Sun OpenDS Standard Edition 2.0 Architectural Reference

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–6172
July 2009

090717@22510

Contents

The Directory Server Access Control Model

This section contains reference information about the directory server access control model. For information about configuring access control in the directory server, see "Controlling Access To Data" in *Sun OpenDS Standard Edition 2.0 Administration Guide*"Controlling Access To Data" in *Sun OpenDS Standard Edition 2.0 Administration Guide*. This section covers the following topics.

Access Control Principles

This section describes the principles of the access control mechanism provided with the directory server.

Access Control Overview

When the directory server receives a request, it uses the authentication information provided by the user in the bind operation, and the access control instructions (ACIs) defined in the server to allow or deny access to directory information. The server can allow or deny permissions such as read, write, search, or compare. The permission level granted to a user might depend on the authentication information that the user provides.

Using access control, you can control access to the entire directory, a subtree of the directory, specific entries in the directory (including entries that define configuration tasks), a specific set of entry attributes, or specific entry attribute values. You can set permissions for a particular user, for all users who belong to a specific group or role, or for all users of the directory. Finally, you can define access for a specific client, identified by its IP address or DNS name.

ACI Structure

Access control instructions are stored in the directory as attributes of entries. The `aci` attribute is an operational attribute that is available for use on every entry in the directory, regardless of whether it is defined for the object class of the entry. This attribute is used by the directory server to evaluate what rights are granted or denied when the directory server receives an LDAP request from a client. The `aci` attribute is returned in an `ldapsearch` operation only if it is specifically requested.

An ACI statement includes three main parts:

Target	Determines the entry or attributes to which permissions apply.
Permission	Defines what operations are allowed or denied.
Bind Rule	Determines who is subject to the ACI, based on their bind DN.

The permission and bind rule portions of the ACI are set as a pair, also called an Access Control Rule (ACR). The specified permission to access the target is granted or denied depending on whether the accompanying rule is evaluated to be true. For more information, see "ACI Syntax" on page 9.

If an entry that contains an ACI does not have child entries, the ACI applies to that entry only. If the entry has child entries, the ACI applies to the entry itself and to all entries below it. Therefore, when the directory server evaluates access permissions to an entry, it verifies the ACIs for every entry between the one that was requested and the base of its root suffix.

The `aci` attribute is multivalued, which means that you can define several ACIs for the same entry or subtree.

You can create an ACI on an entry that does not apply directly to that entry but to some or all of the entries in the subtree below it. The advantage of this is that you can place at a high level in the directory tree a general ACI that effectively applies to entries that are more likely to be located lower in the tree. For example, at the level of an `organizationalUnit` entry or a `locality` entry, you could create an ACI that targets entries that include the `inetorgperson` object class.

You can use this feature to minimize the number of ACIs in the directory tree by placing general rules at high-level branch points. To limit the scope of more specific rules, place them as close as possible to leaf entries.

Note – ACIs that are placed in the root DSE entry (with the DN "") apply only to that entry.

Directory Server Global ACIs

You can configure access control centrally by using `dsconfig` to modify the properties of the Access Control Handler.

The following default global ACIs apply to all suffixes that are defined in the directory server because the rules do not specify a `target` expression:

```
Property   : Value(s)
-----------:-------------------------------------------------------------------
global-aci : "(targetattr!="userPassword||authPassword")(version 3.0; acl
: "Anonymous read access"; allow (read,search,compare)
: userdn="ldap:///anyone";)", (targetattr="*")(version 3.0; acl
: "Self entry modification"; allow (write) userdn="ldap:///self";),
: "(targetattr="createTimestamp||creatorsName||modifiersName||modify
: Timestamp||entryDN||entryUUID||subschemaSubentry")(version 3.0;
: acl "User-Visible Operational Attributes"; allow
: (read,search,compare) userdn="ldap:///anyone";)",
```

For more information, see "Managing Global ACIs With dsconfig" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

ACI Evaluation

To evaluate the access rights to a particular entry, the server compiles a list of the ACIs present on the entry itself and on the parent entries back up to the base of the entry's root suffix. During evaluation, the server processes the ACIs in this order. ACIs are evaluated in all of the suffixes and subsuffixes between an entry and the base of its root suffix, but not across chained suffixes on other servers.

Note – Access control does not apply to any user who has the `bypass-acl` privilege. The Directory Manager has this privilege. When a client is bound to the directory as the Directory Manager, the directory server does not evaluate any ACIs before performing operations. As a result, performance of LDAP operations as Directory Manager is not comparable to the expected performance of other users. You should always test directory performance with a typical user identity.

By default, if no ACI applies to an entry, access is denied to all users except those with the `bypass-acl` privilege. Access must be explicitly granted by an ACI for a user to access any entry in the directory. The default ACIs define anonymous read access and allow users to modify their own entries, except for attributes needed for security. For more information, see "Default Global ACIs" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Although the directory server processes the ACIs that are closest to the target entry first, the effect of all ACIs that apply to an entry is cumulative. Access granted by any ACI is allowed unless any other ACI denies it. ACIs that deny access, no matter where they appear in the list, take precedence over ACIs that allow access to the same resource.

For example, if you deny write permission at the directory's root level, none of the users can write to the directory regardless of the specific permissions you grant them. To grant a specific user write permissions to the directory, you must restrict the scope of the original denial for write permission so that it does not include that user.

ACI Limitations

Be aware of the following limitations when you create an access control policy for your directory service:

- If your directory tree is distributed over several directory servers, some restrictions apply to the keywords that you can use in access control statements. ACIs that depend on group entries (`groupdn` keyword) must be located on the same directory server as the group entry. If the group is dynamic, all members of that group must also have an entry on the directory server. If the group is static, the members' entries can be located on remote directory servers. However, you can do value matching of values stored in the target entry with values stored in the entry of the bind user (for example, using the `userattr` keyword). Access is evaluated normally even if the bind user does not have an entry on the directory server that holds the ACI.
- Access control rules are always evaluated on the local directory server. You must not specify the host name or port number of the directory server in LDAP URLs used in ACI keywords. If you do, the LDAP URL is not taken into account at all.

Access Control and Replication

ACIs are stored as attributes of entries, so if an entry containing ACIs is part of a replicated suffix, the ACIs are replicated like any other attribute.

See Also

"Managing Global ACIs With dsconfig" in *Sun OpenDS Standard Edition 2.0 Administration Guide*

ACI Syntax

ACIs are complex structures with many possible variations. The following sections describe the syntax of an ACI in detail.

ACI Syntax Overview

The `aci` attribute has the following syntax:

```
aci: (target)(version 3.0;acl "name";permissionBindRules;)
```

where:

- *target* specifies the entry, attributes, or set of entries and attributes for which you want to control access. The target can be a distinguished name, one or more attributes, or a single LDAP filter. The target is optional. When the target is not specified, the ACI applies to the entire entry where it is defined and all of its children.
- `version 3.0` is a required string that identifies the ACI version.
- *name* is a name for the ACI. The name can be any string that identifies the ACI. The ACI name is required and should describe the effect of the ACI. Although there are no restrictions on the name, it is good practice to use unique names for ACIs. If you use unique names, the Get Effective Rights control enables you to determine which ACI is in force.
- *permission* specifically states what rights you are either allowing or denying, for example read or search rights.
- *bindRules* specify the credentials and bind parameters that a user has to provide to be granted access. Bind rules can also be based on user or group membership or connection properties of the client.

You can specify multiple targets and permission-bind rule pairs. This allows you to refine both the entry and attributes being targeted and efficiently set multiple access controls for a given target, as shown here:

```
aci: (target)...(target)(version 3.0;acl "name"; permissionBindRule; permissionBindRule; ...; permissionBindRule;)
```

The following example shows a complete LDIF ACI:

```
aci: (target="ldap:///uid=bjensen,dc=example,dc=com")
(targetattr="*")(version 3.0; acl "example"; allow (write)
userdn="ldap:///self";)
```

In this example, the ACI states that the user `bjensen` has rights to modify all attributes in her own directory entry.

Defining Targets

The target identifies what the ACI applies to. When a client requests an operation on attributes in an entry, the directory server evaluates the target to see if the ACI must be evaluated to allow or deny the operation. If the target is not specified, the ACI applies to all attributes in the entry containing the `aci` attribute and to the entries below it.

The general syntax for a target is one of the following:

```
(keyword = "expression")
(keyword!= "expression")
```

where:

- *keyword* indicates the type of target. The following types of targets are defined by the keywords in Table 1:
  - A directory entry or its subtree
  - The attributes of an entry
  - A set of entries or attributes that match an LDAP filter
  - An attribute value or combination of values that match an LDAP filter
  - The scope of the ACI
  - An LDAP control
  - An extended operation
- The equal sign (=) indicates that the target is the object specified in the expression, and not equal (!=) indicates that the target is any object not specified in the expression.

  Note – The not-equal operator is not supported for the `targattrfilters` and `targetscope` keywords.

- *expression* is dependent on the keyword and identifies the target. The quotation marks ("") around *expression* are syntactically required, although the current implementation accepts expressions like `targetattr=*`. In future versions, syntax checking might become more strict, so you should always use quotation marks.

The following table lists each keyword and the associated expressions.

TABLE 1 LDIF Target Keywords

Keyword	Valid Expressions	Wildcard Allowed?
`target`	`ldap:///`*distinguishedName*	Allowed
`targetattr`	*attribute*	Allowed

TABLE 1 LDIF Target Keywords *(Continued)*

Keyword	Valid Expressions	Wildcard Allowed?
`targetfilter`	*LDAPfilter*	Allowed
`targattrfilters`	*LDAPoperation*:*LDAPfilter*	Allowed
`targetscope`	`base, onelevel, subtree, subordinate`	Not Allowed
`targetcontrol`	*oid*	Not Allowed
`extop`	*oid*	Not Allowed

Targeting a Directory Entry

Use the target keyword and a DN inside an LDAP URL to target a specific directory entry and any entries below it. The targeted DN must be located in the entry where the ACI is defined or in the subtree below the entry. The target expression has the following syntax:

```
(target = "ldap:///distinguishedName")
(target != "ldap:///distinguishedName")
```

The distinguished name must be located in the entry where the ACI is defined or in the subtree below the entry. For example, the following target can be used in an ACI on `ou=People,dc=example,dc=com`:

```
(target = "ldap:///uid=bjensen,ou=People,dc=example,dc=com")
```

The keyword `target` is optional. If it is not present, the default target for the ACI is the entry where the ACI is stored.

Note – The DN of the entry must be a distinguished name in string representation (defined in RFC 2253). Therefore, characters syntactically significant for a DN, such as commas, must be escaped with a single backslash (\). For example:

```
(target="ldap:///uid=cfuentes,o=Example Bolivia\, S.A.")
```

You can also use a wildcard in the DN to target any number of entries that match the LDAP URL. The following are legal examples of wildcard usage:

- `(target="ldap:///uid=*,dc=example,dc=com")` Matches every immediate child of the `example.com` branch entry that has the `uid` attribute in the entry's RDN, as shown in this example.

  ```
  uid=tmorris,dc=example,dc=com
  uid=yyorgens,dc=example,dc=com
  uid=bjensen,dc=example,dc=com
  ```

- `(target="ldap:///uid=*,**,dc=example,dc=com")` Matches every entry more than one level below the `example.com` branch entry that has the `uid` attribute in the entry's RDN, as shown in this example.

  ```
  uid=tmorris,ou=sales,dc=example,dc=com
  uid=yyorgens,ou=marketing,dc=example,dc=com
  uid=bjensen,ou=eng,ou=east,dc=example,dc=com
  ```

- `(target="ldap:///uid=*Anderson,ou=People,dc=example,dc=com")` Matches every entry immediately below the `ou=People` branch entry with a `uid` ending in Anderson.
- `(target="ldap:///*=*Anderson,ou=People,dc=example,dc=com")` Matches every entry immediately below the `ou=People` branch whose RDN ends with Anderson, regardless of the naming attribute.

Multiple wildcards are allowed, such as in `uid=*,ou=*,dc=example,dc=com`, which matches every entry in the `example.com` tree whose distinguished name contains the `uid` and `ou` attributes in the specified positions.

To Target Attributes

In addition to targeting directory entries, you can also target one or more attributes, or all but one or more user attributes, that occur in the targeted entries. This is useful when you want to deny or allow access to partial information about an entry. For example, you can allow access to only the common name, surname, and telephone number attributes of a given entry. Similarly, you can deny access to sensitive information such as personal data.

If no `targetattr` rule is present, no attributes can be accessed by default. To access all user attributes, the rule must be `targetattr="*"`. Operational attributes must be explicitly named.

The targeted attributes do not need to exist on the target entry or its subtree, but the ACI applies whenever they do. The attributes you target do not need to be defined in the schema. The absence of schema checking makes it possible to implement an access control policy before importing your data and its schema.

To target attributes, you use the `targetattr` keyword and provide the attribute names. The `targetattr` keyword uses the following syntax:

```
(targetattr = "attribute")
(targetattr != "attribute")
```

You can target multiple attributes by using the `targetattr` keyword with the following syntax:

```
(targetattr = "attribute1 || attribute2 ... || attributeN")
(targetattr != "attribute1 || attribute2 ... || attributeN")
```

For example, to target an entry's common name, surname, and uid attributes, you would use the following:

```
(targetattr = "cn || sn || uid")
```

To target all of an entry's user attributes, except `carlicense`, you would use the following target:

```
(targetattr != "carlicense")
```

The preceding example does not return operational attributes.

Targeted attributes include all subtypes of the named attribute. For example, `(targetattr = "locality")` also targets `locality;lang-fr`. You can also target subtypes specifically, for example, `(targetattr = "locality;lang-fr-ca")`.

You can use a wildcard as a stand-alone character in a `targetattr` rule (such as `targetattr="*"`), but this use is discouraged because it serves no particular purpose and can have a negative performance impact.

To Target an Entry and Attributes

By default, the entry targeted by an ACI containing a `targetattr` keyword is the entry on which the ACI is placed. That is, if you apply the ACI `aci: (`*targetattr* `= "`*uid*`")(`*accessControlRules*`;)` to the `ou=Marketing, dc=example,dc=com` entry, then the ACI applies to the entire Marketing subtree. However, you can also explicitly specify a target using the target keyword, as shown in the following example:

```
aci: (target="ldap:///uid=*,ou=Marketing,dc=example,dc=com")
(targetattr="uid") (accessControlRules;)
```

The order in which you specify the target and the `targetattr` keywords is irrelevant.

To Target Entries or Attributes Using LDAP Filters

You can use LDAP filters to target a set of entries that match certain criteria. To do this, use the `targetfilter` keyword with an LDAP filter. The ACI applies to all entries that match the filter at the level of the target DN and in the subtree below it.

The `targetfilter` keyword uses this syntax:

```
(targetfilter = "LDAPfilter")
```

where *LDAPfilter* is a standard LDAP search filter. For more information about filter syntax, see "LDAP search filter" in *Sun OpenDS Standard Edition 2.0 Glossary of LDAP and Directory Terminology*.

For example, suppose that all entries representing employees have a status of salaried or contractor and an attribute representing the number of hours worked, as a percentage of a full-time position. To target all the entries representing contractors or part-time employees, you could use the following filter:

```
(targetfilter = "(|(status=contractor)(fulltime<=79))")
```

The Netscape extended filter syntax is not supported in ACIs. For example, the following target filter is not valid:

```
(targetfilter = "(locality:fr:=<= Qu?bec)")
```

Target filters select whole entries as targets of the ACI. You can associate the `targetfilter` and the `targetattr` keywords to create ACIs that apply to a subset of attributes in the targeted entries.

The following LDIF example allows members of the Engineering Admins group to modify the `departmentNumber` and `manager` attributes of all entries in the Engineering business category. This example uses LDAP filtering to select all entries with `businessCategory` attributes set to `Engineering`:

```
dn: dc=example,dc=com
objectClass: top
objectClass: organization
aci: (targetattr="departmentNumber || manager")
(targetfilter="(businessCategory=Engineering)")
(version 3.0; acl "eng-admins-write"; allow (write)
groupdn ="ldap:///cn=Engineering Admins, dc=example,dc=com";)
```

Although using LDAP filters can be useful when you are targeting entries and attributes that are spread across the directory, the results are sometimes unpredictable because filters do not directly name the object for which you are managing access. The set of entries targeted by a filtered ACI is likely to change as attributes are added or deleted. Therefore, if you use LDAP filters in ACIs, you should verify that they target the correct entries and attributes by using the same filter in an `ldapsearch` operation.

To Target Attribute Values Using LDAP Filters

You can use access control to target specific attribute values. This means that you can grant or deny permissions on an attribute if that attribute's value meets the criteria defined in the ACI. An ACI that grants or denies access based on an attribute's value is called a value-based ACI.

For example, you can grant all users in your organization permission to modify the `roomNumber` attribute in their own entries. However, you would also want to ensure that they do not give themselves reserved room numbers, all of which begin with `12`. LDAP filters are used to check that the conditions on attribute values are satisfied.

To create a value-based ACI, you must use the `targattrfilters` keyword with the following syntax:

`(targattrfilters="`*Op*`=`*attr1*`:`*F1*`[(&&`*attr2*`:`*F2*`)*][;`*Op*`=`*attr*`:`*F*`[(&&`*attr*`:`*F*`)*]")`

where:

- *Op* is either an `add` or `delete` operation:
  - `add` represents the operation of creating an attribute.
  - `delete` represents the operation of deleting an attribute.
- *attr* represents the target attributes.
- *F* represents filters that apply only to the associated attribute.

When creating an entry, if a filter applies to an attribute in the new entry, then all values of that attribute must satisfy the filter. When deleting an entry, if a filter applies to an attribute in the entry, then all values of that attribute must also satisfy the filter.

When modifying an entry, if the operation adds an attribute, then the add filter that applies to that attribute must be satisfied. Iif the operation deletes an attribute, then the delete filter that applies to that attribute must be satisfied. If individual values of an attribute already present in the entry are replaced, then both the add and delete filters must be satisfied.

The following example attribute filter allows users to add any `roomNumber` attribute to their own entries except the reserved room numbers, which have a `12` prefix. It also allows users to add a telephone number with a `123` prefix.

```
(targattrfilters="add=roomNumber:(!(roomNumber=12*)) && telephoneNumber:(telephoneNumber=123*)")
```

To Target a Single Directory Entry

There is no explicit way to target a single entry. However, you can achieve this in one of two ways:

- By creating a bind rule that matches user input in the bind request with an attribute value stored in the targeted entry
- By using the `targetfilter` keyword

With the `targetfilter` keyword you can specify an attribute value that appears only in the desired entry. For example, during the installation of the directory server, the following ACI is created:

```
aci: (targetattr="*")(targetfilter=(o=example))
(version 3.0; acl "Default anonymous access";
allow (read, search) userdn="ldap:///anyone";)
```

This ACI can apply only to the `o=example` entry, because that is the only entry with an attribute `o` having the value `example`.

The risk associated with these methods is that your directory tree can change in the future, and you would have to remember to modify this ACI.

To Specify the Scope of an ACI

Usually an ACI has subtree scope. You can restrict the scope of an ACI by using the `targetscope` keyword with the following syntax:

```
(targetscope="expression")
```

where *expression* is one of the following:

`base`	The ACI applies to the target resource only.
`onelevel`	The ACI applies to the target resource's first-generation children.
`subtree`	The ACI applies to the target resource and the subtree below it.
`subordinate`	The ACI applies only to the subtree below the target resource.

If the `targetscope` is not specified, the default value is `subtree`. The following example restricts the ACI target match only to the entry with the distinguished name `uid=bjensen,ou=People,dc=example,dc=com` and any of the children one level below it:

```
(target = "ldap:///uid=bjensen,ou=People,dc=example,dc=com")(targetscope="onelevel")
```

Note – The not-equal operator is not supported for the `targetscope` keyword.

To Target LDAP Controls

To target LDAP controls, use the `targetcontrol` keyword and provide the control OID. The `targetcontrol` keyword uses the following syntax:

```
(targetcontrol = "oid")
```

```
(targetcontrol != "oid")
```

You can target multiple LDAP controls by using the `targetcontrol` keyword with the following syntax:

```
(targetcontrol = "oid1 || oid2 ... || oidN")
```

```
(targetcontrol != "oid1 || oid2 ... || oidN")
```

For example, to target both the get effective rights control and the proxied authorization V2 control, use the following `targetcontrol` expression:

```
(targetcontrol = "1.3.6.1.4.1.42.2.27.9.5.2 || 2.16.840.1.113730.3.4.18")
```

Note – The get effective rights control has OID value of 1.3.6.1.4.1.42.2.27.9.5.2. The proxied authorization V2 control has OID value of 2.16.840.1.113730.3.4.18.

To Target LDAP Extended Operations

To target extended operations, use the `extop` keyword and provide the operation OID. The `extop` keyword uses the following syntax:

`(extop= "`*oid*`")`

`(extop!= "`*oid*`")`

You can target multiple extended operations by using the `extop` keyword with the following syntax:

`(extop = "`*oid1* `||` *oid2* `...` `||` *oidN*`")`

`(extop != "`*oid1* `||` *oid2* `...` `||` *oidN*`")`

For example, to target both the StartTLS extended operation and the Password Modify extended operation, use the following `extop` expression:

```
(extop = "1.3.6.1.4.1.1466.20037 || 1.3.6.1.4.1.4203.1.11.1.")
```

Note – Access control using the `extop` keyword with a StartTLS extended operation target must always be done using Global ACIs. The authorization entry in the StartTLS extended operation is null.

Defining Permissions

Permissions specify the type of access that you are allowing or denying. You can either allow or deny permission to perform specific operations in the directory. The various operations that can be assigned are known as rights.

There are two parts to setting permissions:

- Allowing or denying access
- Assigning rights

To Allow or Deny Access

You can explicitly allow or deny access permissions by using the `allow` or the `deny` keyword.

To Assign Rights

Rights detail the specific operations a user can perform on directory data. You can allow or deny all rights, or you can assign one or more of the following rights:

Read	Indicates whether users can read the directory entries and the attributes of entries specified in the ACI. This permission applies only to the search operation. (Compare the Read permission with the description of the Search permission that follows.)
Write	Indicates whether users can modify an entry by adding, modifying, or deleting attributes. This permission applies to the modify and modRDN operations.
Add	Indicates whether users can create entries. This permission applies only to the add operation.
Delete	Indicates whether users can delete entries. This permission applies only to the delete operation.
Search	Indicates whether users can search on the targets specified in the ACI. This permission applies only to the search operation. The Search right is checked once, and after the search is allowed or denied, it is not checked again. If the search is allowed, the read right is then applied to each entry to be returned as a result of the search and to each attribute of each entry.
Compare	Indicates whether users can compare data they supply with data stored in the directory. With compare rights, the directory returns a success or failure message in response to an inquiry, but the user cannot see the value of the entry or attribute. This permission applies only to the compare operation.
Selfwrite	Indicates whether users can add or delete their own DN in an attribute of the target entry. The syntax of this attribute must be a distinguished name. This right is used only for group management. Selfwrite works with proxy authorization: it grants the right to add or delete the proxy DN from the group entry (not the DN of the bound user).
Proxy	Indicates whether the specified DN can access the target with the rights of another entry. You can grant proxy access using the DN of any user in the directory except the Directory Manager DN. Moreover, you cannot grant proxy rights to the Directory Manager. An example is provided in "Proxy Authorization ACIs" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.
Import	Used by the modify DN operation. This access right indicates whether an entry can be imported to the specified DN.
Export	Used by the modify DN operation. This access right indicates whether an entry can be exported from the specified DN.

All — Indicates that the specified DN has the following rights to the targeted entry: read, write, search, delete, compare, and selfwrite. The All access right does not give the following rights to the target entry: proxy, import, and export.

Rights are granted independently of one another. This means, for example, that a user who is granted add rights can create an entry but cannot delete it if delete rights have not been specifically granted. Therefore, when planning the access control policy for your directory, you must ensure that you grant rights in a way that makes sense for users. For example, it does not usually make sense to grant write permission without granting read and search permissions.

Rights Required for LDAP Operations

This section describes the rights that you need to grant to users depending on the type of LDAP operation that you want to authorize them to perform.

- Adding an entry
  - Grant add permission on the entry being added.
  - Grant write permission on the value of each attribute in the entry. This right is granted by default but could be restricted using the `targattrfilters` keyword.
- Deleting an entry
  - Grant delete permission on the entry to be deleted.
  - Grant write permission on the value of each attribute in the entry. This right is granted by default but could be restricted using the `targattrfilters` keyword.
- Modifying an attribute in an entry
  - Grant write permission on the attribute type.
  - Grant write permission on the value of each attribute type. This right is granted by default but could be restricted using the `targattrfilters` keyword.
- Modifying the RDN of an entry
  - Grant write permission on the entry.
  - Grant write permission on the attribute type used in the new RDN.
  - Grant write permission on the attribute type used in the old RDN, if you want to grant the right to delete the old RDN.
  - Grant write permission on the value of the attribute type used in the new RDN. This right is granted by default but could be restricted using the `targattrfilters` keyword.
- Moving an entry to another subtree
  - Grant export permissions on the entry that you want to move.
  - Grant import permission on the new superior entry of the entry that you want to move.
- Comparing the value of an attribute

    - Grant compare permission on the attribute type.
- Searching for entries
    - Grant search permission on each attribute type used in the search filter.
    - Grant read permission on at least one attribute type used in the entry to ensure that the entry is returned.
    - Grant read permission an each attribute type to be returned with the entry.

The permissions you need to set up to allow users to search the directory are more readily understood with an example. Consider the following search:

```
$ ldapsearch -h host -p port -D "uid=bjensen,dc=example,dc=com" \
  -w password -b "dc=example,dc=com" \
  "(objectclass=*)" mail
```

The following ACI is used to determine whether user `bjensen` can be granted access for searching her own entry:

```
aci: (targetattr = "mail")(version 3.0; acl "self access to \
mail"; allow (read, search) userdn = "ldap:///self";)
```

The search result list is empty because this ACI does not allow `bjensen` the right to search on the `objectclass` attribute. For the search operation to be successful, you must modify the ACI, as shown in the following example:

```
aci: (targetattr = "mail || objectclass")(version 3.0; acl \
"self access to mail"; allow (read, search) userdn = \
"ldap:///self";)
```

Permissions Syntax

In an ACI statement, permissions use the following syntax:

`allow|deny (`*rights*`)`

where *rights* is a list of comma-separated keywords enclosed within parentheses. Valid keywords are `read`, `write`, `add`, `delete`, `search`, `compare`, `selfwrite`, `proxy`, `import`, `export`, or `all`.

The `all` access right does not give the following rights to the target entry: `proxy`, `import`, and `export`.

In the following example, `read`, `search`, and `compare` access is allowed, provided that the bind rule is evaluated to be true:

```
aci: (target="ldap:///dc=example,dc=com") (version 3.0;acl \
"example"; allow (read, search, compare) bindRule;)
```

Bind Rules

Depending on the ACIs defined for the directory, for certain operations, you need to bind to the directory. This section describes how bind rules are used to control access.

Bind Rules Overview

Binding means logging in or authenticating yourself to the directory by providing a bind DN and password, or, if using SSL, a certificate. The credentials provided in the bind operation and the circumstances of the bind determine whether access to the directory is allowed or denied.

Every permission set in an ACI has a corresponding bind rule that details the required credentials and bind parameters.

A simple bind rule might require that the person accessing the directory belong to a specific group. A complex bind rule can state that a person must belong to a specific group and must log in from a machine with a specific IP address between 8 a.m and 5 p.m.

Bind rules define who can access the directory, when, and from where. More specifically, bind rules can specify the following:

- Users, groups, and roles that are granted access
- Location from which an entity must bind (The location from which a user authenticates can be spoofed and can therefore not be trusted. Do not base ACIs on this information alone.)
- Time or day on which binding must occur
- Type of authentication that must be in use during binding
- Security strength factor (that is, the length of encryption key currently in use)

Additionally, bind rules can be complex constructions that combine these criteria by using Boolean operators.

The directory server evaluates the logical expressions used in ACIs according to a three-valued logic similar to the one used to evaluate LDAP filters, as described in RFC 4511 Lightweight Directory Access Protocol (LDAP): The Protocol. In summary, this means that if any component in the expression evaluates to Undefined (for example if the evaluation of the expression aborted due to a resource limitation), then the directory server handles this case correctly: it does not erroneously grant access because an Undefined value occurred in a complex Boolean expression.

Using Boolean Bind Rules

Bind rules can be complex expressions that use the Boolean expressions `AND`,`OR`, and `NOT` to set very precise access rules. When creating boolean bind rules, always use parentheses to define the order in which rules are to be evaluated. A trailing semicolon is a required delimiter that must appear after the final rule.

For example, to bind with `bindRuleA`, and with either `bindRuleB`, or with either `bindRuleC` and `bindRuleD`, use the following syntax:

```
(bindRuleA and (bindRuleB or (bindRuleC and bindRuleD));)
```

Using another example, the following bind rule is evaluated to be true if the bind DN client is accessed from within the `example.com` domain and is a member of either the administrators group or both the mail administrators and calendar administrators groups.

```
(dns = "*.example.com" and (groupdn = "ldap:///cn=administrators,dc=example,dc=com" or
(groupdn = "ldap:///cn=mail administrators,dc=example,dc=com" and
groupdn = "ldap:///cn=calendar administrators,dc=example,dc=com"));)
```

The `||` operator is allowed only in the `groupdn` bind rule keyword expression. For all other bind rule expressions, the `or` operator must be used.

See Also

"Bind Rule Syntax" on page 22

Bind Rule Syntax

Whether access is allowed or denied depends on whether an ACI's bind rule is evaluated to be true. This section describes the bind rule syntax and the various keywords that can be used to allow or deny access.

Bind Rule Syntax Overview

Bind rules use one of the following patterns:

- *keyword* = " *expression*";
- *keyword* != " *expression*";

where equal (=) indicates that the keyword and expression must match in order for the bind rule to be true, and not equal (!=) indicates that the keyword and expression must not match in order for the bind rule to be true.

The quotation marks ("") around the expression and the delimiting semicolon (;) are required. The expressions you can use depend on the associated keyword.

The `timeofday` keyword also supports the inequality expressions (<, <=, >, >=). The `timeofday` keyword is the only keyword that supports these expressions.

The following table lists each keyword and the associated expressions and indicates whether wildcard characters are allowed in the expression.

Keyword	Valid Expressions	Wildcard Allowed?
`userdn`	`ldap:///`*distinguishedName*	Allowed, in DN only
	`ldap:///all`	
	`ldap:///anyone`	
	`ldap:///self`	
	`ldap:///parent`	
	`ldap:///`*suffix*`??sub?(`*filter*`)`	
`groupdn`	`ldap:///`*DN*	Not Allowed
`userattr`	*attribute#bindType* or *attribute#value*	Not Allowed
`ip`	*IPaddress*	Allowed
`dns`	*DNShostName*	Allowed
`dayofweek`	`sun`	Not Allowed
	`mon`	
	`tue`	
	`wed`	
	`thu`	
	`fri`	
	`sat`	
`timeofday`	0 - 2359	Not Allowed

Keyword	Valid Expressions	Wildcard Allowed?
authmethod	none	Not Allowed
	simple	
	ssl	
	sasl	
	authenticationMethod	

The following sections provide additional information about the bind rule syntax for each keyword.

Defining User Access (userdn Keyword)

User access is defined using the userdn keyword. The userdn keyword requires one or more valid distinguished names in the following format:

userdn = "ldap:///*dn* [|| ldap:///*dn*]..."

userdn != "ldap:///*dn* [|| ldap:///*dn*]..."

where *dn* can be a DN or one of the expressions anyone, all, self, or parent. These expressions refer to the following users:

userdn = "ldap:///anyone"	Both anonymous and authenticated users
userdn = "ldap:///all"	Only authenticated users
userdn = "ldap:///self"	Only the same user as the target entry of the ACI
userdn = "ldap:///parent"	Only the parent entry of the ACI target

The userdn keyword can also be expressed as an LDAP filter in this form:

userdn = ldap:///*suffix*??sub?(*filter*)

Characters that are syntactically significant for a DN, such as commas, must be escaped with a single backslash (\).

Defining General Access (all Keyword)

You can use bind rules to indicate that a permission applies to anyone who has successfully bound to the directory. The all keyword therefore allows access by all authenticated users. This allows general access while preventing anonymous access.

For example, to grant read access to the entire tree to all authenticated users, create the following ACI on the dc=example,dc=com node:

```
aci: (version 3.0; acl "all-read"; allow (read)
userdn="ldap:///all";)
```

Defining Anonymous Access (`anyone` Keyword)

Granting anonymous access to the directory means that anyone can access it without providing a bind DN or password, regardless of the circumstances of the bind. You can limit anonymous access to specific types of access (for example, access for read or access for search) or to specific subtrees or individual entries within the directory. Anonymous access using the `anyone` keyword also allows access by any authenticated user.

For example, to allow anonymous read and search access to the entire `example.com` tree, create the following ACI on the `dc=example,dc=com` node:

```
aci: (version 3.0; acl "anonymous-read-search";
allow (read, search) userdn = "ldap:///anyone";)
```

Defining Self Access (`self` Keyword)

Specifies that users are granted or denied access to their own entries. In this case, access is granted or denied if the bind DN matches the DN of the targeted entry. For example, to grant all users in the `example.com` tree write access to their `userPassword` attribute, create the following ACI on the `dc=example,dc=com` node.

```
aci: (targetattr = "userPassword") (version 3.0; acl
"modify own password"; allow (write) userdn = "ldap:///self";)
```

Defining Parent Access (`parent` Keyword)

Specifies that users are granted or denied access to the entry only if their bind DN is the parent of the targeted entry. For example, to allow users to modify any child entries of their bind DN, create the following ACI on the `dc=example,dc=com` node:

```
aci: (version 3.0; acl "parent access";
allow (write) userdn="ldap:///parent";)
```

Specifying Users With LDAP URLs

You can dynamically target users in ACIs using a URL with a filter as shown in the following example:

```
userdn = "ldap:///suffix??sub?(filter)"
```

For example, all users in the accounting and engineering branches of the `example.com` tree would be granted or denied access to the targeted resource dynamically based on the following URL:

```
userdn = "ldap:///dc=example,dc=com??sub?(|(ou=eng)(ou=acct))"
```

Do not specify a host name or port number within the LDAP URL. LDAP URLs always apply to the local directory server.

Specifying Users With Wildcards

You can also specify a set of users by using the wildcard character (`*`). For example, specifying a user DN of `uid=b*,dc=example,dc=com` indicates that only users with a bind DN beginning with the letter `b` is allowed or denied access based on the permissions you set.

Specifying Users With a Logical OR of LDAP URLs

Specify several LDAP URLs or keyword expressions to create complex rules for user access. For example:

```
userdn = "ldap:///uid=b*,c=example.com ||
ldap:///cn=b*,dc=example,dc=com";
```

The bind rule is evaluated to be true for users binding with either of the DN patterns.

Excluding Specific LDAP URLs

Use the not-equal (`!=`) operator to define user access that excludes specific URLs or DNs. For example:

```
userdn != "ldap:///uid=*,ou=Accounting,dc=example,dc=com";
```

The bind rule is evaluated to be true if the client is not binding as a UID-based distinguished name in the accounting subtree. This bind rule makes sense only if the targeted entry is not under the accounting branch of the directory tree.

Defining Group Access (`groupdn` Keyword)

Members of a specific group can access a targeted resource. This is known as group access. Group access is defined using the `groupdn` keyword to specify that access to a targeted entry is granted or denied if the user binds using a DN that belongs to a specific group.

The `groupdn` keyword requires the distinguished name of one or more groups in the following format:

```
groupdn="ldap:///groupDN [|| ldap:///groupDN]..."
```

The bind rule is evaluated to be true if the bind DN belongs to a group specified by any of the group DNs. The following section give examples using the `groupdn` keyword.

Characters that are syntactically significant for a DN, such as commas, must be escaped with a single backslash (\).

Specifying a Group With a Single LDAP URL

```
groupdn = "ldap:///cn=Administrators,dc=example,dc=com";
```

The bind rule is evaluated to be true if the bind DN belongs to the Administrators group. For example, to grant the Administrators group permission to write to the entire directory tree, create the following ACI on the `dc=example,dc=com` node:

```
aci: (version 3.0; acl "Administrators-write"; allow (write)
groupdn="ldap:///cn=Administrators,dc=example,dc=com";)
```

Specifying a Group With a Logical OR of LDAP URLs

```
groupdn = "ldap:///cn=Administrators,dc=example,dc=com ||
ldap:///cn=Mail Administrators,dc=example,dc=com";
```

The bind rule is evaluated to be true if the bind DN belongs to either the Administrators or the Mail Administrators group.

Defining Access Based on Value Matching (`userattr` Keyword)

The `userattr` keyword can be used to specify which attribute values must match between the entry used to bind (bind entry) and the targeted entry. A `userattr` expression has two formats, a bind-type format and an attribute-value format.

Bind-Type Format

This format is named the bind-type format because it uses the bind DN and possibly the bind entry when evaluating a match. It is the more complicated of the two formats. The bind-type format can be used in the following three ways:

- Treat a target entry attribute value as a DN that must match the bind DN
- Treat a target entry attribute value as a group DN that the bind DN must be a member of
- Require that both the bind DN and the bind entry match an LDAP URL specified in a target entry attribute value

The bind-type `userattr` format uses this syntax:

```
userattr = "attrName#bindType"
```

where:

attrName Is the name of the attribute in the target entry.

bindType Must be one of the following:

- `USERDN` The value of *attrName* must match the bind DN.
- `GROUPDN` The value of *attrName* is a group that must contain the bind DN.
- `LDAPURL` The value of *attrName* is an LDAP URL that is treated as a search that the bind DN and entry must match. To satisfy the search, the URL's *dn* value is used as a base DN that the bind DN must match or have as a parent DN. The URL's *scope* value restricts how far below the base DN the bind DN can match. Finally, the bind entry must match the URL's *filter* value.

The bind type `userattr` format has a special parent keyword that allows targeting of entries levels below the current target entry. See "Inheritance" on page 30 for more details on this keyword.

Attribute-Value Format

The attribute-value format requires the following two conditions to match:

- An attribute specified in the `userattr` expression must exist in both the target and bind entries.
- The values of both of these attributes must match a string value specified in the `userattr` expression. This string value cannot be one of the bind type keywords (`USERDN`, `GROUPDN`, `LDAPURL`).

The attribute value `userattr` format uses this syntax:

```
userattr = "attrName#attrValue"
```

where:

attrName The name of the attribute in both the target and bind entries.

attrValue The string representing the attribute value (not `USERDN`, `GROUPDN` or `LDAPURL`).

USERDN Bind Type Example

The following example of a bind rule `userattr` keyword expression specifies a match between the bind DN and the value of the target entry attribute `manager`.

```
userattr = "manager#USERDN"
```

This bind rule is evaluated to be true if the bind DN matches the value of the `manager` attribute in the target entry. The `manager` attribute in the target entry must match the bind DN. Wildcards are not allowed.

The following example ACI grants a manager full access to all user attributes of entries located in the subtree under the DN `dc=example, dc=comm`:

```
aci: (target="ldap:///dc=example,dc=com")(targetattr="*")
(version 3.0;acl "manager all access";
allow (all) userattr = "manager#USERDN";)
```

GROUPDN Bind Type Example

This is an example of a bind rule `userattr` keyword expression specifying an attribute that contains a group DN that the bind DN must be a member of.

```
userattr = "owner#GROUPDN"
```

The bind rule is evaluated to be true if the bind DN is a member of the group specified in the `owner` attribute of the target entry.

LDAPURL Bind Type Example

This is an example of a bind rule `userattr` keyword expression specifying an attribute that contains an LDAP URL that is treated as a search that the bind DN and entry must match.

```
userattr = "aciurl#LDAPURL"
```

The attribute `aciurl` is an example only.

The bind rule is evaluated to true if the bind DN and bind entry satisfy all of the search requirements specified in the LDAP URL. For example, if the value of `aciurl` is `ldap:///dc=example,dc=com??one?(cn=joe*)`, then the bind DN must satisfy a one-level search under the base DN of `dc=example,dc=com` and the bind entry must satisfy the filter `(cn=joe*)`.

Attribute Value Example

The following example of the bind rule `userattr` keyword expression specifies an attribute value that both the bind entry and target entry must match.

```
userattr = "favoriteBeverage#Water"
```

The bind rule is evaluated to be true if the bind and target entries include the `favoriteBeverage` attribute with a value of `Water`.

Inheritance

When you use the `userattr` keyword to associate the entry used to bind with the target entry, the ACI applies only to the target specified and not to the entries below it. In some circumstances, you might want to extend the application of the ACI several levels below the targeted entry. This is possible by using the parent keyword and specifying the number of levels below the target that should inherit the ACI.

When you use the `userattr` keyword in association with the parent keyword, the syntax is as shown in the following example:

`userattr = "parent[[`*inheritanceLevel*`].`*attribute*`#`*bindType*`"`

where :

- *inheritanceLevel* is a comma-separated list that indicates how many levels below the target inherit the ACI. You can include ten levels [0,1,2,3,4,..,9] below the targeted entry. Zero (0) indicates the targeted entry.
- *attribute* is the attribute targeted by the `userattr`.
- *bindType* can be either `USERDN` or `GROUPDN`. The `LDAPURL` bind type is not supported with inheritance.

For example, the `userattr = "parent[[0,1].manager#USERDN"` bind rule is evaluated to be true if the bind DN matches the `manager` attribute of the target entry. Also, the bind rule is evaluated to be true for all entries immediately below the target entry (one level below the target) that have `manager` attributes matching the bind DN.

Inheritance Example

The following example indicates that user `bjensen` is allowed to read and search the `cn=Profiles` entry as well as the first level of child entries, which includes `cn=mail` and `cn=news`.

```
cn=Profiles
aci:(targetattr="*")(version 3.0, acl "profiles access" allow(read, search)
userattr="parent[[0,1].owner#USERDN;)
owner=cn=bjensen, ou=people, dc=example, dc=com
cn=mail, cn=Profiles
mailuser: bjensen
cn=news, cn=Profiles
newuser: bjensen
```

If inheritance were not used in this example, you would need to do one of the following:

- Explicitly set read and search access for user `bjensen` on the `cn=Profiles`, `cn=mail`, and `cn=news` entries in the directory.
- Add the owner attribute and the following ACI to the `cn=mail`, `cn=Profiles` and `cn=news,cn=Profiles` entries:

```
aci: (targetattr="*") (version 3.0; acl "profiles access"; allow
(read,search) userattr="owner#USERDN";)
```

Add Permissions

If you use the `userattr` keyword in conjunction with all or add permissions, you might find that the behavior of the directory server is not what you expect. Typically, when a new entry is created in the directory, the directory server evaluates access rights on the entry being created, and not on the parent entry. However, in the case of ACIs using the `userattr` keyword, this behavior could create a security hole, so the directory server's normal behavior is modified to avoid it.

Consider the following example ACI:

```
aci: (target="ldap:///dc=example,dc=com")(targetattr="*")
(version 3.0; acl "manager-write"; allow (all)
userattr = "manager#USERDN";)
```

This ACI grants managers all rights on the entries of employees that report to them. However, because access rights are evaluated on the entry being created, this type of ACI would also allow any employee to create an entry in which the manager attribute is set to their own DN. For example, disgruntled employee Joe, `cn=Joe,ou=eng,dc=example,dc=com`, might want to create an entry in the Human Resources branch of the tree to use (or misuse) the privileges granted to Human Resources employees.

He could do this by creating the following entry:

```
dn: cn= Trojan Horse,ou=Human Resources,dc=example,dc=com
objectclass: top
...
cn: Trojan Horse
manager: cn=Joe,ou=eng,dc=example,dc=com
```

To avoid this type of security threat, the ACI evaluation process does not grant add permission at *level 0*, that is, to the entry itself. You can, however, use the `parent` keyword to grant add rights below existing entries. You must specify the number of levels below the parent for add rights. For example, the following ACI allows child entries to be added to any entry in the `dc=example,dc=com` that has a manager attribute that matches the bind DN:

```
aci: (target="ldap:///dc=example,dc=com")(targetattr="*")
(version 3.0; acl "parent-access"; allow (add)
userattr = "parent[1].manager#USERDN";)
```

This ACI ensures that add permission is granted only to users whose bind DN matches the `manager` attribute of the parent entry.

Defining Access From a Specific IP Address (`ip` Keyword)

Using bind rules, you can indicate that the bind operation must originate from a specific IP address. This is often used to force all directory updates to occur from a given machine or network domain.

The LDIF syntax for setting a bind rule based on an IP address is shown in the following examples:

```
ip = "IPaddressList"
ip != "IPaddressList"
```

The *IPaddressList* is a list of one or more comma-separated elements from among any of the following:

- A specific IPv4 address, such as `123.45.6.7`
- An IPv4/CIDR-compliant address, such as `192.168.0.0/16`
- An IPv4 address with wildcards to specify a subnetwork, such as `12.3.45.*`
- An IPv4 address or subnetwork with a subnetwork mask, such as `123.45.6.*+255.255.255.192`
- An IPv6 address in any of its legal forms and contained in square brackets `[` and `]`, as defined by RFC 2373 and RFC 2732. The following addresses are equivalent:
  - `ldap://[12AB:0000:0000:CD30:0000:0000:0000:0000]`
  - `ldap://[12AB::CD30:0:0:0:0]`
  - `ldap://[12AB:0:0:CD30::]`
- An IPv6 address with a subnet prefix length, such as `ldap://[12AB::CD30:0:0:0:0]/60`

The bind rule is evaluated to be true if the client accessing the directory is located at the named IP address. This can be useful for allowing certain kinds of directory access only from a specific subnet or machine. Note that the IP address from which a user authenticates can be spoofed, and can therefore not be trusted. Do not base ACIs on this information alone.

Defining Access From a Specific Domain (dns Keyword)

A bind rule can specify that the bind operation must originate from a particular domain or host machine. This is often used to force all directory updates to occur from a given machine or network domain.

The LDIF syntax for setting a bind rule based on the DNS host name is as shown here:

```
dns = "DNShostname"
dns != "DNShostname"
```

Caution – The dns keyword requires that the naming service used on your machine is DNS. If the naming service is not DNS, use the ip keyword instead.

The dns keyword requires a fully qualified DNS domain name. Granting access to a host without specifying the domain creates a potential security threat. For example, the following expression is allowed but not recommended:

```
dns = "legend.eng";
```

You should use a fully qualified name such as:

```
dns = "legend.eng.example.com";
```

The dns keyword allows wildcards. For example:

```
dns = "*.example.com";
```

The bind rule is evaluated to be true if the client accessing the directory is located in the named domain. This can be useful for allowing access only from a specific domain. Note that wildcards do not work if your system uses a naming service other than DNS. In such a case, if you want to restrict access to a particular domain, use the ip keyword, as described in "Defining Access From a Specific IP Address (ip Keyword)" on page 32.

Defining Access at a Specific Time of Day or Day of Week (timeofday and dayofweek Keywords)

You can use bind rules to specify that binding can only occur at a certain time of day or on a certain day of the week. For example, you can set a rule that allows access only if the time is between the hours of 8 a.m. and 5 p.m. Monday through Friday. The time used to evaluate access rights is the time on the directory server, not the time on the client.

The LDIF syntax for setting a bind rule based on the time of day is as shown here:

```
timeofday operator "time"
```

where *operator* can be one of the following symbols:

The time is expressed as four digits representing hours and minutes in the 24-hour clock (`0` to `2359`). For example:

- `timeofday = "1200";` is true if the client is accessing the directory during the minute that the system clock shows noon.
- `timeofday != "0100";` is true for access at any other time than 1 a.m.
- `timeofday > "0800";` is true for access from 8:01 a.m. through 11:59 p.m.
- `timeofday >= "0800";` is true for access from 8:00 a.m. through 11:59 p.m.
- `timeofday < "1800";` is true for access from 12:00 midnight through 5:59 p.m.

The time and date on the directory server are used for the evaluation of the `timeofday` and `dayofweek` bind rules and not the time on the client.

The LDIF syntax for setting a bind rule based on the day in the week is as shown here:

```
dayofweek = "day1, day2 ..."
```

The possible values for the `dayofweek` keyword are the English three-letter abbreviations for the days of the week: `sun`, `mon`, `tue`, `wed`, `thu`, `fri`, `sat`. Specify all days you want to grant access, for example:

```
dayofweek = "mon, tue, wed, thu, fri";
```

The bind rule is true if the directory is being accessed on one of the days listed.

Defining Access Based on Authentication Method (`authmethod` Keyword)

You can set bind rules that state that a client must bind to the directory using a specific authentication method. The following authentication methods are available:

`None`	Authentication is not required. This is the default. It represents anonymous access.
`Simple`	The client must provide a user name and password to bind to the directory.
`SSL`	The client must bind to the directory over a Secure Sockets Layer (SSL) or Transport Layer Security (TLS) connection. In the case of SSL, the connection is established to the LDAPS second port. In the case of TLS, the connection is established through a Start TLS operation. In both

cases, a certificate must be provided. For information on setting up SSL, see "Using SASL Authentication" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

`SASL` The client must bind to the directory using a Simple Authentication and Security Layer (SASL) mechanism, such as DIGEST-MD5 or GSSAPI.

The LDIF syntax for setting a bind rule based on an authentication method is as shown here:

```
authmethod = "authentication_method"
```

where `authentication_method` is `none`, `simple`, `ssl`, or `sasl sasl_mechanism`.

Authentication Method Examples

The following examples show typical specifications of the `authmethod` keyword:

`authmethod = "none"`	Authentication is not checked during bind rule evaluation.
`authmethod = "simple"`	The bind rule is evaluated to be true if the client is accessing the directory using a user name and password.
`authmethod = "ssl"`	The bind rule is evaluated to be true if the client authenticates to the directory using a certificate over LDAPS. It is not true if the client authenticates using simple authentication (bind DN and password) over LDAPS.
`authmethod = "sasl DIGEST-MD5"`	The bind rule is evaluated to be true if the client is accessing the directory using the SASL DIGEST-MD5 mechanism. Other supported SASL mechanisms are EXTERNAL and GSSAPI.

Defining Access Based on a Connection's Security Strength Factor (`ssf` Keyword)

You can use bind rules to specify that binding can only occur based on a specific level of Security Strength Factor (SSF) enforced on the established connection. A connection's SSF is based on the key strength of the cipher enforced on the connection and pertains only to TLS/SSL or DIGEST-MD5/GSSAPI confidentiality or integrity connections.

The LDIF syntax for setting a bind rule based on the Security Strength Factor is shown here:

`ssf` *operator* "*strength*"

where *operator* can be one of the following symbols:

- = (equal to)
- != (not equal to)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= less than or equal to

The strength is a value representing the cipher key strength required on the connection and is a value (`1` to `1024`). DIGEST-MD5/GSSAPI connections with integrity enforced have an SSF of `1`. TLS/SSL and DIGEST-MD5/GSSAPI confidentiality connections can have variable values of SSF based on the cipher negotiation performed between the directory server and client. The higher a connection's negotiated SSF is, the stronger the encryption is on the connection, as shown in these examples:

- `ssf = "1";` is true for access if integrity `ssf = 1` only is enforced on the connection.
- `ssf != "40";` is true for access if `ssf not equal 40` is enforced on the connection.
- `ssf > "128";` is true for access if `ssf greater than 128` is enforced on the connection.
- `ssf >= "128";` is true for access if `ssf greater than or equal 128` is enforced on the connection.
- `ssf < "56";` is true for access if `ssf less than 56` is enforced on the connection.

Clear connections have an SSF of `0`.

DIGEST-MD5 QOP Key Size Mapping

The following table illustrates the Quality of Protection (QOP) to cipher key size mapping.

Cipher	QOP	Description
RC4 (40)	Low	RC4 cipher with 40-bit key (obsolete)
RC4 (56)	Medium	RC4 cipher with 56-bit key
DES	Medium	Data Encryption Standard (DES) cipher in cipher block chaining (CBC) mode with a 56–bit key
RC4 (128)	High	RC4 cipher with 128–bit key
Triple DES	High	Triple DES cipher in CBC mode with EDE with the same key for each E stage (also called “two keys mode”) for a total key length of 112 bits

TLS Cipher Key Size Mapping

Cipher	TLS RFC	Key Size	Description
RC2_CBC_40	4346	40	RC2 cipher in cipher block chaining (CBC) mode (obsolete)
RC4_40	4346	40	RC4 cipher (obsolete)
DES40_CBC	4346	40	DES 40–bit cipher in cipher block chaining (CBC) mode (obsolete)
DES_CBC	4346	56	DES 56–bit in cipher block chaining (CBC) mode cipher
3DES_EDE_CBC	4346	112	TDES
RC4_128	4346	128	RC4 cipher
IDEA_CBC	4346	128	International Data Encryption Algorithm (IDEA) cipher in cipher block chaining (CBC) mode
SEED_CBC	4162	128	SEED cipher in cipher block chaining (CBC) mode
CAMELLIA_128_CBC	4132	128	Camellia cipher in cipher block chaining (CBC) mode
AES_128_CBC	3268	128	Advanced Encryption Standard (AES) in cipher block chaining (CBC) mode
AES_256_CBC	3268	256	Advanced Encryption Standard (AES) in cipher block chaining (CBC) mode
CAMELLIA_256_CBC	4132	256	Camellia cipher in cipher block chaining (CBC) mode
AES_256_GCM	5288	256	AES in Galois Counter Mode (GCM)

Example

The following ACI only allows user to change their own passwords over a connection with an SSF strength equal to or greater than 128:

```
(targetattr="userPassword||authPassword")(version 3.0; acl "User change pwd";
(allow (write) userdn="ldap:///self" and ssf >= "128");)
```

Compatibility With the Sun Java System Directory Server Access Control Model

This section describes how the directory server's access control model differs from the access control model provided with the Sun Java™ System directory server.

Global ACI

Global ACI configuration differs from Sun Java System Directory Server, Version 6, global ACI implementation in two ways:

- The `ds-config-global-aci` attribute specifies a global ACI in the `cn=Access Control Handler,cn=config` entry (see "Access Control Principles" on page 5) rather than placing the ACI in the root DSE entry as in Sun Java System Directory Server, Version 6.
- The scope of the global ACI can be narrowed by specifying a target keyword in the ACI. For example, the following global ACI restricts anonymous read access to entries under the suffix `dc=example,dc=com`:

```
ds-cfg-global-aci: (target="dc=example,dc=com")
(targetattr!="userPassword||authPassword")
(version 3.0; acl "Anonymous read access only under dc=example,dc=com suffix";
allow (read,search,compare) userdn="ldap:///anyone";)
```

Removing the `(target="dc=example,dc=com")` expression would make the ACI global to all entries in the directory server.

Global ACIs are not supported in Sun Java System Directory Server versions earlier than Version 6.

All Attributes `targetattr` Rule `(targetattr="*")`

The all attributes `targetattr` rule only applies to non-operational attributes. Operational attributes must be explicitly specified in a `targetattr` ACI statement. This differs from Sun Java System directory server behavior, which allows the all attributes `targetattr` rule to apply to both operational and non-operational attributes.

It is also illegal to use a not-equal operator when an operational attribute is specified in a `targetattr` rule. For example, the `targetattr` rule below is invalid because the operational attribute `aclRights` is used with a not-equal operator:

```
(targetattr != aclRights)
```

Note – A non-equal operator in a `targetattr` rule specifying non-operational attributes is valid, but the rule is restricted to applying to other non-operational attributes only.

It is illegal to specify both operational and non-operational attributes in the same `targetattr` statement.

It is illegal to specify both the all attributes `targetattr` rule and an attribute in the same expression (for example, `targetattr="cn || *"`).

Distinguished Name (DN) Wildcard Matching

The ACI DN wildcard matching implementation supports the following usage:

- Any number of wildcards can appear in Relative Distinguished Name (RDN) attribute values, where they match zero or more characters (similar to substring filters). For example, the bind rule matches the following DNs: `uid=bob jensen,dc=example,dc=com` and `uid=bjensen,dc=example,dc=com`:

  ```
  userdn="ldap///uid=b*jensen*,dc=example,dc=com"
  ```

  It does not match the DN `cn=bill jensen,dc=example,dc=com` because the attribute type of the first RDN does not match.
- A single wildcard can also be used to match any RDN attribute type. (The wildcard in this case can be omitted as a shorthand). For example, these two bind rules behave exactly the same:

  ```
  userdn="ldap:///*=bjensen, dc=example, dc=com"
  userdn="ldap:///bjensen, dc=example, dc=com"
  ```

  They both match the following DNs: `uid=bjensen,dc=example,dc=com` and `cn=bjensen,dc=example,dc=com`.
- A single wildcard can be used to match exactly one RDN component, which can be single or multivalued). For example, the following bind rule matches the DNs `uid=jensen,dc=example,dc=com` and `cn=smith,dc=example,dc=com`:

  ```
  userdn="ldap:///*,dc=example,dc=com"
  ```
- A double wildcard can be used to match one or more RDN components. For example, the following bind rule matches the DNs `uid=jensen,ou=people,dc=example,dc=com` and `uid=jensen,ou=sales,ou=people,dc=example,dc=com`:

  ```
  userdn="ldap:///uid=bjensen,**,dc=example,dc=com"
  ```

Privilege Subsystem Impact

The Sun Java System directory server has no support for privileges. The privilege subsystem (discussed in "Root Users and the Privilege Subsystem" on page 113) impacts ACIs in two ways:

- Users with `ds-privilege-name: bypass-acl` privileges can bypass access control evaluation.
- Users needing to modify access control rules need the `ds-privilege-name: modify-acl` privilege.

Note – Use of the Lightweight Directory Access Protocol (LDAP) Proxied Authorization Control requires the bind user to have the `ds-privilege-name: proxied-auth` privilege. When the proxied authorization control is used, evaluation of the `ds-privilege-name: bypass-acl` privilege is performed using the bind user, not the proxied user.

In general, a user should not have both the `ds-privilege-name: proxied-auth` and `ds-privilege-name: bypass-acl` privileges simultaneously since this allows a proxied user to bypass ACI access evaluation.

The `targetscope` Keyword

The `targetscope` keyword differs from Sun Java System Directory Server, Version 6, by including a new scope:

`subordinate` Restricts the ACI to the subtree below the target resource only.

The `targetscope` keyword is not supported in Sun Java System Directory Server versions earlier than Version 6.

LDAP Modify Increment

The directory server supports the LDAP Modify-Increment Extension. This extension is not supported in teh Sun Java System directory server. Attributes that are to be incremented must have write permissions.

Macro Support

There is no support for macros in ACIs.

The roledn Keyword

Roles are not supported in the directory server, so the roledn keyword should not be used. Equivalent functionality can be achieved by using groups.

Understanding the Directory Server Schema

Schema is a very important part of LDAP directory services. Although many people may have a basic understanding of attribute types and object classes, there is a great deal of information about LDAP schema that many people do not know. This section provides an in-depth description of schema elements in general, and illustrates the ways in which these schema elements are used in the directory server.

For instructions on viewing the schema using the `ldapsearch` command, see "Managing Attribute Types" in *Sun OpenDS Standard Edition 2.0 Administration Guide* and "Managing Object Classes" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Understanding Matching Rules

Matching rules are used by the directory server to compare two values for the same attribute, that is, to perform matching operations on them. There are several different types of matching rules, including:

Equality matching rules	These matching rules are used to determine whether two values are logically equal to each other. Different implementations of equality matching rules can use different criteria for making this determination (for example, whether to ignore differences in capitalization or deciding which spaces are significant).
Ordering matching rules	These matching rules are used to determine the relative order for two values, for example, when evaluating greater-or-equal or less-or-equal searches, or when the results need to be sorted.

Substring matching rules	These matching rules are used to determine whether a given substring assertion should match a particular value. A substring assertion is composed of at least one element from the following sets: at most one `subInitial` element, zero or more `subAny` elements, and at most one `subFinal` element.
Approximate matching rules	These matching rules are used to determine whether two values are approximately equal to each other. This is frequently based on "sounds like" or some other kind of fuzzy algorithm. Approximate matching rules are not part of the official LDAP specification, but they are included in the directory server for added flexibility.

Matching Rule Description Format

The matching rule description format is described in RFC 4512, section 4.1.3. This is the format that is used to display matching rules in the `matchingRules` attribute of the schema subentry, and it shows the properties that can be associated with a matching rule. The following example shows the definition of the matching rule description format:

```
MatchingRuleDescription = LPAREN WSP
numericoid                 ; object identifier
[ SP "NAME" SP qdescrs ]   ; short names (descriptors)
[ SP "DESC" SP qdstring ]  ; description
[ SP "OBSOLETE" ]          ; not active
SP "SYNTAX" SP numericoid  ; assertion syntax
extensions WSP RPAREN      ; extensions
```

The matching rule description includes these elements:

`numericoid`	The numeric OID is used to uniquely identify the matching rule in the directory server. Every matching rule must have a unique OID.
`NAME`	The name elements are human-readable names assigned to the matching rule that can be used to refer to it in place of the OID. A matching rule is not required to have any human-readable names. If it has only a single name, then it is enclosed in single quotes. If there are multiple names for a matching rule, each is enclosed in single quotes with spaces between the names, and parentheses around the entire set of names.
`DESC`	The description element is a human-readable description for the matching rule. There can be at most one description, and if it is present, it should be enclosed in single quotation marks.

`OBSOLETE` The `OBSOLETE` flag indicates whether this matching rule should be considered available for use. If a matching rule is marked `OBSOLETE`, then it should not be possible to create any new attribute types or matching rule uses that reference this matching rule.

`SYNTAX` The syntax element identifies the attribute syntax with which the matching rule is associated. This is used to indicate the acceptable format for values on which the matching rule operates. More information about attribute syntaxes can be found in "Understanding Attribute Syntaxes" on page 47. The syntax OID must be included in all matching rule descriptions.

extensions The extensions for a matching rule can be used to identify other properties for that matching rule that might not be included in the standard definition. The directory server does not currently support any extensions for use in matching rules.

For example, the following is the matching rule description for the standard `caseIgnoreMatch` matching rule:

```
( 2.5.13.2 NAME 'caseIgnoreMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )
```

In this case, the OID is `2.5.13.2`. There is one name, which is `caseIgnoreMatch`. There is no description. The OID of the associated syntax is `1.3.6.1.4.1.1466.115.121.1.15` (which is the Directory String syntax). There are no extensions.

Commonly Used Matching Rules

There are a number of matching rules defined in LDAP, both in the core protocol specification as well as in other related RFCs and Internet Drafts. Many of these matching rules are defined in RFC 4517 (LDAP Syntaxes and Matching Rules), in section 4.2. Some of the most commonly used matching rules include:

`caseIgnoreMatch`, `caseIgnoreOrderingMatch`, `caseIgnoreSubstringsMatch`
: These are equality, ordering, and substring matching rules, respectively, that ignore differences in capitalization and also treat multiple consecutive spaces as a single space.

`caseExactMatch`, `caseExactOrderingMatch`, `caseExactSubstringsMatch`
: These are equality, ordering, and substring matching rules, respectively, that treat values in a case-sensitive manner but do treat multiple consecutive spaces as a single space.

`octetStringMatch`, `octetStringOrderingMatch`, `octetStringSubstringsMatch`
: These are equality, ordering, and substring matching rules, respectively, that perform byte-for-byte comparisons of the values, treating them as binary data rather than strings.

`numericStringMatch`, `numericStringOrderingMatch`, `numericStringSubstringsMatch`
: These are equality, ordering, and substring matching rules, respectively, that operate on values that start with a numeric digit, and contain only numeric digits and spaces. Spaces are ignored when performing matching with these matching rules.

`distinguishedNameMatch`
: This is an equality matching rule that operates on distinguished name (DN) values. It ignores spaces around the commas or semicolons that separate DN components, spaces around plus signs that separate RDN components, and spaces around equal signs that separate RDN attribute type names from their corresponding values. Differences in capitalization are ignored for attribute type names. Equality matching for attribute values is performed using the equality matching rule for the corresponding attribute type.

`doubleMetaphoneApproximateMatch`
: This is an approximate matching rule that uses the double metaphone algorithm to perform a "sounds like" comparison. Note that this matching rule is not part of any official LDAP specification, but it is included in the directory server for added flexibility.

Value Normalization

One of the tasks that most matching rules need to perform is value normalization. This is the process of transforming a given value to a form that can be used to compare values efficiently. In most cases, the normalization process should reduce all logically equivalent values to the same string so that a very simple string comparison can be performed to determine whether the strings are equal. For example, the `caseIgnoreMatch` matching rule typically normalizes values by converting all characters to lowercase and replacing occurrences of multiple consecutive spaces with a single space. A more complicated example is the `distinguishedNameMatch` matching rule, which removes all unnecessary spaces (for example, around commas, equal signs, and plus signs), converts all attribute types to lowercase, and then uses the appropriate matching rules to normalize the attribute values for each RDN component.

Note that in some cases, normalization alone is not sufficient for determining whether two values are logically equivalent. This is particularly true for cases in which the value is transformed, and there can be multiple different transformations for the same value. For example, multiple transformations are often performed for the `userPassword` attribute type, where values can be encoded using a one-way message digest algorithm, and if that algorithm includes a salt then each time a given value is encoded it can result in a different string. (In fact, this is the intended behavior, which helps prevent dictionary attacks.) In cases like this, the matching rule simply needs to use different logic to determine equality, rather than relying solely on normalization.

Understanding Attribute Syntaxes

Attribute syntaxes are essentially data type definitions. The syntax for an attribute type indicates the type of data meant to be held by the corresponding values. This can be used to determine whether a particular value is acceptable for a given attribute, as well as to provide information about how the directory server should interact with existing values.

Sun Java System Directory Server 5 releases do not support the ability to reject values that violates the constraints of the associated attribute syntax. This capability has been added in Sun Java System Directory Server Version 6, although it is off by default for compatibility reasons (in order to support existing deployments where there are attribute values that violate the associated syntax). The directory server also supports the ability to reject values that violate the associated attribute syntax, and this is the default behavior for the purposes of standards compliance. For compatibility reasons, it is possible to completely disable this attribute syntax checking if necessary, but it is also possible to accept values that violate the associated syntax but log a warning message to the directory server's error log every time this occurs. Note, however, that because the directory server is much more standards compliant in its use of schema elements than the Sun Java System directory server, if attributes are allowed to have values that violate their associated syntax, then matching operations might not behave as expected with such values.

The Attribute Syntax Description Format

The attribute syntax description format is described in RFC 4512, section 4.1.5, as shown in this example:

```
SyntaxDescription = LPAREN WSP
numericoid                 ; object identifier
[ SP "DESC" SP qdstring ]  ; description
extensions WSP RPAREN      ; extensions
```

The attribute syntax description includes these elements:

`numericoid` The numeric OID used to uniquely identify the attribute syntax in the directory server.

`DESC` An optional description for the syntax. If it is provided, then it must be enclosed in single quotation marks.

extensions An optional set of extensions for the attribute syntax. The directory server does not currently support any extensions for use in attribute syntaxes.

The following example shows the attribute syntax description for the standard directory string syntax:

```
( 1.3.6.1.4.1.1466.115.121.1.15 DESC 'Directory String' )
```

In this case, the OID is `1.3.6.1.4.1.1466.115.121.1.15`, and the description is `Directory String`. There are no extensions.

Commonly Used Attribute Syntaxes

There are a number of attribute syntaxes defined in LDAP, both in the core protocol specification and in other related RFCs and Internet Drafts. Many of these attribute syntaxes are defined in RFC 4517 (LDAP Syntaxes and Matching Rules) in section 3.3. Some of the most commonly used attribute syntaxes include:

Directory String	The Directory String syntax is used to hold general-purpose string values containing one or more UTF-8 characters. Technically, empty values (that is, those with zero characters) are not allowed. Because the Sun Java System directory server has historically allowed empty values, the directory server offers a configuration option that can be used to allow it as well although it is disabled by default for standards compliance.
IA5 String	The IA5 String syntax is used to hold string values based on the IA5 character set, which is also known as the ASCII character set.
Printable String	The Printable String syntax is used to hold string values that contain one or more characters from the set of uppercase and lowercase letters, numeric digits, single quotes, left and right parentheses, plus sign, comma, hyphen, period, and equal sign.
Boolean	The Boolean syntax is used to hold values of either `TRUE` or `FALSE`. No other values are allowed for attributes with this syntax.
Integer	The Integer syntax is used to hold integer values, which must contain at least one digit. It can start with a hyphen to indicate a negative value. Zero can be used as the first digit only when the value is zero.
Octet String	The Octet String syntax is used to hold a set of zero or more bytes. It has been used to replace the former Binary syntax.
DN	The DN syntax is used to hold distinguished name values, comprised of zero or more RDN components. Values should be in the format specified in RFC 4514 (LDAP String Representation of Distinguished Names).

Understanding Attribute Types

Attribute types define the set of attributes that can be used in the directory server and how operations involving those attributes should be conducted. Among other things, it combines an attribute syntax and set of matching rules with a unique OID and human-readable names.

Attribute Type Description Format

The attribute type description format is described in RFC 4512, section 4.1.2 as shown here:

```
AttributeTypeDescription = LPAREN WSP
    numericoid                    ; object identifier
    [ SP "NAME" SP qdescrs ]      ; short names (descriptors)
    [ SP "DESC" SP qdstring ]     ; description
    [ SP "OBSOLETE" ]             ; not active
    [ SP "SUP" SP oid ]           ; supertype
    [ SP "EQUALITY" SP oid ]      ; equality matching rule
    [ SP "ORDERING" SP oid ]      ; ordering matching rule
    [ SP "SUBSTR" SP oid ]        ; substrings matching rule
    [ SP "SYNTAX" SP noidlen ]    ; value syntax
    [ SP "SINGLE-VALUE" ]         ; single-value
    [ SP "COLLECTIVE" ]           ; collective
    [ SP "NO-USER-MODIFICATION" ] ; not user modifiable
    [ SP "USAGE" SP usage ]       ; usage
    extensions WSP RPAREN         ; extensions

usage = "userApplications"     /  ; user
        "directoryOperation"   /  ; directory operational
        "distributedOperation" /  ; DSA-shared operational
        "dSAOperation"            ; DSA-specific operational
```

The attribute type description includes these elements:

`numericoid` — The numeric OID used to uniquely identify the attribute type in the directory server. Although the specification requires a numeric OID, the directory server also allows a non-numeric OID for the purpose of convenience and better compatibility with the Sun Java System directory server. In this case, the non-numeric OID should be the same as the name of the attribute type followed by the string `-oid`.

`NAME` — An optional set of human-readable names that can also be used to refer to the attribute type. If there is a single name, then it should be enclosed in single quotes. If there are multiple names, then they

	should each be enclosed in single quotes separated by spaces, and the entire set of names should be enclosed in parentheses.
DESC	An optional human-readable description. If there is a description, then it should be enclosed in single quotation marks.
OBSOLETE	An optional OBSOLETE flag that can be used to indicate whether the attribute type is active. If an attribute type is marked as OBSOLETE, then it means that it should not be referenced by any new elements created in the directory server.
SUP	An optional reference to the superior attribute type. If there is a superior type, then it may be referenced by either its OID or any of its human-readable names.
EQUALITY	An optional equality matching rule definition. If a specific equality matching rule is provided, then it can be referenced by either its OID or any of its human-readable names. If no equality matching rule is given, then the attribute type uses the default equality matching rule for the associated attribute syntax. If the attribute syntax does not have a default equality matching rule, then equality matching operations are not allowed for attributes of that type.
ORDERING	An optional ordering matching rule definition. If a specific ordering matching rule is provided, then it can be referenced by either its OID or any of its human-readable names. If no ordering matching rule is given, then the attribute type uses the default ordering matching rule for the associated attribute syntax. If the attribute syntax does not have a default ordering matching rule, then ordering matching operations are not allowed for attributes of that type.
SUBSTR	An optional substring matching rule definition. If a specific substring matching rule is provided, then it can be referenced by either its OID or any of its human-readable names. If no substring matching rule is given, then the attribute type uses the default substring matching rule for the associated attribute syntax. If the attribute syntax does not have a default substring matching rule, then substring matching operations are not allowed for attributes of that type.
SYNTAX	An optional attribute syntax for use with the attribute type. If it is provided, then it should be given as a numeric OID. The syntax identifier can also optionally contain an integer value enclosed in curly braces directly following the OID (without any spaces between the last digit of the OID and the opening curly brace),

which may be used to suggest a minimum upper bound on the length of values for attributes of that type. The directory server does not enforce any maximum length restrictions for attribute values, so if a length is given, then it is ignored.

`SINGLE-VALUE` An optional `SINGLE-VALUE` flag that indicates that attributes of that type are allowed to have only a single value in any entry in which they appear. If this flag is not present in the attribute type description, then attributes of that type are allowed to have multiple distinct values in the same entry.

`COLLECTIVE` An optional `COLLECTIVE` flag that indicates that the attributes of that type are assigned their values by virtue in their membership in some collection. Collective attributes are described in RFC 3671 (Collective Attributes in LDAP) and are one of the types of virtual attributes that are supported in the directory server.

`NO-USER-MODIFICATION` An optional `NO-USER-MODIFICATION` flag that indicates that values of attributes of that type cannot be modified by external clients (that is, the values can be modified only by internal processing within the directory server).

`USAGE` An optional usage specification that indicates how the attribute type is to be used. The following attribute usages are allowed:

`userApplications`	Used to store user data.
`directoryOperation`	Used to store data required for internal processing within the directory server.
`distributedOperation`	Used to store operational data that must be synchronized across servers in the topology.
`dSAOperation`	Used to store operational data that is specific to a particular directory server and should not be synchronized across the topology.

extensions An optional set of extensions for the attribute type. The directory server currently uses the following extensions for attribute types:

`X-ORIGIN`	Provides information about where the attribute type is defined (for example, whether it is defined by a particular RFC or Internet Draft or whether it is defined within the project).

X-SCHEMA-FILE	Indicates which schema file contains the attribute type definition (This extension is generally used for internal purposes only and is exposed to clients.)
X-APPROX	Indicates which approximate matching rule should be used for the attribute type. If this is specified, then its value should be the name or OID of a registered approximate matching rule.

For example, the following is the attribute type description for the standard uid attribute type:

```
( 0.9.2342.19200300.100.1.1 NAME 'uid' EQUALITY caseIgnoreMatch
SUBSTR caseIgnoreSubstringsMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{256}
X-ORIGIN 'RFC 4519' )
```

In this case, the OID is 0.9.2342.19200300.100.1.1. There is a single human-readable name of uid. The caseIgnoreMatch rule should be used for equality matching, and the caseIgnoreSubstringsMatch rule should be used for substring matching. The attribute type uses the directory string syntax with a suggested minimum upper bound of 256 characters, and the attribute type definition was taken from RFC 4519. There is no description or superior type specified. The attribute type is not marked OBSOLETE, SINGLE-VALUE, COLLECTIVE, or NO-USER-MODIFICATION. There is no ordering matching rule specified, which means that the directory server falls back on the default ordering rule used by the directory string syntax. There is no X-APPROX extension to specify the approximate matching rule so the default approximate rule for the directory string syntax is used there as well.

Attribute Type Inheritance

One attribute type can reference another as its superior class. This has two primary effects:

- The matching rule and attribute syntax specifications from the superior attribute type can be inherited by the subordinate type if the subordinate does not override the superior definition. For example, if the superior attribute type uses the IA5 String syntax, then the subordinate attribute type also uses the IA5 String syntax unless its definition overrides that by specifying an alternate syntax. According to the specification in RFC 4512, section 2.5.1, an attribute type can have a different syntax than its superior type only if the syntax for the subordinate type is a refinement of (that is, allows a subset of the values of) the syntax for the superior attribute type.
- The OID, any of the human-readable names associated with the superior attribute type, or both can be used to collectively reference all of the subordinate types. For example, the name attribute type is referenced as the superior type for the cn, sn, c, l, st, o, ou, title,

`givenName`, `initials`, `generationQualifier`, and `dmdName` attribute types. Therefore, a filter of `(name=test)` should match an entry if any attribute with one of those types has a value of `test`.

A subordinate attribute type cannot have a different usage than its superior type. That is, if the superior type is `userApplications`, then the subordinate type must also be `userApplications`. Similarly, if a superior type is declared `COLLECTIVE`, then the subtype must also be `COLLECTIVE`, but if the superior type is not `COLLECTIVE`, then the subordinate type must also not be `COLLECTIVE`.

Attribute Type Implementation

Attribute types in the directory server do not require any custom logic (beyond that already provided by the associated attribute syntax and matching rules). The attribute type description provides all the information that the directory server needs to define an attribute type. As such, all of the attribute types for use in the directory server should be specified in the schema configuration files. These files exist in the `resource/schema` directory in the source repository, and they are put in `config/schema` in the actual directory server instance. The directory server treats these files in much the same way as they are handled in the Sun Java System directory server.

All attribute type objects are instances of the `org.opends.server.types.AttributeType` class. This class primarily serves as a simple data structure that contains all of the properties of the attribute type description as specified in "Attribute Type Description Format" on page 49. Attribute type objects can be retrieved from the directory server schema using their OIDs or any of their human-readable names.

At the present time, the mechanism used to handle attribute types varies from the LDAPv3 specification in the following ways:

- The LDAPv3 specification states that a subordinate attribute type must have the same syntax as the superior type, or a refinement of that syntax. The directory server does not enforce this constraint because it does not have any way to determine whether one attribute syntax is a refinement of the syntax of the supertype.
- The directory server does not yet include support for collective attributes.
- The synchronization subsystem does not take attribute usage into account (for example, so that attribute types with a usage of `dSAOperation` are not synchronized).

Understanding Object Classes

Object classes are essentially named sets of attribute types that can be used to control the type of data that can be stored in entries. Note that the terms “object class” and “objectclass” (that is, with and without a space between the words) are generally used interchangeably.

Object Class Description Format

The object class description format is described in RFC 4512, section 4.1.1.

```
ObjectClassDescription = LPAREN WSP
numericoid                 ; object identifier
[ SP "NAME" SP qdescrs ]   ; short names (descriptors)
[ SP "DESC" SP qdstring ]  ; description
[ SP "OBSOLETE" ]          ; not active
[ SP "SUP" SP oids ]       ; superior object classes
[ SP kind ]                ; kind of class
[ SP "MUST" SP oids ]      ; attribute types
[ SP "MAY" SP oids ]       ; attribute types
extensions WSP RPAREN
kind = "ABSTRACT" / "STRUCTURAL" / "AUXILIARY"
```

The object class description includes these elements:

`numericoid` The numeric OID used to uniquely identify the object class in the directory server. Although the specification requires a numeric OID, the directory server also allows a non-numeric OID for the purpose of convenience and better compatibility with the Sun Java System directory server. In this case, the non-numeric OID should be the same as the name of the object class followed by the string `-oid`.

`NAME` An optional set of human-readable names that can be used to refer to the object class. If there is a single name, then it should be enclosed in single quotes. If there are multiple names, then they should each be enclosed in single quotes separated by spaces, and the entire set of names should be enclosed in parentheses.

`DESC` An optional human-readable description. If there is a description, then it should be enclosed in single quotation marks.

`OBSOLETE` An optional `OBSOLETE` flag that can be used to indicate whether the object class is active. If an object class is marked as `OBSOLETE`, then it should not be referenced by any new elements created in the directory server.

`SUP` An optional set of one or more superior classes for the object class. Note that although technically the specification allows an object class to have multiple

superior classes, the directory server currently only supports a single superior class. In this case, the SUP keyword should be followed by a space and the name or OID of the superior class. If there are multiple superior classes, then they should be separated by dollar signs and the entire set of superior classes should be enclosed in parentheses.

`kind` — An optional keyword that specifies the kind of object class that is being defined. If this is specified, then it must be one of ABSTRACT, STRUCTURAL, or AUXILIARY. If no value is specified, then the object class is considered STRUCTURAL.

`MUST` — An optional set of attribute types for attributes that are required to be present (that is, have at least one value) in entries with that object class. If there is only a single required attribute, then the MUSTkeyword should be followed by the name or OID of that attribute type. If there are multiple required attribute types, then they should be separated by dollar signs and the entire set of required attribute types should be enclosed in parentheses.

`MAY` — An optional set of optional attribute types for attributes that are allowed (but not required) to be present in entries with that object class. If there is only a single optional attribute, then the MAYkeyword should be followed by the name or OID of that attribute type. If there are multiple optional attribute types, then they should be separated by dollar signs and the entire set of optional attribute types should be enclosed in parentheses.

extensions — An optional set of extensions for the object class. The directory server currently uses the following extensions for object classes:

- `X-ORIGIN` — Provides information about where the object class is defined (for example, whether it came from a particular RFC or Internet Draft or if it is defined within the project).
- `X-SCHEMA-FILE` — Indicates which schema file contains the object class definition (This extension is generally used for internal purposes only and is exposed to clients.)

For example, the following is the object class description for the standard `person` object class:

```
( 2.5.6.6 NAME 'person' SUP top STRUCTURAL MUST ( sn $ cn )
MAY ( userPassword $ telephoneNumber $ seeAlso $ description )
X-ORIGIN 'RFC 4519' )
```

In this case, the OID is `2.5.6.6`. There is a single human-readable name of `person`. The superior class is `top`. The kind is `STRUCTURAL`. Any entry containing the `person` object class is required to include the `sn` and `cn` attributes and is allowed to include the `userPassword`, `telephoneNumber`, `seeAlso`, and `description` attributes. The object class definition is taken from RFC 4519. There is no description, and the object class is not considered `OBSOLETE`.

Object Class Kinds

As described in "Object Class Description Format" on page 54, all object classes must have a kind of either `ABSTRACT`, `STRUCTURAL`, or `AUXILIARY`:

- `ABSTRACT` object classes are intended only to be extended by other object classes. An entry must not contain any abstract class unless it also contains a structural or auxiliary class that derives from that abstract class (that is, includes a non-abstract object class which has the abstract class in its inheritance chain). All entries must contain at least the `top` abstract object class in the inheritance chain for their structural class. They may or may not contain other abstract classes in the inheritance chains for their structural class or any of their auxiliary classes.
- `STRUCTURAL` object classes are intended to define the crux of what an entry represents. Every entry must include exactly one structural object class chain, and the root of that chain must ultimately be the `top` abstract object class. The structural object class for an entry cannot be changed.
- `AUXILIARY` object classes are intended to define additional qualities of entries. An entry can contain zero or more auxiliary classes, and the set of auxiliary classes associated with an entry can change over time.

The model represented by object class kinds translates very neatly to the model used by the Java programming language. Abstract LDAP object classes map directly to Java abstract classes, auxiliary LDAP object classes map directly to Java interfaces, and structural LDAP object classes map directly to Java concrete (non-abstract) classes. Just as Java classes must extend exactly one superclass but can implement any number of interfaces, so must LDAP entries contain exactly one structural class chain but can include any number of auxiliary class chains. Similarly, just as it is not possible to directly instantiate an abstract Java class, it is also not possible to create an LDAP entry containing only abstract object classes.

The Sun Java System directory server has never enforced many of the restrictions noted here around object class kinds. In particular, it would allow the creation of entries that did not contain any structural object class chain and would also allow the creation of entries containing multiple structural object class chains. This means that deployments using the Sun Java System directory server can contain entries that violate this constraint. The directory server does not allow this behavior by default, but for the sake of compatibility with existing Sun Java System directory server deployments, it is possible to configure the directory server to allow entries to violate this constraint, optionally writing a message to the directory server's error log each time this condition is detected. However, if there are entries that do not contain exactly one structural object class, then some schema elements like name forms and DIT content rules that depend on this constraint might not work as expected in all cases.

Object Class Inheritance

As specified in "Object Class Description Format" on page 54, object classes can have zero or more superior classes (although at the present time, the directory server supports at most one superior class). If an object class references a superior class, then all of the required and optional attributes associated with that superior class are also associated with the subordinate class.

The following restrictions exist for object class inheritance:

- `ABSTRACT` object classes can inherit only from other abstract classes. They cannot be subordinate to structural or auxiliary classes.
- `STRUCTURAL` object classes can inherit only from abstract classes or other structural classes. They cannot be subordinate to auxiliary object classes.
- `AUXILIARY` object classes can inherit only from abstract classes or other auxiliary classes. They cannot be subordinate to structural object classes.
- All `STRUCTURAL` object classes must ultimately inherit from the top abstract object class. The net effect of this is that every entry in the directory server must include the `top` object class and so must also include the `objectClass` attribute type, which is required by `the` top object class).

Directory Server Object Class Implementation

Object classes in the directory server do not require any custom logic, so they can be implemented as data structures that are populated from information contained in the schema configuration files, much in the same way as for attribute type definitions. All object class instances in the directory server are instances of the `org.opends.server.types.ObjectClass` class. Object class objects can be retrieved from the directory server schema using their OIDs or any of their human-readable names.

At the present time, the mechanism used to handle object classes varies from the LDAPv3 specification in that object classes are allowed to have at most one superior class, whereas the specification allows multiple superior classes in some cases.

Understanding Name Forms

Name forms can be used to define a mechanism for naming entries in the directory server. In particular, a name form specifies one or more attribute types that must be present in the RDN of an entry with a given structural object class. A name form can also specify zero or more attribute types, which can optionally be present in the RDN.

Each structural object class defined in the directory server schema can be associated with at most one name form. If a name form is defined for a given structural object class, then the associated name form is enforced for any add or modify DN operations for entries containing that object class. If a structural object class is not associated with a name form, then any attribute type that is allowed to exist in the target entry can be used as a naming attribute type.

Name forms are a new feature in , and they are not supported in the Sun Java System directory server.

Name Form Description Format

The name form description format is described in RFC 4512, section 4.1.7.2, as shown here:

```
NameFormDescription = LPAREN WSP
numericoid                 ; object identifier
[ SP "NAME" SP qdescrs ]   ; short names (descriptors)
[ SP "DESC" SP qdstring ]  ; description
[ SP "OBSOLETE" ]          ; not active
SP "OC" SP oid             ; structural object class
SP "MUST" SP oids          ; attribute types
[ SP "MAY" SP oids ]       ; attribute types
extensions WSP RPAREN      ; extensions
```

The name form description includes these elements:

`numericoid` The numeric OID used to uniquely identify the name form in the directory server. Although the specification requires a numeric OID, the directory server also allows a non-numeric OID for the purpose of convenience. In this case, the non-numeric OID should be the same as the name of the name form followed by the string `-oid`.

`NAME` An optional set of human-readable names that can be used to refer to the name form. If there is a single name, then it should be enclosed in single quotes. If there are multiple names, then they should each be enclosed in single quotes separated by spaces, and the entire set of names should be enclosed in parentheses.

`DESC` An optional human-readable description. If a description is present, then it should be enclosed in single quotation marks.

`OBSOLETE` An optional `OBSOLETE` flag that can be used to indicate whether the name form is active. If a name form is marked as `OBSOLETE`, then it should not be in effect within the directory server, nor should it be possible to create any other elements that depend on it.

`OC` The name or OID of the structural object class with which the name form is associated.

`MUST` The names or OIDs of one or more attribute types that must be present in the RDN for any entry with the specified structural class. If there is only a single required attribute type, then only its name or OID needs to be given. If there are multiple required attribute types, then they should be separated by spaces and dollar signs, and the entire set of required attribute types should be enclosed in parentheses.

`MAY` The names or OIDs of zero or more attribute types that can optionally be present in the RDN for any entry with the specified structural class. If there is only a single optional attribute type, then only its name or OID needs to be given. If there are multiple optional attribute types, then they should be separated by spaces and dollar signs, and the entire set of optional attribute types should be enclosed in parentheses.

extensions An optional set of extensions for the name form. The directory server currently uses the following extensions for name forms:

- `X-ORIGIN`Provides information about where the name form is defined (for example, whether it came from a particular RFC or Internet Draft or whether it is defined within the project.).
- `X-SCHEMA-FILE`Indicates which schema file contains the name form definition (This extension is generally used for internal purposes only and is exposed to clients.)

For example, the following is the name form description for the `uddiBusinessEntityNameForm` name form defined in RFC 4403:

```
( 1.3.6.1.1.10.15.1 NAME 'uddiBusinessEntityNameForm'
OC uddiBusinessEntity MUST ( uddiBusinessKey ) X-ORIGIN 'RFC 4403' )
```

In this case, the numeric OID is `1.3.6.1.1.10.15.1` and the human-readable name is `uddiBusinessEntityNameForm`. Entries with the `uddiBusinessEntity` structural object class are required to use `uddiBusinessKey` as their only RDN attribute type. There is no description, nor are there any other attribute types that can optionally be included in the associated entries. The name form is not marked `OBSOLETE`.

Name Form Implementation

As for attribute types and object classes, name forms can be defined purely from the schema configuration files, in this case using the name form description syntax detailed in "Name Form Description Format" on page 58. All name form instances in the directory server are instances of the `org.opends.server.types.NameForm` class. Name form objects can be retrieved by the

directory server schema using their OID, any of their human-readable names, or by the structural object class with which they are associated.

Understanding DIT Content Rules

DIT content rules provide a mechanism for defining the content that can appear in an entry. As with name forms, at most one DIT content rule can be associated with an entry based on its structural object class. If such a rule exists for an entry, then it works in conjunction with the object classes contained in that entry to define which attribute types must, may, and must not be present in the entry, as well as which auxiliary classes that it may include.

DIT content rules are a new feature that is available in the directory server and are not supported in the Sun Java System directory server.

DIT Content Rule Description Format

The DIT content rule description format is described in RFC 4512, section 4.1.6, as shown here:

```
DITContentRuleDescription = LPAREN WSP
numericoid                 ; object identifier
[ SP "NAME" SP qdescrs ]   ; short names (descriptors)
[ SP "DESC" SP qdstring ]  ; description
[ SP "OBSOLETE" ]          ; not active
[ SP "AUX" SP oids ]       ; auxiliary object classes
[ SP "MUST" SP oids ]      ; attribute types
[ SP "MAY" SP oids ]       ; attribute types
[ SP "NOT" SP oids ]       ; attribute types
extensions WSP RPAREN      ; extensions
```

The DIT content rule description includes these elements:

`numericoid` The numeric OID of the structural object class with which the DIT content rule is associated. Although the specification requires a numeric OID, this `numericoid` should match the OID specified for the associated object class, so if the object class OID was non-numeric, then this OID should be as well.

`NAME` An optional set of human-readable names used to refer to the DIT content rule. If there is a single name, then it should be enclosed in single quotes. If there are multiple names, then they should each be enclosed in single quotes separated by spaces, and the entire set of names should be enclosed in parentheses.

`DESC` An optional human-readable description. If a description is provided, then it should be enclosed in single quotation marks.

`OBSOLETE` An optional `OBSOLETE` flag that can be used to indicate whether the DIT content rule is active. If a DIT content rule is marked as `OBSOLETE`, then it should not be in effect within the directory server.

`AUX` An optional list of auxiliary object classes that can be present in entries with the associated structural class. If no values are provided, then such entries are not allowed to have any auxiliary object classes. Values should be specified as one or more of the names or OIDs of the allowed auxiliary classes. If multiple auxiliary classes are allowed, then separate them by spaces and dollar signs, and enclose the entire set of names in parentheses.

`MUST` An optional list of attribute types that are required to be present in entries with the associated structural class. This is in addition to the attribute types required by the object classes included in the entry, and these additional attribute types do not need to be allowed by any of those object classes. Values should be specified as one or more of the names or OIDs of the required attribute types. If multiple attribute types are required, then separate them by spaces and dollar signs, and enclose the entire set of required attribute types in parentheses.

`MAY` An optional list of attribute types that can optionally be present in entries with the associated structural class. This is in addition to the attribute types allowed by the object classes included in the entry. Values should be specified as one or more of the names or OIDs of the optional attribute types. If there are multiple optional attribute types, separate them by spaces and dollar signs and enclose the entire set of optional attribute types in parentheses.

`NOT` An optional list of attribute types that are prohibited from being present in entries with the associated structural class. This list cannot include any attribute types that are required by the structural class or any of the allowed auxiliary classes, but it can be used to prevent the inclusion of attribute types that would otherwise be allowed by one of those object classes. Values should be specified as one or more of the names or OIDs of the prohibited attribute types. If multiple types are prohibited , then separate them by spaces and dollar signs, and enclose the entire set of prohibited attribute types in parentheses.

extensions An optional set of extensions for the DIT content rule. The directory server currently uses the following extensions for DIT content rules:

`X-ORIGIN` Provides information about where the DIT content rule is defined (for example, whether it came from a particular RFC or Internet Draft, or whether it is defined within the project)

`X-SCHEMA-FILE` Indicates which schema file contains the DIT content rule definition (This extension is generally used for internal purposes only and is exposed to clients.)

The following provides an example of a DIT content rule description:

```
( 2.16.840.1.113730.3.2.2 NAME 'inetOrgPersonContentRule'
AUX ( posixAccount $ shadowAccount $ authPasswordObject )
MUST uid )
```

In this case, the numeric OID is `2.16.840.1.113730.3.2.2`, which is the OID for the `inetOrgPerson` structural object class. It has a human-readable name of `inetOrgPersonContentRule` and no description. It allows entries containing the `inetOrgPerson` object class to also contain the `posixAccount`, `shadowAccount`, and `authPasswordObject` auxiliary classes, and those entries must contain the `uid` attribute type. It is not marked `OBSOLETE`, and it does not define any additional optional or prohibited attribute types, nor does it include any extensions.

DIT Content Rule Implementation

DIT content rules can be defined purely from the schema configuration files using the DIT content rule description syntax provided in "DIT Content Rule Description Format" on page 60. All DIT content rule objects are instances of the `org.opends.server.types.DITContentRule` class. DIT content rule objects can be retrieved from the directory server schema using the structural object class with which they are associated.

At the present time, the mechanism used to handle DIT content rules varies from the LDAPv3 specification. The LDAPv3 specification states that if the structural object class used in an entry does not have a corresponding DIT content rule, then that entry is not allowed to contain any auxiliary object classes. Because the Sun Java System directory server does not support DIT content rules, the directory server does not prevent the use of auxiliary object classes in entries for which there is no corresponding DIT content rule. If it is desirable to prevent the inclusion of auxiliary classes in a given type of entry, then a DIT content rule should be created with no allowed auxiliary classes to cover entries with the appropriate structural object class.

Understanding DIT Structure Rules

DIT structure rules can be used to define the allowed hierarchical structure of the directory data. In particular, they make it possible to specify what types of entries are allowed to exist as immediate children of entries with a specified structural object class. For example, only entries with the `inetOrgPerson` structural class can be immediate children of entries with an `organizationalUnit` structural object class.

DIT structure rules are themselves hierarchical. Each DIT structure rule is assigned a rule ID, which is an integer value, and is also associated with a name form (which in turn links it to a

structural object class). DIT structure rules can also reference one or more superior DIT structure rules, and this provides the mechanism for controlling the data hierarchy. If a DIT structure rule does not specify any superior rules, then entries containing its associated structural object class are allowed to exist at the root of the associated schema. If a DIT structure does specify one or more superior rules, then entries with an associated structural object class are allowed to exist only below entries containing the structural object class of one of those superior rules.

DIT Structure Rule Description Format

The DIT structure rule description format is described in RFC 4512, section 4.1.7.1, as shown here:

```
DITStructureRuleDescription = LPAREN WSP
ruleid                      ; rule identifier
[ SP "NAME" SP qdescrs ]    ; short names (descriptors)
[ SP "DESC" SP qdstring ]   ; description
[ SP "OBSOLETE" ]           ; not active
SP "FORM" SP oid            ; NameForm
[ SP "SUP" ruleids ]        ; superior rules
extensions WSP RPAREN       ; extensions
ruleids = ruleid / ( LPAREN WSP ruleidlist WSP RPAREN )
ruleidlist = ruleid *( SP ruleid )
ruleid = number
```

The DIT structure rule description includes these elements:

ruleid — The integer rule ID assigned to the DIT structure rule. It must be unique among all other DIT structure rules in the schema.

`NAME` — An optional set of human-readable names that can be used to refer to the DIT structure rule. If there is a single name, then it should be enclosed in single quotes. If there are multiple names, then they should each be enclosed in single quotes separated by spaces, and the entire set of names should be enclosed in parentheses.

`DESC` — An optional human-readable description. If a description is provided, then it should be enclosed in single quotes.

`OBSOLETE` — An optional `OBSOLETE` flag that can be used to indicate whether the DIT structure rule is active. If it is marked `OBSOLETE`, then it should not be taken into account when entries are created or moved.

`FORM` — The name or OID of the name form with which the DIT structure rule is associated. As mentioned in "Understanding DIT Structure Rules" on page 62, the name form associates the DIT structure rule with a structural object class.

`SUP` An optional set of superior rule IDs for the DIT structure rule. If there are multiple superior rule IDs, then separate them by spaces, and enclose the entire set of superior rule IDs in parentheses. It is permissible for multiple DIT structure rules to use overlapping sets of superior rule IDs.

extensions An optional set of extensions for the DIT structure rule. The directory server currently uses the following extensions for DIT structure rules:

- `X-ORIGIN` Provides information about where the DIT structure rule is defined (for example, whether it came from a particular RFC or Internet Draft, or whether it is defined within the project)
- `X-SCHEMA-FILE` Indicates which schema file contains the DIT structure rule definition (This extension is generally used for internal purposes only and is exposed to clients.)

The following example is the DIT structure rule definition for the `uddiContactStructureRule` DIT structure rule:

```
dITStructureRule:
( 2 NAME 'uddiContactStructureRule' FORM uddiContactNameForm SUP ( 1 )
X-ORIGIN 'RFC 4403' )
```

In this case, the rule ID is `2`, and the human-readable name is `uddiContactStructureRule`. It is associated with the `uddiContactNameForm` name form (which in turn links it to the `uddiContact` object class), and it has a superior rule ID of `1`. It was defined in RFC 4403. It does not have a description, nor is it marked `OBSOLETE`.

DIT Structure Rules and Multiple Schemas

DIT structure rules can provide a mechanism for placing constraints on the directory server hierarchy, but in order to maximize their utility, it may be necessary to use them in conjunction with support for multiple schemas. For example, consider a directory with a naming context of `dc=example,dc=com`, below which are two branches: `ou=People,dc=example,dc=com` and `ou=Groups,dc=example,dc=com`. If you want to allow only `inetOrgPerson` entries below the `ou=People` branch and only `groupOfNames` entries below the `ou=Groups` branch, then that can be fully accomplished only if there are different schemas that govern the `ou=People` and `ou=Groups` branches.

If there were a single schema governing the entire directory server, then you can imagine that it would have four DIT structure rules:

- `dITStructureRule: ( 11 NAME 'domainStructureRule' FORM domainNameForm )`
- `dITStructureRule: ( 12 NAME 'organizationalUnitStructureRule' FORM organizationalUnitNameForm SUP 11 )`

- `dITStructureRule: ( 13 NAME 'inetOrgPersonStructureRule' FORM inetOrgPersonNameForm SUP 12 )`
- `dITStructureRule: ( 14 NAME 'groupOfNamesStructureRule' FORM groupOfNamesNameForm SUP 12 )`

This set of DIT structure rules would allow the structure described above, but it would also allow the creation of group entries below the `ou=People` branch and the creation of user entries below the `ou=Groups` branch. The only way to prevent that using DIT structure rules would be to define separate schemas for the `ou=People` and `ou=Groups` branches and define only the `inetOrgPersonStructureRule` rule in the schema for the `ou=People` branch, and only define the `groupOfNamesStructureRule` rule in the schema for the `ou=Groups` branch.

DIT Structure Rule Implementation

DIT structure rules can be defined purely from the information contained in the schema configuration files using the DIT structure rule description syntax provided in "DIT Structure Rule Description Format" on page 63. All DIT structure rule objects are instances of the `org.opends.server.types.DITStructureRule` class. DIT structure rules can be retrieved from the directory server schema using either the integer rule IDs or their associated name forms.

Understanding Matching Rule Uses

Matching rule uses can be used to specify which attribute types can be used in conjunction with a given matching rule when processing a search request with an extensible match filter component. If that extensible match component includes both an attribute type and a matching rule ID, then the directory server checks to see if there is a matching rule use for the associated matching rule, and if there is, it ensures that it allows the specified attribute type to be used with that matching rule.

The matching rule use description format is described in RFC 4512, section 4.1.4, as shown here:

```
MatchingRuleUseDescription = LPAREN WSP
numericoid                 ; object identifier
[ SP "NAME" SP qdescrs ]   ; short names (descriptors)
[ SP "DESC" SP qdstring ]  ; description
[ SP "OBSOLETE" ]          ; not active
SP "APPLIES" SP oids       ; attribute types
extensions WSP RPAREN      ; extensions
```

The matching rule use description includes these elements:

numericoid The numeric OID of the matching rule with which the matching rule use is associated. There can be only one matching rule use associated with a given matching rule.

`NAME` An optional set of human-readable names that may be used to refer to the matching rule use. If there is a single name, then it should be enclosed in single quotes. If there are multiple names, then they should each be enclosed in single quotes and separated by spaces, and the entire set of names should be enclosed in parentheses.

`DESC` An optional human-readable description. If there is a description, then it should be enclosed in single quotes.

`OBSOLETE` An optional `OBSOLETE` flag that can be used to indicate whether the matching rule use is active. If it is marked `OBSOLETE`, then it should not be taken into account when determining whether to allow an extensible match filter.

`APPLIES` A set of one or more attribute types that can be used in conjunction with the associated matching rule. If there is an associated attribute type, then its name or OID can be used. If there are multiple attribute types, then separate them by spaces and dollar signs, and enclose the entire set of associated attribute types in parentheses.

extensions An optional set of extensions for the matching rule use. The directory server currently uses the following extensions for matching rule uses:

- `X-ORIGIN` Provides information about where the matching rule use is defined (for example, whether it came from a particular RFC or Internet Draft, or whether it is defined within the project)
- `X-SCHEMA-FILE` Indicates which schema file contains the matching rule use definition (This extension is generally used for internal purposes only and is exposed to clients.)

The following example shows a matching rule use description:

```
( 1.3.6.1.4.1.26027.1.999.10 NAME 'testAddMRUSuccessful' APPLIES cn )
```

In this case, the numeric OID is `1.3.6.1.4.1.26027.1.999.10`, the single human-readable name is `testAddMRUSuccessful`, and it can be used in conjunction with the `cn` attribute. It does not have a description, it is not marked `OBSOLETE`, and it does not have any extensions.

Matching Rule Use Implementation

Matching rule uses can be created using only the information in the schema configuration files, in the matching rule use description syntax. All matching rule use objects are instances of the `org.opends.server.types.MatchingRuleUse` class. Matching rule uses can be retrieved from the directory server schema by their associated matching rule.

Index Databases

Index databases enable search requests to be processed efficiently. The children and subtree system indexes are dedicated to processing one-level and subtree search scopes respectively. Additional attribute indexes are configured as needed to process components of a search filter. Each record in an index database maps a key to an Entry ID List.

The following sections describe the index databases and the search evaluation process:

Index Entry Limit

The index entry limit is a configuration limit that can be used to control the maximum number of entries that is allowed to match any given index key (that is, the maximum size of an ID list). This provides a mechanism for limiting the performance impact for maintaining index keys that match a large percentage of the entries in the server. In cases where large ID lists might be required, performing an unindexed search can often be faster than one that is indexed.

Attribute Equality Index

An equality index is a type of indexwhich is used to identify efficiently which entries are exactly equal to a given assertion value. An equality index may only be maintained for attributes that have a corresponding equality matching rule. That matching rule will be used to normalize values to use as index keys, and the value for that key will be the ID list containing the entry ID of the entries with values that are equal to that normalized value.

Attribute Presence Index

A presence index is a type of index that is used to keep track of the entries that have at least one value for a specified attribute. There is only a single presence index key per attribute, and itsID list contains the entry IDs for all entries that contain the specified attribute.

Attribute Substring Index

A substring index is a type of index that is used to keep track of which entries contain specific substrings. Index keys for a substring index consist of six-character substrings taken from attribute values and the corresponding values are ID list containing the entry ID of the entries containing those substrings. The attribute's substring matching rule is used to normalize the values for the index keys, and substring indexes cannot be defined for attributes that do not contain substring matching rules.

Attribute Ordering Index

An ordering index is a type of index that is used to keep track of the relative order of values for an attribute. It is very similar to an equality index except that it uses an ordering matching rule instead of an equality matching rule to normalized value the values. Ordering indexes may not be maintained for attributes that do not have a corresponding ordering matching rule.

Search Evaluation

To process an LDAP search operation, the JE back end first applies the search filter to the attribute indexes to obtain an initial set of candidate entry IDs. Then the candidates can be further refined by fetching subordinates of the base entry from either the children or subtree databases (depending on the search scope).

If a candidate set is obtained, the search is considered to be “indexed”. Each candidate entry is fetched from the entry database and returned to the client if it matches the search scope and filter.

If no candidate set is obtained (because of a lack of indexes or some of the index values having exceeded the index entry limit), the search is considered to be “not-indexed“ In this case, a cursor is opened on the DN database at the base entry to iterate through the DN/ID records, fetching and filtering the corresponding entries until all the entries under the search base have been processed.

Whenever the number of candidate entry IDs from the indexes is found to be 10 or less, no further attempt is made to reduce the number of candidates. Instead those entries are

immediately fetched from the entry database and filtered, on the assumption that this is quicker than continuing to read the index databases. This can pay off for AND search filters in which the first component is the most specific.

Search AND filters are also rearranged so that components that are slow to evaluate (greater-than-or-equal, less-than-or-equal) come after components that are generally faster (for example, equality).

Understanding Directory Server Plug-Ins

Plug-ins are responsible for injecting custom logic into the course of processing an operation or other well-defined points within the directory server. This section describes the following topics:

Plug-In Types

The directory server supports several kinds of plug-ins, including:

Pre-parse plug-ins	Can be used to alter the contents of a request before the directory server begins to process it.
Pre-operation plug-ins	Can be used to perform any processing that might be required immediately before the directory server performs the core action associated with that operation (for example, before an updated entry is stored in the back end for a modify operation).
Post-operation plug-ins	Can be used to perform any processing that might be required immediately after the directory server performs the core action associated with that operation but before the response is sent to the client.
Post-response plug-ins	Can be used to perform any processing that might be required for an operation that does not need to occur before the response is sent to the client.
Search result entry plug-ins	Can be used to perform any processing that might be required immediately before the directory server sends a search result entry to a client (including altering the contents of the entry or preventing it from being sent).
Search result reference plug-ins	Can be used to perform any processing that might be required immediately before the directory server sends a

	search reference (that is, a referral) to a client (including altering the contents of the referral or preventing it from being sent).
Intermediate response plug-ins	Can be used to perform any processing that might be required immediately before the directory server sends an intermediate response to a client (including altering the contents of the intermediate response or preventing it from being sent).
Post-connect plug-ins	Can be used to perform any processing that might be required immediately after the directory server accepts a new connection from a client (including terminating that connection).
Post-disconnect plug-ins	Can be used to perform any processing that might be required immediately after a client connection is closed for some reason (regardless of whether the closure was initiated by the client or the directory server).
Startup plug-ins	Can be used to perform any processing that might be required when the directory server is in the process of being started.
Shutdown plug-ins	Can be used to perform any processing that might be required when the directory server is in the process of performing a graceful shutdown.
LDIF import plug-ins	Can be used to perform any processing that might be required for any entries read during the LDIF import process (including altering the contents of the entry or preventing it from being imported).
LDIF export plug-ins	Can be used to perform any processing that might be required for any entries to be written during the LDIF export process (including altering the contents of the entry or preventing it from being exported).

A plug-in can have multiple plug-in types, and multiple plug-ins with a given plug-in type can be defined in the directory server. If multiple plug-ins are defined with the same plug-in type, the order in which they are invoked during processing is undefined by default. If you require the plug-ins to be invoked in a specific order (for example, if the processing performed by one plug-in depends on the result of another), you can specify the order in which the plug-ins are invoked. For more information, see "Modifying the Plug-In Configuration" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

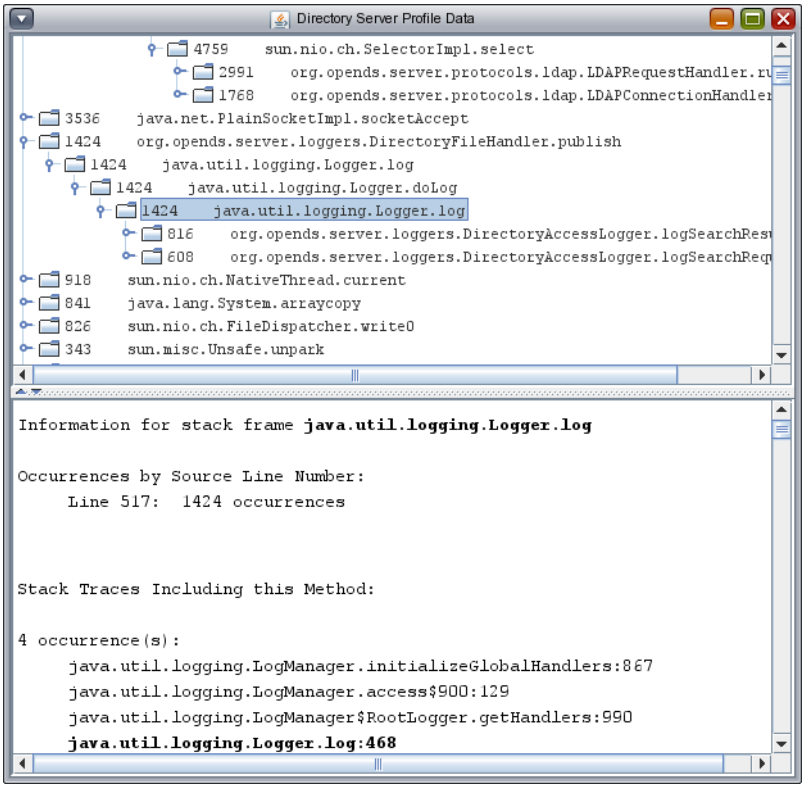
Understanding the Profiler Plug-In and the Profiler Tool

The profiler plug-in provides a basic in-server profiling capability that can be used to monitor performance characteristics and identify potential hot spots in the directory server. It is based on the relatively simple principle of periodically capturing stack traces from all threads in the JVM and collecting them so that they can be aggregated and correlated for later analysis. This tool can be useful for debugging performance problems without the need for any other external tools and without the need to restart the directory server to start and stop profiling. The plug-in has no impact on performance while it is not in use, and it can be relatively inexpensive even when it is capturing data.

The data file that the profiler collects is written in a binary form, and it is necessary to use a tool provided with the directory server in order to be able to view its contents. It can be invoked as a purely command-line tool, but it can also provide a graphical interface to examine the data. To run the tool, go into the directory server instance root and issue the following command (all on one line):

```
$ java -cp lib/OpenDS.jar org.opends.server.plugins.profiler.ProfileViewer \
-f dataFile -g
```

where `dataFile` is the path to the data file containing the data that was collected by the profiler plug-in. This command displays the profiler GUI interface, which looks like the following:



The top panel of the data displayed includes a list of all the leaf methods seen by the profiler, ordered by the number of profile intervals in which that method was seen. Each node can be expanded to show the methods that called it and the number of times they appeared in that particular stack. The bottom panel displays a list of all stack traces where the highlighted method appears.

To invoke the profiler data viewer in command-line mode (to create text-only output), remove the `-g` option from the command line.

Directory Server Replication

Directory server replication uses a loosely consistent multi-master model. All directory servers that are part of a replication topology can accept read and write operations.

The following architectural topics are targeted at developers and at users who want to understand the internals of the replication mechanism. It is not necessary to read these topics just to be able to use replication. For information about configuring and using replication, see "Replicating Data" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

The following topics describe the architecture of the directory server replication functionality.

Overview of the Directory Server Replication Architecture

The directory server replication model is a loosely consistent, multi-master model. In other words, all directory servers in a replicated topology can process both read and write operations.

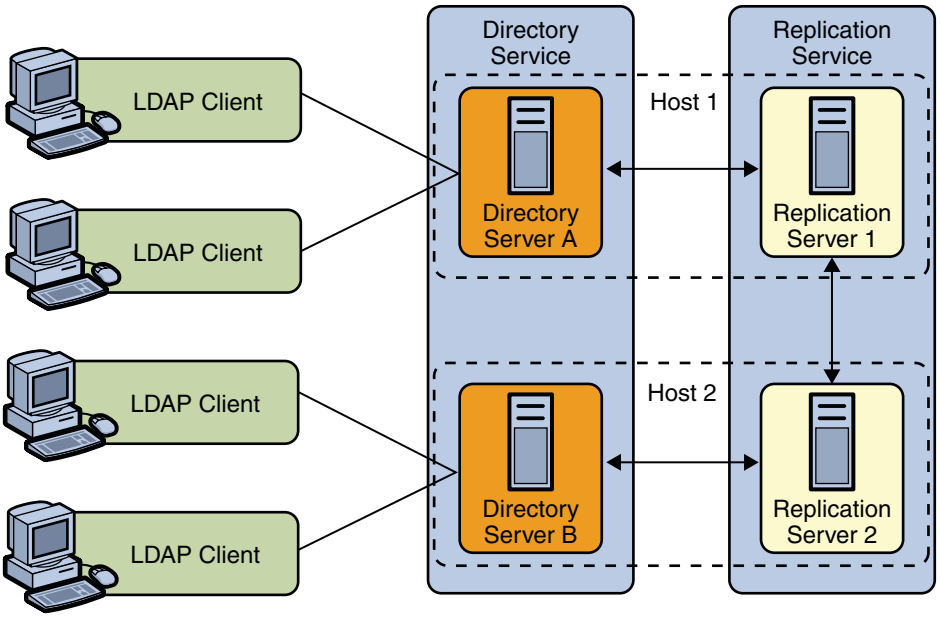
Replication is built around a centralized publish-subscribe architecture. Each directory server communicates with a central service, and uses the central service to publish its own changes and to receive notification about changes on other directory servers. This central service is called the *replication service*.

The replication service can be made highly available by using multiple server instances running on multiple hosts. Within the replication architecture, a server instance that provides the replication service is called a *replication server*. A server instance that provides the directory service is called a *directory server*.

The topics in this section describe the replication architecture and the various elements that make up this architecture.

Basic Replication Architecture

The basic replication architecture is shown in the following illustration.



Note – In the current release, the `setup` and `dsreplication` commands only support a scenario in which the replication server and the directory server exist on the same host (or virtual machine). However, the replication architecture is designed to support replication server and directory server instances on different machines. You can use `dsconfig` to configure such a scenario, but a good understanding of the replication architecture is required to do so.

At startup time, each directory server selects a single replication server and connects to it. The directory server sends all changes that it processes to that replication server, and receives all changes from other servers in the topology through that replication server. Each replication server is connected to every other replication server in the topology.

When a replication server receives a change from a directory server, the replication server forwards the change to all the other replication servers in the topology. These replication servers in turn forward the change to all the directory servers to which they are connected. When a replication server receives a change from another replication server, the replication server forwards the change to the directory servers to which it is connected, but not to other

replication servers. A directory server never sends a change directly to another directory server. This architecture ensures that all changes are forwarded to all servers without requiring complex negotiation.

Every change is assigned a *change number* by the directory server that originally processed the change. The change number is used to identify the change throughout its processing. A replication server maintains changes in stable storage so that older changes can be resent to directory servers that were not connected when the change occurred or that fell behind, becoming temporarily unable to receive all the changes at the time they were processed. For more information, see "Replication Change Numbers" on page 80.

The current update state of each directory server is maintained by keeping a record of the last changes that the directory server processed. When a directory server connects to a replication server, the replication server uses this record to determine the first change in the list of updates to send to the directory server.

Because multiple directory servers can process updates simultaneously, an update operation on one directory server can conflict with another update operation that affects the same entries on another directory server. Each directory server resolves conflicts when it replays operations from other directory servers, so that all directory server data eventually converges.

Conflicts can occur because of conflicting modify operations, called *modify conflicts*. Conflicts can also occur because of conflicting add, delete, or modRDN operations, called *naming conflicts*. To resolve conflicts in a coherent way, directory servers maintain a history of successive changes to each entry. This history is called *historical information*. Historical information is stored as an operational attribute inside the entry on which the changes occurred. For more information, see "Historical Information and Conflict Resolution" on page 84.

Replication Servers

A replication server performs the following tasks:

- Manages connections from directory servers
- Connects to other replication servers
- Listens for connections from other replication servers
- Receives changes from directory servers
- Forwards changes to directory servers and other replication servers
- Saves changes to stable storage, which includes trimming older operations

Replication servers are not the same as directory servers. However, like directory servers, replication servers use a configuration file, can be configured and monitored online, and can be backed up and restored. Replication servers are therefore always LDAP servers or JMX servers, even though replication servers do not store directory data.

Replication Change Numbers

Change numbers uniquely identify changes that are made on an LDAP directory server. Change numbers also provide a consistent ordering of changes. The change number order is used to resolve conflicts and to determine the order in which forwarded changes should be replayed.

A *change number* consists of the following elements:

- **Time stamp, in milliseconds.** Time stamps are generated using the system clock. The change number is also generated such that each change number is always greater than all the change numbers that have already been processed by the server. Constantly increasing change numbers guarantees that operations that depend on previous operations are consistently replayed in the correct order. An example of an operation that depends on a previous operation is a modify operation that directly follows the add operation for that entry.
- **Sequence number.** A sequential number, incremented for each change that occurs within the same millisecond.
- **Replica identifier.** A unique integer identifier that is assigned to each replica in a topology. (A *replication topology* is the set of all replicas of a given data set. For example, the replication topology for `example.com` might be all copies of the `dc=example,dc=com` suffix across a directory service.)

  The replica identifier ensures that two different servers do not assign the same identifier to two different changes. In a future directory server release, an algorithm might be used to assign replica identifiers automatically.

Replication Server State

When a directory server connects to a replication server, the replication server must determine how up to date the directory server data is before the replication server can send changes that the directory server has not yet seen. This "up to date" state of the directory server is called the *replication server state*.

Server state is maintained as a vector. A server might have missed relatively old changes from another remote server, yet might already have seen and processed more recent changes from a server that is close by. Server state is therefore maintained by recording the last change number processed by each replica, according to the replica identifier.

Because administrators can stop and restart servers, the server state must be saved to stable storage. Ideally saving the server state would be done after each local or replicated change is made. Saving information to the database after each change would add significant overhead, however. Server state is therefore kept in memory and saved to the database on a regular basis, and when the server is properly shut down.

A drawback of this approach is that brutal interruptions such as kills and crashes can cause the server to lose track of changes that have already been processed. This can result in the need to fix inconsistencies when the server restarts. For an explanation of how crash recovery is managed, see "Directory Server Crashes" on page 84.

Operation Dependencies

Sometimes an operation cannot be replayed until another operation is complete. For example, when an add operation is followed by a modify operation on the same entry, the server must wait for the add operation to be completed before starting the modify operation.

Such dependencies are quite rare and are generally necessary for a few operations only. Most operations do not have dependencies (since most are modify operations). In such cases, it is necessary to replay operations in parallel to obtain the best performance with multi-CPU servers.

The replication model is built on the assumption that operation dependencies are rare. The replication mechanism therefore always tries to replay operations in parallel, and only switches to processing operation dependencies if an operation fails to replay.

How Replication Works

The topics in this section describe the mechanics involved in the replication process and how specific functionality is achieved.

Directory Server Change Processing

When a modification occurs on a directory server, replication code on the directory server performs the following tasks:

- Assigns a change number
- Generates historical information
- Forwards the change to a replication server
- Updates the server state

Historical information is stored in the entry and must therefore be included in the operation before the server writes to the back end. The server uses the change number when generating

historical information. The change number is therefore generated before the historical information. Both the change number and the historical information are performed as part of the pre-operation phase.

The operation is sent to the replication server before an acknowledgment for the update is sent to the client application that requested the operation. This ensures that a synchronous, assured replication mode can be implemented. For more information, see "Assured Replication" on page 94. The acknowledgment is therefore sent as part of the post-operation phase.

Changes are sent in the order defined by their change numbers. The correct order enables replication servers to make sure that all the changes are forwarded to other directory servers.

Because a directory server is multi-threaded, post-operation plug-ins can be called in a different order to pre-operation plug-ins, for the same operation. The replication code maintains a list of *pending changes*. This list includes changes that have started, and for which change numbers have already been generated, but that have not yet been sent to the replication server. Changes are added to the list of pending changes in the pre-operation phase. Changes are removed from the list when they are sent to the replication server. If a specific operation reaches the post-operation phase ahead of its change number-defined position, that operation waits until previous operations are sent before being sent to the replication server.

The server state is updated when the operation is sent to the replication server. For more information, see "Replication Server State" on page 80.

Change Replay

The replay of changes on replicated directory servers is efficient on multi-core and multi-CPU systems. On a directory server, multiple threads read the changes sent by the replication server.

Dependency information is used to decide whether an operation can be replayed immediately. The server checks the server state and the list of operations on which the current operation depends to determine whether those operations have been replayed. If the operations have not been replayed, the server puts the operation in a queue that holds dependency operations. If the operation can be replayed, the server builds an internal operation from information sent by replication servers. The server then runs the internal replay operation.

Internal replay operations built from the operations that are sent by a replication server can conflict with prior operations. Such internal operations cannot therefore always be replayed as if they were taking place on the original directory server. The server checks for conflicts when processing the `handleConflictResolution` phase.

In the majority of cases, the internal replay operations do not conflict with prior operations. In such cases, the `handleConflictResolution` phase does nothing. The replication code is therefore optimized to return quickly.

When a conflict does occur, the `handleConflictResolution` code takes the appropriate action to resolve the conflict. For modify conflicts, the `handleConflictResolution` code changes the modifications to retain the most recent changes.

When conflict resolution is handled, historical information is updated as for local operations. The operation can then be processed by the core server. Finally, at the end of the operation, the server state is updated.

After completing an operation, the server thread processing the operation checks whether an operation in the dependency queue was waiting for the current operation to complete. If so, that operation is eligible to be replayed, so the thread starts the replay process for the eligible operation. If not, the thread listens for further operations from the replication server.

Auto Repair

Despite efforts to keep servers in sync, directory servers can begin to show incoherent data. Typically, this occurs in the following circumstances:

- A disk error taints the stored data
- A memory error leads to an error in processing data
- A software bug leads to bad data or missing changes

In such cases, tracking and replaying changes is not sufficient to synchronize the incoherent data.

An automatic repair mechanism is provided, which leverages historical information inside entries to determine what the coherent data should be. The replication mechanism then repairs the data on directory servers where the data is bad or missing. The *auto repair* mechanism is implemented as an LDAP application, and runs on the hosts that run replication servers.

The auto repair application can run in different modes. Depending on the mode in which it is run, the auto repair application performs the following tasks:

- Repairs inconsistencies manifested as an error when the server was replaying modifications
- Repairs inconsistencies detected by the administrator
- Periodically scans directory entries to detect and repair inconsistencies

---

Note – In the current directory server release, the auto repair mechanism must be run manually. For more information, see "Detecting and Resolving Replication Inconsistencies" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Directory Server Crashes

When a directory server crashes, its connection to the replication server is lost. Recent changes that the directory server has processed and committed to its database might not yet have been transmitted to any replication server.

When a directory server restarts, therefore, it must compare its state with the server state of the replication servers to which the directory server connects. If the directory server detects that changes are missing and not yet sent to a replication server, the directory server constructs fake operations from historical information. The directory server sends these fake operations to its replication server.

Because teh local server state is not saved after each operation, the directory server cannot trust its saved server state after a crash. Instead, it recalculates its server update state, based on historical information.

Replication Server Crashes

When a replication server crashes, directory servers connect to another replication server in the topology. The directory servers then check for and, if necessary, resend missing changes.

Historical Information and Conflict Resolution

The topics in this section describe how historical information is retained and used to resolve replication conflicts.

What is a Replication Conflict?

A *conflict* occurs when one or more entries are updated simultaneously on multiple servers and the changes are incompatible, or causes some interaction between the updates. Conflict occurs because no update operation is carried out simultaneously on every replica in the replication topology. Instead, updates are first processed on one server, then replicated to other servers.

The following example describes a conflict that occurs when an attribute is modified at the same time on two different directory servers.

Consider a topology with two read-write replicas. A modify operation changes the surname, `sn`, attribute of an entry to `Smith` on one server. Before the server that is processing the change can synchronize with the other server, the `sn` attribute value for that entry is replaced with the value `Jones` on the other server. Unless the conflict is managed, replication would replay the change (`Smith`) on the server that now contains the value `Jones`. At the same time, replication would replay the change (`Jones`) on the server that contains the value `Smith`. The servers would therefore end up with inconsistent values for the `sn` attribute on the modified entry.

The following list describes additional conflicts that can occur.

- An entry is deleted on one server while one of its attribute values is modified on another server.
- An entry is renamed on one server while one of its attribute values is remodified on another server.
- An entry is deleted and another entry with the same Distinguished Name (DN) is added on one server while one of its attribute values is modified on another server.
- A parent entry is deleted and a child of that entry is created on another server, either through an add operation or a rename operation.
- Two different entries with the same DN are added at the same time on two different servers.
- Two different values are used to replace a single-valued attribute on the same entry on different servers at the same time.

Conflicts that involve only modifications of the same entry are called *modify conflicts*. Conflicts that involve at least one operation other than modify are called *naming conflicts*.

All modify conflicts and the vast majority of naming conflicts can be solved automatically by replaying the operations in their order of occurrence. However, the following naming conflicts, which have very little chance of occurring, cannot be solved automatically.

- Two entries with the same DN are created at the same time on different servers, either by adding new entries or by renaming existing entries.
- A parent entry is deleted and a child of the parent entry is created at the same time. The child entry can be created either when a new entry is added or when an existing entry is renamed.

Resolving Modify Conflicts

Modify conflicts only occur with modification operations.

Operations are globally and logically ordered to determine the outcome of a given set of operations. Change numbers are used to define the order.

The replication conflict resolution functionality ensures that all servers eventually reach the same state, as if all operations were replayed everywhere in the order defined by the change numbers. This remains true even though changes might be replayed in a different order on different servers. In the modify conflict example with the `sn` values of `Smith` and `Jones`, described previously, assume that the value was set to `Jones` on the second server *after* it was set to `Smith` on the first server. The resulting attribute value should be `Jones` on both servers, even after the replace modification of `Smith` is replayed on the second server.

Historical information about each entry is retained to check whether a conflicting operation has already been played using a change number newer than that of a current conflicting operation. For each modify operation, historical information is used, first to check if there is a conflict, and, if there is a conflict, to determine the correct result of the operation.

When a modify conflict occurs, the server determines whether the current attribute values must be retained or whether the modification must be applied. The current attribute values alone are not sufficient to make this assessment. The server also determines when (at which change number) prior modifications were made. Historical information therefore includes the following elements:

- The date when the attribute was last deleted
- The date when a given value was last added
- The date when a given value was last deleted

When an attribute is deleted or fully replaced, older information is no longer relevant. At that point the older historical information is removed.

Historical information undergoes the following processing:

- Saved in the `ds-sync-hist` attribute (This attribute can only be viewed by an administrator.)
- Updated (but not used) for normal operations
- Updated and used for replicated operations

Conflict resolution is carried out when operations are replayed, after the pre-operation during the `handleConflictResolution` phase.

Conflict resolution is carried out by changing the `List<Modification>` field of the `modifyOperation` to match the actual modifications required on the user attributes of the entry, and to change the `ds-sync-hist` attribute that is used to store historical information.

Resolving Naming Conflicts

Naming conflicts only happen for replayed operations. The server uses the following methods to resolve naming conflicts:

- Uses unique IDs to identify entries, including entries that have been renamed
- Tries to replay each operation first and only takes action if a conflict occurs
- Checks during the pre-operation phase for conflicts that cannot be detected when operations are replayed
- Retains no *tombstone entries*, which are entries that have been marked for deletion but not yet removed

Because directory entries can be renamed, the DN is not an immutable value of the entry. DNs cannot therefore be used to identify the entry for replication purposes. A unique and immutable identifier is therefore generated when an entry is created, and added as an operational attribute of the entry. This unique ID is used, instead of the DN, to identify the entry in changes that are sent between directory servers and replication servers.

A replication context is attached to the operation. The replication context stores private replication information such as change number, entry ID, and parent entry ID that is required to solve the conflict.

Purging Historical Information

Historical information is stored in the server database. Historical information therefore consumes space, I/O bandwidth, and cache efficiency. Historical information can be removed as soon as more recent changes have been seen from all the other servers in the topology.

Historical information is purged in the following ways:

- When a new change is performed on the entry.
- By a purge process that can be triggered at regular intervals. The purge process saves space, at the cost of more CPU for processing the purge. The purge process is therefore configurable. For more information, see "Changing the Replication Purge Delay" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Schema Replication

This section describe how schema replication is implemented. and is aimed at users who require an in-depth understanding of the schema replication architecture.

Schema describe the entries that can be stored in a directory server. Schema management is a core feature of the directory service. Replication is also a central feature of the directory service and is essential to a scalable, highly available service.

Any changes made to the schema of an individual directory server must therefore be replicated on all the directory servers that contribute to the same service.

Schema replication occurs when the schema is modified in any of the following ways:

- By modifying the `cn=schema` suffix when the server is online
- By using a dedicated task to perform dynamic schema updates by means of a file when the server is online
- By modifying the underlying back-end files directly when the server is offline

Generally, schema modifications occur only when deploying new applications or new types of data. The rate of change for schema is therefore low. For this reason, the schema replication implementation favors simplicity over scalability.

Schema replication is enabled by default. In certain specific cases, it might be necessary to have different schema on different directory servers, even when the servers share all or part of their data. In such cases you can disable schema replication, or specify a restricted list of servers that

participate in schema replication. For more information, see "Configuring Schema Replication" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Schema Replication Architecture

The schema replication architecture relies on the general replication architecture. You should therefore have an understanding of the general replication architecture before reading this section. For more information, see "Overview of the Directory Server Replication Architecture" on page 77.

Directory servers notify replication servers about any changes to their local schema. As in the case of data replication, the replication servers propagate schema changes to other replication servers, which in turn replay the changes on the other directory servers in the topology.

Schema replication shares the same replication configuration used for any subtree:

```
dn: cn=example,cn=domains,cn=Multimaster Synchronization,\
  cn=Synchronization Providers,cn=config
objectClass: top
objectClass: ds-cfg-replication-domain
cn: example
ds-cfg-base-dn: cn=schema
ds-cfg-replication-server: <server1>:<port1>
ds-cfg-replication-server: <server2>:<port2>
ds-cfg-server-id: <unique-server-id>
```

Schema replication differs from data replication in the following ways:

- **Entry Unique ID.** A unique ID is required for data replication because entries can be renamed or deleted.

  In the case of the schema, there is only one entry and that entry cannot be deleted or renamed. The unique ID used for the schema entry is therefore the DN of the schema entry.
- **Historical information.** Historical information is used to save a history of relevant modifications to an entry. This information makes it possible to solve modification conflicts.

  For schema replication, the only possible operations are adding values and deleting values. Historical information is therefore not maintained for modifications to the schema.
- **Persistent server state.** When a directory server starts up, the replication plug-in establishes a connection with a replication server. The replication server looks for changes in its change log and sends any changes that have not yet been applied to the directory server.

  In order to know where to start in the change log, the replication plug-in stores information that is persistent across server stop and start operations. This persistent information is stored in the replication `base-dn` entry.

The schema back end allows the specific operational attribute used to store the persistent state, `ds-sync-state`, to be modified.

Replication Status

Each replicated domain in a replicated topology has a certain *replication status*, depending on its connections within the topology, and on how up to date it is with regard to the changes that have occurred throughout the topology.

Knowledge of a domain's replication status enables a replicated topology to do the following:

- Manage certain aspects of assured replication
- Enable certain administrative tasks
- Administer and monitor replication effectively

For more information, see "Monitoring a Replicated Topology" in *Sun OpenDS Standard Edition 2.0 Administration Guide*. The following section outlines the different statuses that a replicated domain can have.

Replication Status Definitions

The following list provides a description of each possible replication status that can be held by a replicated domain.

`NOT_CONNECTED_STATUS` — The local replicated domain is not connected to any replication server. Replication cannot occur until a connection to a replication server is established. This is the only possible status if there is no connection to a replication server.

`NORMAL_STATUS` — The local replicated domain is almost in sync with its peers (that is, with the updates received on the replication server). The client LDAP requests have been processed normally.

`DEGRADED_STATUS` — The local replicated domain is too late regarding updates that have been queued by the replication server. What constitutes *too late* is defined by the *degraded status threshold*, that is, the number of changes that the replication server has in its queue for the directory server. With this status, the local directory server might be slow in replaying changes. This can have an impact on assured replication.

`FULL_UPDATE_STATUS` — An online full update is currently being performed on the local replicated domain (in other words, the domain is receiving entries from a remote directory server). The full update must be completed before the status can be changed and before the replicated domain can participate in replication again.

BAD_GEN_ID_STATUS	The local replicated domain does not have the same generation ID as the replication server to which it is connected. Replication cannot run until the local domain is initialized with a data set that has the same generation ID as its replication server. To initialize the local domain, perform an online full update, an LDIF import, or a binary copy of the database, retaining the domain entries.

Degraded Status

A directory server that is slow in replaying changes is assigned a DEGRADED_STATUS. The stage at which the server is regarded as "too slow" is defined by the *degraded status threshold* and is configurable, based on the number of updates queued in the replication server for that directory server.

When the degraded status threshold is reached, the directory server assumes a degraded status and is considered to be unable to send acknowledgments in time. A server with this status can have an impact on assured replication, as replication servers no longer wait for an acknowledgment from this server before returning their own acknowledgments.

Full Update Status and Bad Generation ID Status

Apart from being assigned a degraded status, a directory server can change status if an administrator performs one of the following tasks on the topology:

- **Full update.** When a replicated domain is initialized online from another server in the topology, the directory server status for that domain changes to FULL_UPDATE_STATUS. When the full update has completed, the directory server reinitializes its connection to the topology, and the status is reset to NORMAL_STATUS.
- **Local import or restore.** When a replicated domain is reinitialized by using a local import or restore procedure, the directory server status for that domain changes to NOT_CONNECTED_STATUS.
- **Resetting the generation ID.** If a replicated domain connects to a replication server with a generation ID that is different from its own, the domain is assigned a BAD_GEN_ID status. A domain can also be assigned this status if a reconnection occurs after a full online update, a local import, or a restore with a set of data that has a different generation ID to that of the replication server.

  In addition, you might need to reset the generation ID of all the replication servers in the topology by running the reset generation ID task on the directory server. This causes all the replication servers in the topology to have a different ID to the ID of the directory servers to which they are connected. In this case, the directory servers are assigned a BAD_GEN_ID status.

Replication Groups

Replication groups are designed to support multi-data center deployments and disaster recovery scenarios. Replication groups are defined by a group ID. A group ID is a unique number that is assigned to a replicated domain on a directory server (one group ID per replicated domain). A *group ID* is also assigned to a replication server (one group ID for the whole replication server).

Group IDs determine how a directory server domain connects to an available replication server. From the list of configured replication servers, a directory server first tries to connect to a replication server that has the same group ID as that of the directory server. If no replication server with a compatible group ID is available, the directory server connects to a replication server with a different group ID. This selection process is described in greater detail in the following section.

For information about how to configure replication groups, see "Configuring Replication Groups" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Note – Assured replication does not cross group boundaries. For more information, see "Assured Replication" on page 94.

Replication Server Selection

When a directory server starts, or when a replication server that is connected to the directory server stops, the directory server must choose an adequate replication server for publishing and receiving changes. Two parameters affect which replication server is selected.

- The geographic identifier of the replication server, or group ID
- The server state of the replication server

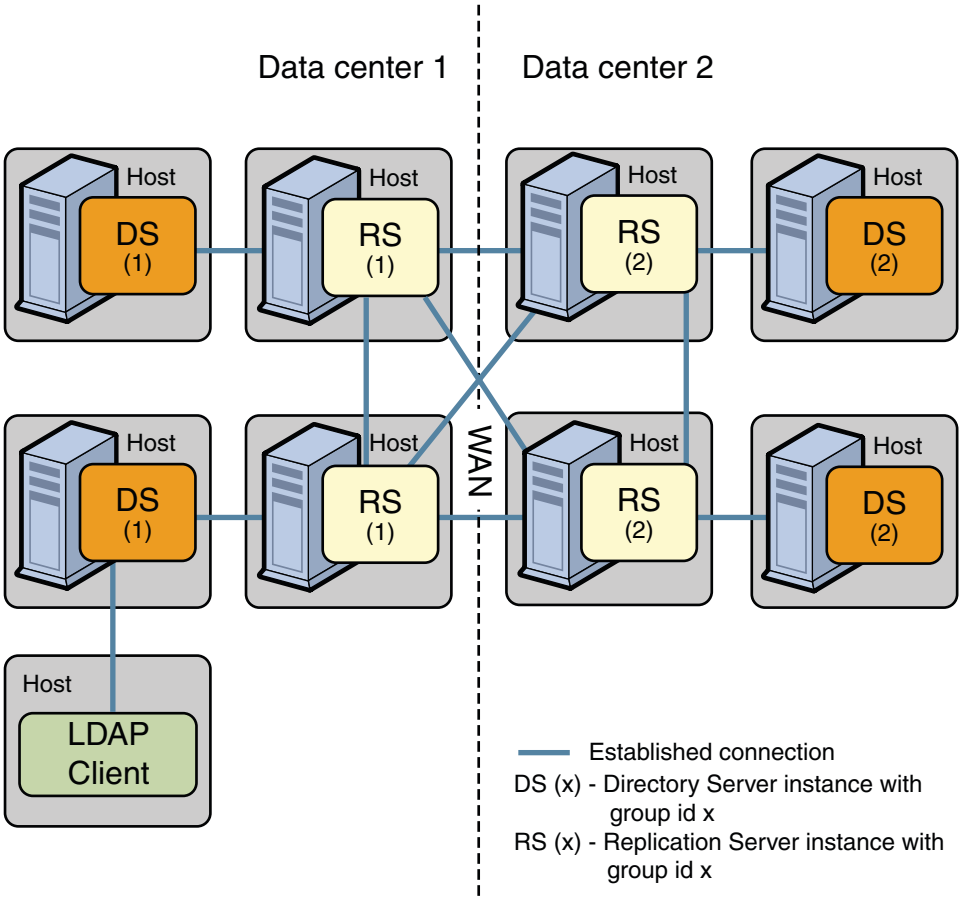
A directory server selects a replication server according to the following criteria, in this order:

1. A replication server with the same group ID, that has received all the changes from that directory server, and that has the greatest number of available updates not yet replayed on that directory server.
2. A replication server with the same group ID and that has the greatest number of available updates not yet replayed on that directory server.
3. A replication server that has received all the changes from that directory server, and that has the greatest number of available updates not yet replayed on that directory server.
4. A replication server that has the greatest number of available updates not yet replayed on that directory server.

Replication Groups in a Multi-Data Center Deployment

This sample deployment shows the use of replication groups across multiple data centers. The deployment assumes two data centers, connected by a wide area network (WAN), with the following configuration:

- Each replication server and directory server within a single data center has the same group ID.
- There is a unique group ID for the entire data center (one group ID per data center).



Disaster Recovery deployment: 2 data centers with different group ids

FIGURE 1 Replication Groups Over WAN

In this deployment, each directory server will attempt to connect to a replication server in its own data center, avoiding the latency associated with connection over a WAN. If all the replication servers in a data center fail, the directory server will connect to a remote replication server. This ensures that the replication service is maintained, albeit in a degraded manner (if the connection between data centers is slow). When one or more local replication servers is back online, the directory servers will automatically reconnect to a local replication server.

Assured Replication

Before you read the following sections , you should have an understanding of basic replication concepts. You must know what a replication server is, as opposed to a directory server, and have an understanding of how replication servers work in a replicated topology. If this is not the case, read at least the "Overview of the Directory Server Replication Architecture" on page 77 to obtain an understanding of how regular replication works in the directory server.

In a standard replicated topology, changes are replayed to other replicated servers in a "best effort" mode. A change made on an LDAP server is replayed on the other servers in the topology as soon as possible, but in an unsynchronized manner. This is convenient for performance but does not ensure that a change has been propagated to other servers when the initial LDAP client call is finished.

In some deployments this might be acceptable, that is, the time period between the change on the first server and the replay on peer servers might be short enough to fulfill the requirements of the deployment. For example, an international organization might store employee user accounts in a replicated topology across various geographical locations. If a new employee is hired and a new account is created for him on one LDAP server in a specific location, it might be acceptable that the replay of the creation occurs in other LDAP servers a few milliseconds after the LDAP client call terminates. The user is unlikely to perform a host login that would access one of the other LDAP servers in the same second that the user account is created.

However, there might be cases in which more synchronization is required from the replication process. If a specific host fails, it might be imperative that any changes made on that host have been propagated elsewhere in the topology. In addition, the deployment might require assurance that once the LDAP client call of a change is returned by a server, all of the peer servers in the topology have received that change. Any other clients that read the entry from anywhere in the topology would be sure to obtain the modification.

Assured replication is a method of making regular replication work in a more synchronized manner. The topics in this section describe how assured replication works, from an architectural perspective. For information about configuring assured replication, see "Configuring Assured Replication" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Assured Replication Modes

The directory server currently supports two assured replication modes, depending on the level of synchronization that is required, the goal of the replicated topology, and the acceptable performance impact.

- Safe Data Mode
- Safe Read Mode

Safe Data Mode

In safe data mode, any change is propagated to a specified number of servers in the topology before the LDAP client call returns. If the LDAP server on which the change was made fails, it is guaranteed that the change has already been propagated to at least the specified number of servers.

This specified number of servers (N) defines the *safe data level*. The safe data level is based on acknowledgments from the replication servers only. In other words, an update message that is sent from an LDAP server must be acknowledged by at least N (N>=1) replication servers before the LDAP client call that initiated the update returns.

The higher the safe data level, the greater the number of machines that are assured to have the update and, consequently, the more reliable the data. However, as the safe data level increases, the overall performance decreases because additional acknowledgments are required before the LDAP client call returns.

The safe data level functions in best effort mode. That is, if the safe data level is set to 3 and there are temporarily only two replication servers available in the topology, an acknowledgment from the third (unavailable) replication server will not be expected until this server is available again.

Safe data mode is affected by the use of *replication groups*. Because assured replication does not cross group boundaries, a replication server with a group ID of 1 waits for an acknowledgment from other replication servers with the same group ID but not for acknowledgments from replication servers with a different group ID. For more information, see "Replication Groups" on page 91.

Note – In the current replication implementation, the `setup` and `dsreplication` commands support only a scenario in which the main replication server is physically located in the same VM as the LDAP server (that is, on the same machine). However, the fundamental replication design is to support deployments where the replication servers run on separate machines, to increase reliability.

Such deployments can currently be configured only by using the `dsconfig` command and are not supported by the `setup` and `dsreplication` commands. However, these deployments provide better failover and availability, and are expected to be supported in the future. In such deployments, if the safe data level is set to 1 (acknowledgment of only one replication server is expected), this replication server *must* run on a separate machine to the LDAP server.

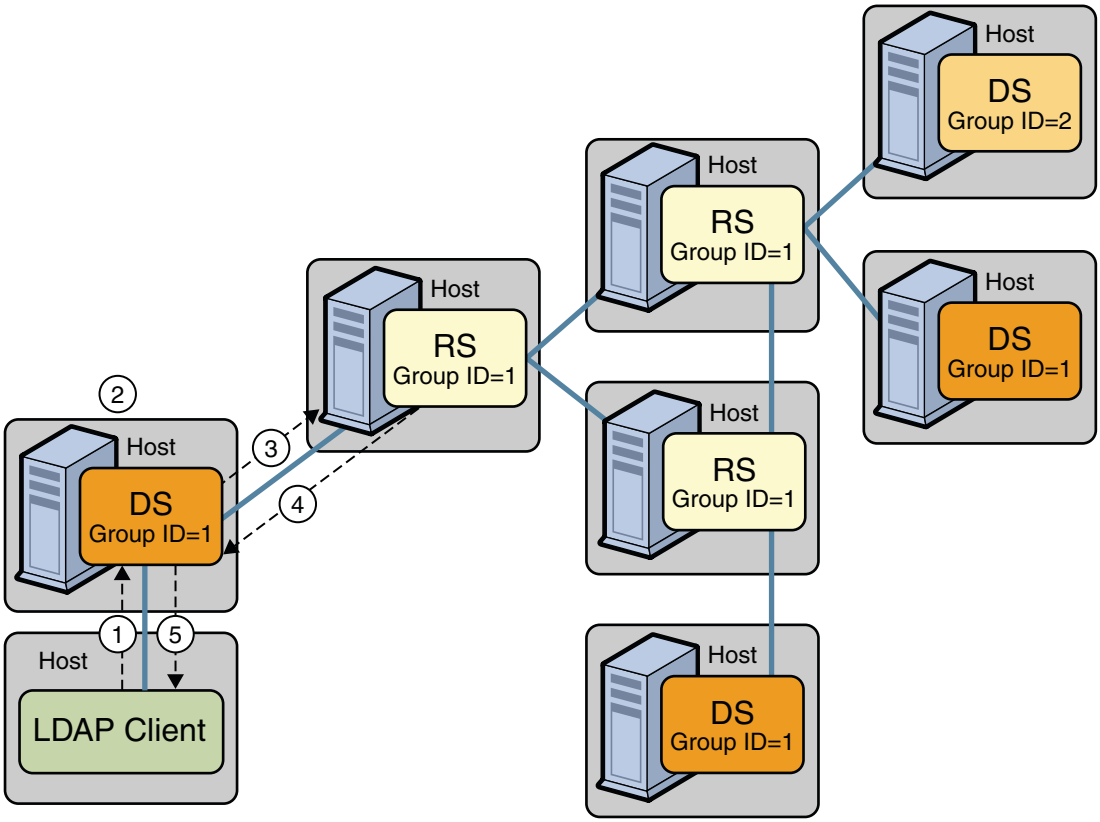
EXAMPLE 1 Safe Data Level = 1

Setting the safe data level to 1 ensures that the first replication server returns an acknowledgment to the directory server immediately after receiving the update. The replication

EXAMPLE 1 Safe Data Level = 1 *(Continued)*

server does not wait for acknowledgments from other replication servers in the topology. The modification is guaranteed to exist on one additional server (other than the directory server on which the change was made).

This example can only be configured with `dsconfig` and is not yet supported by the `setup` or `dsreplication` commands.



Connections configured between servers

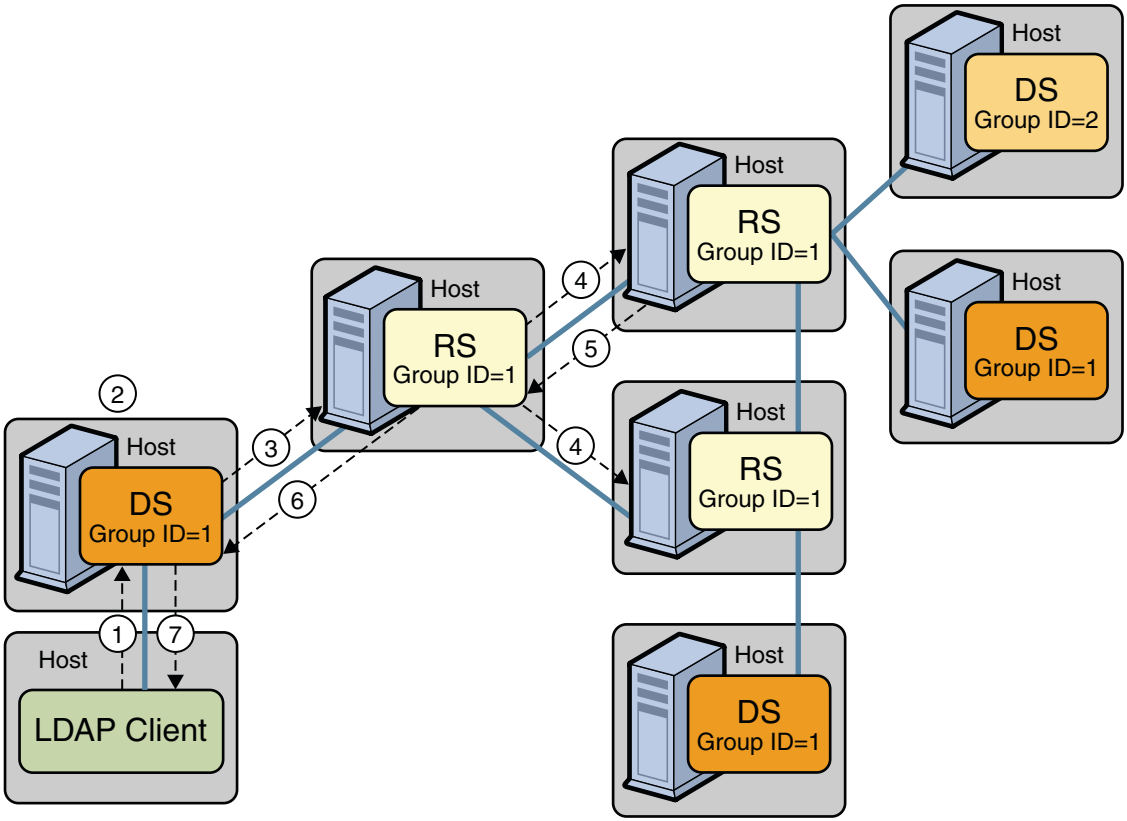
DS - Directory Server Instance
RS - Replication Server Instance
1. LDAP change operation initiated
2. LDAP operation executed locally
3. Matching update info sent, wait for ack
4. Ack from first RS that receives update (does not wait until the update is forwarded to other Replication Servers).
5. LDAP operation call returns

EXAMPLE 2 Safe Data Level = 2 (RS and DS on Different Hosts)

Setting the safe data level to 2 ensures that the first replication server will wait for an acknowledgment from one peer replication server before returning an acknowledgment to the directory server. The modification is guaranteed to exist on two additional servers (other than the directory server on which the change was made).

EXAMPLE 2 Safe Data Level = 2 (RS and DS on Different Hosts) *(Continued)*

This example can only be configured with `dsconfig` and is not yet supported by the `setup` or `dsreplication` commands.



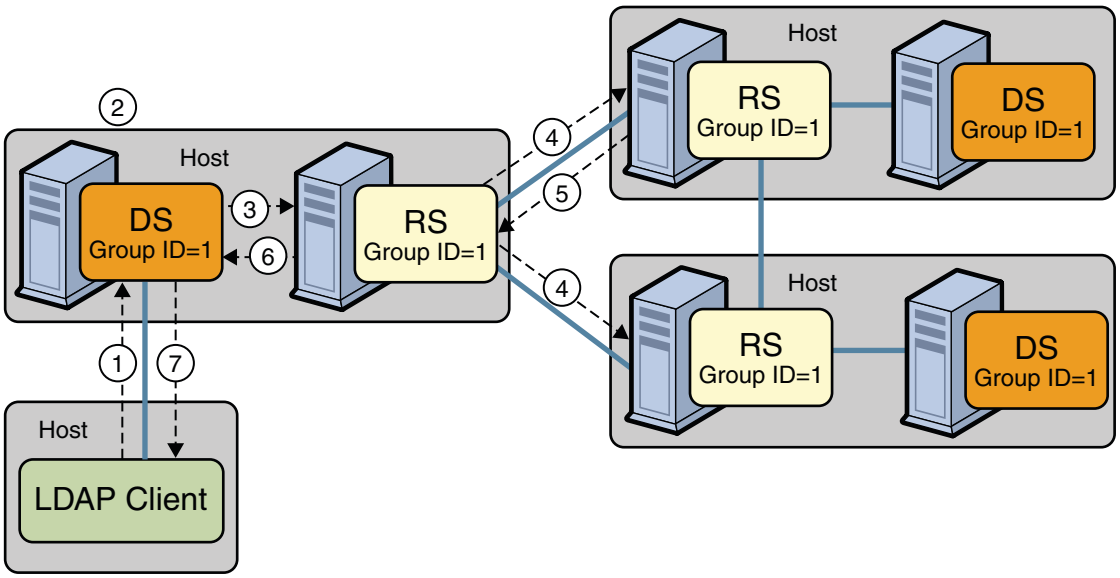
Connections configured between servers

DS - Directory Server Instance
RS - Replication Server Instance

1. LDAP change operation initiated
2. LDAP operation executed locally
3. Matching update info sent, wait for ack
4. Update info sent to peer RS with group ID=1, wait for ack
5. This RS is the first to send an ack of the update
6. The RS has received one ack. Including itself, there are two RSs in the topology that have acknowledged the update. The RS can therefore send an acknowledgment to the initial DS.
7. LDAP operation call returns

EXAMPLE 3 Safe Data Level = 2 (RS and DS on Same Host)

In the current replication implementation, the `setup` and `dsreplication` commands only support configurations in which the replication is on the same machine as the directory server. With this implementation, if you want to ensure that a change is sent to at least one additional host, you must set the safe data level to 2.



Connections configured between servers

DS - Directory Server Instance
RS - Replication Server Instance

1. LDAP change operation initiated
2. LDAP operation executed locally
3. Matching update info sent, wait for ack
4. Update info sent to peer RS with group ID=1, wait for ack
5. This RS is the first to send an ack of the update
6. The RS has received one ack. Including itself, there are two RSs in the topology that have acknowledged the update. The RS can therefore send an acknowledgment to the initial DS.
7. LDAP operation call returns

Safe Read Mode

Safe read mode ensures that any modification made on a specific directory server has been replayed to all other directory servers within the topology before the LDAP call returns. In this mode, if another LDAP client performs a read operation on another directory server in the topology, that client is assured of reading the modification that has just been performed. Safe read mode is the most synchronized manner in which replication can be configured. However, this mode also has the biggest performance impact in terms of write time.

Safe read mode is based on acknowledgments from the LDAP servers rather than the replication servers in a topology. When a modification is made on a directory server, the update is sent to the corresponding replication server. The replication server then forwards the update to the other replication servers in the topology. These replication servers wait for acknowledgment of the modification being replayed on all the directory servers to which the modification is forwarded. When the modification has been replayed on all directory servers in the topology, the replication servers send their acknowledgment back to the first replication server, which in turn sends an acknowledgment to the original directory server.

The first replication server also waits for an acknowledgment from any other directory servers that are directly connected to it before sending the acknowledgment to the original directory server. Only when the original directory server has received an acknowledgment from its replication server does it finally return the end of the operation call to the LDAP client.

At this point, all directory servers in the topology contain the modification. If an LDAP client reads the data from any directory server, it is therefore certain of obtaining the modification.

Safe Read Mode and Replication Groups

Replication groups support multi-data center replication and high availability. For more information about replication groups, see "Replication Groups" on page 91. In the context of assured replication, replication groups enable a set of directory servers to work together in safe read mode. All directory servers that work together in a synchronized manner require the same group ID. This group ID should also be assigned to all the replication servers working in the synchronized topology. Assured replication does not cross group boundaries.

When a change occurs on a directory server with certain group ID (N), the LDAP call is not returned before every other directory server with group ID N has returned an acknowledgment of the change.

The use of replication groups provides more flexibility in a replicated topology that uses safe read mode.

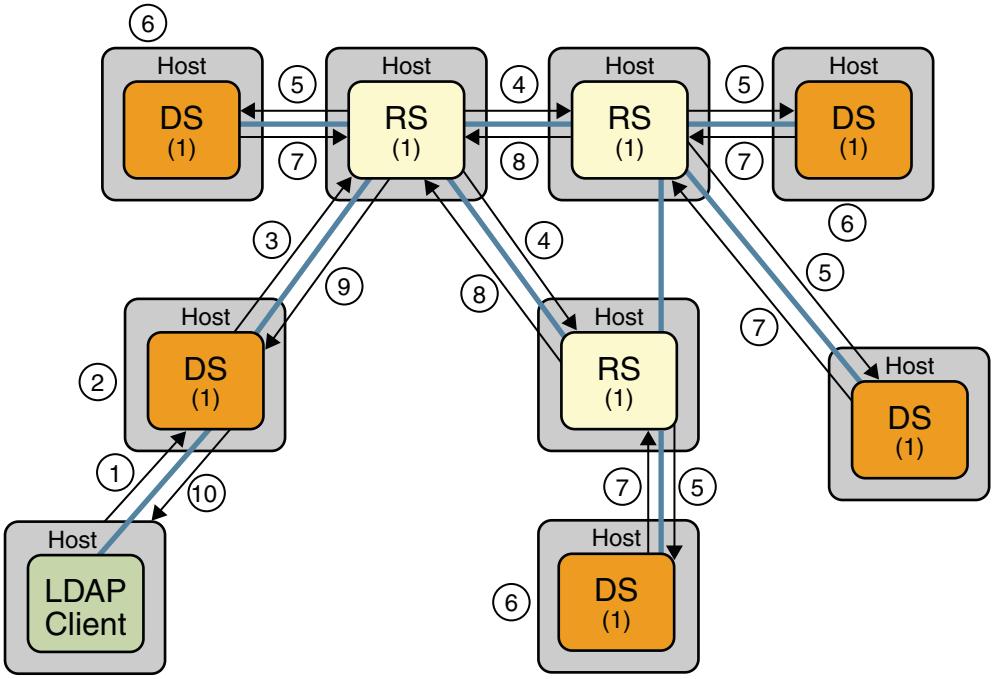
- In a single data center deployment, you can define a subset of the directory servers that should be fully synchronized. Only the directory servers with the same group ID will wait for an acknowledgment from their peers with the same group ID. All the replication servers will have the same group ID.

- In a multi-data center deployment, you can specify that all the directory servers within a single data center are fully synchronized. A directory servers will wait for acknowledgment only from its peers located in the same data center before returning an LDAP call. Acknowledgment is expected only if the directory server is connected to a replication server with the same group ID.

EXAMPLE 4 Safe Read Mode in a Single Data Center With One Group

The following illustration shows a deployment in which all nodes are in the same data center and are part of the same replication group. Each directory server and replication server has the same group ID. Any modification must be replayed on every directory server in the topology before an LDAP client call returns. Any subsequent LDAP read operation on any directory server in the topology is assured of reading the modification.

Such a scenario might be convenient, for example, if there is an LDAP load balancer in front of the replicated directory server pool. Because it is impossible to determine the directory server to which the load balancer will redirect an LDAP modification, a subsequent read operation is not necessarily routed to the directory server on which the modification was made. In this case, it is imperative that the change is made on all servers in the topology before the LDAP client call is returned.



Established connection
DS (x) - OpenDS instance with group id x
RS (x) - Replication Server instance with group id x

1. LDAP change operation initiated
2. LDAP operation executed locally
3. Matching update info sent, wait for ack
4. Update info sent to peer RS with group ID=1, wait for ack
5. Update info sent to DSs with group id=1, wait for ack
6. LDAP operation locally replayed
7. DS sends an ack to RS
8. RS sends an ack to initial RS
9. RS sends an ack to initial DS
10. LDAP operation call returns

Assured Mode Replication: Safe Read Mode, 1 data center deployment (every directory servers with Safe Read Mode)

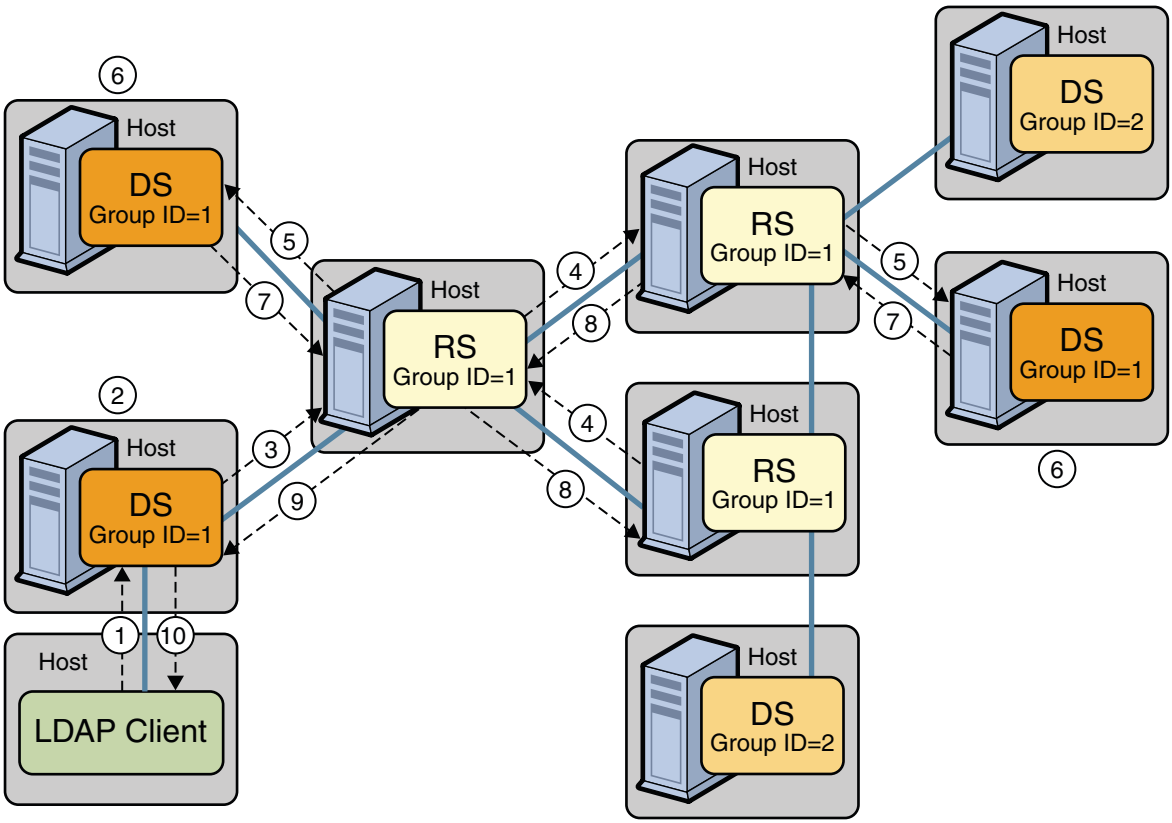
EXAMPLE 5 Safe Read Mode in a Single Data Center With More Than One Group

The following illustration shows a deployment in which all nodes are in the same data center but in which assured replication is configured on only a subset of the directory servers. This subset of servers constitutes a replication group, and each server is assigned the same group ID

EXAMPLE 5 Safe Read Mode in a Single Data Center With More Than One Group *(Continued)*

(1). When a change is made on one of the directory servers in the replication group, an acknowledgment must be received from all the directory servers in the group before the initial LDAP call is returned to the client. The remaining directory servers in the topology will still replay the change, but their acknowledgment is not required before the LDAP call is returned. If a change made on one of the servers outside of the group, no acknowledgment from other directory servers is required before the LDAP call is returned to the client.

In this example, the replication server that is connected to directory servers outside of the replication group is still assigned a group ID of 1. This configuration ensures failover in the case of another replication server being offline. In this case, if a directory server within the replication group connects to this particular replication server, assured replication must still work. For the purpose of failover, any replication server must be assigned the same group ID if there is a chance that a directory server within the group might connect to it at some stage.



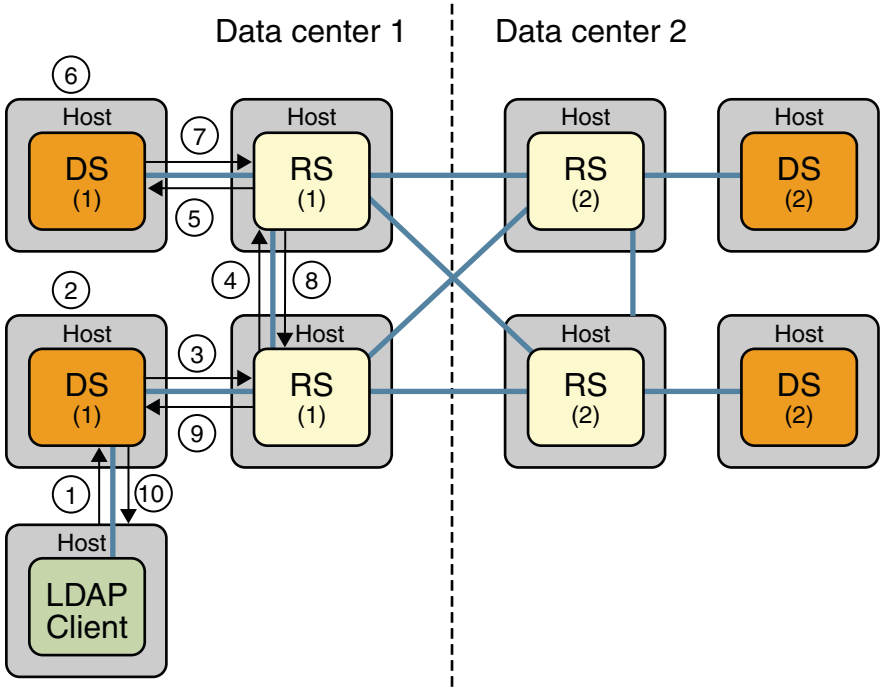
EXAMPLE 6 Safe Read Mode in a Multi-Data Center Deployment

The following illustration shows a deployment with two data centers (in different geographical locations). Each data center has safe read mode configured locally within the data center. All of the directory servers and the replication servers within the same data center are assigned the same group ID (1 for the first data center and 2 for the second data center). The directory

EXAMPLE 6 Safe Read Mode in a Multi-Data Center Deployment *(Continued)*

servers within the same data center operate in a more tightly consistent synchronized manner. Any change made on a directory server must be replayed and acknowledged from all directory servers within that data center before the LDAP client call returns.

In this example, data is synchronized between the two data centers, but any change made on a specific directory server is immediately visible on all other directory servers within the same data center. This scenario is convenient if there is an LDAP load balancer in front of the directory servers of a data center. The performance impact in terms of writes is not too great because no acknowledgments are requested from the servers of the remote data center.



Established connection

DS (x) - Directory Server instance with group id x
RS (x) - Replication Server instance with group id x

1. LDAP change operation initiated
2. LDAP operation executed locally
3. Matching update info sent, wait for ack
4. Update info sent to peer RS with group ID=1, wait for ack
5. Update info sent to DSs with group id=1, wait for ack
6. LDAP operation locally replayed
7. DS sends an ack to RS
8) RS sends an ack to initial RS
9) RS sends an ack to initial DS
10) LDAP operation call returns

Assured Mode Replication: Safe Read Mode, 2 data centers deployment

EXAMPLE 6 Safe Read Mode in a Multi-Data Center Deployment *(Continued)*

The group ID of the replication server is important in this scenario. If a change arrives from a directory server with group ID N, the replication server compares N with its own group ID and takes the following action:

EXAMPLE 6 Safe Read Mode in a Multi-Data Center Deployment *(Continued)*

- If the replication server has the same group ID (N), it forwards the change to all the replication servers and directory servers to which it is directly connected. However, it waits for an acknowledgment only from the servers with the same group ID (N) before sending its own acknowledgment back to the original directory server.
- If the replication server has a different group ID, it forwards the change to all the replication servers and directory servers but does not wait for any acknowledgment.

Assured Replication Connection Algorithm

In implementing the scenarios described in the previous sections, a directory server in a topology uses the following algorithm to select the replication server to which that directory server should connect:

1. Connect to each replication server in the list of configured replication servers and obtain its server state and group ID.
2. From the list of replication servers that are up to date with the changes on the directory server, and that have same group ID as the directory server, select the one that has the most updates from other directory servers in the topology. If no replication server exists with the same group ID as the directory server, select the replication server that is most up to date.

This algorithm ensures that a higher priority is given to replication servers with the same group ID as the directory server's group ID. A directory server will therefore favor a replication server located in its own data center.

Connecting to a replication server with the same group ID (in the same data center) provides the safe read mode functionality. Connecting to a replication server with a different group ID provides failover to another data center (if all the replication servers in the local data center fail). In this case, safe read mode is disabled as no acknowledgment is requested when sending update messages to replication servers with a different group ID. Replication continues, but in degraded mode (that is, the safe read mode requested at configuration time is not applied.)

To return replication to normal, a directory server periodically polls the configuration list for the arrival of replication servers with the same group ID as its own. If the directory server detects that a replication server with its own group ID is available, it disconnects from the current replication server (with a different group ID), and reconnects to the recovered replication server with the same group ID. Safe read mode is thus re-enabled and replication returns to the mode in which it was configured.

Assured Replication and Replication Status

When a replication server detects that a directory server is out of sync regarding the overall updates made in the topology, that directory server is said to have a *degraded status*. A directory server that is out of sync is unlikely to be able to send the expected acknowledgments in time for the replication server to avoid a timeout situation. The server therefore has a degraded status until it has an acceptable level of updates. With a degraded status, a directory server is no longer expected to send acknowledgments to the replication server, until it returns to having a *normal status*.

Because a directory server with a degraded status is unable to send acknowledgments, the synchronization of an LDAP operation in safe read mode cannot be assured. In other words, data read from this directory server might not contain the modifications made on another directory server in the topology.

For more information, see "Replication Status Definitions" on page 89.

Assured Replication Monitoring

The assured replication mechanism includes several attributes defined to monitor how well the mechanism is working. The following tables list the monitoring attributes defined on the directory servers and on the replication servers in a topology.

On a directory server, the attributes are located under the monitor entry for that replicated DN. For example, monitoring information related to the replicated domain `dc=example,dc=com` is located under the monitoring entry `cn=Replication Domain,dc=example,dc=com,`*`server-id`*`,cn=monitor`.

On a replication server, the monitoring information related to assured replication is on a per connection basis. Monitoring attributes are found in the monitoring entry of a directory server or replication server that is connected to the current replication server. For example, on a particular replication server, the monitoring information related to a connected directory server would be under the monitoring entry `cn=Directory Server dc=example,dc=com` *`ds-host`*`,`*`server-id`*`,cn=monitor`. The monitoring information related to a connected replication server would be under the monitoring entry `cn=Remote Replication Server dc=example,dc=com` *`repl-server-host`*`:`*`repl-port`*`,`*`server-id`*`,cn=monitor`.

TABLE 2 Monitoring Attributes on the Directory Server

Attribute Name	Attribute Type	Purpose
`assured-sr-sent-updates`	Integer (0..N)	Number of updates sent in assured replication, safe read mode

TABLE 2 Monitoring Attributes on the Directory Server *(Continued)*

Attribute Name	Attribute Type	Purpose
`assured-sr-acknowledged-updates`	Integer (0..N)	Number of updates sent in assured replication, safe read mode, that have been successfully acknowledged
`assured-sr-not-acknowledged-updates`	Integer (0..N)	Number of updates sent in assured replication, safe read mode, that have not been successfully acknowledged (either because of timeout, wrong status, or error at replay)
`assured-sr-timeout-updates`	Integer (0..N)	Number of updates sent in assured replication, safe read mode, that have not been successfully acknowledged because of timeout
`assured-sr-wrong-status-updates`	Integer (0..N)	Number of updates sent in assured replication, safe read mode, that have not been successfully acknowledged because of wrong status
`assured-sr-replay-error-updates`	Integer (0..N)	Number of updates sent in assured replication, safe read mode, that have not been successfully acknowledged because of replay error
`assured-sr-server-not-acknowledged-updates`	String	Multiple values allowed: number of updates sent in assured replication, safe read mode, that have not been successfully acknowledged (either because of timeout, wrong status or error at replay) for a particular server (directory server or replication server). String format: *server id*:*number of failed updates*
`assured-sr-received-updates`	Integer (0..N)	Number of updates received in assured replication, safe read mode
`assured-sr-received-updates-acked`	Integer (0..N)	Number of updates received in assured replication, safe read mode that have been acknowledged without errors
`assured-sr-received-updates-not-acked`	Integer (0..N)	Number of updates received in assured replication, safe read mode, that have been acknowledged with errors
`assured-sd-sent-updates`	Integer (0..N)	Number of updates sent in assured replication, safe data mode
`assured-sd-acknowledged-updates`	Integer (0..N)	Number of updates sent in assured replication, safe data mode, that have been successfully acknowledged

TABLE 2 Monitoring Attributes on the Directory Server *(Continued)*

Attribute Name	Attribute Type	Purpose
`assured-sd-timeout-updates`	Integer (0..N)	Number of updates sent in assured replication, safe data mode, that have not been successfully acknowledged because of timeout
`assured-sd-server-timeout-updates`	String	Multiple values allowed: number of updates sent in assured replication, safe data mode, that have not been successfully acknowledged (because of timeout) for a particular RS. String format: *server id*:*number of failed updates*

TABLE 3 Monitoring Attributes on the Replication Server

Attribute Name	Attribute Type	Purpose
`assured-sr-received-updates`	Integer (0..N)	Number of updates received from the remote server in assured replication, safe read mode
`assured-sr-received-updates-timeout`	Integer (0..N)	Number of updates received from the remote server in assure replication, safe read mode, that timed out when forwarding them
`assured-sr-sent-updates`	Integer (0..N)	Number of updates sent to the remote server in assured replication, safe read mode
`assured-sr-sent-updates-timeout`	Integer (0..N)	Number of updates sent to the remote server in assured replication, safe read mode, that timed out
`assured-sd-received-updates`	Integer (0..N)	Number of updates received from the remote server in Assured Replication, Safe Data
`assured-sd-received-updates-timeout`	Integer (0..N)	Number of updates received from the remote server in assured replication, safe date mode, that timed out when forwarding them. This attribute is meaningless if the remote server is a replication server.
`assured-sd-sent-updates`	Integer (0..N)	Number of updates sent to the remote server in assured replication, safe data mode. This attribute is meaningless if the remote server is a directory server.
`assured-sd-sent-updates-timeout`	Integer (0..N)	Number of updates sent to the remote server in assured replication, safe data mode, that timed out. This attribute is meaningless if the remote server is a directory server.

Root Users and the Privilege Subsystem

Most LDAP directory servers typically have a single superuser (for example, `cn=Directory Manager` in the Sun Java System directory server), which is much like the root account in traditional UNIX® systems. This account can bypass access controls and other restrictions that can be enforced for other users. In the directory server , however, two key changes are made to this model: it is possible to define multiple root users, and a privilege subsystem that makes it possible to control capabilities on a more fine-grained level.

This section discusses the following topics:

Root User Accounts

Root user accounts in the directory server are defined below the `cn=Root DNs,cn=config` branch in the server configuration. Each root account should be defined as a regular user entry, with the exception that it should include the `ds-cfg-root-dn-user` auxiliary object class. It can also have one or more values for the `ds-cfg-alternate-bind-dn` attribute. this attribute specifies alternate DNs that can be used to authenticate as that user (for example, so you can bind as `cn=Directory Manager` instead of having to use `cn=Directory Manager,cn=Root DNs,cn=config`, which is the actual entry DN).

Providing the ability to have multiple root users and breaking each of them out into their own entries provides a number of advantages:

- Each administrator that needs root access to the directory server can have their own account with their own credentials. This makes it easier to keep an audit trail of who does what in the directory server than if all of the administrators had to share a single root account.
- Since each root user account has its own set of credentials, the credentials for one root user can be changed without impacting any of the other root users. It is not necessary to coordinate root password changes among all of the administrators since each of them has their own account. If an administrator leaves, then that account can simply be deactivated or removed.

- Since each root user has its own entry, and you can put whatever attributes and object classes you want in that entry (as long as it also has the `ds-cfg-root-dn-user` auxiliary object class), root users are capable of using strong authentication like the EXTERNAL or GSSAPI SASL mechanisms.
- Root users are subject to password policy enforcement, which means that it is possible to do things like force root users to change their passwords on a regular basis, ensure that they are only allowed to authenticate or change their passwords using secure mechanisms, and ensure that they choose strong passwords. It is possible to use custom password policies for these users, so they are subject to different sets of password policy requirements than other users in the directory.
- It is also possible to define different resource limits for root users than for regular users. Since each root account has its own entry, operational attributes like `ds-rlim-size-limit`, `ds-rlim-time-limit`, and `ds-rlim-lookthrough-limit` work for root users just as they do with normal user accounts.

Privilege Subsystem

As mentioned above, root user accounts in traditional directories are special because they can to bypass access controls and other restrictions, and there are some kinds of operations that only root users can perform. This is much like the concept of root users in traditional UNIX operating systems, and it often creates a bit of a paradigm because there may be cases in which a user needs to do something that only root can do. If users are given root access, then they are given far more power than they actually need to do their job, and system administrators have to hope that they use it responsibly and not intentionally or unintentionally impact some other part of the system. Alternately, the user may not be given root access and either not be able to perform a vital function or have to get one of the system administrators to perform the task.

Solaris 10 has addressed this problem in UNIX systems by creating a privilege subsystem (also called "process rights management"). The engineers developing Solaris realized that it is dangerous and undesirable to be forced to give someone root access just to perform one specific task. For example, just because a user may need to start a process that listens on a port below 1024 does not mean that they should also be able to bypass filesystem permissions, change network interface settings, or mount and unmount file systems. With the privilege subsystem in Solaris 10, it is possible to give a user just the specific capability that they need, for example, the ability to bind to privileged ports, without giving them full root access. Similarly, it is possible to take away privileges that might otherwise be available. For example, an account that is only used to run a specific daemon does not need to be able to see processes owned by other users on the system.

The directory server has embraced this concept, and it too has a privilege subsystem that defines distinct capabilities that users might need and makes it possible to give them just the level of access that they require. Normal users may be granted privileges that they would not otherwise have, and it is even possible to take certain privileges away from root users. The set of privileges currently defined in the directory server includes:

`bypass-acl`	Allows the user to bypass access control evaluation
`modify-acl`	Allows the user to make changes to the access controls defined in the server
`config-read`	Allows the user to have read access to the server configuration
`config-write`	Allows the user to have write access to the server configuration
`jmx-read`	Allows the user to read JMX attribute values
`jmx-write`	Allows the user to update JMX attribute values
`jmx-notify`	* Allows the user to subscribe to JMX notifications
`ldif-import`	Allows the user to request the LDIF import task
`ldif-export`	Allows the user to request the LDIF export task
`backend-backup`	Allows the user to request the backend backup task
`backend-restore`	Allows the user to request the backend restore task
`server-shutdown`	Allows the user to request the server shutdown task
`server-restart`	Allows the user to request the server restart task
`proxied-auth`	Allows the user to use the proxied authorization control or request an alternate SASL authorization ID
`disconnect-client`	Allows the user to terminate arbitrary client connections
`cancel-request`	* Allows the user to cancel arbitrary client requests
`unindexed-search`	Allows the user to request unindexed search operations
`password-reset`	Allows the user to reset the passwords for other users
`data-sync`	* Allows the user to participate in the data synchronization environment
`update-schema`	Allows the user to update the server schema
`privilege-change`	Allows the user to change the set of privileges assigned to a user, or to change the set of default root privileges

At the present time, the privileges marked with an asterisk (*) are not yet implemented in the server and therefore have no effect.

Note that the privilege subsystem is largely independent from the access control subsystem. Unless the user also has the `bypass-acl` privilege, operations may still be subject to access control checking. For example, if a user has the `config-read` privilege, then that user can see

only those parts of the configuration that are allowed by access control. As a rule, whenever an operation is covered by both the privilege subsystem and access control, both mechanisms must allow that operation.

Assigning Privileges to Normal Users

By default, normal users are not granted any of the privileges listed above. Therefore, if a user should be allowed to perform any of the associated operations, they must be granted the appropriate privileges. This can be done by adding the `ds-privilege-name` operational attribute to the user's entry.

Note – Adding a privilege with a value such as `modify-acl` is not sufficient for granting a user the right to add, replace, or delete an ACI. Appropriate access control for the user to modify the ACI for another entry is also required. See "ACI Syntax" on page 9 for more information.

`ds-privilege-name` is a multivalued attribute, and if a user is to be given multiple privileges, then a separate value should be used for each one. When the virtual attribute subsystem is in place, it should also be possible to grant privileges to groups of users automatically by making `ds-privilege-name` a virtual attribute in those user entries.

As an example, the following modification can be used to add the `proxied-auth` privilege to the user `cn=Proxy User,dc=example,dc=com`:

```
dn: cn=Proxy User,dc=example,dc=com
changetype: modify
add: ds-privilege-name
ds-privilege-name: proxied-auth
```

Assigning Privileges to Root Users

With the introduction of the privilege subsystem, the primary distinguishing characteristics of root users that separate them from other accounts in the server are that they exist in the configuration rather than in the user data, and that because they are root users they automatically inherit a certain set of privileges. The set of privileges automatically granted to root users is defined in the `ds-cfg-default-root-privilege-name` attribute of the `cn=Root DNs,cn=config` entry. By default, root users are automatically granted the following privileges:

- `bypass-acl`
- `modify-acl`
- `config-read`
- `config-write`

- ldif-import
- ldif-export
- backend-backup
- backend-restore
- server-shutdown
- server-restart
- disconnect-client
- cancel-request
- unindexed-search
- password-reset
- update-schema
- privilege-change

If you want to alter the set of privileges that are automatically assigned to root users, then you may do so by editing the ds-cfg-default-root-privilege-name attribute. Further, if you want to have a different set of privileges for a specific root user, then you can accomplish that using the ds-privilege-name attribute in that root user's entry, just like for a normal user. For example, the following modification may be used to give a specific root user (in this case cn=Test Root User,cn=Root DNs,cn=config) the ability to use proxied authorization while removing the ability to change user privileges or access the configuration. (The minus sign before the privilege indicates that it is being removed rather than granted.):

```
dn: cn=Test Root User,cn=Root DNs,cn=config
changetype: modify
add: ds-privilege-name
ds-privilege-name: proxied-auth
ds-privilege-name: -config-read
ds-privilege-name: -config-write
```

In this case, the cn=Test Root User,cn=Root DNs,cn=config user inherits all privileges automatically granted to root users with the exception of the config-read and config-write privileges and is also given the proxied-auth privilege.

Supported Controls and Operations

The directory server supports a number of standard LDAP controls and extended operations. The following topics list these controls and extended operations.

For information about using the LDAP controls, see "Searching Using Controls" in *Sun OpenDS Standard Edition 2.0 Administration Guide*.

Supported LDAP Controls

A supported control is a mechanism for identifying the request controls supported by the Directory Server. The OIDs of these controls are listed in the `supportedControl` attribute of the server's root DSE.

The directory server supports the following controls:

1.2.826.0.1.3344810.2.3	Matched values control
1.2.840.113556.1.4.805	Subtree delete control
1.2.840.113556.1.4.319	Simple paged results control
1.2.840.113556.1.4.473	Server-side sort control
1.3.6.1.1.12	LDAP assertion control
1.3.6.1.1.13.1	LDAP pre-read control
1.3.6.1.1.13.2	LDAP post-read control
1.3.6.1.4.1.42.2.27.8.5.1	Password policy control
1.3.6.1.4.1.42.2.27.9.5.2	Get effective rights control
1.3.6.1.4.1.42.2.27.9.5.8	Account usability control
1.3.6.1.4.1.7628.5.101.1	Subentry
1.3.6.1.4.1.26027.1.5.2	Replication repair control
2.16.840.1.113730.3.4.2	Manage DSA IT control

2.16.840.1.113730.3.4.3	Persistent search control
2.16.840.1.113730.3.4.4	Password expired control
2.16.840.1.113730.3.4.5	Password expiration warning control
2.16.840.1.113730.3.4.7	Entry change notification control
2.16.840.1.113730.3.4.9	Virtual list view control
2.16.840.1.113730.3.4.12	Proxied authorization control
2.16.840.1.113730.3.4.16	Authorization identity control
2.16.840.1.113730.3.4.17	Real attributes only control
2.16.840.1.113730.3.4.18	Proxied authorization control
2.16.840.1.113730.3.4.19	Virtual attributes only control

Supported Extended Operations

A supported extension is a mechanism for identifying the extended operations supported by the Directory Server. The OIDs of these extended operations are listed in the `supportedExtension` attribute of the server's root DSE.

The supported extensions for the directory server include:

1.3.6.1.1.8	The Cancel extended operation
1.3.6.1.4.1.1466.20037	The StartTLS extended operation
1.3.6.1.4.1.26027.1.6.1	The Password Policy State extended operation
1.3.6.1.4.1.26027.1.6.2	The Get Connection ID extended operation
1.3.6.1.4.1.26027.1.6.3	The Get Symmetric key extended operation
1.3.6.1.4.1.4203.1.10.2	The LDAP no-op control
1.3.6.1.4.1.4203.1.11.1	The Password Modify extended operation
1.3.6.1.4.1.4203.1.11.3	The "Who Am I?" extended operation