Referral Handling Changes

Introduction

Since 1.0 the way referrals are handled in the core and in various layers above have changed. This document is intended to updater those interested in the details of handling referrals in the server.

According to RFC 2251/4511, a referral is returned inside a LdapResult, if the result code is set to **referral**, or as one or more **SearchResultReference**.

Info on Referrals and JNDI

- RFC 2251
- RFC 3296
- JNDI Tutorial on Referral Behavoir

ApacheDS Implementation Notes

Referral handling must be accounted for in two respects. At the protocol level in the MINA LDAP protocol provider (protocol-ldap module) and at the JNDI level in the core ApacheDS JNDI Provider (core-jndi module). Both must behave according to their respective specifications when dealing with referrals.

Changes Since 1.0

The ReferralInterceptor has been removed. The core of ApacheDS no longer handles referrals at all. All entries stored in the core are represented as standard entries returned as ServerEntry objects. No longer does the core handle modes of throwing of exceptions, chasing them or ignoring them. They're just entries in the core.

It is completely the responsibility of the core-jndi module to handle the expected modes and behavior for JNDI LDAP providers while dealing with referrals. Likewise the protocol-ldap module is responsible for complying with LDAP specification requirements concerning the handling of referral entries with and without the presence of the ManageDSAiT Control.

We will restore a ReferralInterceptor for another reason: to avoid a performance penalty we have when doing a lookup to determinate if an entry has a referral ancestor. This interceptor will manage the Reefrral cache when adding/deleting or modifying a referral.

Motivation for Changes

The big bang effort to refactor JNDI constructs out of the server achieved many of it's intended goals. JNDI was complicating the picture and often causing an impedance mismatch if not complete confusion on how to bridge between JNDI and the protocol. There was too much complexity as a result.

With referrals we were mixing into the core, the requirements of JNDI LDAP providers and the LDAP protocol resulting in a mess when handling referrals in a clear fashion. The code was hard to maintain and difficult to understand. So we removed all the referral handling code which was mostly JNDI specific out of the core which removed the need to have a ReferralInterceptor all together.

Now it's the job of the LDAP protocol fontend and the core JNDI provider to apply the appropriate behavior in their own specific way. This is cleaner because it does not mix the two different ways in which these layers above the core must deal with referrals.

Before the protocol-ldap module sat on top of the core which included the JNDI provider implementation. Now the core JNDI provider has been moved out of the core into it's own module: core-jndi. The protocol-ldap module no longer sits on top of the core JNDI provider but uses a new API to directly tap into the core without having to understand JNDI and deal with it's quirks.

ManageDsaIT Control

The ASN.1 subsystem understands the ManageDsaIT control and the server publishes that it supports this control in the RootDSE. The control determines how the LDAP protocol provider handles responses when present and referrals are encountered. There is no longer any JNDI in the protocol provider, so it does not need to pass controls down into the JNDI provider for the core to make critical decisions about request handing behavior.

Context.REFERRAL Property

The *Context.REFERRAL* property in the JNDI environment affects the way referrals are handled by JNDI LDAP providers including the core JNDI provider. According to JNDI specifications:

```
A JNDI application uses the Context.REFERRAL(in the API reference documentation)
("java.naming.referral") environment property to indicate to the service providers how
to handle referrals. The following table shows the values defined for this property. If this
property has not been set, then the default is to ignore referrals.
```

Property Setting	Description
ignore	**Ignore referrals (they are considered as normal entries)**
follow	**Automatically follow any referrals**
throw	**Throw a ReferralException(in the API reference documentation) for each referral**

Based on the entry point, (via protocol or embedded JNDI) two mechanisms exist for controlling the underlying Referral handing mechanism. One uses the ManageDsaIT control and the other uses the Context.REFERRAL property. The presence of the ManageDsaIT control is the same as setting the environment property to **ignore** or not even setting the property in the environment since by absence the property is defaulted to **ignore**.

Odd as it sounds, adding an entry which is a subordinate of an existing referral into the server seems to be a possibility, using the ManageDSAIT control. This is not the case. As any modification of the data should keep the server in a consistent state, such an addition is obviously forbidden. (this has been discussed here

The core-JNDI provider does not, at the moment, support the 'follow' property.

Referral Handling Scenarios

Here's a slightly modified example DIT used in RFC 3296. We'll also use this to elaborate on the behavior of operations based on the different scenarios outlined in 3296.

Legend

Green nodes are actual entries. Red nodes are referrals.

Finding target in non-search operations

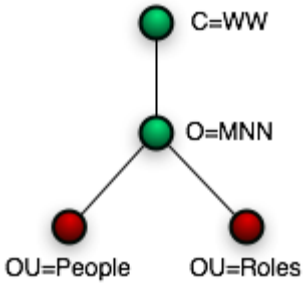
The handling for **add**, **compare**, **delete**, **modify** and **modify DN** operations to the target entry operated on is the same. The RFC gets a bit confusing when describing different scenarios and it's examples are lacking. They could have picked referrals where the **DN** is not the same as the reference to better demonstrate what they exactly meant. Regardless there seems to be 3 cases worth considering (whether the added entry is a referral or not is irrelevant) :

1. target is present, and has no ancestor which is a referral
2. target is not present, and no ancestor is a referral
3. target is not present, but an ancestor is a referral

(the special case "target is present, and has an ancestor which is a referral" is impossible...).

If we consider the tree we are using for our samples, those 3 cases can be represented as :

1. target's DN is "o=MNN,c=WW" or "ou=people, o=MNN, c=WW" (in this last example, the associated entry will be a referral.
2. target's DN is "o=absent,c=WW"
3. target's DN is "cn=Alex karasulu,ou=people,o=MNN,c=WW"



OU=People,O=MNN,C=WW

```
ou: People
ref: ldap://hostb/OU=People,
DC=example,DC=com
ref: ldap://hostc/OU=People,
O=MNN,C=WW
objectClass: referral
objectClass: extensibleObject
```

OU=Roles,O=MNN,C=WW

```
ou: Roles
ref: ldap://hostd/ou=Roles,
dc=apache,dc=org
objectClass: referral
objectClass: extensibleObject
```

Referrals and LDAP operations

We now will describe the way Referrals are handled, depending on the operation the server will receive. We will consider the three different cases :

- through JNDI
- through the server own API (CoreAPI)

Add Operation handling

test	target exists	is a referral	has an ancestor	JNDI/Core handling	Description
1	no	no	no	Irrelevant	Adds the entry into the server
2			yes	JNDI ignore	The JNDI provider will throw a PartialResultException
3				JNDI throw	The JNDI provider will throw a LdapReferralException
4				Core API	The Core API will throw a LdapReferralException Equivalent to the JNDI throw handling
5				Core API+ManageDsaIT	The Core API will throw a PartialResultException Equivalent to the JNDI ignore handling
6	yes	irrelevant	no	irrelevant	Throws an EntryAlreadyExists error

Test 1

We try to add the following entry :

```
dn: cn=Alex karasulu, c=MNN, c=WW
ObjectClass: top
ObjectClass:person
cn: Alex Karasulu
sn: alex
```

As a result, we should be able to find the entry in the local server.

Test 2 & 5

We try to add the following entry, using the Context.REFERRAL=ignore property (JNDI) or adding the ManageDsaIT control (Core API) :

```
dn: cn=Alex karasulu, ou=users, ou=people , c=MNN, c=WW
ObjectClass: top
ObjectClass:person
cn: Alex Karasulu
sn: alex
```

We should get a PartialResultException containing the **ou=people,c=MNN,c=WW** result

Test 3 & 4

We try to add the following entry, using the Context.REFERRAL=throw property (JNDI) or without the ManageDsaIT control (Core API) : :

```
dn: cn=Alex karasulu, ou=users, ou=people , c=MNN, c=WW
ObjectClass: top
ObjectClass:person
cn: Alex Karasulu
sn: alex
```

We should get a LdapReferralException.

Test 6

We try to add the following entry twice :

```
dn: cn=Alex karasulu, c=MNN, c=WW
ObjectClass: top
ObjectClass:person
cn: Alex Karasulu
sn: alex
```

As a result, we should get an EntryAlreadyExist exception

Delete Operation handling

test	target exists	is a referral	has an ancestor	JNDI/Core handling	Description
1	no	irrelevant	no	Irrelevant	Returns a NoSuchObject exception
2			yes	JNDI+throw	Returns a PartialResult exception
				JNDI+ignore	Throws a LdapReferralExceptionexception
3				Core API	Returns a PartialResult exception
				Core API+ManageDsaIt	Throws a LdapReferralException exception
4	yes	no	no	Irrelevant	Remove the entry from the server
5		yes	no	JND+throw	Throw a LdapReferralException exception
6				JNDI+ignore	Remove the entry from the server
7				CoreAPI	Throw a LdapReferralException exception
8				Core API+ManageDsaIt	Remove the entry from the server

Test 1

Trying to delete dn: **cn=not present, c=MNN, c=WW** should fail

Test 2 & 3

Trying to delete dn: **cn=alex karasulu, ou=people, c=MNN, c=WW** should throw a PartialResultException

Test 4

We should be able to delete dn: **o=MNN, c=WW**

Test 5 & 7

Trying to delete dn: **ou=people, c=MNN, c=WW** should throw a ReferralException

Test 6 & 8

We should be able to delete dn: **ou=people, c=MNN, c=WW**

Compare Operation handling

test	target exists	is a referral	has an ancestor	JNDI/Core handling	Description
1	no	irrelevant	no	Irrelevant	Returns a NoSuchObject exception
2			yes	JNDI+throw	Returns a PartialResult exception
				JNDI+ignore	Throws a LdapReferralException exception
3				Core API	Returns a PartialResult exception
				Core API+ManageDsaIT	Throws a LdapReferralException exception
4	yes	no	no	Irrelevant	Returns the comparison result
5		yes	no	JNDI+throw	Throws a LdapReferralException exception
6				JNDI+ignore	Returns the comparison result
7				CoreAPI	Throws a LdapReferralException exception
8				Core API+ManageDsaIT	Returns the comparison result

test 1

Doing a compare on attribute **Objectclass** for **cn=not present, c=MNN, c=WW** should fail with a NoSuchObject exception

test 2 & 3

Doing a compare on attribute **Objectclass** for **cn=alex karasulu, ou=people, c=MNN, c=WW** should throw a PartialResultexception

test 4

Doing a compare on attribute **Objectclass** for **c=MNN, c=WW** should return a success

test 5 & 7

Doing a compare on attribute **Objectclass** for **ou=people, c=MNN, c=WW** should throw a ReferralException

test 6 & 8

Doing a compare on attribute **Objectclass** for **ou=people, c=MNN, c=WW** should should return a success

Modify Operation handling

test	Target exists	is a referral	has an ancestor	JNDI/Protocol handling	Description
1	no	irrelevant	no	irrelevent	Throws a NoSuchObject exception
2			yes	JNDI+throw	Throws a PartialResultException
				JNDI+ignore	Throws a LdapReferralException exception
3				Core API	Throws a PartialResultException
				Core API+ManageDsaIT	Throws a LdapReferralException exception
4	yes	no	no	irrelevant	Modify the entry on the server
5		yes	no	JNDI+throw	Throws a LdapReferralException

6				JNDI+ignore	Modify the referral on the server
7				Core API	Throws a LdapReferralException exception
8				Core API+ManageDsaIT	Modify the referral on the server

Test 1

Doing a modify on **cn=not present, c=MNN, c=WW** should fail with a NoSuchObjectException

Test 2 & 3

Doing a modify on **cn=alex karasulu, ou=people, c=MNN, c=WW** should throw a LdapPartialResult exception

Test 4

Doing a modify on **c=MNN, c=WW** should modify the entry on the server

Test 5 & 7

Doing a modify on **ou=people, c=MNN, c=WW** should throw a LdapReferral exception

Test 6 & 8

Doing a modify on **ou=people, c=MNN, c=WW** should modify the referral on the server

ModifyDN Operation handling

The ModifyDN operation is slightly more complicated, as we may change two things which might affect the operation :

- the new DN (it's a rename)

- the new Superior (it's a move)

And we can combine those two modifications (it's a move and rename).

One more important thing : the RFC states that :

RFC 3296 Section 5.6.2

```
If the newSuperior is a referral object or is subordinate to a
    referral object, the server SHOULD return affectsMultipleDSAs.  If
    the newRDN already exists but is a referral object, the server SHOULD
    return affectsMultipleDSAs instead of entryAlreadyExists.
```

We will analyze those three kind of modifications separately.

Rename operation

test	Entry exists	New RDN exists	is a referral	has an ancestor	JNDI/Protocol handling	Description
1	no	irrelevant	irrelevant	no	irrelevent	Throws a NameNotFoundException exception
2				yes	JNDI+throw	Throws a ReferralException
3					JNDI+ignore	Throws a PartialResultException
4					Core API	Throws a ReferralException
5					Core API+ManageDsaIT	Throws a PartialResultException
6	yes	no	no	irrelevant	irrelevant	Renames the entry on the server
7			yes	no	JNDI+throw	Throws a ReferralException
8					JNDI+ignore	Renames the referral on the server
9					Core API	Throws a ReferralException
10					Core API+ManageDsaIT	Renames the referral on the server
11		yes	no	irrelevant	irrelevant	Throws a NameAlreadyBoundException exception
12			yes	irrelevant	JNDI+throw	Throws a ReferralException
13					JNDI+ignore	Throws a NameAlreadyBoundException exception
14					Core API	Throws a ReferralException
15					Core API+ManageDsaIT	Throws a NameAlreadyBoundException exception

Test 1

Renaming **cn=not present, c=MNN, c=WW** to **cn=akarasulu, c=MNN, c=WW** on should fail with a NameNotFoundException

Test 2 & 4

Renaming **cn=not present, ou=people, c=MNN, c=WW** to **cn=new name, ou=people, c=MNN, c=WW** should throw a ReferralException exception

Test 3 & 5

Renaming **cn=not present, ou=people, c=MNN, c=WW** to **cn=new name, ou=people, c=MNN, c=WW** should throw a LdapPartialResult exception

Test 6

Renaming **cn=Alex Karasulu, c=MNN, o=WW** to **cn=Alex, c=MNN, o=WW** should rename the entry on the server

Test 7 & 9

Renaming **ou=people, c=MNN, c=WW** to **cn=new name, c=MNN, c=WW** should throw a LdapReferral exception

Test 8 & 10

Renaming **ou=people, c=MNN, c=WW** to **cn=new name, c=MNN, c=WW** shouldrename the referral on the server

Test 11

Renaming **ou=Alex Karasulu, c=MNN, c=WW** to **cn=Emmanuel Lecharny, c=MNN, c=WW** should throw a NameAlreadyBoundException exception (both entry already exist)

Test 12 & 14

Renaming **ou=people, c=MNN, c=WW** to **ou=roles, c=MNN, c=WW** should throw a ReferralException

Test 13 & 15

Renaming **ou=people, c=MNN, c=WW** to **ou=roles, c=MNN, c=WW** should throw a NameAlreadyBoundException

Move operation

It's a bit different than the rename operation, as we may have an existing new superior, but with a non existing combinaison of the oldRDN + new superior.

test	OldSuperior exists	OldSuperior has an ancestor	OldSuperior is a referral	New superior exists	NewSuperior is a referral	NewSuperior has an ancestor	JNDI/Protocol handling	Description
1	no	no	irrelevant	irrelevant	irrelevant	irrelevant	irrelevent	Throws a NameNotFoundException exception
2		yes	irrelevant	irrelevant	irrelevant	irrelevant	JND+throw	Throws a ReferralException exception
3							JNDI+ignore	Throws a PartialResult exception
4							Core API	Throws a ReferralException exception
5							Core API+ManageDsaIT	Throws a PartialResult exception
6	yes	no	no	no	no	no	irrelevent	Move the branch Handle inner referrals
7				yes	no	yes	irrelevant	Returns a AffectMultipleDsasresult
8					yes	irrelevant	irrelevant	Returns a AffectMultipleDsasresult
9			yes	irrelevant	irrelevant	irrelevant	JND+throw	Throws a ReferralException exception
10							JNDI+ignore	Throws a PartialResult exception
11							Core API	Throws a ReferralException exception
12							Core API+ManageDsaIT	Throws a PartialResult exception

Test 1

Moving **c=MNN, cn=not present** to **c=MNN, c=XX** on should fail with a NameNotFoundException

Test 2 & 4

Moving **cn=Emmanuel Lecharny, ou=apache, ou=People, c=MNN, c=WW** to **cn=Emmanuel Lecharny, ou=apache, ou=org** should throw a ReferralException exception.

Test 3 & 5

Moving **cn=alex karasulu, ou=apache, ou=people, c=MNN, c=WW** to **cn=alex karasulu, ou=asf, ou=people, c=MNN, c=WW** should throw a PartialResultException exception.

Test 6

Moving **cn=Alex Karasulu, c=MNN, c=WW** to **cn=Alex Karasulu, c=MNN, c=XX** should move the branch to it's new place, with all the children.

Test 7

Moving **cn=Alex, c=MNN, c=WW** to **cn=Alex, ou=people, c=MNN, c=WW** should throw a AffetcsMultipleDSAs result.

Test 8

Moving **cn=Alex, c=MNN, c=WW** to **cn=Alex, ou=apache, ou=people, c=MNN, c=WW** should throw a AffetcsMultipleDSAs result.

Test 9 & 11

Moving **cn=Alex, ou=apache, ou=people c=MNN, c=WW** to **cn=Alex, ou=people, c=MNN, c=WW** should throw a ReferralException exception.

Test 10 & 12

Moving **cn=Alex,ou=apache, ou=people, c=MNN, c=WW** to **cn=Alex, c=not present** should throw a PartialResult exception.

Move and rename operation

test	Old DN exists	Old Superior is a referral or has an ancestor	New DN exists	New Superior is a referral or has an ancestor	JNDI/Core API	Description
1	no	no	no	irrelevant	irrelevant	Throws a NameNotFoundException exception
2		yes(has an ancestor)	irrelevant	irrelevant	JNDI+throw	Throws a ReferralException exception
3					JNDI+ignore	Throws a PartialResult exception
4					Core API	Throws a ReferralException exception
5					Core API+ManageDsaIT	Throws a PartialResult exception
6	yes	no	no	no	irrelevant	Moves and renames the entry, and the children
7				yes	irrelevant	Throws a AffectMultipleDsas
8			yes	irrelevant	irrelevant	Throws an NameAlreadyBoundException exception
9		yes (is a referral)	irrelevant	irrelevant	JNDI+throw	Throws a ReferralException exception
10					JNDI+ignore	Throws a PartialResult exception
11					Core API	Throws a ReferralException exception
12					Core API+ManageDsaIT	Throws a PartialResult exception

Test 1

Trying to move and rename **ou=not present, o=MNN, c=WW** to whatever DN should throw a NameNotFoundException exception

Test 2 & 4

Trying to move and rename **cn=alex karasulu, ou=apache, ou=people, o=MNN, c=WW** to whatever DN should throw a ReferralException exception

test 3 & 5

Trying to move and rename **cn=alex karasulu,ou=apache, ou=people, o=MNN, c=WW** to whatever DN should throw a partialResultException exception

test 6

Trying to move and rename **cn=alex karasulu,o=MNN, c=WW** to **cn=Alex,o=PNN,c=WW** should move and rename the entry

test 7

Trying to move and rename **cn=alex karasulu,o=MNN, c=WW** to **cn=Alex, ou=People, o=MNN, c=WW** should give a AffectsMultipleDSAs result

test 8

Trying to move and rename **cn=Alex Karasulu,o=MNN, c=WW** to **cn=Emmanuel Lecharny, o=PNN, c=WW** DN should throw a NameAlreadyBoundException exception

test 9 & 11

Trying to move and rename **cn=Alex Karasulu,ou=People,c=MNN, c=WW** to **cn=Alex, c=PNN, c=WW** should throw a ReferralException exception

test 10 & 12

Trying to move and rename**cn=Alex Karasulu,ou=People,c=MNN, c=WW** to **cn=Alex, c=PNN, c=WW** should throw a PartialResultException exception

Search Operation handling

test	Target exists	is a referral	has an ancestor	JNDI/Protocol handling	Description
1	no	no	no	irrelevant	Throws a NameNotFoundException exception
2			yes	JNDI+throw	Returns a ReferralException exception
3				JNDI+ignore	Throws a PartialResultException
4				Core API	Returns a SearchResultReference
5				Core API+ManageDsaIT	Throws a PartialResultException
6	yes	no	irrelevant	irrelevant	Returns the search result
7		yes	irrelevant	JNDI+throw	Returns a SearchResultReference
8				JNDI+ignore	Returns the entry
9				Core API	Returns a SearchResultReference
10				Core API+ManageDsaIT	Returns the entry

Test 1

Searching for **cn=not present, c=WW** should return an empty result

Test 2 & 4

Searching for **cn=alex karasulu, ou=apache, ou=people, c=MNN, c=WW** should return a SearchResultReference

Test 3 & 5

Searching for **cn=alex karasulu, ou=apache, ou=people, c=MNN, c=WW** should throw a Partial Result Exception

Test 6

Searching for **c=MNN, c=WW** should return the entry.

Test 7 & 9

Searching for *ou=people, c=MNN, c=WW" should return a SearchResultReference

Test 8 & 10

Searching for *ou=people, c=MNN, c=WW" should return the entry.

Other technical considerations

case #1: Target is not a referral, has no ancestor which is a referraThe presence of the ManageDsaIT control is irrelevent.
JNDI handlingAs the entry is not a referral, whatever value is set to the Context.REFERRAL property, the response will be the same : the server simply returns the entry if it existsMINA provider handling

Without the ManageDsaIT control

When the target is a referral, the refs are returned back to the client with a resultCode of **REFFERAL** (example from RFC). If for example the client issues a **modify** for the target of "OU=People,O=MNN,C=WW", the server will return the following when the **ManageDsaIT** control is **NOT** present:

Server Response

```
ModifyResponse (referral) {
    ldap://hostb/OU=People,DC=example,DC=com
    ldap://hostc/OU=People,O=MNN,C=WW
}
```


Referral Modifications

The ref attribute values **SHOULD** be modified to exclude any scope, filter or attribute list from the URI if it is an LDAP URL. These search specific URL elements must be removed because the operation to be continued by chasing the referred are not be search operations.

Let's consider how the request is handled regarding to the two layers : MINA provider and JNDI provider.

JNDI handling

In this situation, without the **ManageDsaIT** control, the ApacheDS LDAP frontend (MINA provider) will set the value of the Context.REFFERAL property to "**throw**" before issuing JNDI calls to the core. The JNDI operation on the ApacheDS JNDI DirContext will throw a ReferralException which shall contain everything needed for the LDAP frontend to respond properly. This also allows, embedding applications to see the same results they would encounter from the SUN JNDI LDAP Provider operating against a remote LDAP server.

Case #3: Target's parent is a referral

According to the RFC 3296 it appears as though the remaining name past the referral is appended to the DN of the ref attributes, if the values are LDAP URLs. Also if they are LDAP URLs the scope, filter and attribute terms are removed. The result is returned back. To illustrate this let's consider the example from the RFC where an add operation is performed with the target DN of "CN=Manager,OU=Roles,O=MNN,C=WW".

The dynamics of the add operation must be considered first WRT the ApacheDS JNDI provider. This operation can proceed in two ways. First via the lookup of the parent context, "OU=Roles,O=MNN,C=WW", followed by a createSubcontext() operation on it using the RDN of the target entry. Other way to perform the add operation is by looking up an ancestor context above the parent, "O=MNN,C=WW" for example, followed by a createSubcontext() operation using a name fragment like "CN=Manager,OU=Roles". The last situation is not performed by the ApacheDS LDAP frontend but it can be performed by an embedding application against the JNDI interfaces.

When the Context.REFERRAL environment property is not set (an implicit ignore) or is explicity set to the "**ignore**" String, the createSubcontext() operation, regardless of what parent or ancestor it is issued upon will create the target entry under the referral parent. Remember when referrals are ignored all referrals are processed as regular entries. The dynamics get interesting when the Context.REFFERAL property is set to "**throw**". Incidentally we will ignore the "**follow**" value for the property for the time being. In the first case where the createSubcontext() operation is performed on the parent, which is the the referral entry "OU=Roles,O=MNN,C=WW", the attempt to create a non-existing child will succeed unless logic is put into place. The logic must allow the Context implementation to detect the fact that it is a referral, and that the createSubcontext() operation being performed with the Context.REFERRAL property set to "**throw**" must be prevented. BTW If an attempt is made to lookup the parent context with the Context.REFERRAL property set to "**throw**" then a ReferralException will occur. So to get the parent we would have had to look it up with referral's ignored.

In the latter case, the createSubcontext() operation is being performed upon a (non-referral) ancestor with a name fragment for the target, "CN=Manager, OU=Role". The context used to perform the operation does not care what mode the Context.REFERRAL property is set to since it is not a referral itself. It will issue the add request against the nexus with the computed target DN. The only way to prevent incorrect creation of this entry, is to check through the target's lineage for an ancestor that is a referral. If no ancestor up to the root suffix context is a referral then the operation may proceed. We cannot just check if the parent is a referral because the parent may be a regular entry hidden under another referral: meaning the target's parent may have been created while ignoring referrals. So we must exhaust the entire lineage or short the process when we find a parent or ancestor that is a referral. This must happen within the JNDI provider when the Context.REFERRAL mode is set to "**ignore**".

Case #4: Target's ancestor (not parent) is a referral

This is very similar to the latter half of case #3 above. When Context.REFERRAL="**throw**", The ApacheDS JNDI provider must test to see if any ancestors of the target entry are referrals. The biggest difference here is in the processing of ref attribute value DN fields (if they are LDAP URLs). Here the remaining name after the referral ancestor is tacked onto the DN components of the ref value. So if we were performing a createSubcontext() to add "CN=OneDown,CN=Manager,OU=Role,O=MNN,C=WW", the ancestor "OU=Role,O=MNN,C=WW" is a referral. Now the parent "CN=Manager" may or may not exist. Whether the parent exists or not we have to check for the presence of a referral ancestor before allowing the add operation to proceed. Again the best place for this is within the JNDI provider. In this example the returned AddResponse would be:

AddResponse

```
AddResponse (referral) {
        ldap://hostd/CN=OneDown,CN=Manager,ou=Roles,dc=apache,dc=org
    }
```

"Review & update for 1.5 stopped here!!!"

~akarasulu

Protocol Handlers

Since the LDAP protocol provider no longer sits on the JNDI Provider it no longer delegates checks for referrals to the JNDI provider. Each handler must handle referral semantics as specified by the protocol where referrals are concerned. In most write based handlers this just means the handlers must check if the target entry is a referral and issue the correct result codes and URIs.

Finding base of search operations

Here we discuss referral handling for finding the search base which is very similar to finding the targets of other operations. Unlike the other operations if we encounter a referral while finding the search base we must add a search scope specifier to the ref if it's value is an LDAP URL. Also critical extensions MUST NOT be trimmed nor modified.

Another difference to calculating ref values is in factoring in alias dereferrencing. This is where things seem to get a little tricky but not really. Whether or not the name for the discovered referral is an aliased name or the primary processing of the URL DN in ref values is the same. Only the remaining name, the remaining part of the search base DN after the referral's DN, is needed and appended to the the DN of the URL in the ref value. Either it's easier in ApacheDS because of the architecture or we're way oversimplifying this. Now let's review the cases for referral handling while finding the search base.

1. base is a normal entry (**default**)
2. base is a referral
3. base's parent is a referral
4. base's parent is or is not present, but an ancestor is a referral

Case #1 is the default case and will be skipped for consideration/elaboration.

Case #2: Same as for finding target entries with URL handling differences

An example is best for this case. We'll take the one in the RFC. If the client issues a subtree search in which the base object is "OU=Roles,O=MNN, C=WW", the server will return:

SearchResultDone

```
SearchResultDone (referral) {
        ldap://hostd/ou=Roles,dc=apache,dc=org??sub
    }
```

Notice the extra subtree scope parameter tacked onto the URL.

Case #3: Same as for finding target entries with URL handling differences

Again an example is best for this case. We'll take the one in the RFC. If the client issues a base scoped search in which the base object is "CN=Manaager, OU=Roles,O=MNN,C=WW", the server will return:

SearchResultDone

```
SearchResultDone (referral) {
        ldap://hostd/CN=Manager,ou=Roles,dc=apache,dc=org??base
    }
```

Notice the extra subtree scope parameter tacked onto the URL.

I won't elaborate on Case #4 since it's pretty much the same concept.

Back on track with search continuations

So for each referral within scope, we have to return a SearchResultReference using the URI compoents of the ref attribute. Here's what we have to do to transform that URI:

- If the URI component is not a LDAP URL, it should be returned as is.
- If the LDAP URL's DN part is absent or empty, the DN part must be modified to contain the DN of the referral object.
- If the URI component is a LDAP URL, the URI SHOULD be modified to add an explicit scope specifier.

Subtree Example (From RFC 3296)

Subtree search of "O=MNN,C=WW" with filter (objectClass=*) might return:

Subtree Results

```
SearchEntry for O=MNN,C=WW
SearchResultReference {
    ldap://hostb/OU=People,DC=example,DC=com??sub
    ldap://hostc/OU=People,O=MNN,C=WW??sub
}
SearchEntry for CN=Manager,O=MNN,C=WW
SearchResultReference {
    ldap://hostd/OU=Roles,dc=apache,dc=org??sub
}
SearchResultDone (success)
```


Smart Filter Alteration

If the filter contains an objectClass=* OR branch there is no point to altering it. Might want to look into a simple test for this before altering the filter to add a new branch node and OR term. (objectClass=*) is common and it makes (objectClass=referral) redundant.

One Level Example (From RFC 3296)

Same search but scope is one level on the same base:

Subtree Results

```
SearchResultReference {
    ldap://hostb/OU=People,DC=example,DC=com??sub
    ldap://hostc/OU=People,O=MNN,C=WW??sub
}
SearchEntry for CN=Manager,O=MNN,C=WW
SearchResultReference {
    ldap://hostd/OU=Roles,dc=apache,dc=org??sub
}
SearchResultDone (success)
```

Processing Considerations for Other Operations

Operations

We won't have to implement every operations in the interceptor : some of them are not necessary, like operations which do not modify the entries. For instance, **bind()** operation is not implemented.

Here is the list of operations defined in the interface, and the list of operations we implement in **ReferralInterceptor** (the missing methods are already implemented in the intermediate abstract class) :

Interface	ReferralInterceptor
add	✓
addContextPartition	✓
bind	✗
compare	✓
delete	✓
destroy	✗

Interface	ReferralInterceptor
getMatchedName	✗

getRootDSE	✗
getSuffix	✗
hasEntry	✗
init	✓

Interface	ReferralInterceptor
isSuffix	✗
list	✗
listSuffixes	✗
lookup	✗
modify	✓

Interface	ReferralInterceptor
modifyRn	✓
move	✓
removeContextPartition	✓
search	✓
unbind	✗

Conclusion

We will need to alter the ApacheDS JNDI provider, and the LDAP server frontend (MINA LDAP protocol provider) to handle referrals correctly. Here are the changes required for each subsystem.

Changes to JNDI Provider

- Add logic to check for parent and ancestor referrals when referral handling is not ignored
- Throw the appropriate ReferralExceptions with ref modification with referral handling set to throw
- Implement follow handling to chase referrals
- Add special handling for search to properly modify referral LDAP URLs
- Add code to alter the search operation when referral handling is not ignored
- Create and add search result enumeration filter to collect referrals and save them for returning last after the underlying enumeration has been exhausted of regular entries. This way we can return named continuation referrences last as JNDI LDAP providers are supposed to do.

Changes to MINA LDAP Frontend (Protocol Provider)

- Make handlers set the Context.REFERRAL property approapriate (ignore or throw)
- Make handlers correctly deal with ReferralExceptions for non-search operations
- Handle search continuations properly