to the usual directory traversal routine, NUCM must handle versions and attributes. However, this procedure is of special interest because it is tightly related to the way artifacts are distributed across the network.

As we have seen, NUCM allows every artifact, either atom or collection, to be stored at a different site. All the versions of an artifact are stored together. Figure 6 shows a data model together with the physical location of each artifact.

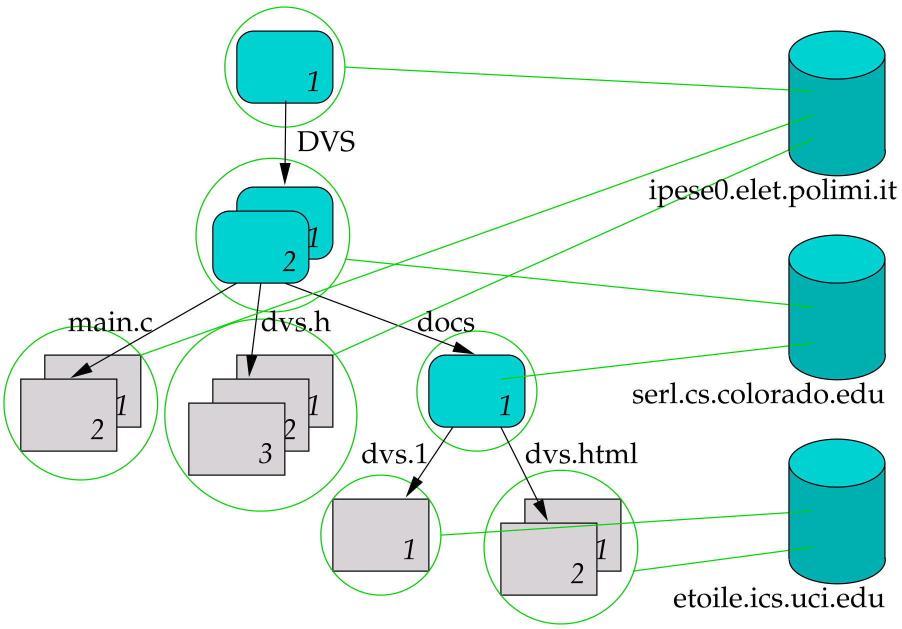

Figure 6: Distribution of artifacts

Of course, the mapping can be established by the user. In fact, artifact are initially stored at one particular site specified by the user's default server when they are created and they can also be moved using specific NUCM primitives.

Anyhow, in fetching an artifact, NUCM must perform a collection traversal search that may span multiple sites. To do this, NUCM implements an *internal* interface function, i.e., a function that is not directly exported to the user-level API, that interprets a path locally as far as it can and returns either the final destination or a pointer to the next site and the residual path that remains to be interpreted. The client issuing the request calls this function that iterates until the final destination is reached.

As an example, consider the path:
`//ipese0.elet.polimi.it/DVS:2/docs/dvs.html:1`
A client that wants to access that artifact starts off by connecting to the NUCM server on `ipese0.elet.polimi.it` requesting path `//DVS:2/docs/dvs.html:1`. The server on `ipese0` replies that it can not fulfill the request completely and provides the physical address of the first unknown artifact and the unresolved path. The physical address of an artifact is composed by three pieces of information: a server address (IP address and port number), a physical artifact id that is conceptually similar to the i-node of a file, and its version id that is taken verbatim from the path. Thus, in this case, the client will receive a reply containing of a partial resolution of the path:

server =	`serl.cs.colorado.edu:1234`
artifact =	`492:2`
residual-path =	`docs/dvs.html:1`

At this point the client function iterates sending a second request to the server running on `serl.cs.colorado.edu` providing artifact number `492` version `2` as a starting point and `docs/dvs.html:1` to be interpreted. The server reads artifact 492 version 2, a collection, resolves the reference to artifact `docs` locally and finds a non local reference corresponding to `dvs.html:1`. That reference is returned to the client in this form:

server =	`etoile.ics.uci.edu:1234`
artifact =	`239:1`
residual-path =	*null*

Finally, the client can directly access artifact `239` version `1` stored on server `etoile.ics.uci.edu`.

4.5 Benefits of the file system approach

The model of a *CM specific* extension of a file system has the obvious advantage of inheriting all the nice properties of file systems incorporating them in an orthogonal and elegant manner with its CM specific features. Although we will not go into details on how to program a NUCM-based CM system, we will mention here a couple of examples that highlight the composition of file-system and CM features of NUCM.

As an example, we can consider the change control policy with strict locks on artifacts that is adopted by DVS. This policy is implemented in DVS in its two commands: *check out* (for modifications) and *check in*. Their implementation in pseudo code looks like this:[2]

check out :

```
dvs_checkout_lock(string artifact) {
   int lockby_fd, artifact_fd;
   string who_locks;

   lockby_fd = open(artifact + "^lockedby", O_CREAT | O_RDWR);
   flock(lockby_fd, LOCK_EX);
```

[2]Proper error handling is omitted.

```
      who_locks = readline(lockby_fd);
      if (who_locks == "") {
         writeline(lockby_fd, current_user());
      }
      else if (who_locks != current_user()) {
         error(artifact + " is already locked by " + who_locks);
         flock(lockby_fd, UNLOCK);
         close(lockby_fd);
         return ALREADY_LOCKED;
      }
      artifact_fd = open(artifact, O_RDWR);
      /* ... */
      close(artifact_fd);
      flock(lockby_fd, UNLOCK);
      close(lockby_fd);
   }
```

check in :

```
   dvs_checkin_unlock(string artifact) {
      int lockby_fd, artifact_fd;
      string who_locks;

      lockby_fd = open(artifact + "^lockedby", O_CREAT | O_RDWR);
      flock(lockby_fd, LOCK_EX);
      who_locks = readline(lockby_fd);
      if (who_locks == "") {
         error("You must set a lock for "+ artifact + " first");
         flock(lockby_fd, UNLOCK);
         close(lockby_fd);
         return NOT_LOCKED;
      }
      else if (who_locks != current_user()) {
         error(artifact + " is locked by " + who_locks);
         flock(lockby_fd, UNLOCK);
         close(lockby_fd);
         return ALREADY_LOCKED;
      }
      artifact_fd = open(artifact, O_RDWR);
      /* ... copy contents ... */
      close(artifact_fd);
      writeline(lockby_fd, current_user());
      flock(lockby_fd, UNLOCK);
      close(lockby_fd);
   }
```

Another example is the versioning schema. DVS uses a simple linear versioning schema. The code for generating a new version from a previous one looks like this:

```
dvs_checkin_newversion(string artifact) {
   int lockby_fd, artifact_fd;
   string version_tag, artifact_id;

   artifact_id = strip_version_tag(artifact);

   newart_fd = dvs_create(artifact_id + ":" + dvs_new_version(artifact));
   /* ... */
}
```

The function `dvs_new_version()` implements the versioning schema by supplying a new version tag. In DVS, version tags are integer numbers and this function is simply an increment. However, a new and more complex versioning schema could be plugged in simply by re-implementing that function.

4.6 NUCM interface functions

The functions described in this section constitute the application programming interface of NUCM. As we said, these interface functions provide a set of user-level primitives for the implementation of the NUCM file system. Most of these functions operate *together* or *through* the real file system of the client machine. This means that, for example, `nc_open` implements the equivalent of the NUCM-client `open()` function by creating an *image* of the artifact in the local file system.

Table 3 presents these functions 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. Below we introduce, per class, the individual interface functions that constitute the programmatic interface to a NUCM distributed repository.

4.7 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.

Access functions	
`nc_open`	Materializes an artifact in a workspace.
`nc_close`	Removes an artifact from a workspace.
Versioning functions	
`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 functions	
`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 functions	
`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 functions	
`nc_destroyversion`	Physically removes a version of an artifact from a repository.
Distribution functions	
`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 functions	
`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 3: NUCM Interface Functions.

4.8 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`.

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 4.9). 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 4.10 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.

4.9 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`.

4.10 Attribute Functions

Virtually every configuration management system attaches a certain amount of meta-data to the artifacts that it maintains. These meta-data 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 meta-data 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 meta-data 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 [32] and CCC/Harvest [28]. 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 [13]) 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.

4.11 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.

NUCM also provides an automatic garbage collection mechanism for artifacts that are no longer referenced. An artifact is referenced as long as at least one of its versions is referenced in a collection. Root collections are always referenced by definition.

4.12 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.

4.13 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 the 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 DVS Functions

DVS is a command line tool. It provides a set of commands that give access to its various functionalities. There are thirteen basic functionalities provided by DVS. They are *check-out*, , *check-in close*, *link*, *unlink*, *lock*, *unlock*, *list-collection*, *print-version-log*, *set-version-log*, *print-locks*, *whatsnew*, and *sync*. Most of these functions operate on a list of *artifacts*, each one identified by a *path* expression (see Section 4.1).

Check-out populates the local workspace by retrieving artifacts from the repository while *check-in* can be used to register new artifacts or new versions of existing artifacts. When checking out an artifact a lock on that artifact can be set by the check out function, conversely, the check in function removes the lock. These functions can be applied to both atoms and collections. In case of collections, they can also execute recursively on the artifacts of that collection. *Close* removes an artifact from the local workspace.

Link (and *unlink*) can be used to incorporate (or remove) existing artifacts into (from) a collection. These commands are necessary for creating data models that share artifacts among collections like the ones displayed in Figure 3.

Lock and *unlock* directly manipulate locks for artifacts that have been already checked out. Locks in DVS are defined on a per-user basis. These operations are idempotent and they are each one the inverse of the other one.

List-collection, *print-log*, *print-locks*, and *whatsnew* are utility functions that can be used to examine the state and the contents of collections and atoms in the workspace or in the repository. The most interesting one is *whatsnew* that lists all the artifacts that have newer versions. This function can be applied recursively to collections.

Sync is another useful function that checks for new versions of artifacts. If one artifact in the workspace is obsolete, i.e., if there is a newer version of that artifact in the repository, *sync* checks out that version. *sync* too can be used recursively on collections.

A detailed description of the DVS commands in the form of a user manual can be found in Appendix A.

6 Experience in Using DVS and NUCM

This section refers to the experience we gained in using DVS and NUCM and the lessons learned thereof. In particular, the experience highlights some of the benefits of the solutions implemented by DVS and NUCM, but also uncovers their limitations and suggests design and implementation improvements.

6.1 Use of NUCM

Apart from the realization of DVS, NUCM has been used as a basis for the implementation of two other CM systems. One system is an experimental implementation of a subset of WebDAV, an emerging standard in Web versioning [36]. The other system is SRM [34], a system that implements a distributed repository of software releases. SRM is currently in use as the release manager of the Software Engineering Research Laboratory at the University of Colorado at Boulder, and at the Software Engineering Institute to support the release of software within the Evolutionary Design of Complex Systems (EDCS) project, a project funded by DARPA involving several research groups from both academia and industry. A more detailed evaluation of NUCM is discussed in [33].

6.2 Experience with DVS

DVS has been used in some real scenarios as well. The most significant ones are two collaborative authoring efforts involving each one seven authors distributed across sites in the United States including University of Colorado at Boulder, University of California Irvine, Northrop Grumman Corporation, Aerospace Corporation, and University of Hawaii. During these projects, more than thirty different documents in three collections with a total of over 300 versions were created and managed by DVS.

In general, the usage experience of DVS proved a good reliability of DVS and of the NUCM server and a discrete usability of DVS as a CM tool. Some criticisms have also been made and some new requirements have been posed. To improve DVS apart from some routine bug fixing, during the first project, some counter-intuitive behaviors have been modified and also some new features have been implemented mainly to provide utility functions that automate common tasks that previously required combined DVS and file-system operations.

6.3 Lessons learned from DVS and improvements to DVS and NUCM

DVS, both from its usage and in its implementation, provided useful feedback to the design of NUCM. Here we examine a set of requirements that stem from designing and using DVS. These requirements can be sufficiently general for other CM systems and other policies. Here we list these requirements paying special attention to the impact they have on the data model and/or on the overall architecture of NUCM:

Storage policies NUCM is designed to be a policy-neutral platform, however, while the majority of the policy programming can be confined to the client and the workspace, there are a number of policies that heavily affect the storage of artifacts and thus the server side. In particular, in a CM system that stores text files, it is desirable to have an incremental storage of consecutive versions of artifacts. This can be achieved with NUCM by storing deltas and relating versions with additional attributes. This would allow to assemble any requested version on the client side. Unfortunately this solution has two major shortcomings:

1. in case the artifact resides on a remote server, the network traffic generated by a simple request could be heavy because the client has to assemble a succession of deltas, each one requiring a separate network access;

2. being implemented on the client side, this storage policy would undermine the orthogonality between versioning and access. In other words, another client would have to implement the same change set assembly to retrieve any version of an artifact.

This problem suggests that delta storage be implemented on the server side, perhaps as an optional feature that can be selected on a per-artifact basis.

Server-side *plug-ins* In general, it would be good to have a generic mechanism to augment the capabilities of NUCM servers by plugging additional modules into the server rather than programming them on the client side. A first example is the delta storage proposed in the previous paragraph. Another example is a compression module that could condense the data transferred over the network between clients and servers. A similar module could add security features to the server such as secure connections with public/private key authentication and with data encryption.

Piggy backing on HTTP or other protocols NUCM primitives use an ad hoc communication protocol. This protocol has been designed with two requirements in mind:

1. the protocol must be efficient in transferring big files, thus encoding should be avoided or it should be minimal. Also, a protocol similar to FTP is preferable as opposed to some sort of RPC-based protocol.[3]

2. the protocol should use only one port. In any case it is not appropriate to use dynamically allocate ports on the server side. This requirements is intended to meet the restrictions posed by firewalls.

In essence, the NUCM protocol is very simple and, thanks to its simplicity, it proved to be reasonably efficient and open. The performances of a basic check out operation are comparable to a plain FTP of the file. As far as openness, the protocol has been seamlessly extended to accommodate new primitive calls.

Nevertheless, it would be desirable to have the option of piggy backing NUCM requests on top of other standard protocols, typically HTTP. The main reason to make this request is that many sites are closed to the rest of the net for security reasons and the only protocols that are not blocked are HTTP and SMTP.

We have not studied in details this requirement, but from a first analysis, it seems that implementing NUCM requests on top of HTTP would not affect the NUCM data and distribution models.

[3] An early implementation of NUCM that was based on CORBA showed that, while providing a nice abstraction, CORBA proved to be quite heavyweight for requests that are mostly transfers of files

Packing requests into batch jobs When programming CM policies with the NUCM primitives, one single operation may very likely result in several consecutive and related NUCM requests. Since every request involves a separate network message sent to a NUCM server. In case these requests are applied to remote artifacts, the network latency could seriously affect performances of the CM client.

To a certain extent, network delays can not be completely avoided, however there are several techniques that can be proficiently applied to improve performances of CM clients that access distributed artifacts. One idea is to use replication (caching) of artifacts. Another approach is to move artifacts near their clients dynamically. Both these solutions have been studied extensively in similar domains. Caching and dynamic relocation of objects have been implemented for distributed file systems and for Web objects that, in many respects, pose challenges similar to NUCM. However, both these solutions imply a major re-design of the NUCM server and the client library.

As a relatively simple improvement of the NUCM protocol, we could pack series of related requests in batch jobs. In a way, this solution corresponds to the *Keep-Alive* extension of the HTTP protocol [11] that allows several separate requests and their responses to flow through the same TCP/IP connection. With this type of modification, we pay the price of adding a minimal encoding to requests and we gain in performances for long series of requests.

Utility functions for the server A minimal set of utilities for the NUCM server should be provided. It should include a dump/restore facility possibly accessible remotely. NUCM already supports some utility functions: one is similar to `ping` and can be used to test is a NUCM server is running correctly and accepting requests. The other one is a remote shutdown of the server. New administrator functions should be provided. We already mentioned a dump/restore function, but also other functions that check the status of the server more in details, e.g., listing the active client connections, possibly allowing the administrator to kill stale connections. The same way, similar function could allow the administrator to change server parameters on the fly.

Library for CM clients Programming CM clients based on NUCM showed that many NUCM primitives requires low level operations that should be handled by utility client libraries. Typical functions manipulate NUCM paths, in fact, DVS implements a whole set of path manipulation utilities that extract the base name from a path, strip the version tag, extract the server part if one exists, etc.

Other useful functions are iterators over collections. It is a common task to iterate through all the elements of a collection executing some specific function. The iteration part that reads the elements of a collection could be easily set apart as a library routine. To a certain extent, it would also be nice to have a set of pre-programmed policies in the form of library functions.

It is desirable that these all function be available to CM system programmers in the form of an extended NUCM client library.

Finer access control The current implementation of the NUCM server offers an access control list to protect the server from unwanted access. This mechanism allows to control access to a NUCM server on a per-host basis. The list can contains expressions matching a single host ID (one IP address) as well as ranges of IP addresses and entire subnets. Examples of entries in the access list are:

```
128.138.242.69
131.175.70.100-200
131.175.21.*
128.138.240-241.*
```

The access list can be modified dynamically.

This access control mechanism offers a rudimentary protection. In all the scenarios we used NUCM, this has been sufficient. However, it is known to suffer from two major drawbacks:

- it is vulnerable to IP spoofing attacks
- it is too coarse grained

A finer and more robust access control should be provided by NUCM servers. As an initial cut, users identifiers and per user permissions could be added, then of course, the system could be extended in the direction of the capabilities of the access control system, for example, by adding per-artifact access lists and classes of user profiles. Another direction is the authentication system that could be initially done in a simple UNIX-like style with user name and password. Subsequent refinements could introduce public/private key authentication protocols.

7 Conclusions

The field of configuration management is well established as a software engineering discipline. In this phase, configuration management research results have been consolidated and transferred to commercial products. However, the connectivity offered by wide-area networks and the distributed nature of many development or editing processes have introduced new challenges and opportunities for new research projects. The work described in this paper is certainly stimulated by these new technologies and scenarios.

In particular, current CM systems lack a native support for distribution of artifacts and in general they are either heavyweight and expensive or simplistic in their data model and their policy. Hence, in short, our goal is to develop *lightweight CM systems that support distributed development and authoring with flexible, easy to program, CM policies*.

The approach we follow is therefore twofold:

- we realize a low-level CM repository that implements a distributed generic data model;
- we implement different high-level policies with different CM clients on top of the generic repository.

In this paper we presented NUCM, a low-level generic distributed CM infrastructure, together with DVS, a CM system that implements a strict locking policy. With NUCM, we describe its *CM file system* abstraction and its core functionalities. Of DVS we illustrate the data model, the distribution model, and the detail of its CM operations.

Here we also report our experience in using NUCM for the implementation of some CM systems (DVS being the main one) and in using DVS in some real distributed authoring efforts. Our general conclusion is that separating a policy-neutral CM platform from the policy implementations is a good approach for supporting flexibility and distribution. This experience also highlights some shortcomings of our implementation that need better engineering and optimization.

Future developments could pursue two main directions. We could work on improving both NUCM and DVS following some of the guidelines presented in this paper. Alternatively, we could expand the set of policies implemented on top of NUCM. For example, a natural candidate would be a CM system based on change sets. More experience would give us more arguments and confidence in favor of the NUCM approach. Pushing this idea a little further we could come up with a library of pre-packaged possibly orthogonal libraries of CM policies.

Both DVS and NUCM are available for download free of charge. Additional information, installation and usage instructions, and binary packages for UNIX (Sun, DEC, HP, and Linux) and Win32 (95/98/NT) can be found on [6].

Acknowledgments

This study has been carried out together with André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf at the Department of Computer Science of the University of Colorado at Boulder. In particular, the design and implementation of NUCM 2 is the result of a joint effort with André van der Hoek.

I wish to thank Prof. Vincenzo Piuri of Politecnico di Milano for his comments and suggestions about this work and for his advises during the writing of this paper.

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, 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, 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 home page. Available on the WEB at `http://www.elet.polimi.it/~carzanig/dvs` and at `http://www.cs.colorado.edu/~carzanig/dvs`.

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

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

[9] 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, 1994.

[10] P. 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] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Barnes-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet Request For Comments (RFC) 2068, Internet Engineering Task Force, January 1997.

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

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

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

[15] J. Hunt, F. Lamers, J. Reuter, and W. 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, 1997. Springer-Verlag.

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

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

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

[19] 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, Nov. 1988.

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

[21] 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, 1997. Springer-Verlag.

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

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

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

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

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

[27] 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, 1995. Springer-Verlag.

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

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

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

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

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

[33] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. L. Wolf. A Reusable, Distributed Repository for Configuration Management Policy Programming. Technical Report CU-CS-864-98, Department of Computer Science, University of Colorado, September 1998.

[34] A. van der Hoek, R. Hall, D. Heimbigner, and A. 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, Sept. 1997. Springer-Verlag.

[35] A. van der Hoek, D. Heimbigner, and A. 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, Mar. 1996.

[36] E. Whitehead, Jr. World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction. *StandardView*, 5(1):3–8, Mar. 1997.

A DVS Manual

DVS is a command line tool. It provides a set of commands that give access to its various functionalities. When invoked, DVS accept one command as a first parameter. Some of the commands also accept additional parameters. The list of parameters, if anyone exists, precedes the list of artifacts to which the command applies. The syntax for DVS is the following:

`dvs` ⟨*command*⟩ [*options*] [*artifacts*]

This section describes the functionality of DVS by listing its commands and their options in the form of a practical user guide.

A.1 Default Server

For all the operations that require to connect to a NUCM server, DVS uses either the server that is specified in each *path* or the *default server*. The host name for the default server is specified by the `NUCMHOST` environment variable. If `NUCMHOST` is not defined, `localhost` is used. Similarly, the port number is specified by the `NUCMPORT` variable the default value being `1234`.

A.2 Commands

There are thirteen basic commands. They are `co` (check-out), `ci` (check-in), `close`, `link`, `unlink`, `lock`, `unlock`, `list` (list collection), `log` (print versions log), `setlog`, `printlocks`, `whatsnew`, and `sync`.

co [`-R`] [`-l`] [`-f`] [`-last`] *path* [*path* ...]

check out artifacts. `co` checks out *path* from the NUCM repository. If *path* is an atom, the corresponding file is created in the current working directory. If *path* is a collection then a directory is created. If *path* is a collection and the `-R` option is specified, then every element of *path* is recursively checked-out.

If the `-l` option is specified, DVS tries to lock *path* and prepares it for change. If `-l` is specified on a collection together with `-R`, only that collection is locked while all its components are checked out read-only.

If the a version of *path* other than the one determined by *path* is already present in the workspace, `co` prints a warning message and prompts for closing it. The `-f` option forces DVS to close all the artifacts that conflict with the check out operations.

When trying to resolve a *path* for check-out, DVS automatically determines the version number in case it is not already specified in the *path*. The following criteria apply: if the current working directory corresponds to a *collection* that contains *path*, the version referred by that *collection* is checked out. If the current working directory does not correspond to any *collection* or if it corresponds to a *collection* that does not contain *path*, then the *last* version of *path* is checked-out. If the `-last` option is specified, the last version is checked out regardless of the content of the current collection. If `-last` is specified together with `-R`, the `-last` flag is applied recursively to all the artifacts that are checked out.

In all the check-out operations, if *path* can not be resolved and if it does not contain a reference to a NUCM server, DVS automatically looks up the *default server*.

`ci` [`-R`] [`-m` *`message`*] *path* [*path*...]
check-in artifacts. `ci` checks in each *path* in the database.

`ci` has two slightly different semantics depending on the fact that *path* is already linked to the current working collection or not.

If *path* is already referenced in the current directory, a new version is created and checked in, provided that the current version is locked.

If *path* is not already linked to the current working collection, DVS creates a new artifact in the database and adds it to the current version of the current collection. To do so, such a collection must exist and it must be locked. The initial version number of an artifact is 1. Only in this case, if the `-R` option is specified and *path* is a directory, all the sub-tree of files in *path* is recursively checked in.

Note that checking in the first version or a new version of an artifact does not imply a check in of the *collection* (or *collections*) that refers to the previous version of that artifact. Collections have to be explicitly checked in.

Checking in a directory (i.e., a *collection*) causes that *collection* to be updated with all the version that are currently present in the file system. Checking in a collection also requires that collection to be locked.

In any case, for every artifact that is checked in, DVS prompts for a log message to be attached to the new version. If the `-m` option is specified, *message* is used as a log message for every check-in operations.

After a check-in is completed, the previous version is automatically unlocked. Note that if the artifact is new, then the collection that contains it is not automatically checked in and unlocked.

`close` [`-f`] *path* [*path*...]
closes each *path*. Closing a *path* means removing the file or the directory and all its content from the local workspace.

`close` fails whenever *path* or an artifact contained in *path* has been checked out with lock. Also, in case *path* is a directory, DVS checks for the presence of other non-dvs files in that directory, and, in case, it prompts the user for a confirmation before proceeding to remove them. In these cases, the `-f` option can be used to force DVS to `close` *path* anyway.

Note that `close` has no effect on the database, i.e., it does not commit changes to the NUCM database nor it unlocks artifacts.

`link` [`-last`] *path* [*path*. . .]
: links each *path* to the current collection, i.e., it makes *path* a member of the current collection. If *path* doesn't refer to an artifact that exists in the workspace or if the `-last` option is specified, the latest version is linked. Otherwise, the version referred in the workspace is linked. `link` is an operation that affects the current collection, thus, the current collection must be locked. Similarly to `ci`, `link` does not implicitly check in the current collection. *path* must be an existing artifact.

`unlink` [`-f`] *path* [*path*. . .]
: removes *path* from the current collection. Similarly to `link`, `unlink` requires the current collection to be locked. *path* is not actually deleted from the database as long as there exist at least one version of a collection that refers to any version of *path*.

`unlink` has the same effect of `close` on the workspace, i.e., `unlink` removes *path* from the workspace unless some *path* or any artifact contained in *path* has been checked out for change. If *path* is a collection, DVS also checks that *path* does not contain any other non-dvs files that would be removed together with *path*. In both these cases, `-f` forces the execution of the command.

`lock` [`-last`] *path* [*path*. . .]
: locks each *path* and prepares it for change. The locks are assigned on a per version basis, i.e., two users can lock two different versions of the same artifact at the same time. `lock` fails when *path* is already locked by another user. If *path* does not specify a version, the version referred in the workspace is locked. If the `-last` option is specified then the latest version of *path* is locked, regardless of the version specified in *path* or the one referred in the workspace.

After setting the lock information, *path* is prepared for change, i.e., it is made user-writable. `lock` fails if *path* is not open in the workspace.

Locks are based on the identity of the user that executes DVS. By default, the login name is used, however, if the `USERADDRESS` environment variable is set (e.g., to the e-mail address), its value is used to mark locks instead of the login name.

This variable is provided in case the login name is not adequate. For example, when one user has different login names on different machines or when two different users can have the same login name.

Note that DVS can not control the authenticity of `USERADDRESS`.

`unlock` [`-last`] [`-f`] *path* [*path*. . .]
: removes the lock from *path* and reverts changes restoring the version of *path* that was previously in the workspace. `unlock` fails if *path* is locked by another user or if *path* is not open in the workspace. Similarly to `lock`, `unlock` determines the version from *path* if it is explicitly specified or from the current collection or it uses the latest if the `-last` option is specified.

In restoring the previous version, DVS closes the artifact that is currently in the workspace, thus, similarly to `close`, `unlock` fails whenever *path* or an artifact contained in *path* has been checked out with lock. Also, in case *path* is a directory, DVS checks for the existence of other non-dvs files in that directory, and, in case, it prompts the user for a confirmation before proceeding to remove them. In these cases, the `-f` option can be used to force DVS to `close` *path* anyway.

`list` [`-v`] [`-o` *`filter`*] *path* [*path*...]
lists the contents of *path*. `list` fails if *path* is not a collection. By default, `list` produces a list of artifact names, but if `-v` is specified, DVS outputs a verbose list containing version information. Here is an example of a verbose output:

```
% dvs list -v .
! A.bod v.3                          3   (2)
  B.bod v.1                          1   (1)
- figure.eps                         -   (4)
  paper.tex v.9                      9   (9)
+ paper.bib v.2                      2   (-)
* paper.ps                           -   (-)
```

The first number on the right side is the version that is currently in the workspace (*path*). The number in parentheses is the version that is referenced in the collection (*path*).

Files marked with '-' are those that are referenced in the collection, but are not present in the workspace; '+' indicates that the artifact is in the workspace, but it is not referenced in the collection; '`*`' marks non-dvs files, and '`!`' says that the version in the workspace doesn't match the one in the collection.

By default, all these artifacts are listed, however a *filter* (`-o` flag) can be used to list just one or more of these classes. In particular, *filter* is a string of one or more of the following characters:

`c` (*collection*) artifacts that are members of *path*

`w` (*workspace*) artifacts managed by DVS that are in the workspace

`b` (*both*) artifacts managed by DVS that are members of the collection defined by *path* and that have been checked out into the workspace

`o` (*other*) files that don't correspond to artifacts managed by DVS.

`log` [`-n` *num*] *path* [*path*...]
prints the log of the versions of *path*. The log reports the list of versions for each path *path*. For each version, the creation date, the author, lock information, and the log message are displayed.

DVS prints the log for all the version of *path* in reverse order, i.e., from the latest version (highest version number) to the first version. `-n` can be used to limit the log output to the *num* most recent versions.

setlog [-m *message*] *path* [*path*...]
: sets the log message for each *path*. If the -m option is specified, *message* is used as a log message for every *path*. Otherwise DVS prompts for a log message for every *path*.

printlocks [-R] [-v] *path* [*path*...]
: prints the lock information for all the versions of *path* that are locked. If *path* is a collection and the -R option is specified, *path* is recursively checked for locks. By default, this command outputs only *paths* that have at least one locked version. The -v option can be used to produce a more verbose output with all the versions that are locked and the name of the users that hold the locks.

whatsnew [-R] [-v] *path* [*path*...]
: prints out *path* if the version referenced in the workspace does not correspond to the latest version checked-in in the repository. If *path* is a collection and the -R option is specified, then all the components of *path* are recursively scanned for new versions. The -v option causes DVS to print out the current version and the latest version for each *path*.

sync [-R] [-f] *path* [*path*...]
: this command is useful to refresh the workspace with the latest versions of the artifacts. `sync` checks whether the version of *path* in the workspace is actually the latest version of *path* in the repository. In case the workspace contains an earlier version, DVS checks out the latest version. With -R, if *path* is a collection/directory, DVS scans *path* recursively refreshing all its components. The effect of refreshing an artifact is identical to a 'co -last', thus, `sync` uses the -f flag to force the check-out operation, in case it would overwrite other versions already checked out for changes or in case it would remove other non-dvs files.