LDAP Stored Procedures and Triggers in ApacheDS

Ersin Er

The Apache Software Foundation
&
Department of Computer Engineering, Hacettepe University

ersiner@apache.org & ersin.er@cs.hacettepe.edu.tr

Abstract

LDAP directories lack stored procedure and trigger facilities which have been provided by relational database management systems for many years. The need for these rich integration tier constructs drives information architects towards the use of relational databases for managing centralized information that would have best been served by directories. A novel model for specifying stored procedures and triggers in LDAP directories is presented. Rather than introducing incompatible changes to the protocol, these features were designed by making use of LDAP extension points. LDAP stored procedures allow users to define their own server side routines and to call them via an LDAP extended operation. LDAP triggers raised by standard LDAP operations invoke LDAP stored procedures. Triggers can be defined on individual entries or on sets of entries using subtree specifications based on the X.500 administrative model adopted by LDAP. The proposed models have been realized and tested within the Apache Directory Server.

1. Introduction and Motivation

X.500 [1] and LDAP [4] directory specification designers primarily focused on the use of directories to serve static white pages. As read optimized, non-transactional stores, some sophisticated facilities of relational database systems were overlooked and not included in their specifications. The main strengths of directories center around data distribution and replication for high availability and load balancing since as centralized services, directories potentially impose a single point of failure. Today, the use of LDAP has exceeded its foreseen purpose; it plays a key role in enterprise integration and provisioning. LDAP directories centrally store and serve configuration information, enterprise user profiles and inventory information. This way, directories provide a common centralized view of information needed by several systems. This common view facilitates interoperability across these systems yet it also leads to the need for provisioning resources as new shared information is added, changed or removed from directories. Multiple systems within an enterprise may need to respond to directory system events where users are added or new inventory items are introduced.

Stored procedures and triggers are popular facilities present in most relational database systems. Stored procedures enable the storage of data manipulation code close to server side data and provides a way to run them within the server for better performance. Triggers allow defining custom actions to be taken upon the occurrence of certain events and generally are used to satisfy referential integrity. In its current role, LDAP would greatly benefit from these features and would in turn better meet enterprise requirements. The need for these features has driven many architects towards the use of relational databases for managing centralized information that would have best been served by directory systems.

This paper describes a novel model for specifying stored procedures and triggers in LDAP directories. Rather than introducing incompatible changes to the protocol, these features were designed by making use of LDAP extensions. LDAP stored procedures allow users to define their own server side routines that can be called via an LDAP extended operation [6]. LDAP triggers raised by standard LDAP operations invoke LDAP stored procedures. Triggers can be defined on individual entries or on sets of entries using subtree specifications based on the X.500 Administrative Model [2] adopted by LDAP [7].

2. LDAP Stored Procedures

2.1. Abstract Representation of LDAP Stored Procedures

Basically an LDAP stored procedure is represented with several custom schema elements and stored in Directory Information Tree (DIT) as any other data. An entry which stores an LDAP stored procedure should have the objectClass value storedProcUnit whose definition is given in Figure 1.

```
Objectclass( storedProcUnit_oid
    NAME 'storedProcUnit'
    SUP top
    ABSTRACT
    MUST ( storedProcLangId $ storedProcUnitName ) )

attributetype ( storedProcLangId_oid
    NAME 'storedProcLangId'
    EQUALITY caseExactIA5Match
    SYNTAX IA5StringSyntax_oid
    SINGLE-VALUE )

attributetype ( storedProcUnitName_oid
    NAME 'storedProcUnitName'
    EQUALITY caseExactIA5Match
    SYNTAX IA5StringSyntax_oid
    SINGLE-VALUE )
```

Figure 1.

The object class is named as storedProcUnit instead of storedProc, because many current programming languages support composition of more than one procedure into a single programmatic unit. For example, in the Java language, procedures are composed into classes and there is no way to store a single procedure alone into a compilable source file.

All LDAP stored procedures are associated with a language identifier. Using this information, the server can automatically involve the appropriate language specific means to invoke a stored procedure.

A stored procedure entry contains only the name of the stored procedure unit, not names of all stored procedures stored in the unit. Name resolution for stored procedures should be handled by language specific engines upon invocation requests.

2.2. Representation of LDAP Java Stored Procedures

As ApacheDS is a pure Java based LDAP server, the initial choice of language for stored procedures was Java. An LDAP Java stored procedure is a static method of a compiled Java class. Schema elements required for representing an LDAP Java stored procedure are given in Figure 2.

```
objectclass ( javaStoredProcUnit_oid
    NAME 'javaStoredProcUnit'
    SUP storedProcUnit
    STRUCTURAL
    MUST ( javaByteCode ) )

attributetype ( javaByteCode_oid
    NAME 'javaByteCode'
    SYNTAX binarySyntax_oid
    SINGLE-VALUE )
```

Figure 2.

The only added attribute for an LDAP Java stored procedure is javaByteCode which is proposed to store compiled bytecode form of a Java class. A single attribute is enough for storing even more than one LDAP Java stored procedure. Any language specific implementation can define more than one attribute to represent its stored procedures. storedProcUnitName inherited from storedProcUnit object class is used for holding the fully qualified name of the Java class stored.

LDAP stored procedures are not different from any other DIT data for their representation but they need to be handled specially for invocation purposes. Invocation models of LDAP stored procedures are explained in the following sections.

2.3. Generalized Invocation of LDAP Stored Procedures by External Clients

An LDAP stored procedure can be invoked in two ways: via an external call to the server or by server side means. For calling an LDAP stored procedure externally a simple LDAP extended operation was defined with the components shown in Figure 3.

```
StoredProcedureExecutionRequestValue ::= SEQUENCE {
    name           IA5String,
    parameters     SEQUENCE OF Parameter OPTIONAL }

Parameter ::= SEQUENCE OF {
    type       OCTET STRING OPTIONAL,
    value      [0] OCTET STRING }

StoredProcedureExecutionResponseValue ::= SEQUENCE {
    returnType     OCTET STRING OPTIONAL,
    returnValue    [0] OCTET STRING OPTIONAL }
```

Figure 3.

An LDAP stored procedure execution request contains the name of the stored procedure to be invoked and parameters to be passed to the stored procedure. Name of the stored procedure must be given in a special format where name of the unit and actual stored procedure are separated with a semi-column. Using the part before the semi-column, the server can locate the stored procedure and determine the appropriate language specific handler. The remaining part after the semi-column should be decoded by the language specific engine. All details of parameter encoding are subject to the specifications of the implementation language. The type component of Parameter is optional because some language environments provide run time type information capabilities. When the type information can be gathered from the value itself the server does not need such an external information to locate the correct stored procedure to execute. In all cases value and type information should be compatible with the target stored procedure. Implementations may provide "overloading" like capabilities but still a call should exactly match with one stored procedure.

The actual location of the stored procedure to be invoked is left as an implementation preference for

vendors. It may be a configurable setting for users.

Invocation of stored procedures by local, server side, means is out of scope of this section. Information on integration of LDAP stored procedures and LDAP triggers, which also serves as an example of execution of stored procedures by local means, are given in the following sections.

2.4. Invocation of LDAP Java Stored Procedures by External Clients

A language specific implementation of LDAP stored procedure execution framework is responsible for decoding name and parameters and invoking the stored procedure. In necessary cases, the execution engine should also forward the value returned by the stored procedure to the caller.

For LDAP Java stored procedures name component of StoredProcedureExecutionRequestValue is a string which specifies a fully qualified name of a Java class and name of a static method in it. An example is given in Figure 4.

com.example.ldap.util.sp.DITUtilities:deleteSubtree
Figure 4.

As Java provides run time type information via reflection, types of arguments are not necessary to call a stored procedure. All arguments are sent as serialized Java objects. After decoding the name and parameters for the request, the LDAP Java stored procedure execution engine should locate the correct stored procedure and invoke the stored procedure again via reflection. This scheme also allows overloading of LDAP Java stored procedures.

The implementation for LDAP Java stored procedures in ApacheDS also provides a special parameter type where the caller specify the distinguished name of an directory entry and the server injects a handle (an LdapContext object) on that entry during the stored procedure call process. While many languages are expected to provide such a data structure which serves as a handle on a directory entry, this feature can be defined within the generalized model too.

storedProcLangId attribute of an entry containing an LDAP Java stored procedure must be 'Java'. This identifier should be unique among all language specific languages. If another implementation is developed in order to handle LDAP Java stored procedures in a different way, its language identifier should be something else.

2.5. LDAP Java Stored Procedures in Action

This section states a problem which cannot be easily solved with current tools and gives a solution using LDAP Java stored procedures as implemented in ApacheDS.

Considering the LDAP Delete operation, [6] states the following restriction:

> *“Only leaf entries (those with no subordinate entries) can be deleted with this operation”.*

However it may be needed to delete a subtree completely at once. A delete operation control for complete subtree deletion was proposed [9] but it has never been adopted by the mainstream. So if a user needs such a feature he/she will have to wait the developers to implement it.

A (administrator type) user can write his/her own LDAP stored procedure to meet this requirement and execute it using the extended operation defined in previous sections. An example Java stored procedure for this task and a sample driver program to demonstrate its usage are given in Figure 5 and Figure 6 respectively.

```
/* Imports are not shown. */
public class DITUtilities
{
    public static Integer deleteSubtree(
        LdapContext ctx,
        Name rdn ) throws NamingException
    {
        Integer count = 0;
        int ctxSize = new LdapDN( ctx.getNameInNamespace() ).size();
        NamingEnumeration<SearchResult> results =
            ctx.search( rdn, "(objectClass=*)", new SearchControls() );
        while ( results.hasMore() )
        {
            SearchResult result = results.next();
            Name childRdn = new LdapDN( result.getName() );
            for(int i = 0; i < ctxSize; i++ )
            {
                  childRdn.remove( 0 );
            }
            count += deleteSubtree( ctx, childRdn );
        }
        ctx.destroySubcontext( rdn );
        count++;
        return count;
    }
}
```

Figure 5.

```
/* Imports are not shown. */
public class JavaStoredProcDemo
{
    public static void main( String[] args )
    {
        LdapContext ctx = new LdapContext( "dc=example,dc=com );
        JavaStoredProcUtils.loadStoredProcUnit(
            ctx,
            DITUtilities.class );

        String spName = DITUtilities.class.getName() + ".deleteSubtree";
        Object[] spParams = new Object[] {
            new LdapContextParameter( "dc=example,dc=com" ),
            new LdapDN( "ou=People" ) };

        JavaStoredProcUtils.callStoredProc( ctx, spName, spParams );
    }
}
```

Figure 6.

Execution of this stored procedure deletes all entries subordinate to "ou=People,dc=example,dc=com". The stored procedure loading phase could be done via any LDAP client. This example just uses some abstractions provided by ApacheDS for developers.

3. LDAP Triggers

3.1. Representation and Execution Semantics of LDAP Triggers

An LDAP trigger is a specification of an action, particularly an LDAP stored procedure, raised by a standard LDAP operation at a certain respective time. LDAP triggers can be defined on individual entries or on sets of entries which are subject to the LDAP operations.

An LDAP Trigger can be specified with four basic components:

- Triggering Event
- Triggered Action
- Action Time (with respect to the Event time)
- Trigger Scope

Triggering Event for LDAP Triggers is a change inducing LDAP operation, one of Modify, Add, Delete and ModifyDN. The ModifyDN operations is handled as three distinct sub-operations, Export, Import and Rename. These are the more granular operations defined in [3]. Triggered Action is an LDAP stored procedure. The only supported Action Time can be identified with AFTER keyword which means triggered actions will be executed just after the triggering event ends.

Trigger Scope can be either a single entry or a set of entries. The trigger scope as a set of entries, Trigger Execution Domain, is provided via the powerful X.500 Administrative Model and will be discussed in the following sections in detail.

When the scope of a trigger is a single entry, the trigger can be specified in the entry as a regular LDAP attribute with a special syntax. entryTriggerSpecification attribute located in an entry makes that entry a candidate for trigger execution process. A simple example of a trigger specification is given in Figure 7.

```
AFTER Delete
    CALL "com.example.ldap.util.sp.BackupTools:backupDeletedUserEntry"
        ( $deletedEntry );
```

Figure 7.

A delete operation on an entry containing the trigger specification given in Figure 6 will cause a trigger to be executed. The activated trigger will call the specified LDAP stored procedure while injecting the specified operation specific argument as a parameter to the stored procedure. Operation specific parameters for all change inducing LDAP operations are provided with meaningful names (with names in the LDAP Protocol specification where appropriate) to be used in stored procedure calls in trigger specifications. Some generic parameters like '$operationPrincipal', '$ldapContext "<DN>"' (which was discussed in the previous sections) are also provided to make working with triggers and stored procedures easier and more valuable. List of operation specific parameters cover all the standard parameters as well as some parameters for convenience like '$deletedEntry' for Delete operation and '$oldEntry' / '$newEntry' for Modify operation.

Being able to provide operation specific and generic parameters to stored procedures is a powerful aspect of triggers and stored procedures integration. Encoding of these parameters are subject to the specification of the stored procedure language. Again the location of stored procedures is also left as an implementation preference to LDAP servers. It should be noted that this integration model is not built on top of model of Invocation of LDAP Stored Procedures by External Clients defined in previous sections but they share similar semantics. The trigger execution framework is supposed to invoke stored procedures without going over the wire.

Specifying more than one stored procedure call in one trigger is possible which provides guarantied order of execution of different stored procedures upon an event.

3.2. Trigger Execution Domains

One of the most sophisticated features of the Trigger Execution Model is the ability to specify sets of entries as the scope of triggers and to manage them efficiently. To achieve this capability the model makes use of the X.501 Administrative Model as recently been adopted by LDAP via [7]. Subentries are special entries which are direct subordinates of administrative entries which are again special entries. Administrative entries include an attribute named administrativeRole which defines the implicit administrative areas starting from that entry as their root. Subentries subordinate to such administrative entries affect entries in their administrative areas with their associated administrative aspects. Trigger Execution is proposed to be such an administrative aspect. The X.501 Access Control Model was also taken as a reference while many similarities exist between these two aspects of administration. Details of the X.501 Administrative Model and Access Control model are defined in [2] and suggested to be read for better understanding inner workings of Trigger Execution.

A few schema elements shown in Figure 8 are defined to support Trigger Execution on top of the X.501 Administrative Model. ApacheDS provides a powerful infrastructure to add new aspects of administration based on the X.501 Administrative Model.

```
objectclass ( triggerExecutionSubentry_oid
    NAME 'triggerExecutionSubentry'
    AUXILIARY
    MAY prescriptiveTriggerSpecification )

attributetype ( prescriptiveTriggerSpecification_oid
    NAME 'prescriptiveTriggerSpecification'
    EQUALITY directoryStringFirstComponentMatch
    SYNTAX triggerSpecificationSyntax_oid
    USAGE directoryOperation )
```

Figure 8.

A trigger execution subentry with objectClass value triggerExecutionSubentry is effective on a Trigger Execution Domain defined by the subtreeSpecification attribute. subtreeSpecification attribute is required in all subentries and it provides a grammar to specify a portion of the DIT with LDAP semantics. The prescriptiveTriggerSpecification has the same syntax as the entryTriggerSpecification attribute. Both are multivalued attributes and this makes it easier to define more than one triggers for a Trigger Execution Domain. Examples of possible DIT partitioning via subtree specifications are provided in Figure 9. Although the original specifications, [2] and [7], do not allow usage of LDAP filters in subtree specifications, ApacheDS provides this as an option.

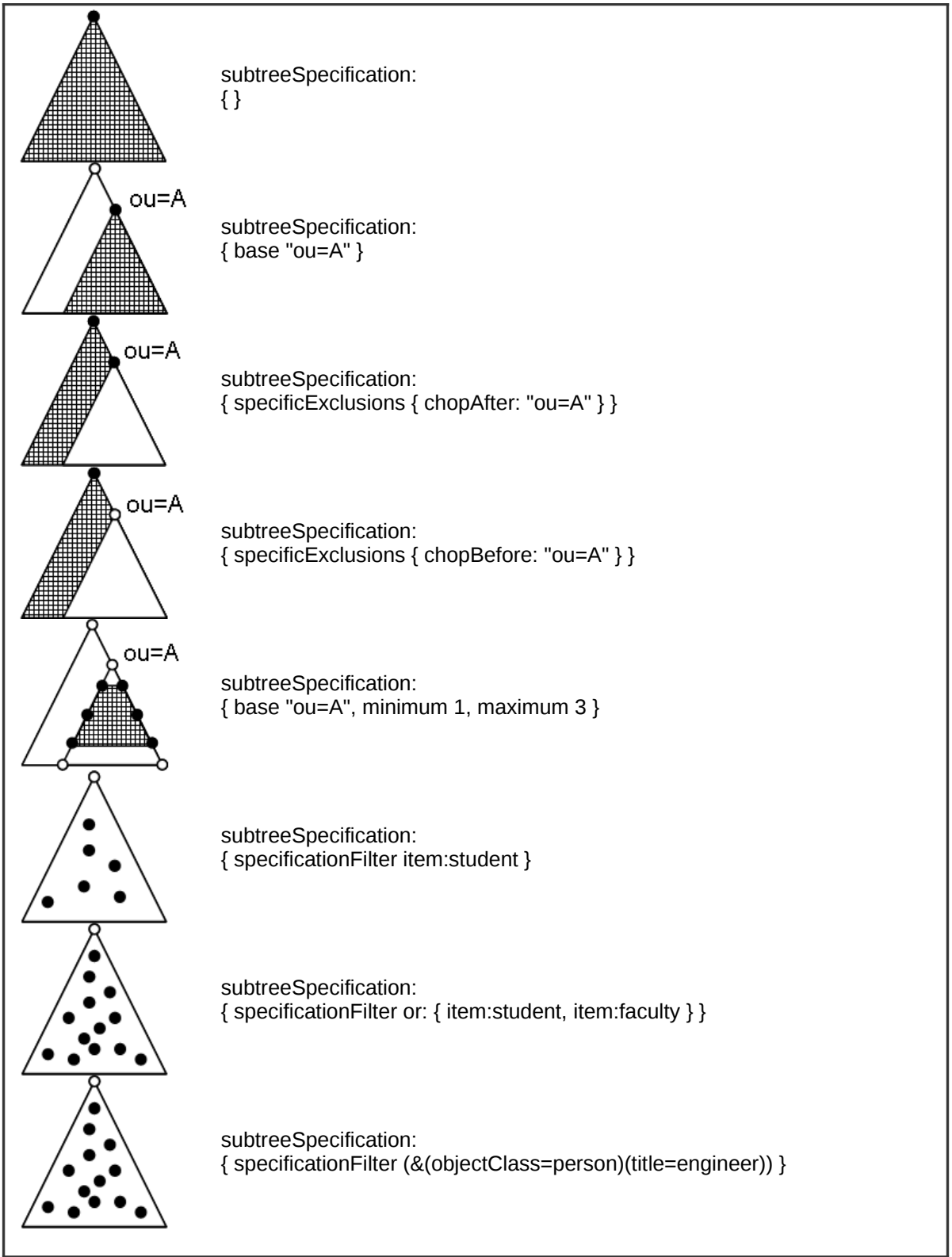

Figure 9.

3.3. LDAP Triggers with LDAP Java Stored Procedures

The integrations of generic LDAP trigger model and LDAP Java stored procedures relies in the determination of data types for the parameters passed to stored procedures. For each supported

parameter, an existing Java type was chosen in order to make the bridge between the two models. Some examples of identifier to Java type associations are shown in Figure 10. Further details for all operations and parameters can be found in [9].

Modify::modification	javax.naming.directory.ModificationItem[ ]
Add::entry	javax.naming.Name
Delete::deletedEntry	javax.naming.directory.Attributes
ModifyDN.Rename::deleteoldrdn	java.lang.Boolean

Figure 10.

3.4. LDAP Triggers with LDAP Java Stored Procedures in Action

It may be necessary to log some information upon modification of certain entries. For example entries of manager users are important for a company and it is desired to monitor all operations on those entries. Except some details two basic things are needed to achieve that goal: a stored procedure and a trigger execution subentry with the following attributes:

- subtreeSpecification attribute which selects the manager entries,
- a prescriptive trigger specification.

The signature of the stored procedure can be as shown in Figure 11.

```
com.example.ldap.util.sp.Logging:logManagerModification(
    Name operationPrincipal,
    Name object,
    ModificationItem[] modifications )
```

Figure 11.

The subtreeSpecification can be as shown in Figure 12. This subtreeSpecification uses an LDAP filter to fine tune the selection.

```
{ base "ou=People", minimum 1,
    specificationFilter
        (&(objectClass=inetOrgPerson)(title=manager)) }
```

Figure 12.

The prescriptiveTrigger can be as shown in Figure 13.

```
AFTER Modify
    CALL "com.example.ldap.util.sp.Logging:logManagerModification"
        ( $operationPrincipal, $object, $modification );
```

Figure 13.

Besides these changes, appropriate modifications should be made in order to meet some administrative model requirements. Further details are explained in [9].

4. Conclusions and Future Work

LDAP lacks rich integration tier constructs required for modern use cases. Triggers and stored procedures are useful elements which enable directories to be better utilized in environments where

enterprise integration and provisioning are critical. Although modeled with different semantics from their relational counterparts, these constructs were shown to better equip a directory so it may respond to changes and support more complex applications. Models were defined which use extension points of the LDAP protocol and elements of the X.500 Administrative model to meet these requirements while complying with directory service semantics.

Although the developed models have been defined independent of implementation technologies as much as possible, there is room for improvement. Specifically with stored procedures, callers are required to be aware of the implementation of the stored procedure. This requirement can be lifted with proper parameter to language bindings for callers so they need not be aware of what language was used to implement a stored procedure.

Research is currently being conducted regarding the utility of the current Trigger specification and whether or not it will meet the needs of users. The open source user community at Apache is providing feedback on the Trigger implementation within ApacheDS as well as the specification which will be compiled for further feedback. Extensions to the trigger implementation are also foreseeable for BEFORE and INSTEADOF Triggers in addition to AFTER triggers.

Several security considerations have come about as a result of introducing LDAP triggers and stored procedures. Stored procedures naturally bring about the need to sandbox code running within an LDAP server. The question of who can execute this code as well as what permissions it runs with are extremely relevant. Does a stored procedure execute with the permissions of the owner/creator of the procedure or does it execute with the permissions of the user that invoked or triggered it? These are valid questions to consider which where not addressed by the scope of this work. Future work may address these valid concerns which need to be resolved for real world operation.

5. References

1. International Telecommunication Union - Telecommunication Standardization Sector, "The Directory - Overview of concepts, models and services", X.500 (1993)
2. International Telecommunication Union - Telecommunication Standardization Sector, "The Directory - Models", X.501 (1993)
3. International Telecommunication Union - Telecommunication Standardization Sector, "The Directory - Abstract service definition", X.511 (1993)
4. K. Zeilenga, "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map", RFC 4510, June 2006
5. K. Zeilenga, "Lightweight Directory Access Protocol (LDAP): Directory Information Models", RFC 4512, June 2006
6. J. Sermersheim, "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, June 2006
7. K. Zeilenga, S. Legg, "Subentries in the Lightweight Directory Access Protocol (LDAP)", RFC 3672, December 2003
8. Michael P. Armijo, Tree Delete Control, Internet draft, August 2000
9. Apache Directory Server - Triggers, http://directory.apache.org/apacheds/1.5/triggers.html