# A SECURE MULTI-SITED VERSION CONTROL SYSTEM

Indrajit Ray and Junxing Zhang
*Computer Science Department, Colorado State University*

**Abstract:** The software development process is increasingly taking the form of a collaborative effort among several teams which are hosted at widely dispersed sites networked over the Internet. In this model of software development, multi-sited version control systems play a very important role to maintain the revision history of software and facilitate software evolution. In this paper we look into the security requirements of such multi-sited version control systems. We identify the security deficiencies in current systems and propose a new framework for secure multi-sited version control.

**Key words:** revision control, versions, software development, collaboration, integrity

## 1. INTRODUCTION

A version control system is used to maintain the revision history of software and facilitate software evolution. When software evolves it produces many revisions. A version control system stores these revisions, organizes them into meaningful structures that conform to software development principles, and provide operations to work with the different versions so that the integrity of the different versions of the software is independently ensured. A multi-sited version control (MVC) system is a version control system that operates at multiple sites to coordinate the software development effort of several teams that are collaborating with each other to develop a single piece of software. Even at the same geographical location such a system is useful to allow independent groups to work with the same development data, to enable interoperation in a heterogeneous environment, or to be a backup mechanism.

Version control systems were being used even when software development was primarily an individual effort. During that time the security of such systems was not of a major concern. This was because the data in the system was not shared. Later when version control systems became "team-ware" to share data among team members, security became somewhat more important. Still it was not considerably so; teams were located at the same site and the data sharing happened over a company's proprietary network. Since multi-sited version control system appeared, security concerns have become critical. This is because the data in the system now needs to be shared among multiple sites that are connected by open networks. Development teams may want to protect proprietary technologies from competitors. More important perhaps, is the need to ensure the integrity of the different revisions submitted over the Internet by different teams and the non-repudiation of their origin. This is particularly true about in the Open Source community. The various teams within this community generally do not care about the confidentiality of their software due to their commitment to the open source model; however they do need to ensure the integrity and non-repudiation of their code, and their developers do submit the contributed code via Internet.

In this work we propose a new secure multi-sited version control (MVC) system that can be easily implemented using COTS components. We utilize the Directory Information Tree (DIT) structure of X.500 directories [1, 2] to represent the MVC data model, network model and user account management model. Next we propose a novel authorization scheme for the MVC system that is based on integrating the access control list model [13] with the role-based access control model [12]. To support this new access model we propose to extend the Lightweight Directory Access Protocol (LDAP) [8, 9, 10, 15]. Finally we utilize the Simple Authentication and Security Layer (SASL) protocol [7] to solve the authentication problem and protect data confidentiality and integrity in the MVC system.

The rest of the paper is organized as follows. In section 2 we discuss some of the more well-known multi-sited version control systems to identify their security deficiencies. Section 3 develops the model for our multi-sited version control system. Section 4 describes our MVC framework. Finally we conclude in section 5.

## 2. RELATED WORK

There are a number of version control systems available today both commercial as well as open-source. The most widely used among them are ClearCase [14] (by the erstwhile Rational Software Corporation now owned

by IBM), the UNIX Source Code Control System (SCCS) [16], the GNU Revision Control System (RCS) [17] and the Open Group's Concurrent Versions Systems (CVS) [11]. These differ from in each other mostly in terms of functionality and convenience. However, none of these systems were explicitly designed with security in mind. Consequently each of them have at least one of the following security deficiencies – (i) authentication deficiencies, (ii) authorization flaws, (iii) confidentiality and integrity problems and (iv) user account management problems – and cannot be used, without consequences, as multi-sited version control systems. In the following we discuss these deficiencies in more details.

**Authentication Deficiencies** - Most MVC systems use the native operating systems' authentication mechanisms for local access. This causes a dependency on the operating systems. Also remote access authentication varies among different systems. ClearCase doesn't have any support for remote access authentication, so do not SCCS and RCS. CVS supports several remote authentication mechanisms. Some of them are rather weak for authentication over open networks – such as connecting with *rsh* (which requires a machine at one site completely trust other machines at other sites) and simple password based authentication. Other remote authentication mechanisms that CVS supports, like authentication with GSSAPI [5] and Kerberos [6], are considerably stronger but are more suitable for authentication within a single administrative domain than over the Internet.

**Authorization Flaws** – We categorize authorization flaws into (a) missing dimensions, (b) encapsulation failures, (c) inconsistent controls, (d) write and checkout problems and (e) authorization coordination failure.

- Missing Dimensions – The most common authorization flaw is the authorization granularity is not as fine as the granularity of accessible objects. For example, CVS doesn't have access control at the version tree layer at all although it does support branches. ClearCase has controls at all three layers, but it doesn't control access at the version level.
- Encapsulation Failure – Many systems require users to check the permissions of the internal devices. SCCS, RCS, CVS users need to check the permission of data repository directories and even their subdirectories. ClearCase users need to check the permission of virtual workspace devices; in many cases these devices are not on the same machines where workspaces are used. Because these systems take internal attributes as the external attributes, users have to understand their internal mechanisms. This often confuses them and makes systems hard to use.
- Inconsistent Controls – Some systems are inconsistent in controlling different permissions. ClearCase doesn't control read and execute

access at branch level, but it does control the write access (checkout, checkin, and uncheckout) at the branch level.

- Write and Checkout Problems – Some systems (ClearCase, for example) do not treat the write permission in the same way as the checkout permission; instead they use the checkout permission to replace the write permission. The problem with this approach is that users cannot determine from the access control lists of files in the virtual workspace, who can change the content of the files or directories (elements). This is against the design purpose of using the virtual workspace to simulate the work environment in a file system.
- Authorization Coordination Failure – Some systems (CVS, for example) do not have an authorization coordination mechanism; so they use one central data repository to represent the whole repository family. This approach can't support local access model. Since every operation has to be done remotely, it incurs lots of resource overhead. ClearCase has the coordination scheme, but because it doesn't support remote access model it can only use the passive access mode of the scheme. More details will be discussed in the proposed framework.

**Confidentiality and Integrity Problems** – None of the four systems encrypt the messages exchanged in any manner by default for remote access. CVS has the provision for encrypting messages but does not do this without special configuration. This is a serious security problem; not only is the data in the messages endangered, attackers can also plant malicious code or data in the packets to damage the confidentiality and integrity of the importing data repositories.

**User Account Management Concern** – All current systems rely on the operating system to provide the network-wide database of user and group names. This causes a dependency on the operating system and undermines interoperability if the system needs to run in a heterogeneous environment.

## 3. MODELS OF THE MVC SYSTEM

The proposed MVC system is defined in terms of four different parameters (i) the data model, (ii) the network model, (iii) the access model and (iv) the user management model. We discuss each in details beginning with the data model.

**The Data Model** – The MVC system data model defines how the data is stored in the system. The data model defines four components – (i) data elements, (ii) data repository, (iii) data repository family and (iv) virtual workspace. We use the X.500 directories [1] to represent the data model.

A *data element* is a unit of data that is stored in the system. It can have different granularities. The smallest data element in the version control system is a *version.* A version is a particular revision of a file or directory. Versions of one line of development form a linear sequence called a *branch.* Branches are used to separate different development efforts and allow parallel development. For example, one branch may contain all the versions used to add a new feature to the software; another branch may be composed of the versions contributing to a software bug fixing. Branches of the same file or directory are organized into a version tree. Every tree has one main branch (called main), which represents the principal line of development. Files and directories are called *elements.* Unlike in a file system file and directory elements in a version control system are not flat. They have the version tree structure. Like in a file system, however, file and directory elements are also located somewhere in a directory tree.

A *data repository* is used to store different versions of files and directories. It also holds the derived data and meta-data associated with them. Data repositories at multiple sites form a *data repository family.* The data repository family stores data relevant to a single project – that is, the data elements in the repository family are all semantically related.

A *virtual workspace* is an environment where users can have access to a set of versions selected from the data repository. The versions are selected via a user-defined filter, which is a part of the environment. Most virtual workspace is used by one user for individual development. Some are shared by several users for integration or integration testing.

The data model is illustrated by an example in figure 1. In this figure, the directory element "/home/junxing" has one branch and two versions on this branch, 0 and 1. Version 1 contains a file element named "Hello.c". This element has four branches: main, bug102, feature23 and Linux_port. Each of its branches has several versions to store the revision history of a separate development line. As the diagram shows in the MVC system each element is represented as a version tree no matter it is a directory or file. The contents of directory versions indicate the sub directory and file elements they include. The directory tree is organized in this way just as it is designed in a file system.

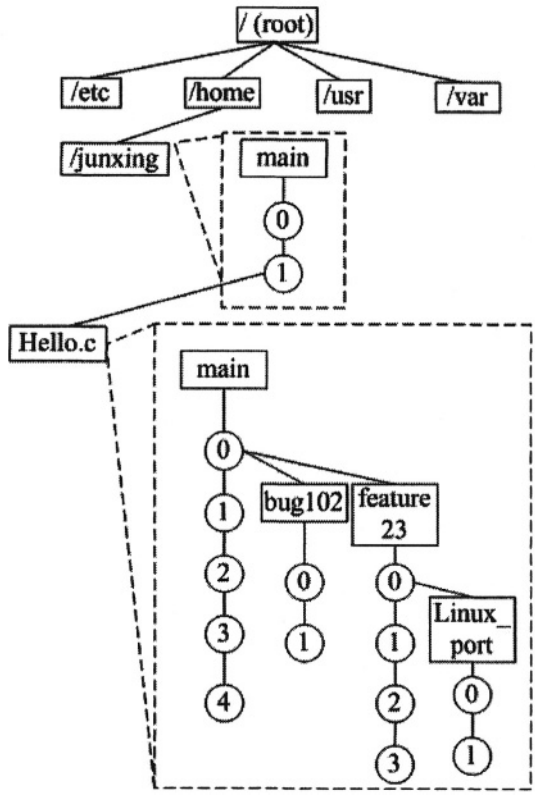


*Figure 1.* Data Model for the MVC System

**The Network Model** – The MVC system network model defines who accesses and/or manages the different versions of the data that are stored in the entire system. It comprises of *servers* and *clients* connected over a local area network at *local sites,* a number of which participate in the same MVC system.

We define a *local site* to be one that is responsible for one and only one data repository. The MVC system comprises of a number of such local sights. From the point of view of a local site, other local sites are called *remote sites.* Each local site consists of one server machine and several client machines that are connected in a local area network. A local site is under a single administrative control. The server at the local site maintains the data repository. The clients manage the virtual workspaces and accept and respond to users' requests. The clients also communicate with local or remote servers to retrieve, add, delete or update versions, meta-data and/or derived data.

A specific data repository at a server holds only one copy of the data. This copy has the latest local revision changes. However, it may not have the latest, up-to-date revision changes from other sites. In order to get the latest changes, repositories must be synchronized with each other. To synchronize the data at the local site with data at remote sites, each server communicates periodically with servers at remote sites. We adopt a peer-to-peer model for such server-server communication.

The network model for the MVC system is shown in figure 2.

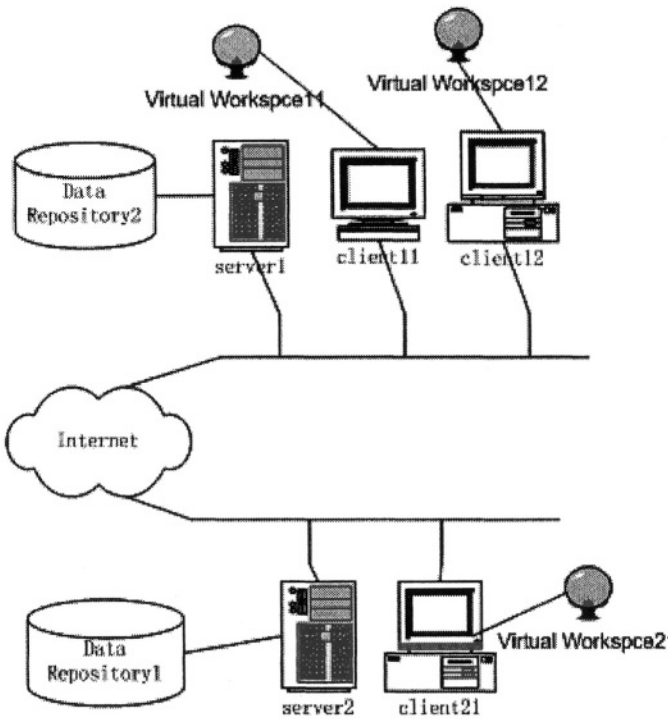


*Figure 2.* Network Model for the MVC System

**The Access Model** – The access model defines how the MVC system controls access to different versions in the data repository family. The access is of two types – local access and remote access. If a client contacts a local server, then this is a local access. For a local access a user must access the versions in a data repository via a virtual workspace. She needs to specify which versions of the files she wants to access. This is done by defining rules in the filter of a virtual workspace. We use a SQL like syntax for defining such rules. An example of such a rule on the data model shown in figure 1 is as follows.

SELECT Hello.c /main/LATEST

This rule will select the version numbered 4 (the latest version) on the main branch of Hello.c.

We define a *well-formed rule* to be one that retrieves only one version of any element. The virtual workspace is usable only after such well-formed rules are defined. When the user requests an operation on a file or directory element, the client that manages the workspace contacts a server to apply the operation to the version of the element selected by the filter. If the user wants to operate on a different version or on more than one version, she explicitly gives the version name in the request.

Sometimes the local server does not have the permissions to execute certain operations on the specified version. In this case a remote server has to be contacted; either the client can directly contact the remote server that has the permission to complete this request, or the client can request the local server to contact the relevant remote server. Now a server typically has to handle multiple client requests. So it is more efficient for a local server to gather together multiple client requests to access a remote server, and schedule it in a batch mode than to handle each client request as it comes. One the other hand, if we allow a client to directly access the relevant remote server then the client will be serviced promptly. For this reason we choose to have the client contact the remote server directly in the MVC system. This is termed as *remote client-server access.* The only time a local server needs to communicate with remote servers is to synchronize data in the containing repositories. This is called *remote server-server access.*

Access to different versions is achieved by executing a set of atomic operations. These are (i) get, (ii) checkout, (iii) check-in, and (iv) un-checkout.

- Get – Get is the action of getting the copy of a version from the data repository to a virtual workspace.
- Checkout – Checkout is the action of locking a branch in the data repository for adding a new version. This prevents other users from checking the same branch out, though they are not restricted from getting versions in the branch.
- Check-in – Check-in is the action of adding a new version to the branch locked by a checkout and then releasing the lock.
- Un-checkout – Un-checkout is the action of releasing the lock created by a checkout without adding a new version.

Check-in, checkout and un-checkout all are essentially write transactions. Checkout starts the transaction, check-in commits it, and un-checkout rolls back the transaction. The definition of checkout operation implies that a branch cannot be checked out by a user if it has already been checked out by

another one. However, the data model of the repository family indicates that the local repository may not know if a branch has been checked out at other sites because it does not have the latest revision changes at other sites. This means the authorization need be coordinated among sites. A check-in or un-checkout operation is based on a previous checkout operation. The user who can check in or un-checkout a branch must be the one who checked it out. Optionally the system may allow others who are in charge of the branch or its contents, to execute the operation when the user is unavailable.

Checkin, checkout and un-checkout all require permissions similar to write permissions. The execute permission has additional connotation when it is applied to the objects in a MVC system. The execute permission of directory elements, file elements and branches is the permission to list or search their included objects, while the execute permission of versions is to run operations defined in owners' data. This is because owners' data is contained in files in a file system, but it is kept in versions in a MVC system.

A user is authorized at three layers in order to access a particular version. The first layer consists of virtual workspaces. Here the user needs permissions to operate on the temporary copies of versions she selected from the data repository. The second layer is the directory tree. Here she needs the permission to operation on the directory structure. The third layer is the version tree. Here she needs the permission to access a particular version on a specific branch, which presents a development line.

The authorization granularity is decided by the granularity of accessible objects. In the workspace layer the accessible objects are just virtual workspaces. In the directory tree layer the accessible objects are the data repository and elements (directories and files). In the version tree layer the accessible objects are branches and versions.

Finally, although in the local access model the MVC system may not need strong encryption mechanisms to ensure data confidentiality, they are needed for integrity and non-repudiation of data origin; strong cryptographic techniques are indispensable in the remote access model.

**The User Account Management Model** – This model defines how user account information is maintained across the entire MVC system. Every user in the system belongs to a primary site. All changes to user account information are made at the server at the primary site. To meet the requirements of parallel development at multiple sites, the account information at the primary site must be available at all sites and be consistent. This is achieved by a remote server-server access.

# 4. THE PROPOSED MVC SYSTEM

The proposed MVC system is developed by adopting and/or extending widely industry standards and COTS components.

We use X.500 directories [1] to represent the data model, network model and user account management model of the MVC system.

2. We extend the Lightweight Directory Access Protocol (LDAP) [8, 9, 10] to support the access model of the MVC system. In particular, we develop the get, checkin, checkout and un-checkout operations as extensions to LDAP.
3. We develop a novel authorization scheme for the MVC system by borrowing from and integrating the concepts of access control lists, role-based access control and the concept of mastership.
4. We adopt the Simple Authentication and Security Layer (SASL) [7] protocol to address the authentication problems in the MVC system. This protocol also helps to protect data confidentiality and integrity in the MVC system by virtue of its built in strong cryptographic techniques.

In the following we briefly discuss how we implement each of the above modules of the MVC system.

## 4.1 X.500 schemas for Data, Network and User Account Management Models

The MVC system stores information about version trees; thus it is natural to use another tree structure to implement the system. We use the Directory Information Tree (DIT) of X.500 directories to represent the system tree structure. The structure is defined X.500 schemas according to RFC 2252 specifications, which is based on BNF (Backus-Naur Form meta-language). For lack of space we show the declarations for only the security related elements. Since X.500 is used, each schema element for a project must have a globally unique Object Identifier (OID)[1]. For this discussion, the OIDs under 1.1 are used. This is in keeping with the conventions of ASN.1. Each element has at least one textual name. To reduce the potential for name clashes, the name prefix "mvc" is used in the convention. The following is an example of these definitions for a project eduColostate.

**objectIdentifier** eduColostateOID 1.1 ; Organization OID

[1] OIDs are basically strings of numbers. They are commonly found in protocols described by ASN.1. The formal definitions of OIDs come from ITU-T Rec. X.208 (ASN.1). OIDs are allocated in a hierarchical manner and managed by the IANA. For private experiments and research purposes the OIDs under the 1.1 branch are typically used.

**objectIdentifier** mvc eduColostateOID: 1 ; Name prefix and Application OID
**objectIdentifier** mvcObjClass mvc: 1 ; MVC object class

**The Data Model** – The data elements, data repository, data repository family and virtual workspace are all defined as object classes in X.500 directories. The type definition of the data element *version* is given in the following schema. Its object Id is derived by appending 1 to the object Id of "mvcObjClass". Its name is "mvcVersion". It has a text description and inherited from the superior object class "top". Each instance of this class must have five attributes: "mvcUserId", "mvcGroupId", "mvcUserPermission", "mvcGroupPermission" and "mvcOtherPermission". These attributes are introduced in the authorization scheme section. Other data model components can be defined similarly. For lack of space we omit them from here.

```
objectclass
  (
    mvcObjClass:1
    NAME 'mvcVersion'
    DESC 'Data Version Type'
    SUP top
    MUST  (
                mvcUserId $
                mvcGroupId $
                mvcUserPermission $
                mvcGroupPermission $
                mvcOtherPermission
            )
  ); Version Class
```

**The Network Model** – To conform to the MVC network model, each MVC site should have one server sub tree and multiple client sub trees. The server sub tree is composed of data model components: version, branch, element, data repository and data repository family. The client sub tree consists of one root element and several workspace elements. The root element is used to organize workspace elements and identify the client. Its type doesn't concern the MVC security, so the definition is not given here.

**The User Account Management Model** – User account management service required by MVC systems are represented using user schemas defined in RFC 2256. Since these are very well defined and standardized we do not discuss them here.

## 4.2 LDAP Extensions to Support the Access Model

The MVC framework uses X.500 directories to represent the MVC system structures. LDAP is the access protocol to X.500 directories. Thus it is reasonable to support the access model of the MVC system by extending LDAP in the secure framework. Another advantage gained by extending LDAP is the convenient access to SASL. Since SASL is the association security services of LDAP, the framework can use it directly instead of building a new one from scratch.

The MVC framework uses a special operation in LDAPv3 support the extension. This operation is defined in order to allow additional operations to be defined for services not available elsewhere in LDAP protocol. It is described in ASN.1 (Abstracted Syntax Notation 1) [3, 4], and is typically transferred using a subset of ASN.1 Basic Encoding Roles. The special operation that we use is as follows.

```
ExtendedRequest ::= [APPLICATION 23]
SEQUENCE
  {
    requestName [0] LDAPOID,
    requestValue [1] OCTET STRING OPTIONAL
  }

ExtendedResponse ::= [APPLICATION 24]
SEQUENCE
  {
    COMPONENTS OF LDAPResult,
    responseName  [10] LDAPOID OPTIONAL,
    response  [11] OCTET STRING OPTIONAL
  }
```

The MVC access model operations get, checkout, check-in and un-checkout, are implemented using the above LDAP extended operation definition.

## 4.3 Authorization Scheme for the MVC System

Problems with authorization are often caused by the intention to reuse the authorization scheme of a native operating system. To avoid them, we provide the MVC system with its own authorization scheme. Due to the system's special requirements, this scheme combines three authorization

mechanisms: ACL (Access Control List), RBAC (Role Based Access Control) and mastership.

**Access Control List** – ACL is used as the basic authorization mechanism. There are two reasons to use ACL. Firstly, MVC systems are object centralized and subject distributed systems. Secondly,it makes simulating file systems easier. Most file systems use ACL, so if the MVC system also uses ACL it will be easier for it to translate the permissions of a specific version in the data repository into the permissions of the corresponding file in the virtual workspace. ACL is defined as attribute types in the following LDAP/X.500 schema. In the first line of the schema the object Id of "mvcAttType" is created by appending 2 to the object Id of "mvc". There are two attribute types. One is used to represent the user Id, and another is for the user permission. Their object Ids are created by extending the Id of "mvcAttType". They use the matching rule of numeric strings. They have the syntax of integers. And they may only hold one single value. Similarly the group Id, group permission and other's permission can be defined. These definitions consist of an ACL implementation in X.500 directories.

```
objectIdentifier mvcAttType mvc:2 ; MVC attribute type
attributetype
(
        mvcAttType: 1
        NAME 'mvcUid'
        DESC 'User Id'
        EQUALITY numericStringMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        SINGLE-VALUE
)   ; User Id Type

attributetype
(
        mvcAttType:2
        NAME 'mvcUperm'
        DESC 'User Permission'
        EQUALITY numericStringMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
        SINGLE-VALUE
)   ; User Permission Type
```

In the data model design, every object class has mandatory attributes of ACL attribute types. This means that every object in the MVC system is in

ACL control, so the system's access control granularity is as fine as the accessible objects.

**Role Based Access Control** – ACL is not enough to meet the MVC system's authorization requirements. Operations such as checkin and un-checkout require the system to check if the requestor has certain relationship with the user who executed the corresponding checkout operation. This kind of requirements calls for Role Based Access Control. This mechanism makes use of the attribute types and object classes defined in the schema below.

```
attributetype
(
        mvcAttType:6
        NAME 'mvcOperationId'
        DESC 'Operation Id'
        EQUALITY numericStringMatch
        SYNTAX
1.3.6.1.4.1.1466.115.121.1.27
        SINGLE-VALUE
); Operation Id Type
```

```
attributetype
(
        mvcAttType:7
        NAME 'mvcUids'
        DESC 'User Id List'
        EQUALITY numericStringMatch
        SYNTAX
1.3.6.1.4.1.1466.115.121.1.27
); User Id List Type
```

```
objectclass
(
        mvcObjClass:6
        NAME 'mvcOperationRoles'
        DESC 'Roles can execute the
Operation'
        SUP top
        MUST mvcOperationId
); Roles allowed for an operation
```

```
objectclass
(
        mvcObjClass:7
        NAME 'mvcRoleUsers'
        DESC 'Users in the Role'
        SUP top
        MUST mvcUids
); User Ids in a role
```

The above types and classes are used to build RBAC trees. A RBAC tree is a directory information sub tree that has only a root and leaves. The root is an entry of " mvcOperationRoles" type. It has an "mvcOperationId" attribute that identifies the operation. The leaves are entries of "mvcRoleUsers " type. Each of them has an "mvcUids" attribute that includes all user Ids in the role. RBAC trees are used to implement RBAC. One operation such as checkout creates and inserts a RBAC tree to the checked out branch. Another operation such as checkin or un-checkout examines the RBAC tree in the target branch. If there is no such tree or the requestor Id can't be found in the

tree, the operation fails; otherwise the operation is executed and the RBAC tree is deleted.

**Mastership** – is introduced in the MVC system to coordinate the authorization among multiple sites. This is needed because changes made at different sites can potentially conflict when they are imported into one data repository; the authorization must be coordinated to prevent conflicts from happening. Mastership is "site based access control". It ensures that only one site in a repository family has the permission to change a controlled object (e.g. branch) at any given time. This exclusive permission to modify ensures that no parallel changes can be made to controlled objects; thus conflicts are avoided. Users at sites that do not have mastership-permission of an object have two access modes to make changes. The active mode allows one to modify the object using remote access model at the site that has the permission. The passive mode allows one to request the mastership-permission to be transferred to the local site. This scheme is made possible by the following attribute types and object classes:

**attributetype**
(
mvcAttType:8
NAME 'MVC Mastership'
DESC 'Master Site Id'
EQUALITY numericStringMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
SINGLE-VALUE
); A type used to define attributes that identify the sites that have the mastership-permissions of the controlled objects

**objectclass**
(
mvcObjClass:8
NAME 'MVC MastershipObject'
DESC 'Mastership Container'
SUP top
MUST MVCMastership
); A class used to define entries that hold the attribute of the previous type.

## 4.4 Authentication, Confidentiality and Integrity Measures

Simple Authentication and Security Layer protocol is used to provide both strong authentication mechanisms needed in the remote access model and regular mechanisms required in the local access model of the MVC system. To ensure secure remote access, the MVC system imposes the following requirements to LDAP configuration: Remote MVC operations are allowed only when SASL mechanisms whose Security Strength Factor (SSF) are greater than 1, are in effect. Remote MVC operations are allowed only when mechanisms that are able to negotiate a security layer are in force. To ensure secure remote access, the mechanisms that support the security layer negotiates a privacy protection layer after the successful authentication for remote MVC operations. There are no mandatory configuration requirements for the local access model.

# 5. CONCLUSION

In this paper we look into multi-sited version control systems. Such systems are very important in distributed collaborative environments where many development teams work together on a large software system and the system goes over several versions before being finally available. We identify the security requirements of multi-sited version control systems in general. We observe that it is more important that such systems ensure the integrity of the software versions rather than their confidentiality. We then identify the security deficiencies in current implementations of multi-sited version control systems. We propose a new framework for secure multi-sited version control. The specific contributions include:

- Using X.500 directories and extending LDAP v3 to provide version control service at multiple sites.
- The application of SASL to improve the security of MVC systems.
- The application of the concept of mastership in the remote access model so that conflicts due to concurrent access is eliminated.
- An authorization scheme combining ACL, RBAC and mastership.

The paper also uses the LDAP extension for Internet Domain Name service as the reference for extending the protocol.

## REFERENCE:


[1] ISO/IEC 9594-1, "X.500 The Directory: Overview of Concepts, Models and Services", International Standards Organization, 1993.

[2] ISO/IEC 9594-2, "X.501 The Directory: Models", International Standards Organization, 1993.

[3] ITU-T Rec. X.680, "Abstract Syntax Notation One (ASN.1): Specification of Basic Notation", International Telecommunication Union, 1994.

[4] ITU-T Rec. X.690, "Specification of ASN.1 Encoding Rules: Basic, Canonical and Distinguished Encoding Rules", International Telecommunication Union, 1994.

[5] J. G. Meyers, "Simple Authentication and Security Layer (SASL)", RFC 2222, Network Working Group, The Internet Society, October 1997.

[6] M. Wahl, "A Summary of the X.500(96) User Schema for Use with LDAPv3", RFC 2256, Network Working Group, The Internet Society, December 1997.

[7] M. Wahl, A. Coulbeck, T. Howes and S. Kille, "Lightweight Directory Access Protocol (V3): Attribute Syntax Definitions", RFC 2252, Network Working Group, The Internet Society, December 1997.

[8] M. Wahl, T. Howes and S. Kille, "Lightweight Directory Access Protocol (v3)", RFC 2251, Network Working Group, The Internet Society, December 1997.

[9] P. Cederqvist, et al. "Version Management with CVS", The Open Group, Available from http://www.cvshome.org/docs/manual/cvs-1.11.6/cvs.html

[10] R. Sandhu, "Role-Based Access Control", in Advances in Computers, volume 48, M. Zerkowitz editor, Academic Press, 1998.

[11] R. Sandhu and P. Samarati, "Access Control: Principles and Practice", IEEE Communications, 32(9), 1994.

[12] Rational Software Corporation, "Rational® ClearCase Multisite® Documentation", October 2001.

[13] S. Kille, M. Wahl, A. Grimstad, R. Huber and S. Sataluri, "Using Domains in LDAP/X.500 Distinguished Names", RFC 2247, Network Working Group, The Internet Society, January 1998.

[14] Unix SCCS, "Source Code Control System", The Regents of the University of California, 1986.

[15] W. F. Tichy, "RCS – A System for Version Control", Software – Practice and Experience, 15(7), 1985.