Aliases and Search

by Alex Karasulu

LDAPv3 Aliases

LDAP aliases provide alternative distinguished names for entries. They bypass the hierarchical structure of the directory by pointing anywhere without constraint. The objectClass definition for an alias is provided below along with the attributeType definition for its sole required attribute aliasedObjectName:

(2.5.6.1 NAME 'alias' SUP top STRUCTURAL MUST aliasedObjectName)

(2.5.4.1 NAME 'aliasedObjectName' EQUALITY distinguishedNameMatch SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 SINGLE-VALUE)

The aliasedObjectName uses the distinguishedNameSyntax. Hence an LDAP alias is an LDAP entry that contains the distinguished name of another LDAP entry on the same LDAP server. Dereferencing parameters associated with search requests are used to optionally dereference aliases on the server before returning results to the client. According to section 4.5.1 of RFC2251, search requests contain a search operation modifier, 'derefAliases', which affects the manner in which aliases are handled during the course of a search operation. Parts of the RFC's section defining the values of the derefAliases search parameter are listed in the block below:

Search Request Indicator ASN.1 Notation Snippet:

```
derefAliases ENUMERATED
      {
      neverDerefAliases       (0),
      derefInSearching        (1),
      derefFindingBaseObj     (2),
      derefAlways             (3)
      }
```

Indicator Commentary Snippet:

```
derefAliases: An indicator as to how alias objects (as defined in
X.501) are to be handled in searching.  The semantics of the
possible values of this field are:

      neverDerefAliases: do not dereference aliases in searching
      or in locating the base object of the search ;

      derefInSearching: dereference aliases in subordinates of the
      base object in searching, but not in locating the base
      object of the search ;
```

```
derefFindingBaseObj: dereference aliases in locating the
base object of the search, but not when searching
subordinates of the base object ;

derefAlways: dereference aliases both in searching and in
locating the base object of the search.
```

The definitions for the four modes of alias handling raise questions regarding alias handling behavior. These questions deal with the order in which dereferencing and search selection criteria are applied. If alias dereferencing is applied before the application of search criteria then the filter takes affect on the target entry referred to by the alias. Conversely if dereferencing is applied after the application of search criteria then the filter is applied to the alias itself and not on the target entry. In the later case the alias may never be a candidate for dereferencing if the search filter does not select it. Although a subtle nuance in alias handling, the order of handling can produce dramatic differences in the returned search result set. These questions must be answered before choosing a design approach that incorporates the handling of aliases.

The existing LDAP and DAP standards do not explicitly confront these crucial details. Perhaps the lack of standardization is the reason why so many directory servers do not support aliases. The SUN One Directory Server, formerly known as the iPlanet Directory Server, does not support aliases. Although a sticky point, aliases provide an immeasurable degree of flexibility, and should be supported. Ultimately we need to pose these questions to both the LDAP and DAP communities to find the appropriate answers. For the time being some inferences can be drawn from what little information we do have regarding alias handling. These inferences can be presented to the community for definitive answers after we draw them out. Unfortunately formal documentation on or specifications describing the exact behavior of aliases is scant. There seems to be a defunct IETF draft that never made it to RFC which discusses the expected behavior of aliases here. We do not trust or recommend the content within this draft since it has been rejected and conflicts with existing RFC's.

There are several questions that need to be answered before we implement aliases. Finding the answers to these questions and the constraints they impose on the design are the goals of this document:

1. Does alias dereferencing occur before or after the application of a filter?
2. Does dereferencing follow aliases chains? In other words, when dereferencing is enabled, does the process continue dereferencing an alias chain until a non-alias entry is found?
3. How do we handle alias imposed cycles while conducting search? References to parents and other relatives could impose cycles. What mechanisms will be put in place to bound searches preventing infinite loops within alias imposed cycles?

4. How do we elegantly make JNDI environment parameters and other search modifying aspects of a JNDI Context available deep within the search engine?
5. When traversing aliases to return entries beyond the expected scope of the search base, which absolute path should be returned for entry results: the primary distinguished name or the alternative distinguished name through the alias?

Our goal is to eventually be able to backend both LDAP and X.500 directories. To do so we must also consider X.500 nuances associated with aliases.

X.500 Aliases

According to X.501 aliases are alternative distinguished names for objects provided by the use of 'alias' entries. An X.501 alias uses the aliasedEntryName attribute instead of the aliasedObjectName attribute used in LDAP. Regardless X.501(02/2001) section 9.8 note 2 states that the aliasedEntryName can be the primary distinguished name or any other alternative distinguished name. Hence X.501(02/2001) allows alias chaining, however it warns that chaining may introduce inconsistencies between the 02/2001 specification and specifications before 1997. X.501 goes on to define the process of alias dereferencing which accounts for alias chains and finally explicitly states that aliases can point to other aliases:

> X.501 Section 9.8
>
> The conversion of an alias name to an object name is termed (alias) dereferencing and comprises the systematic replacement of alias names, where found within a purported name, by the value of the corresponding **aliasedEntryName** attribute. The process may require the examination of more than one alias entry.
>
> Any particular entry in the DIT may have zero or more alias names. It therefore follows that several alias entries may point to the same entry. An alias entry may point to an entry that is not a leaf entry and may point to another alias entry.
>
> An alias entry shall have no subordinates, so that an alias entry is always a leaf entry.

X.501 also scratches the surface of directory inconsistencies resulting from the deletion of entries with aliases pointing to them. “Stale aliases”, as we coin them, can exist while the directory is in operation. Administrators are expected to remove or make stale aliases consistent:

> X.501 02/2001 Section 21.3
>
> During the process of modification of entries it is possible that the Directory may become inconsistent. This will be particularly likely if modification involves aliases or aliased objects which may be in different DSAs. The inconsistency shall be corrected by specific administrator action, for example to delete aliases if the corresponding aliased objects have been deleted. The Directory continues to operate during this period of inconsistency.

Other security implications with regard to alias handling are discussed however we omit these sections due to immediate relevance. Security, namely authorization is handled outside of backends.

Reaching Conclusions By Way Of Common Sense

Above we listed the modes for search alias handling. If we consider alias handling in each of these modes using very special search scenarios we can draw some conclusions regarding the behavior of aliases.

An Alias Entry for the Search Base

A particularly special situation is when the search base is an alias. The base is constrained by the DSA to be a leaf node since aliases can only be leaves. When using the neverDerefAliases mode the alias entry is handled like any other regular leaf entry. There are two possible outcomes to the search. One where the alias entry is returned and one where nothing is returned. The scope and filter will determine the result. If we presume the filter accepts the alias then the scope will determine the outcome. The base and subtree scopes should return a single entry: the alias base. A one level scope search never returns the base in the result, so with one level scope, the search returns nothing.

The derefInSearching mode also will not dereference aliases while finding a base. For any alias used for the search base, the derefInSearching mode should behave just like neverDerefAliases mode. Presuming the filter accepts the alias entry, the scope of the search determines whether the alias entry is returned or if nothing at all is returned. If base or subtree level scope is used, the alias entry will be returned. If one level scope is used no entries are returned. These conclusions are directly a result of following the definitions for the alias handling modes within RFC 2251.

The derefFindingBaseObj finally begins to introduce alias dereferencing. Before investigating the dynamics of this mode some discussion regarding expected alias semantics are germane. White pages often have an ou=People and an ou=Groups container for managing the users and their roles within an organization. Often an ou=Admin area is created specifically to manage the accounts of directory administrators. The probability of the need to point an alias entry within the ou=Admin entry to a person entry under the ou=People entry is highly likely. Presume the person entry has the Rdn uid=akarasulu and the alias referring to it under ou=Admin has the Rdn uid=alex. If a search with base scope on the alias entry is attempted using (objectClass=person) then one would expect to get the entry under ou=People back. This is common sense and the basis for alias semantics. Now if the filter was applied before the dereferencing took place, the alias entry would not have been selected since it is not an instance of objectClass person. Only if applied after the dereferencing phase of search can the filter return the expected result.

If the situation were reversed and dereferencing occurred after the application of the

search filter, the user would have to know that the entry searched is an alias. To return the entry, the following filter would have to be used instead: (objectClass=alias). The faults of dereferencing after filter application are becoming apparent. The entire point to aliasing is to not have to be cognizant of an object's objectclass as an alias or otherwise. Directory users should only have to set the alias handing mode to deal with all aliases within the directory. Dereferencing aliases before allows a single setting to handle the gambit of conditions. The user then only needs to know whether or not the possibility of encountering an alias exists. Dereferencing aliases after the application of a filter requires users to be aware of the exact objectclass of every entry composing the search filter. The latter is completely impractical and misses the implied semantics of an alias. Alias dereferencing must be applied before filter application to preserve the meaning and expected behavior of an alias.

To be absolutely clear, take the example a step further. A one level search at the ou=People base using the (objectClass=person) filter is a natural construct to use when listing the people within the organization. If dereferencing occurs after filter application all aliases would be missed even when the derefAlways mode is in effect. If dereferencing occurs before filter application then aliases are dereferenced to their targets and returned when derefAlways or derefInSearching is enabled. However when these modes are not in effect then as expected the aliases are ignored. Alias dereferencing must always occur before filter application otherwise there is no point to having aliases in the first place. With this clarification we can continue.

When the derefAliases parameter is set to derefFindingBaseObj, dereferencing only occurs when finding the search base. If the search scope is set to base scope then the entry the alias refers to is dereferenced and returned as a candidate for selection by the search filter. If the search scope is set to one level or subtree level scope, the search operates as if the base were replaced with the distinguished name of the entry pointed to by the alias. Hence the effective search base when dereferencing is the primary DN of the aliased entry. Depending on the structure of the DIT at the point of the entry aliased, search could return one or more results with sub tree scope and zero or more results with one level scope.

The remaining derefAlways mode is a composite mode based on the derefInSearching and the derefFindingBaseObj modes. It's behavior on search can be inferred by combining the behaviors of the composing two modes. The search base would be replaced with the primary DN of the entry referred to by the alias. The difference now is that another alias encountered during the search will have to expand the search if it increases the search scope.

The simple example helped deduce some very critical aspects of how aliases are to be handled. In conclusion, alias dereferencing must occur before the filter is applied upon a candidate to determine a candidate's eligibility.

Dereferencing In Search and Alias Chaining

Both X.501 and LDAP allow for alias chaining where one alias can refer to another. According to X.501 the dereferencing of an alias continues until a primary DN is resolved. "Primary," as opposed to alternate DN refers to the DN of a non-alias entry. The statement conotates an atomicity to alias dereferencing. Meaning dereferencing does not stop after the first alias is dereferenced if it points to another aliased entry. If dereferencing occurs every alias is dereferenced until a target primary distinguished name is found at the end of the chain.

Before debating the necessity for alias chains questions regarding cyclic alias chains must be raised. If alias chaining is allowed then there is the possibility of having one alias refer to another second alias which refers back to the first alias. This is a cycle with two links. The number of links in the cycle could be arbitrarily large and could throw alias dereferencing code into an infinite loop if cycle detection does not occur. Cyclic chaining is an absolute possibility. It emerges from the fact that alias inconsistency is an acceptable state in both X.500 and LDAP. According to specifications, aliases can be created arbitrarily without constraint to nonexistant entries that do not yet exist. The target of the first alias can later be be added as an aliase referring back to the first alias. This way dangerous cycles with two links can be created by malicious users to slow the DSA down to a crawl effectively conducting denial of service attacts.

Why then would cycles be allowed in the first place? When dereferencing aliases, the DSA must dereference completely until it reaches a non-alias target entry. Why not dereference aliases to other aliases in advance at the point of addition instead of introducing extra dereferencing steps during time critical search operations. Allowing users to chain aliases becomes moot, if the server bypasses chains on entry addition. Two different approaches can be used to remove the wasteful effects of alias chains.

One approach would be to allow for the addition of an alias, however the DSA would dereference the alias to a primary distinguished name and use that name instead of the user provided alias DN which causes the chain. The net affect is another direct alias to a non-alias entry rather than an alias chain. However, in this situation the DUA is unaware of the tampering or that alias dereferencing took place to bypass the chain. The tampering really is a form of optimization since the direct alias will perform faster than one getting to the target indirectly through a chain. Changes to user provided inputs without their notification is a dangerous endeavor altogether and we choose to avoid it.

An alternative approach would be to detect chaining, and reject the addition of the chained alias. An informative report directing the user to modify the request using a direct alias instead will alert the user to the problem. If dereferencing is performed then as a bonus the alias target can be recommended within the rejection message.

Both X.500 and LDAP allow for alias chaining it seems. Although seemingly pointless, X.500 aliases may have at some point had a reason for allowing alias chaining. Perhaps the ability of X.500 aliases to traverse servers and later LDAP's adoption of it without allowing the alias to point beyond a single server, lead to the loss of the need for alias chaining. Why this feature was allowed is irrelevant. Our goal is to enable aliases while preventing its misuse without detracting from its utility. Alias chaining will not be allowed and should not factor into backend designs for implementing aliases. For all practical purposes chaining can be ignored while designing the alias handling mechanism in search.

Cycles within the directory can still result without alias chaining. With sub tree scoped searches, encountering an alias referring back to a relative will cycle back down through the relative. The loop could be executed several times until time or size limits in the search are reached if specified at all. If aliases are not allowed to point to their relatives the problem goes away. We choose to reject aliases to parents and other relatives on addition to avoid unnecessary cycle detection code which consumes space and time during time critical search operations.

Common Sense Conclusions

We've concluded on constraining aliases in two ways. First we shall prevent alias chaining by rejecting chain formation on addition. Secondly we shall avoid wasteful search cycles by preventing aliases to parents and other relatives to the alias again on addition. These two measures to constrain aliases on addition reduce the complexity of implementations where alias dereferencing occurs. Furthermore these constraints reduce complexity while enabling aliases in a safe and sensible fashion.

Specialized Alias Indices in xldbm

Tim Howes published a paper on the xldbm database for backing X.500 and LDAP directories[1]. Two special indices are used in the recommended database design specifically for handling alias dereferencing. Briefly we quote the description of the dereferencing mechanism and the indices used within the paper. We will use it for the basis of our implementation discussions to follow:

> The key to handling aliases is to identify those aliases that point outside the scope of the search. If an alias does not "escape" the scope of the search, the entry it points to will be searched automatically (because it is contained within the scope, not because an alias points to it - why it gets searched is immaterial, as long as it does). Once such aliases are identified, the search is continued with the entries to which they point (either the entry itself for a one-level search, or the entry and all its descendants for a subtree search). Base object searches are easy to handle by examining the entry directly, and do not require any special indexing.
>
> To efficiently identify aliases that need searching, two new indexes are maintained, one for one-level scopes, one for subtree scopes. For each non-leaf entry, the one-level index contains an entry containing the entry IDs of alias children of the entry that do not point to other children (i.e., aliases that escape the onelevel scope). Similarly, the subtree index contains entry IDs of alias descendants of the entry that do not point to other descendants. During a search, the list of candidate entries is generated as before, and then the appropriate alias-scope index is consulted to determine if there are entries outside the scope that should be searched.

1 An X.500 and LDAP Database: Design and Implementation by Timothy A. Howes. A copy of this paper is available here.

Implementing An Alias Dereferencing Phase To Search

With slight modifications, we will follow the same techniques used by xldbm to manage alias dereferencing. The two specific indices for onelevel and subtree aliases respectively will be constructed and maintained. Another third optimizing index will be used to map the target dn of an alias to the id of alias. On alias addition index entries are added. On alias deletion index entries are removed. The use of these indices to implement the search phase for alias dereferencing is discussed below categorized by scope.

Handing Base Scope

Base scoped searches are simple and can be handled directly using point lookups in the search engine before returning the entry if at all. The four alias dereferencing modes can be reduced down to two modes where base dereferencing occurs across all scopes. Here's how the four modes reduce down to two modes:

Table I.

Dereference Base	***Do not Dereference Base***
derefFindingBaseObj	derefInSearching
derefAlways	neverDerefAliases

When an alias is encountered on the search base, the alias is dereferenced only if the dereferencing mode is derefAlways or derefFindingBaseObj. If dereferencing is to occur, the target entry pointed to by the alias is retrieved. The filter is applied against the entry and if accepted, it is returned in an enumeration with a single result. If the filter rejects the dereferenced entry then an empty enumeration is returned. If dereferencing does not occur, the filter is then applied to the alias entry in the same way it would have been applied to the target.

Handing Onelevel Scope

Like other scopes the onelevel scoped search must determine the effective search base first. To do so it uses the same technique outlined in Table I. to dereference the base. Once the effective base is known the scope node is built using this new effective base.

If derefAlways and derefInSearching are enabled a special onelevel scope constraint is assembled for filter evaluation. This constraint is composed of the disjunction of two assertions. The first assertion tests if the candidate is a child of the search base. The

second assertion tests if the candidate is contained as a value within the onelevel alias index for the effective base. If one or the other of these assertions evaluates to true the candidate is scope accepted and must be evaluated by the rest of the AST for return.

Handling Subtree Scope

The same process to deduce the effective base is applied as summarized in Table I. The truly interesting aspects of subtree scoped search appear within the derefInSearching and derefAlways modes.

Subtree scoped search where dereferencing is to occur while searching must factor in aliases that extend the search beyond the intended scope of the search. A special scope node on the filter AST is assembled using a disjunction to assert whether a candidate is within this wider alias extended scope. At the present moment the scope node is added to the search to constrain candidates on the basis of their being descendants of a single DN. If more than one DN is used this disjunction, the scope extension via aliases occurs naturally the search simply appears to progress as if it had multiple bases. The disjunction can easily be assembled by performing a lookup on the subtree alias index. The DN of each id returned is used to build an assertion to check if a candidate is a descendant of the DN. All these assertions are added to the disjunction of assertions so one must evaluate to true for the candidate to be within scope. The original effective base assertion is also added to the disjunction.

With the previously described alias addition constraints, search scope usually extends either laterally and downward beyond the effective search base when alias dereferencing is enabled in searching. Finding all the aliases that extend the scope is critical and a bit more involved. A single lookup on the subtree alias index returns a subset of the scope expanding aliases. What about the new aliases introduced by the expanded scope? By following the first set of aliases we may hence encounter more aliases along the way. We must account for these aliases but when does the process stop. The point is the process is recursive. The first lookup may return an initial set of aliases. Out of the initial set the process must be conducted again recursively until lookups return no more new alias.

Once the entire set of aliases are found an optimization could be performed to determine overlap. Overlapping alias DNs can then be removed from the set. If an alias extends the scope to ou=people,dc=example,dc=com and another alias extends the scope to ou=special,ou=people,dc=example,dc=com then there is no reason to add the latter to the disjunction in the scope node.

Once the optimal scope is determined using the aliased DNs the scope assertion node is assembled as a disjunction of these nodes and the original effective search base. The subtree search with aliases extending the scope reduces to a search on multiple non-related bases.

JNDI Considerations

With respect to JNDI, alias dereferencing is controled using the following environment property: "java.naming.ldap.derefAliases". The dereference links flag obtained via the getDerefLinkFlag() on SearchControls has nothing to do with alias dereferencing. This is clearly stated in the JNDI specification.

When dereferencing aliases, entries are returned out of the specified scope because of paths through aliases. When returning the names of entries via the NameClassPair getName() method returns a name which can be relative or absolute. There is no mention of the name being primary. A primary name is absolute: it is the real distinguished name. Any absolute distinguished name however need not be a primary name. An absolute path can traverse over an alias while dereferencing. A path through an alias will create an alternative name for potentially many entries in the set of SearchResults. The problem of determining which DN to use for returned entries going through aliases extends beyond just the JNDI. The problem is at the level of the protocol. JNDI just takes a precaution to parameterize whether or not the returned name is relative or absolute leaving it up to the provider to set the flag. The precaution perhaps comes as a consequence of the following complex problem.

Lets consider a situation where a subtree scoped search is conducted. Two separate aliases A and B below the search base point to the same aliased entry C. Without alias dereferencing C would never have been in scope and hence never returned. Both aliases A and B bring C into scope through separate pathways when dereferencing while searching is enabled. How does one determine which alternative path to return? When entry C is returned in the search result should we use the alternative distinguished name through A or through B. Or should we return C twice using both alternative distinguished names through A and through B? Two copies of C would have to be returned as two separate entries with different alternative distinguished names through both aliases A and B. If one were only returned which one would be returned: the path through A or the path through B? What algorithm would be used to make the returned distinguished name consistant across multiple DSAs? There are two possible behaviors that could be implemented: primary names could be returned with a single copy of C or alternative DNs could be returned with multiple copies of C which represents paths through aliases A and B. Generalize the the problem beyond just two pathways to N alternative paths to C and we have a big problem indeed.

When alias dereferencing is enabled and only primary distinguished names are returned with a single copy of C there is no way to determine how C is arrived at with more than one alternative path: through alias A or B. The path information is lost in the dereferencing. There is no way for the JNDI provider to return relative names to the client, it must default to returning absolute primary distinguished names. This is why JNDI is designed to report the relative flag of a NameClassPair. Only when alias

dereferencing while searching is disabled, can the provider compute the difference between the search base and the primary distinguished name to return a relative path. This is the only way the provider can be certain aliases have not tampered with the path between the base and the returned entry.

Let's take a look at the alternative approach. When multiple copies of C are returned, each with a different alternative distinguished name through aliases A and B, relative names can still be calculated when alias dereferencing is enabled. However the search must go to great lengths to determine both paths through aliases A and B to make that happen. One approach would be to determine if the returned entry is out of search scope first. Next all the paths to the entry must be determined through aliases A and B. The distinguished name and alias index can be used to compute the alternative relative distinguished name and determine if the entry was in scope to begin with. Although not impossible it takes a great deal more effort to get to this point than to simply return one copy using the primary distinguished name and setting names to non-relative.

At this point we must ask if search is meant to return a set of unique paths or if it is to return a set of unique entries that satisfy a filter constraint. Aliases seem to bring with them the devil throwing everything into a loop making distinguished names not so distinguished anymore. The bottom line is, the words 'unique' and 'distinguished' loose their meaning somewhat when aliases are enabled. We're left to semantics. The best approach would be either mechanism which can be toggled using a JNDI environment parameter.

Search presently is geared to return only one copy of an entry. It goes to some lengths to prevent duplicate returns. This is natural if the returned entry also has the same DN. Search should continue to returns primary distinguished names only. So whenever derefInSearching or derefAlways is set the JNDI provider only returns absolute distinguished names rather than relative names in the derefFindingBaseObj and neverDerefAliases modes. For the time being this can be the default operation. If we find that the other semantics apply where alternative distinguished names are to be returned with copies of the entry for each alternative name then we can add a new provider specific JNDI environment property named: enable.alternative.names. The property would probably be prefixed with the package name of the server side provider. When present and set to anything at all, even null the property should toggle on the use of alternative distinguished names rather than primary distinguished names. Then the provider can return one or more copies of entries using different relative alternative names from the base to the target. So when the property is set, the JNDI provider always returns relative alternative names regardless of the dereferencing mode used.

Another complex question arrises specifically when alternative names are enabled. How do you build the alternative name? The alternative DN could include the name components up to but not including the Rdn of the alias entry. The Rdn of the aliased

target object would be used instead to continue the name. The other way to construct the alternative name would be to include the name components up to and including the Rdn of the alias entry. The name would then exclude the target's Rdn and continue from there. Both the alias entry's Rdn and the target entry's Rdn could also be used adding an extra component. How would you build the relative alternative name when you have these three options? Our approach is to leave these questions unanswered until we are told that alternative names are required. If we have to return alternative names then we can apply our mechanism discussed earlier after answering these questions.

Effected Regions

As of this writing, changes to the database implementation for the add, modify, delete, and move operations would be required to manage one, sub and alias indices. The add, delete and modify operations are easily handled. The move operation is far more complex because it rearranges the parent child relationships within the directory. This makes the one and sub index entries inconsistent requiring their recalculation. Although move operations are not supported in LDAP they are supported in the database for administrative purposes.

A new system index on the ALIAS_ATTRIBUTE (aliasedObjectName or aliasedEntryName) is added to forward and reverse map normalized alias target DNs to the entry containing the target DN as its ALIAS_ATTRIBUTE. The move operation in particular would benefit greatest from the creation of another alias index separate from the one and sub . The index will probably never be that large but it will help speed up several operations and is well worth it: with a full database a fraction of a percent of the entries would be aliases at most. This index is used to avoid needlessly resusitating entries to test if they are aliases while conducting protocol operations.

Besides changes to add, modify, delete and move major changes will be required on the search engine to implement the dereferencing itself. These changes will be documented within the module as internal documentation.

Conclusions

We conclude by answering the original questions posed:

1. Does alias dereferencing occur before or after the application of a filter? ***Alias dereferencing is always conducted before filter application takes place. With our implementation scheme the notion of timing is irrelevant. The bottom line is: filters must be applied on primary DNs when dereferencing.***

2. Does dereferencing follow alias chains? In other words when dereferencing is enabled does the process continue dereferencing an alias chain until a non-alias entry is found? ***Alias chains are prevented at the point of addition by rejecting the add operation. This is a constraint applied to alias creation by the server to prevent potentially dangerous and wasteful alias constructs without loosing the benefits of aliases.***

3. How do we handle alias imposed cycles while conducting search? References to parents and other relatives could impose cycles. What mechanisms will be put in place to bound search preventing infinite loops within alias imposed cycles? ***Alias cycles are prevented at the point of addition by rejecting the add operation. This is yet another constraint applied to alias creation by the server to prevent potentially dangerous and wasteful alias constructs without loosing the benefits of aliases.***

4. How do we elegantly make JNDI environment parameters and other search modifying aspects of a JNDI Context available deep within the search engine? ***The mechanism used to extract and assemble these parameters are irrelevant to the manner in which they are used to conduct the search operation. However the JNDI environment parameters need for handling aliases are added to the filter AST as an extra constraint node called a scope node.***

5. When traversing aliases to return entries beyond the expected scope of the search base, which absolute path should be returned for entry results: the primary distinguished name or the alternative distinguished name through the alias? ***A JNDI environment parameter can later be used to modify the semantics of search operations to switch from using primary distinguished names in entry returns to using relative distinguished names in entry returns. The JNDI provider can then react accordingly to determine whether relative names are returned as opposed to absolute distinguished names via the isRelative marker.***

Besides answering these critical questions we have elaborated on the means to implement the added alias dereferencing phase of search.

Appendix A. Alias Error Codes

Because LDAP and X.500 allow aliases to exist in inconsistent states, we thought it would be appropriate to list the error codes associated with them below:

Error Code	***Code Name***	***Description***
33	aliasProblem	An alias has been dereferenced which names no object - a broken link where destination entry does not exist. [X511, Section 12.5]
36	aliasDereferencingProblem	An alias was encountered in a situation where it was not allowed. For example an add that creates an alias to another alias could throw this error or a delete operation going through an alias. If the client does not have read permission for the aliasedObjectName attribute and its value then the error should be returned. [X511, Section 7.11.1.1]