Chapter 5

Traversing the NDS Tree

Overview

This chapter describes the processes involved in resolving a name, or walking the NDS tree.

Most operations on an NDS object require the use of an Entry ID. An Entry ID is valid for an entry on a specific server. That is, it is local to each server and is not used as a global ID. Walking the tree involves finding the physical location of an entry and obtaining that entry's Entry ID. After obtaining an Entry ID, a client can get more information about the entry through a *Read* operation.

Sometimes a server holds a temporary Entry ID because it has a temporary copy of another object's name. A temporary Entry ID is valid for up to eight days.

There are three ways to obtain the Entry ID of an object:

- Indirect by enumeration through the "List" operation
- Direct through the use of the "Resolve Name" operation
- Indirect by enumeration through the "Search" operation

Each of these tree-walking operations will be discussed in detail in the sections that follow.

Note These topics are presented in the order listed above, which proceeds from the most elementary to the most complex operation.s

Concepts to Know

To understand this chapter you should be familiar with the following NDS concepts:

- Directory tree structure

Enumeration Through the List Operation

The "List" operation is the most elementary of the operations used to obtain the Entry ID for an object. This operation does not actually involve "walking the tree" because the information it returns is local to the current name server. Starting with the specified entry, the "List" operation enumerates/itemizes the directly contained entities, that is, it finds the immediate subordinates of the entry in the Directory tree. It then returns requested information about zero or more subordinate objects.

The client can specify three filters (fields in the request) to select the information of interest about each entry. This information indicates the subordinates to be returned.

- The *Name Filter* allows the client to request only objects that have a certain Relative Distinguished Name (RDN) or RDN pattern.
- The *Class Filter* allows the client to request only objects that belong to a certain object class.
- The *Time Filter* allows the client to request only objects modified after a certain time.

A client's rights on the subordinate object determine the information the operation returns. To list objects under a specific container, the client and server interact as follows:

Table 5.1
Listing Objects

Module	Action
Client	1. Sends a *List* request to the server.
Server	2. Checks that the client has Browse rights to each subordinate object. 3. Checks the subordinate entries for those that match the filters. 4. If steps 2 and 3 are successful, places the list of objects in the buffer.

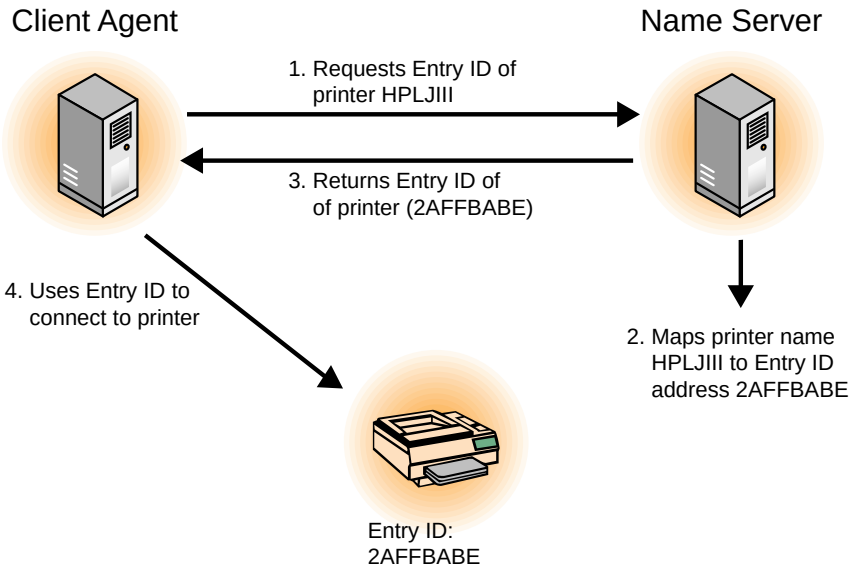
Tree Walking Using the Resolve Name Operation

NDS refers to all objects and resources with names that describe the object's relationship to the database hierarchy. In order for one object or resource to locate another object or resource, it must map, or resolve, this name to an object's Entry ID. The "Resolve Name" operation provides an object's Entry ID directly. The operation starts with the *Distinguished Name* of the object being sought and obtains the Entry ID, which can then be used to access a corresponding NDS entry record in the NDS entries file. The client indicates various characteristics of the desired Entry ID by the choice of request parameters. The flag field in the request packet also

limits the acceptable replies.

This process is often referred to as tree walking. Figure 5.1 presents a simplified view of this process.

Figure 5.1
Mapping an Entry ID to a Resource Name



Resolving names involves a group of functions that find an Entry ID associated with a Distinguished Name. These functions comprise three groups:

- Connection-based requests, which are used to find an Entry ID on a specific server
- Referral requests, which are made by a client process as it walks the distributed tree of entries to locate a specific entry
- Chained request, which are used by clients whose limited resources require that the server handle the referral process

These functions are described in the following sections. The descriptions will refer to three levels of software:

- Application, which provides the basic inputs
- Client, which is the code that controls connections to servers and makes requests through the connections
- Server, which is the code that responds to individual requests from clients.

Connection-Based Requests

These requests are used to locate an Entry ID on a specific server. For this operation, the application must provide both a Distinguished Name (DN) and a connection.

Connection-based requests include the following functions:

- Find Entry ID on a server. This operation finds an Entry ID for that Distinguished Name on the server specified by the connection. The specified server returns either an Entry ID or a NO_SUCH_ENTRY error. This error includes the offset of the name component at which resolution failed.
- Create Entry ID on a server. This performs the same operation as Find Entry ID but creates external references if necessary.

The client and server interact as follows in this type of name resolution:

Table 5.2
Resolving Names through Connection-Based Requests

Module	Action
Client	1. Packages the request and sends it to the server.
Server	2. Parses the Distinguished Name and starts from the most significant (root-most) name component. 3. Using the entry cache and the name hash index in the Name Base subsystem, steps down the name one RDN at a time until it finds the leaf-most entry or cannot find a component of the name. If it does not find a component of the name and the application has requested that an external reference be created, it resolves the name globally to determine that the entry exists. If the entry exists, creates an external reference for all components of the name that are missing on the local server. 4. Returns the Entry ID or an error.

Referral Requests

This type of function is used by the client to walk the Directory tree to locate a specific entry. For this function, the application must specify:

- A Distinguished Name (DN).
- The replica type.
- A dereference aliases flag (refers only to the leaf-most entry).
- A connection hint. (If no hint is given, the client picks a current connection. NDS 4.1 NLM clients always pick a connection to the local server. DOS clients always pick their “primary” connection. If the client has no connections to an NDS server, it returns a *NO_REFERRALS* error.)
- The originating client’s transport selection list if the request is

chained from another client. This *Client Transport* list specifies which transport types the client supports. (If no list is given here, the request uses the client's *Walker Transport* list.)

The client must also specify its *Walker Transport* selection list. This list specifies which transport types are supported by the client or server that is walking the tree. During the process, the client maintains the following:

- *Used Addresses* list. This is the set of addresses that have been tried. It prevents client loops in resolving names.
- *Current Referrals* list. This is a list of referrals to a particular partition. It is replaced by another list when a partition closer to the target entry is found. This list specifies the number of components (depth) in the current partition and whether the list was received with an *Up Referrals, Down Referrals,* or *Across Referrals* tag. When the *Current Referrals* list is replaced, it is sorted by transport cost weighted by the current connection cache contents, favoring addresses that the client is already connected to or servers nearest the client.
- The current Distinguished Name (DN), which may differ from the original DN if aliases have been dereferenced.

Referral requests include two functions:

- *Connect To Entry*. This function finds an acceptable copy of the entry and returns a connection and an Entry ID valid on that connection.
- *Get Entry Addresses.* This function returns a set of addresses where acceptable copies of the entry exist. If the request is chained, only this operation can be performed. 4.0x servers did not directly support this function. When an entry was located, chained request still returned only one address and one Entry ID. In 4.1 this process still returns only one address and ID if it encounters a 4.0x server when resolving a name. If this happens, the *FILTERED_BY_VERSION* flag is set in the reply.

To resolve names using referral requests:

Table 5.3
Resolving Names Using Referral Requests

Module	Action
Client	1. Packages the current Distinguished Name, flags, and both transport lists and sends them to the server on the current connection. If the *GetEntryAddresses* function is being performed, the request also contains a flag that requests that the server return addresses even it contains the entry.

Table 5.3
Resolving Names Using Referral Requests (Continued)

Module	Action
Server	2. Parses the Distinguished Name and starts from the most significant (root-most) name component. 3. Using the entry cache and the name hash index in the Name Base subsystem, steps down the name one RDN at a time until it finds the leaf-most entry. During the process it keeps track of the last partition root it found and the replica type; otherwise it does not matter whether it finds external references. The process terminates under the following conditions, which are described in Table 5.4 on page 7: • If a component is not found and no partition root has been seen, returns an *Up Referrals* tag if it holds a replica; otherwise it returns a *NO_REFERRALS* error. • If a component is not found and the last partition root was a subordinate reference, returns a *DownReferrals* tag. • If a component is not found and the last partition root was *not* a subordinate reference, returns a *NO_SUCH_ENTRY* error and a resolved offset. • If a non-leaf component of the name is an alias, returns an *Alias Referral* tag. • If the leaf-most component is an alias, returns an *Alias Referral* tag if dereference aliases has been requested; otherwise returns Entry IDs and addresses. • If the leaf-most component is an external reference, returns the same information as if the entry was not found. • If an entry is found, returns *Across Referrals,* Entry Addresses, and/or Entry ID(s) depending on the entry type, server transports, and input flags.

Table 5.4 describes the information that may be returned from a referral request.

Table 5.4
Referral Information Returned

Response	Description
Error	Two errors are important: • NO_SUCH_ENTRY indicates that the entry does not exist and includes the offset of the component at which resolution failed. • NO_REFERRALS indicates that the server in the current connection has no replicas and therefore has no suggestions for the client. If the client had no *Current Referrals* list, it must try another connection. If it does have a *Current Referrals* list, it continues processing the list. If the client has tried all its connections, this error is returned.

Table 5.4
Referral Information Returned (Continued)

Response	Description
Entry ID	This information is returned if the server has an acceptable copy of the entry. This returns the server's Entry ID and addresses if they are *ClientTransports.* A 4.1 server will not return this information if the client requested *Get Entry Addresses,* but a 4.0x server will. If the client receives this response while attempting *Get Entry Addresses,* the output flag *FILTERED_BY_VERSION* is set, and only addresses to that server are returned.
Entry Addresses	This response indicates that the server has a copy of the entry, but it may not be acceptable because of transport or replica type restrictions or because the address is unreachable. If the server does have an acceptable copy, its Entry ID and addresses are included in the response. 4.0x servers cannot return this response. • If the application has requested *Get Entry Addresses,* the response addresses are returned to the caller. • If the application has requested *Connect To Entry,* the client then sorts the addresses based on transport cost weighted by the contents of the connection cache. It will then connect to the first address. If the server response contains an Entry ID and it was for this address, the connection and Entry ID are returned; otherwise, the client must do *Find Entry ID on Server.* If an error is encountered, the client continues with the list.
Alias Referral	One of the components of the Distinguished Name is an alias. The server replaces that portion of the DN with the dereferenced alias and returns the new DN. The client replaces the current DN with the new DN, sets the *Current Referrals* list to empty, and begins the process again. **Note** If the client has requested *Get Entry Addresses,* servers with version 4.1 respond with *Entry Addresses.*

Table 5.4
Referral Information Returned (Continued)

Response	Description
Up Referrals	This response means that the server does not hold a copy of any component of the DN but holds a replica of entries in this tree. Addresses are excluded from the response if they are not WalkerTransports or if the address is unreachable. This response specifies the depth of the server's root-most entry and the remaining addresses. • If the depth of the root-most entry is less than the depth of the *Current Referrals* and if the *Current Referrals* were *Up Referrals,* the client replaces the *Current Referrals* with those in the response. • If the depth of the root-most entry is not less than that of the *Current Referrals,* the client connects to the next referral in the list that is a server and has an entry of less depth. The client then begins the process again. In version 4.1, the referring server does not know which addresses point to servers that contain the parent partition, but the server knows that one of these servers holds the parent. The client determines which one holds the parent partition by pinging them to find the depth.
Down Referrals	This indicates that the server holds a copy of one or more components of the DN, but the last component it found while stepping down the name was a subordinate reference. The server returns referrals to the partition represented by the subordinate reference. Addresses are excluded if they are not *Walker Transports,* if they are unreachable, or if they refer to another subordinate reference. The client replaces the *Current Referral* list with this list and begins the process again. **Note** A bug in version 4.1 (fixed in build 477) causes the response to include referrals to other subordinate references. Because 4.0x servers do not keep a *Used Address* list, they could cause endless loops between servers.
Across Referrals	This means that the server has a copy of the entry but it does not support *Client Transports* or the entry is not an acceptable replica type. This response returns referrals to the partition containing the entry. Addresses are excluded if they are not *Walker Transports,* if they are unreachable, or if they are to unacceptable replica types. The client replaces the *Current Referral* list with the response list and begins the process again. **Note** 4.1 servers do not make this response if the client has requested *Get Entry Addresses*. They will return *Entry Addresses*.

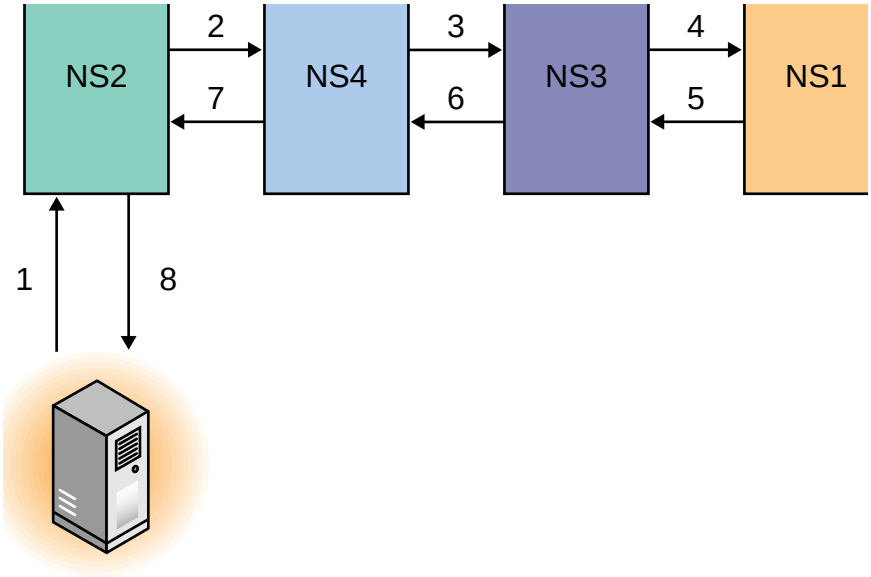
Chaining Requests

This request is used by clients whose limited resources require that the server handle the referral process. These functions require the following input from the application:

- A Distinguished Name (DN)
- A replica type
- A dereference aliases flag
- A connection hint (If no connection hint is given, the client picks a connection through which to make the chained request)

Chaining is a method in which one name server forwards the query to the next name server and so on, until the object is located. The information containing the object's Entry ID is then forwarded along the chain in the reverse order, as illustrated by the following figure.

Figure 5.1
Chaining Method of Name Resolution

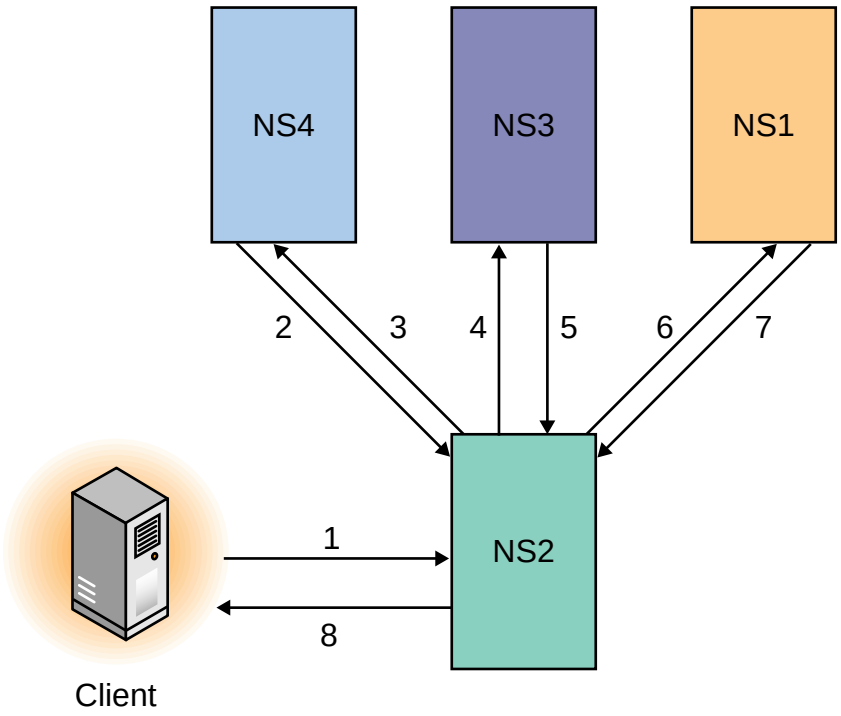


As shown above, in the name resolution chaining method, the client may request a service from Name Server 2, which attempts to resolve the name in its partitions, then if it cannot resolve the name, passes the request to Name Server 4 which repeats the process, then passes the request to Name Server 3, and so on. Once the name is found, or resolved, the request is forwarded back to the client in the reverse order of the chain. In this example, the name is resolved on Name Server 1.

"Resolve Name" is one NDS operation that may perform the chaining method, not just referrals. That is, the NDS server iteratively resolves chains of references to other servers on behalf of the client through the name server's "client agent." It does not always just return the references for the client to follow, however, it does so if that's what the client

requests. NDS uses a variation of the chaining method described above. This variation is illustrated in Figure 5.2.

Figure 5.2
Basic NDS Name Resolution



Basically, in NDS, a client will query a name server, which then is responsible, through its client agent, for querying other name servers until the name is resolved and the resource is found. In Figure 5.2, the client queries Name Server 2, its client agent then queries Name Servers 4, 3 and 1 in turn. When Name Server 1 returns the location of the service, Name Server 2 relays this information to the client.

Usually the client starts with a name it knows, such as the *Distinguished Name*, when navigating an NDS tree. Once the client has an Entry ID, it operates on an entry by sending requests to the NDS server were the Entry ID is valid.

To resolve a name using a chaining request:

Table 5.5
Resolving Names Using Chaining Requests

Module	Action
Client	1. Packages the inputs and sends them to the server on the current connection.

Table 5.5
Resolving Names Using Chaining Requests

Module	Action
Server	2. Checks its local database to see if it contains the specified entry. • If so, responds as described in the "Referral Requests" section. • If not, takes the client's transport selection list and makes the *Get Entry Addresses* request of the embedded NDS Client. 3. Forwards the responses back to the client. These responses are the same as those described in the "Referral Requests" section.

Enumeration Through the Search Operation

The *Search* operation is the more powerful enumeration operation that also does tree walking. That is, it combines capabilities of both the *List* and *Resolve Name* operations. It searches a region of the Directory for objects whose attributes satisfy specified criteria. Unwanted entries are weeded out by placing assertions on attribute values, or by a boolean combination of these assertions. The assertions must conform with the matching rules defined for the syntax of an attribute in the NDS schema. The search is applied not only to a containers directly contained entries, but extends to the entire subordinate subtree. The reply indicates when contact has to be made with other servers to continue the tree walk.

Search Request

This request searches a region of the Directory for objects whose attributes satisfy specified criteria. Fields in the request indicate the criteria for including entries in the response, and what information about each entry to include. The search starts with *Base* Entry ID, and proceeds downward in the tree to the extent indicated by *Scope*, or until examining *Number of Objects* or *Search* items. Searches can be filtered so that only entries meeting specific criteria are returned. The operation succeeds if the Base Entry ID is found, even if there are no entries to return in the reply. This implies that an unfiltered search of a single entry can give a result different from a *Read* operation on that entry. *Read* returns an error if none of the selected attributes exist in the entry, whereas *Search* does not return an error.

To search an entry:

Table 5.6
Searching for an Entry

Module	Action
Client	1. Sends a *Search Request* to the name server.

Table 5.6
Searching for an Entry

Module	Action
Server	2. Returns a *Search Reply* containing the results of the search requested in the *Search Request*.

Search Filter. A search filter indicates how the server screens entries for inclusion in the reply. This filter has several alternative forms.

item

True if and only if the <*assertion*> is true.

or

True if any of the nested <*search filter*>s is True. (False if <*number of items*> = 0.)

and

True unless any of the nested <*search filter*>s is False. (True if <*number of items*> = 0.)

not

True if and only if the nested <*search filters*> is False.

The *Search Filter* is a serialized binary structure in BNF (Backus-Naur Form) format, which is not a text string. Each *Search Filter* contains an assertion, which tests one of the entry's attributes. An assertion has one of three results when applied to an NDS entry: *True*, *False*, or *Undefined*. A result is *Undefined* in the following cases:

- The <*attribute name*> is unknown.
- The <*value*> does not conform to the syntax of the attribute indicated by <*attribute name*>.
- Access control restrictions prevent the assertion from being evaluated.
- The request indicates a comparison that is invalid for the attribute's syntax.

Several assertion types are available, indicated by the <*assertion*>'s tag. If an attribute has several values, the result is *True* if any one of the values satisfies the assertion (for those assertion types involving a value comparison).

A complete *Search Filter* BNF structure might appear as follows:

```
<search filter> ::= <item tag = 0>  <assertion>  |
    <or tag = 1>  <number of items = N> N*<search filter>  |
    <and tag = 2> <number of items = M> M*<search filter>  |
    <not tag = 3>  <search filter>
<assertion<  ::= <equal tag = 7>  <attribute name>  <value>  |
    <greater or equal tag = 8>  <attribute name>  <value>  |
    <less or equal tag = 9>  <attribute name>  <value>  |
    <approx equal tag = 10>  <attribute name>  <value>  |
    <present tag = 15>  <attribute name>  |
```

<rdn tag = 16> <value> |
<base class tag = 17> <value> |
<modification ge tag = 18> <attribute name> <value> |
<time ge tag = 19> <attribute name> <value> |

All these fields are four-byte aligned. The tag fields discriminate variant records. Each tag field is a uint32. Each tag's value is shown after its equal sign in the BNF description. The <*number of items = N*> field (a uint32) determines how many recursive <*search filter*>s follow. An <*attribute name*> is a Ustring. A <*value*> is an attribute value.

The assertion types are presented in Table 5.7 below.

Table 5.7
Search Filter Assertion Types

Tag No.	Filter/Assertion	Description
7	DS_SEARCH_EQUAL	The entry's attribute value matches the specified value.
8	DS_SEARCH_GREATER_ OR_EQUAL	The entry's attribute value is greater than or equal to the specified value.
9	DS_SEARCH_LESS_OR_ EQUAL	The entry's attribute value is less than or equal to the specified value.
10	DS_SEARCH_APPROX	The attribute value approximately matches the specified value. The syntax defines what approximate equals it supports.
15	DS_SEARCH_PRESENT	The specified attribute exists on that entry. That is, *True* if the named attribute exists for the entry.
16	DS_SEARCH_RDN	The specified value matches the entry's *Relative Distinguished Name*. *True* if the entry's *Relative Distinguished Name* matches the <value>.
17	DS_SEARCH_BASE_ CLASS	The entry belongs to the specified base class. *True* if the entry belongs to the class indicated by <value>.
18	DS_SEARCH_MODIFI-CATION_GE	The entry modification time is greater than or equal to the specified value.
19	DS_SEARCH_VALUE_ TIME_GE	The attribute creation time is greater than or equal to the specified value.
20	DS_SEARCH_ REFERENCES	Dereference Aliases and continue the search.