Open Directory Programming Guide

Networking > Mac OS X Server

2007-01-08

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings

Introduction

This manual describes the Open Directory application programming interface for Mac OS X v10.4. Open Directory is a directory service architecture whose programming interface provides a centralized way for applications and services to retrieve information stored in directories. The Open Directory architecture consists of the DirectoryServices daemon, which receives Open Directory client API calls and sends them to the appropriate Open Directory plug-in.

Organization of This Document

This book contains the following chapters:

- "Concepts" (page 9) describes the concepts used in the Open Directory architecture.
- "Working with Nodes" (page 25) explains how to use the Open Directory API interact with nodes.
- "Working with Records" (page 33) explains how to use the Open Directory API to interact with records.

See Also

Refer to the following reference document for Open Directory:

- *Open Directory Reference*

For more information about writing plug-ins for Open Directory, read:

- *Open Directory Plug-in Programming Guide*

Concepts

Open Directory is a directory service architecture whose programming interface provides a centralized way for applications and services to retrieve information stored in directories. Often, the information that is being sought is configuration information stored in a NetInfo database or in flat files, with each file having its own record format and field delimiters. Examples of configuration information include users and groups (`/etc/passwd` and `/etc/group`), and automount information (`/mounts`). Open Directory uses standard record types and attributes to describe configuration information so that Open Directory clients have no need to know the details of record formats and data encoding.

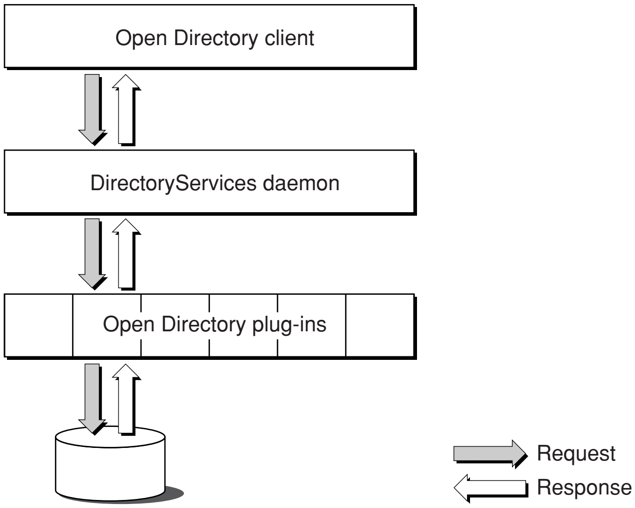
Earlier directory services, such as `lookupd` and NetInfo, took the first steps in providing access to configuration information but were limited in their capabilities. For example, `lookupd` provides support for reading but does not provide support for writing, and it does not provide support for authentication. Open Directory continues the evolution of directory services by providing expanded functionality. For example, Open Directory can write data as well as read it, and Open Directory includes support for a variety of authentication methods.

While providing support for `lookupd` and NetInfo, Open Directory's primary protocol is LDAP (supporting LDAPv2 and LDAPv3). As a result, Open Directory provides a way of accessing and sharing data using both LDAP and NetInfo. Open Directory provides seamless and automatic integration of Apple Computer's directory services and third-party directory services including Active Directory, iPlanet and OpenLDAP.

Open Directory Overview

Open Directory consists of the DirectoryService daemon and Open Directory plug-ins. Apple Computer provides Open Directory plug-ins for LDAPv3 (which supports LDAPv2), NetInfo, AppleTalk, SLP, Windows, and Bonjour. The AppleTalk, SLP, SMB, and Bonjour Open Directory plug-ins discover services that are available on the local network. In Mac OS X, `lookupd` resolves DNS queries through UNIX function calls like `gethostbyname()`. The Open Directory LDAP plug-in provides information about users and groups of users. For information on writing your own Open Directory plug-in, see the document *Open Directory Plug-ins*.

Figure 1-1 Flow of an Open Directory request



The Open Directory programming interface identifies the basic features that are common to many directory services and provides the functions necessary to support the development of high-quality applications that can work with a wide range of dissimilar directory services.

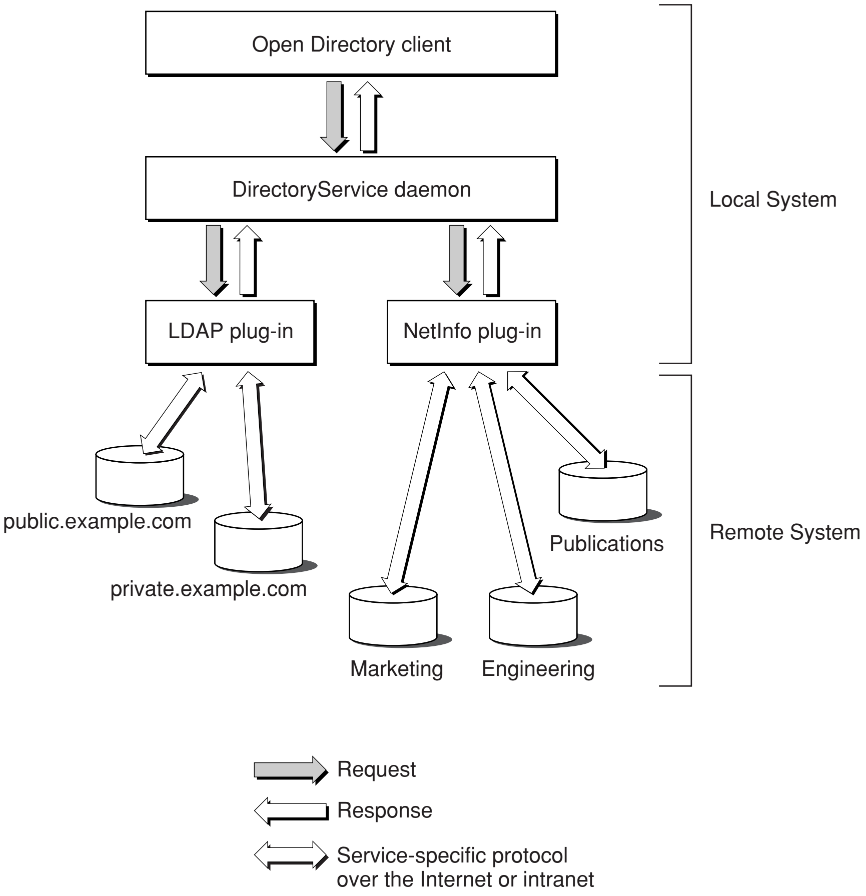
Nodes

From the viewpoint of Open Directory, a directory service is a collection of one or more nodes, where a node is a place that can be searched for information. Each NetInfo database in a hierarchy and each LDAP service configured by the Directory Access tool is a separate node. The following rules apply to nodes.

- A node is either the root of a directory or a child of another node.
- A registered node is a node that an Open Directory plug-in has registered with Open Directory or that an administrator has registered using the Directory Access tool.
- A node is a collection of records and child nodes.
- A record can belong only to one node.
- A record has a type and can be of no more than one type. Examples of record types include user records and group records.
- A record has a name and type that together make the record unique within its node. For example, there can't be two user records that have the name "admin," but there can be a user record named "admin" and a group record named "admin" within the same node.
- Nodes and records can contain any number of attributes.
- An attribute can have a value. Certain attributes can have more than one value.
- An attribute value is arbitrary data whose structure is unknown to the Open Directory programming interface. Open Directory clients are responsible for interpreting the value of any particular attribute.

Figure 1-2 (page 11) shows how Open Directory and the Open Directory LDAPv3 and NetInfo plug-ins might locate nodes over a network.

Figure 1-2 An Open Directory request over a network



Given the topology shown in Figure 1-2, the Open Directory function for listing registered nodes (`dsGetDirNodeList`) might return the following list:

```
/NetInfo/root/AppleMarketing
/NetInfo/root/AppleEngineering
/NetInfo/root/ApplePublications
/LDAPv3/private.example.com
/LDAPv3/public.example.com
```

The first part of the node name (`LDAPv3` and `NetInfo` in this example) is the name of the plug-in that handles that node.

Note: An Open Directory plug-in is not required to return information that conforms exactly to the information that the directory service maintains. A plug-in can generate information "on the fly." In addition, a plug-in may not return information about certain nodes; the plug-in's behavior in this respect can be configurable.

Search Policies and Search Nodes

A search policy defines the locations that are to be searched and the order in which those locations are searched in order to get certain kinds of information. The first location that a search policy defines must be the local NetInfo database.

Search nodes implement search policies, which are configured by administrators through the Directory Access application. Search nodes are easy for Open Directory applications to find and are guaranteed to always be available.

There are four search node types:

- *authentication search node* — Use this search node when you are looking for information that is needed to authenticate a user. Use the pattern matching constant `eDSAuthenticationSearchNodeName` to locate the authentication search node. Examples of applications that use the authentication search node include the login window and applications that set System Preferences. The authentication search node is also used indirectly by all UNIX commands that use `lookupd`.
- *contacts search node* — Use this search node when you are looking for contact information, such as an e-mail address, a telephone number, or a street address. Use the pattern matching constant `eDSContactsSearchNodeName` to locate the contacts search node. Mail.app and Address Book use the contacts search node to look up e-mail addresses and other types of contact information.
- *network search node* — Use this search node, which consolidates all of the nodes that are local to a machine for service discovery purposes, to find services on the local network. When third-party Open Directory plug-ins are loaded, they register their nodes with Open Directory so they can be found by the network search node. Use the pattern matching constant `eDSNetworkSearchNodeName` to locate the network search node.
- *locally hosted nodes* — Use a locally hosted node to find NetInfo domains stored on this machine (that is, the local domain plus any shared domains that are running locally). Locally hosted nodes are a class of nodes that have a special pattern match. Use the pattern matching constant `eDSLocalHostedNodes` to locate locally hosted nodes.

When an Open Directory client application uses a search node to search for information, it can request the fully qualified path for any record that matches a specific search criteria. As a result, Open Directory can perform extremely precise searches and a high degree of control over the type of information that is returned.

Record Types

Apple Computer has defined a series of standard record types. The standard record types include but are not limited to user records, group records, machine records, and printer records.

Providers of services can define their own record types (known as native record types) and are encouraged to publish information about them. Developers are encouraged to use Apple's standard record types whenever possible.

Table 1-1 Standard record types

Constant	Description
`kDSStdRecordTypeUsers`	Standard record for describing users
`kDSStdRecordTypeGroups`	Standard record for describing groups
`kDSStdRecordTypeMachines`	Standard record for describing machines
`kDSStdRecordTypeHosts`	Standard record for describing hosts
`kDSStdRecordTypePrinters`	Standard record for describing printers
`kDSStdRecordTypeNetworks`	Standard record for describing records in the networks file
`kDSStdRecordTypeServices`	Standard record for describing records in the services file
`kDSStdRecordTypeProtocols`	Standard record for describing records in the protocols file
`kDSStdRecordTypeRPC`	Standard record for describing RPC records
`kDSStdRecordTypePrintService`	Standard records for describing print services
`kDSStdRecordTypeConfig`	Standard record for describing configuration records
`kDSStdRecordTypeAFPServer`	Standard record for describing AFP servers
`kDSStdRecordTypeSMBServer`	Standard record for describing SMB servers
`kDSStdRecordTypeFTPServer`	Standard record for describing FTP servers
`kDSStdRecordTypeNFS`	Standard record for describing NFS servers
`kDSStdRecordTypeWebServer`	Standard record for describing Web servers
`kDSStdRecordTypeLDAPServer`	Standard record for describing LDAP servers
`kDSStdRecordTypeQTSServer`	Standard record for describing QuickTime Streaming servers
`kDSStdRecordTypeMounts`	Standard record for entries in the mount file
`kDSStdRecordTypeComputers`	Standard record for storing computer information
`kDSStdRecordTypeComputerLists`	Standard record for storing information about a list of computers
`kDSStdRecordTypePresetUsers`	Standard record for storing “preset” information used to create new user records
`kDSStdRecordTypePresetGroups`	Standard record for storing “preset” information used to create new group records
`kDSStdRecordTypePresetComputerLists`	Standard record for storing “preset” information used to create new computer list records

Constant	Description
kDSStdRecordTypePrintServiceUser	Standard record for storing quota usage for a user in the local node
kDSStdRecordTypeBootp	Standard record for storing bootp information
kDSStdRecordTypeNetDomains	Standard record for storing net domains
kDSStdRecordTypeEthernets	Standard record for storing Ethernets
kDSStdRecordTypeNetGroups	Standard record for storing net groups
kDSStdRecordTypeHostServices	Standard record for storing host services

Standard Attribute Types

Apple Computer has defined a series of standard attributes. Developers can define their own attributes (known as native attributes). An attribute can be required or optional. Each record type defines the attributes that it requires.

Open Directory clients are responsible for interpreting the value of any particular attribute. All configuration and discovery of information in the directory service can be accomplished by requesting the appropriate attribute value.

Table 1-2 lists some of the standard attribute types. See the Open Directory Reference section for the complete list of attributes. Constants for attributes that start with kDS1 represent attributes that can only have one value; constants for attributes that start with kDSN represent attributes that can have multiple values.

Table 1-2 Standard attribute types

Constant	Description
kDS1AttrPassword	Standard attribute for storing a password; commonly found in kDSStdRecordTypeUsers records
kDS1AttrUniqueID	Standard attribute for storing a unique ID; commonly found in kDSStdRecordTypeUsers records
kDS1AttrPrimaryGroupID	Standard attribute for storing a Primary Group ID; commonly found in kDSStdRecordTypeUsers and kDSStdRecordTypeGroups records
kDS1AttrComment	Standard attribute for storing a comment; commonly found in kDSStdRecordTypeGroup, kDSStdRecordTypeUsers, kDSStdRecordTypeComputers and other preset records
kDS1AttrUserShell	Standard attribute for storing the user's shell; commonly found in kDSStdRecordTypeUsers records
kDS1AttrPrintService-UserData	Standard attribute for print quota configuration or statistics; commonly found in kDStdRecordTypePrintServiceUser records

Constant	Description
kDS1AttrPort	Standard attribute for storing the port number at which a service is available; commonly found in kDSStdRecordTypeAFPServer, kDSStdRecordTypeFTPServer, kDSStdRecordTypeLDAPServer, kDSStdRecordTypeWebServer, and other service discovery records
kDSNAttrGroupMembership	Standard attribute for storing group memberships
kDSNAttrAuthentication-Authority	Standard attribute for storing authentication authorities; commonly found in records of type kDSStdRecordTypeUsers and kDSStdRecordType-Computers

Native Attribute Types

Developers can define their own attributes (known as native attributes). Open Directory maps the namespace of each directory system onto native types, while the standard types are the same across all Open Directory plug-ins.

Authentication

Open Directory for Mac OS X v10.2 supports authentication on a per-user basis whereby user records have an authentication authority attribute that specifies the type of authentication that is to be used to authenticate a particular user and all of the information required to use the specified authentication method, such as encoded password information.

Note: The information in this section is of interest to Open Directory clients that create user records or that want to change the authentication authority for a user. These clients must write the authentication authority attribute and may have to do a set password operation to have the change take effect. Open Directory clients that only do directory native authentication or that only change existing passwords do not need to interpret the authentication authority attribute because the Open Directory plug-ins handle the supported authentication authority attribute values.

This version of Mac OS X supports the following types of authentication:

- Basic, which supports Crypt password authentication. For more information, see "Basic Authentication" (page 16).
- Apple Password Server authentication, which uses a Mac OS X Password Server to perform authentication. For more information, see "Apple Password Server Authentication" (page 16).
- Shadow Hash authentication, which uses salted SHA-1 hashes. The hash type of can be configured using the authentication authority data. By default, NT and LAN Manager hashes are not stored in local files, but storing them in local files can be enabled. This is the default authentication for this version of Mac OS X. For more information, see "Shadow Hash Authentication" (page 18).
- Local Windows authentication, which is legacy subset of Shadow Hash authentication. For more information, see "Local Windows Hash Authentication" (page 18).
- Local Cached User authentication, which is appropriate for mobile home directories using directory-based authentication such as LDAP. For more information, see "Local Cached User Authentication" (page 19).

- Kerberos Version 5 authentication, which is used to authenticate users to Kerberos v5 systems. For more information, see "Kerberos Version 5 Authentication" (page 19).
- Disabled User authentication, which prevents any authentication from taking place. For more information, see "Disabled User Authentication" (page 20).

Note: For compatibility with previous versions of Mac OS X, user records that do not have an authentication authority attribute are authenticated using Basic password authentication.

User records contain an optional authentication authority attribute. The authentication authority attribute can have one or more values specifying how authentication and password changing should be conducted for that user. The format of this attribute is a semicolon-delimited string consisting of fields in the following order:

- *version* — a numeric value that identifies the structure of the attribute. This field is currently not used and usually is blank. This field may contain up to three 32-bit integer values (ASCII 0–9) separated by periods (.). If this field is empty or its value is 1, the version is consideration to be 1.0.0. If the second or the third field is empty; the version is interpreted as 0. Most client software will only needs to check the first digit of the version field. This field cannot contain a semi-colon (;) character.
- *authority tag* — a string value containing the authentication type for this user. Each authentication type defines the format of the authority data field and specifies how the authority data field is interpreted. The authority tag field is treated as a UTF8 string in which leading, embedded, and trailing spaces are significant. When compared with the list of known types of authentication, the comparison is case-insensitive. Open Directory clients that encounter an unrecognized type of authentication must treat the authentication attempt as a failure. This field cannot contain a semi-colon character.
- *authority data* — a field whose value depends on the type of authentication in the authority tag field. This field may be empty and is allowed to contain semi-colon characters.

Basic Authentication

An Open Directory client that encounters a user record containing the Basic authentication type should conduct authentication in a manner consistent with the authentication method supported by Mac OS X v10.0 and v10.1, which was crypt password authentication.

If the user record does not have an authentication authority attribute, the Open Directory client should use the Basic authentication type.

Here are some examples of authentication authority attributes that use the Basic authentication type:

```
;basic;
1.0.0;basic;
1;basic;
```

All three examples have the same result: authentication is conducted using crypt.

Apple Password Server Authentication

The Apple Password Server authentication type requires an Open Directory client to contact a Simple Authentication and Security Layer (SASL) password server at the network address stored in the authority data field. After contacting the Password Server, the Open Directory client can interrogate it to determine an appropriate network-based authentication method, such as CRAM-MD5, APOP, NT, LAN Manager, DHX, or Web-DAV Digest. Note that the Password Server's administrator may disable some authentication methods in accordance with local security policies.

The authority data field must contain two strings separated by a single colon (:) character. The first string begins with a SASL ID. The SASL ID is provided to the Password Server to identify who is attempting to authenticate. Apple's Password Server implementation uses a unique pseudo-random 128-bit number encoded as hex-ASCII assigned when the password was created to identify user passwords in its private password database. However, Open Directory clients should not assume that the first string will always be a fixed-size value or a simple number.

The SASL ID is followed by a comma (`,`) and a public key, which is used when the client challenges the Password Server before authentication begins to confirm that the Password Server is not being spoofed.

The second string is a network address consisting of two sub-strings separated by the slash (`/`) character. The first substring is optional and indicates the type of network address specified by the second substring. The second substring is the actual network address. If the first substring and the slash character are not specified, the second substring is assumed to be an IPv4 address.

If specified, there are three possible values for the first substring:

- `IPv4` — The client can expect the second substring to contain a standard 32-bit IPv4 network address in dotted decimal format.
- `IPv6` — The client can expect the second substring to contain a standard 64-bit IPv6 network address.
- `dns` — The client can expect the second substring to contain a fully qualified domain name representing the network location of the password server.

If the authority data field is missing or malformed, the entire authentication authority attribute value must be ignored and any attempt to authenticate using it must be failed.

In the following example of an authentication authority attribute for Mac OS X Password Server authentication, the version field is empty, so the version is assumed to 1.0.0. The SASL ID is `0x3d069e157be9c1bd0000000400000004`. The IP address is not preceded by `ipv6/`, so the IP address is assumed to be an IPv4 address.

```
;ApplePasswordServer;0x3d069e157be9c1bd0000000400000004,1024 35
162238334177531214968844629131367208019989492134080333699347018789801300072
13381175293354694885919239435422606359363041625643403628356164401829095281
75978839978526395971982754647985811845025859418619336892165981073840052570
657008816692626571374650047656107118967420361846115729915621101131109959977
4708458210473 root@pwserver.example.com:17.221.43.124
```

In the following example, the appearance of `dns` indicates that the network address in the second substring is a fully qualified domain name.

```
;ApplePasswordServer;0x3d069e157be9c1bd0000000400000004,1024 35
162238334177531214968844629131367208019989492134080333699347018789801300072
133811752933546948859192394354226063593630416256434036283561644018290952 81
759788399785263959719827546479858118450258594186193368921659810738400525 70
657008816692626571374650047656107118967420361846115729915621101131109959 97
4708458210473 root@pwserver.example.com:dns/sasl.password.example.com
```

Local Windows Hash Authentication

The Local Windows Hash authentication type was used on Mac OS X v10.2 in combination with Basic authentication, but its use is superseded by Shadow Hash authentication in this version of Mac OS X. With Local Windows Hash authentication, hashes for NT and LAN Manager authentication are stored in a local file that is readable only by root. The local file is updated to contain the proper hashes when the password changes.

This authentication type only supports the NT and LAN Manager authentication methods. In order to support other authentication methods, the Local Windows Hash authentication type is recommended for use in combination with the Basic authentication type. In this case, when a password is changed, both stored versions are updated.

Use of the Local Windows Hash authentication type only makes sense for non-network visible directories, such as the local NetInfo domain.

Here are some examples of properly formed authentication authority attribute values for Local Windows Hash authentication:

```
;LocalWindowsHash;
1.0.0;LocalWindowsHash;
1;LocalWindowsHash;
```

Shadow Hash Authentication

The Shadow Hash authentication type is the default password method for Mac OS X v10.3 and later. Starting with Mac OS X v10.4, Mac OS X desktop systems do not store NT and LAN Manager hashes by default, while Mac OS X Server systems store certain hashes by default. When storage of hashes is enabled, only a salted SHA-1 hash is stored. When a password is changed, all stored versions of the password are updated.

If the value of the authority data field is `BetterHashOnly`, only the NT hash is used.

Shadow Hash authentication supports cleartext authentication (used, for example, by `loginwindow`) as well as the NT and LAN Manager authentication methods. Starting with Mac OS X v10.4, ShadowHash authentication also supports the CRAM-MD5, DIGEST-MD5, and APOP authentication methods if the proper hashes are stored.

Here are some examples of properly formed authentication authority attribute values for Shadow Hash authentication:

```
;ShadowHash;
1.0.0;ShadowHash;
1;ShadowHash;
```

With Mac OS X v10.4,the authority data field can be customized with a list of hashes that are to be stored. Here is an example:

```
;ShadowHash;HASHLIST:<SALTED-SHA-1,SMB-NT,SMB-LAN-MANAGER>
```

Other valid hash types are `CRAM-MD5`, `RECOVERABLE`, and `SECURE`.

Local Cached User Authentication

Local Cached User authentication is used for mobile home directories. The authority data field must be present. Its format is

DS Nodename : *DS Recordname* : *DS GUID*

where the colon (:) character delimits the three individual strings. All three strings are required. The first string is any valid node name in UTF-8 format. The second string is any valid record name in UTF-8 format. The third string is any valid generated unique identifier (GUID) in UTF-8 format.

If the authority data field is absent or malformed, the authentication authority attribute value must be ignored and must result in failure to authenticate any client that attempts authentication using it. No other authentication type can be combined with this authentication type.

Here are some examples of properly formed authentication authority attribute values for Local Cached User authentication:

```
;LocalCachedUser;/LDAPv3/bh1234.example.com:bjensen:AFE453BF-284E-4BCE-
ADB2-206C2B169F41
1.0.0;LocalCachedUser;/LDAPv3/bh1234.example.com:bjensen:AFE453BF-284E-
4BCE-ADB2-206C2B169F41
1;LocalCashedUser;/LDAPv3/bh1234.example.com:bjensen:AFE453BF-284E-4BCE-
ADB2-206C2B169F41
```

Kerberos Version 5 Authentication

For Kerberos Version 5 authentication, the authority data field is formatted as follows:

[UID];[user principal (with realm)]; realm; [realm public key]

The optional 128-bit UID is encoded in the same way as for Apple Password Server authentication.

The optional user principal is the user principal for this user within the Kerberos system. If the user principal is not present, the user name and the realm are used to generate the principal name (*user@REALM*). This allows a fixed authentication authority value to be set up and applied to all user records in a database.

The required realm is the name of the Kerberos realm to which the user belongs.

The optional realm public key may be used to authenticate the KDC in a future release.

The following example yields a user principal of kerbdude@LDAP.EXAMPLE.COM:

```
;Kerberosv5;;Kerberosv5;0x3f71f7ed60eb4a19000003dd000003dd;kerbdude@LDAP.
EXAMPLE.COM;LDAP.EXAMPLE.COM;1024 35
148426325667675065063924525312889134704829593528054246269765042088452509
603776033113420195398827648618077455647972657589218029049259485673725023
256091629016867281927895944614676546798044528623395270269558999209123531
180552515499039496134710921013272317922619159540456184957773705432987195
53350982486690712830З root@ldap.example.com
```

Disabled User Authentication

The Disabled User authentication is used to indicate that an account has been disabled. The complete previous authentication attribute value is retained in the authority data field and is enclosed by left and right angle brackets. If the authority data field is absent, Basic authentication is assumed.

Here are some examples of properly formed authentication authority attribute values for Disabled User authentication:

```
;DisabledUser;;ShadowHash;
;DisabledUser;<;ShadowHash;>
```

The left (<) and right (>) angle brackets around the old authentication authority value are optional. Any tool that re-enables the user should check to see if the brackets are used and strip them when restoring the original authentication authority value.

Multiple Authentication Attribute Values

An authentication attribute can have multiple values. When changing a password, all authentication authority values are tried until the password is successfully changed or an error occurs. When verifying a password, the order of authentication authority values determines which value is used first. The first authentication authority that returns something other than `eDSAuthMethodNotSupported` is used. For example, Local Windows Hash returns `eDSAuthMethodNotSupported` for all methods other than the change and set methods, cleartext authentication, and the SMB LM and SMB NT authentication methods.

Authentication Versus Authorization

It is important to distinguish the difference between authentication and authorization:

- *authentication* — a process that uses a piece of information provided by the user (typically a password) to verify the identity of that user
- *authorization* — the determination of whether a user has permission to access a particular set of information

Open Directory allows an Open Directory client to use any method to authenticate a user. Open Directory does not provide any facility for determining whether a user is authorized to access any particular set of information. Moreover, Open Directory does not provide an authorization model. Instead, Open Directory clients are responsible for granting or denying a user access to a particular set of information based on the user's authenticated identity.

When developing an authorization model, Open Directory clients must consider the following:

- the authorization information to store
- where and how to store authorization information
- which applications can see, create, or modify authorization information
- who is authorized to see and change authorization information

Often authorization is based on membership in a particular group. Many directory services store authorization information in the directory service itself. These directory services use the identity that is currently being used to access the directory service to determine whether to grant access to this information.

Other directory services store authorization information outside of the service. By providing an interface between clients of directory services and the directory services themselves, authorization information that is stored outside of the directory service can be shared. For example, you could design a system that controls authorization based on a common token (such as a user entry in a common directory) so that when an administrator creates, deletes, or modifies a token, all services use that same token for authorization. Accordingly, the Open Directory `dsDoDirNodeAuth` function's `inDirNodeAuthOnlyFlag` parameter tells the plug-in whether the proof of identity process is being used to establish access to the foreign directory or whether the proof of identity process is being used only to verify a password.

Here are some ways that could be used to establish an identity that is authorized to access a foreign directory:

- have the Open Directory client use a preference to establish a "configuration" identity that can access a given directory
- configure the Open Directory plug-in with identity information

It will be necessary for the administrator of the foreign directory to set up, provide, or configure an identity with sufficient access so that a service or plug-in can access or modify all of the necessary information in the foreign directory. Allowing anonymous read access is an alternative to storing a username and password on each client machine. Whether this is possible depends on the directory server in use.

Mac OS X v10.4 optionally uses trusted directory binding, which establishes a trust relationship between a client machine and the directory server.

Directory Native Authentication

Open Directory supports a mechanism that frees Open Directory clients from having to provide specific information about a particular authentication method. This mechanism is called directory native authentication.

When using directory native authentication to authenticate a user to a node, the Open Directory client passes to the Open Directory plug-in the user's name, password, and an optional specification that cleartext is not an acceptable authentication method.

Upon receipt of the authentication request, the Open Directory plug-in determines the appropriate authentication method based on its configuration (if the plug-in is configurable) or on authentication methods the plug-in has been coded to handle. When the authentication is successful, the Open Directory client receives the authentication type that the plug-in used.

When cleartext is the only available authentication method, the plug-in would deny the authentication if the Open Directory client specifies that cleartext authentication is unacceptable.

Directory Proxy

In previous versions of Mac OS X, an application could only open an Open Directory session with the local DirectoryService daemon. The Open Directory function `dsOpenDirService` is responsible for opening local Open Directory sessions and returning an Open Directory reference that the application passes to subsequent calls of Open Directory functions.

With Mac OS X v10.2 and later, applications can open an authenticated and encrypted Open Directory session with a remote DirectoryService daemon over TCP/IP. The Open Directory function `dsOpenDirServiceProxy` is responsible for opening remote Open Directory sessions. As with `dsOpenDirService`, `dsOpenDirServiceProxy` returns a Open Directory reference that the application passes to any Open

Directory function that requires such a reference. Once a remote Open Directory session is successfully opened, Open Directory automatically sends all calls to Open Directory functions that use the remote directory reference to the DirectoryService daemon over the encrypted TCP/IP connection. Other than calling `dsOpenDirServiceProxy`, there is nothing the application has to do in order for its actions to take effect on the remote system.

Open Directory, lookupd, and NetInfo

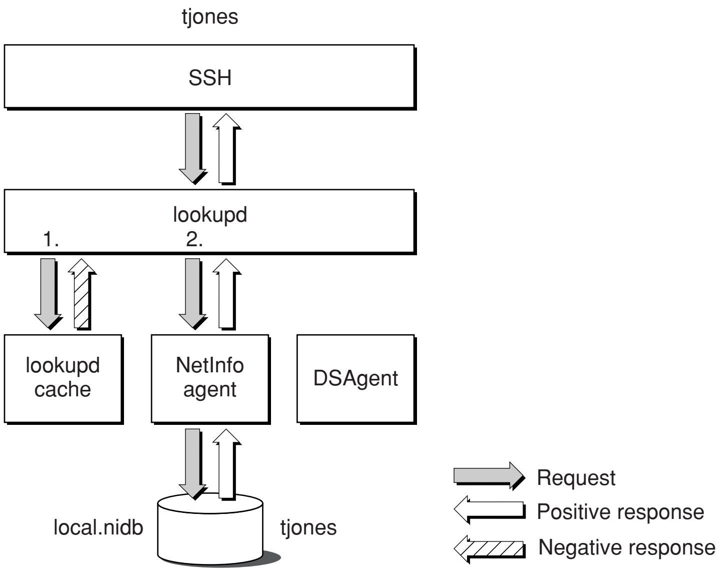
The process `lookupd` can be used to look up various categories of data, typically configuration information, such as users, groups, networks, services, protocols, remote procedure calls, (RPC), mounts, printers, boot parameters, aliases and netgroups, but also DNS information. This section describes how Open Directory works with `lookupd` and NetInfo.

The `lookupd` process works through the use of agents, with each agent designed to obtain a particular type of information. For example, there is an agent for resolving DNS queries, an agent for querying the local NetInfo database and its parents, and agent for querying LDAP directories, an agent for querying the Network Information System (NIS), and an agent for querying UNIX flat files in the file system. There is also an agent for querying the `lookupd` cache, in which `lookupd` stores information that has recently been looked up. The agents and the order in which the agents are used to look up information are configured through command-line utilities. By default, the `lookupd` cache is searched first, followed by NetInfo, and then Open Directory.

Like `lookupd`, Open Directory has a flexible architecture, provided by Open Directory plug-ins, for finding a wide variety of information. Unlike `lookupd`, Open Directory is configured by the graphical tool, Directory Access. While UNIX-based programs use `lookupd` to get information from external sources, Mac OS X applications use Open Directory.

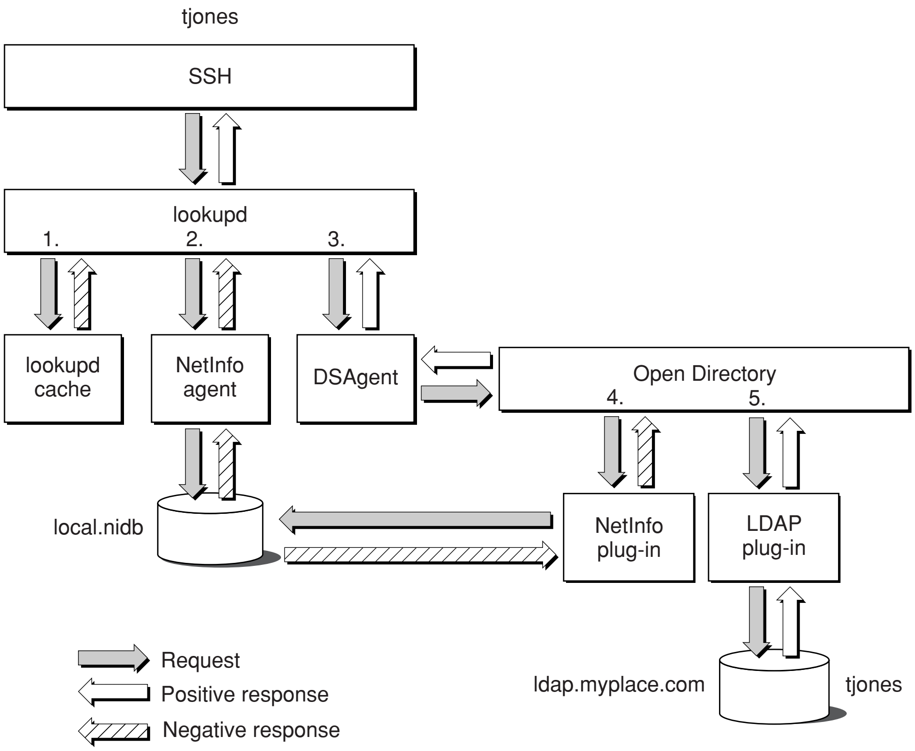
The `lookupd` process can be configured to work with Open Directory through the use of the DSAgent. When a process that uses `lookupd` requires a piece of information, `lookupd` searches its cache and any other configured agents. If no results are found, the DSAgent queries Open Directory. Take, for example, the searching that occurs when “tjones” logs in using SSH, as shown in Figure 1-3 (page 23). In this example, the `lookupd` cache is searched first. The user “tjones” is not in the cache, so `lookupd` queries NetInfo, which finds “tjones” in the local NetInfo database (`local.nidb`). Open Directory does not participate in this particular login process.

Figure 1-3 `lookupd and` NetInfo interaction when using SSH



In Figure 1-3, the user's record is located on the local system. Figure 1-4 shows what happens when the user's record is located on a remote system.

Figure 1-4 `lookupd`, NetInfo, and Open Directory interaction when using SSH



In this example, lookupd queries its local cache and NetInfo, and gets negative responses — “tjones” could not be found in either location. Now, lookupd tells its DSAgent to query Open Directory. Searching the local NetInfo database is the first step in every Open Directory search, so Open Directory tells its NetInfo plug-in to search the local NetInfo database. Again, “tjones” is not found. In this case, Open Directory is configured to conduct LDAP searches next, so Open Directory tells its LDAP plug-in to search for “tjones” in the remote LDAP servers that it knows about. This time, “tjones” is found in ldap.myplace.com.

In summary, lookupd calls Open Directory when its local cache and NetInfo cannot find an answer. Whether Open Directory is called by lookupd or called by another application, Open Directory always searches its local NetInfo database first and then conducts other searches using whatever search technology it has been configured to use. Most of the time, that search technology is LDAP.

Directory Service Command Line Utility

The directory service command line utility, dscl, operates on Open Directory nodes. It is similar to the nicl utility, which only operates on NetInfo nodes. The dscl utility’s options allow you to create, read, and manage Open Directory data. For more information on the dscl utility, see the man page for dscl .

Debugging

You must be root to enter the DirectoryService killall commands that enable and disable debug logging. The following command, run by root, enables debug logging if debug logging is currently off and disables debug logging if debug logging is currently on:

```
killall -USR1 DirectoryService
```

Debugging output is sent to /Library/Logs/DirectoryService/DirectoryService.debug.log. Debugging output includes input to Open Directory API calls, results, and timing, plus any debug information output by Open Directory plug-ins.

The following command, run by root, enables debug logging to /var/log/system.log if debug logging is currently off and disables debug logging if debug logging is currently on:

```
killall -USR2 DirectoryService
```

When debug logging is enabled by -USR2, debug output includes API call results and timing. Debug logging enabled by -USR2 is turned off automatically after five minutes.

Working with Nodes

This chapter provides sample code that shows how to work with nodes. Finding a specific node, opening a session with the node, and authenticating a user to the node are fundamental Open Directory tasks.

Listing Registered Nodes

The sample code in Listing 2-1 demonstrates how to get a list of all registered nodes. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its own `ListNodes` routine.

The `ListNodes` routine calls `dsGetDirNodeCount` to get the number of registered nodes. If the number of registered nodes is not zero, `ListNodes` calls `dsDataBufferAllocate` to allocate a data buffer and then calls `dsGetDirNodeList` to fill the buffer with the list of registered node names. The `ListNodes` routine then calls `dsDataListAllocate` to allocate a data list and `dsGetDirNodeName` to fill the data list with registered node names from the data buffer. The `ListNodes` routine then calls its own `PrintNodeName` routine to print the node names and passes to it a pointer to the data list.

The `PrintNodeName` routine calls `dsGetPathFromList` to get a node name from the data list and prints the name.

When the `PrintNodeName` routine returns, the `ListNodes` routine cleans up by calling `dsDataListDeallocate` and `free()` to deallocate the data list.

Listing 2-1 Listing registered nodes

```
tDirReference gDirRef = NULL;
void main ( )
{
    long dirStatus = eDSNoErr;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        ListNodes();
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}
void ListNodes ( void ) {
    bool done = false;
    long dirStatus = eDSNoErr;
    unsigned long index = 0;
    unsigned long nodeCount = 0;
    unsigned long bufferCount = 0;
    tDataBufferPtr dataBuffer = NULL;
    tDataListPtr nodeName = NULL;
    tContextData context = NULL;

    dirStatus = dsGetDirNodeCount( gDirRef, &nodeCount );
    printf( "Registered node count is: %lu\n", nodeCount  );
    if ( (dirStatus == eDSNoErr) && (nodeCount != 0) )
    {
        //Allocate a 32k buffer.
        dataBuffer = dsDataBufferAllocate( gDirRef, 32 * 1024 );
        if ( dataBuffer != NULL )
        {
            while ( (dirStatus == eDSNoErr) && (done ==  false) )
            {
              dirStatus = dsGetDirNodeList( gDirRef, dataBuffer,  &bufferCount,
  &context );
                if ( dirStatus == eDSNoErr )
                {
                    for ( index = 1; index <= bufferCount; index++  )
                    {
                        dirStatus = dsGetDirNodeName( gDirRef, dataBuffer,
index,  &nodeName );
                        if ( dirStatus == eDSNoErr )
                        {
                            printf( "#%4ld ", index );
                            PrintNodeName( nodeName );
                            //Deallocate the data list containing  the node
name.
                           dirStatus = dsDataListDeallocate( gDirRef,  nodeName
 );
                            free(nodeName);
                        }
                        else
                        {
                           printf("dsGetDirNodeName error  = %ld\n", dirStatus
 );
                        }
                    }
                }
                done = (context == NULL);
            }
            if (context != NULL)
            {
                dsReleaseContinueData( gDirRef, context );
            }
            dsDataBufferDeAllocate( gDirRef, dataBuffer );
            dataBuffer = NULL;
        }
    }
} // ListNodes

void PrintNodeName ( tDataListPtr inNode ) {
    char* pPath;
    pPath = dsGetPathFromList( gDirRef, inNode, "/" );
    printf( "%s\n", pPath );
    if ( pPath != NULL )
    {
        free( pPath );
        pPath = NULL;
    }
```

```
} // PrintNodeName
```

Finding a Node

The sample code in Listing 2-2 (page 27) demonstrates how to find the node for a specific pathname. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its own FindNodes routine and passes to it the pathname for the node that is to be found (/NetInfo/root).

The FindNodes routine calls dsBuildFromPath to build a data list for the pathname and calls dsDataBufferAllocate to allocate a data buffer in which to store the result of calling dsFindDirNodes. The routine then calls dsFindDirNodes to find the node whose name matches the specified pathname.

Then the FindNodes routine calls dsDataListAllocate to allocate a data list and provides that data list as a parameter when it calls dsGetDirNodeName. The dsGetDirNodeName function copies the node name from the data buffer filled in by dsFindDirNodes to the data list. Then the FindNodes routine calls its PrintNodeName routine to print the node name that was found. The PrintNodeName routine is described in the section Listing 2-1 (page 25).

When the PrintNodeName routine returns, the FindNodes routine cleans up by calling dsDataListDeallocate and free() to deallocate the data list.

Listing 2-2 Finding the node for a pathname

```
void main ( )
{
    long dirStatus = eDSNoErr;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        FindNodes("/NetInfo/root");
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}

void FindNodes ( char* inNodePath ){
    bool done = false;
    long dirStatus = eDSNoErr;
    unsigned long index = 0;
    unsigned long bufferCount = 0;
    tDataBufferPtr dataBuffer = NULL;
    tDataListPtr nodeName = NULL;
    tContextData context = NULL;
    nodeName = dsBuildFromPath( gDirRef, inNodePath, "/");
    if ( nodeName != NULL )
    {
        //Allocate a 32k buffer.
        dataBuffer = dsDataBufferAllocate( gDirRef, 32 * 1024 );
        if ( dataBuffer != NULL )
        {
            while ( (dirStatus == eDSNoErr) && (done ==  false) )
            {
```

```
        dirStatus = dsFindDirNodes( gDirRef, dataBuffer,  nodeName,
eDSContains, &bufferCount, &context );
        if ( dirStatus == eDSNoErr )
        {
          for ( index = 1; index <= bufferCount; index++  )
          {
            dirStatus = dsGetDirNodeName( gDirRef, dataBuffer,
index,  &nodeName );
            if ( dirStatus == eDSNoErr )
            {
              printf( "#%4ld ", index );
              PrintNodeName( nodeName );
              //Deallocate the nodes.
             dirStatus = dsDataListDeallocate( gDirRef,  nodeName
 );
              free(nodeName);
            }
            else
            {
              printf("dsGetDirNodeName error  = %ld\n", dirStatus
 );
            }
          }
        }
        done = (context == NULL);
      }
      dirStatus = dsDataBufferDeAllocate( gDirRef, dataBuffer  );
      dataBuffer = NULL;
    }
  }
} // FindNodes
```

Opening and Closing a Node

The sample code in Listing 2-3 demonstrates how to open and close a node. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its `MyOpenDirNode` routine and passes to it the address of the node reference (`nodeRef`) that it has allocated.

The `MyOpenDirNode` routine prints a prompt that solicits the entry of a node to open and calls `scanf()` to get the node's name. It then calls `dsBuildFromPath` to build a data list containing the node name that was entered. It specifies the slash (/) character as the path delimiter. Then the `MyOpenDirNode` routine calls `dsOpenDirNode` to open the node. If the node can be opened, `dsOpenDirNode` stores in the node reference parameter a node reference that the application can use in subsequent calls to Open Directory functions that operate on the open node.

The `MyOpenDirNode` routine cleans up by calling `dsCloseDirNode` to close the node that was opened.

Listing 2-3 Opening a node

```
void main ( )
{
  long dirStatus = eDSNoErr;
  tDirNodeReference nodeRef = NULL;
  dirStatus = dsOpenDirService( &gDirRef );
```

```
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            dsCloseDirNode( nodeRef );
        }
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}

long MyOpenDirNode ( tDirNodeReference *outNodeRef )
{
    long dirStatus = eDSNoErr;
    char nodeName[ 256 ] = "\0";
    tDataListPtr nodePath = NULL;
    printf( "Open Node : " );
    fflush( stdout );
    scanf( "%s", nodeName );
    printf( "Opening: %s.\n", nodeName );
    nodePath = dsBuildFromPath( gDirRef, nodeName, "/"  );
    if ( nodePath != NULL )
    {
        dirStatus = dsOpenDirNode( gDirRef, nodePath, outNodeRef  );
        if ( dirStatus == eDSNoErr )
        {
            printf( "Open succeeded. Node Reference = %lu\n",  (unsigned
long)outNodeRef );
        }
        else
        {
            printf( "Open node failed. Err = %ld\n", dirStatus  );
        }
    }
    dsDataListDeallocate( gDirRef, nodePath );
    free( nodePath );
    return( dirStatus );
} // MyOpenDirNode
```

Authenticating a User to a Node

To authenticate itself to the Open Directory for the purposes of reading, writing, or making changes to a node, an Open Directory client application calls `dsDoDirNodeAuth`. The `dsDoDirNodeAuth` function handles authentication methods that use one or more steps to complete the authentication process.

To determine the authentication methods that a node supports, call `dsGetDirNodeInfo` for the node and request the `kDSNAttrAuthMethod` attribute. For any particular user, some methods may not be supported. The available authentication methods depend on the authentication authority of the user that is being authenticated.

If the authentication methods that an authentication authority implements are known, the authentication authority may be used to deduce those authentication methods that are available for a user. Note, however, that it is possible to disable hash storage on a per-user basis, which has the effect of disabling some authentication methods that would otherwise be available.

Directory Native Authentication

The sample code Listing 2-4 (page 30) demonstrates directory native authentication. In the sample code, the `inDirNodeRef` parameter contains a node reference for the node, `inUserName` parameter contains the user name that is to be authenticated to the node, the `inUserPassword` contains the password in cleartext that is to be used to authenticate the user name.

Listing 2-4 Authenticating using directory native authentication

```
Bool DoNodeNativeAuthentication ( const tDirReference inDirRef,
                            const tDirNodeReference inDirNodeRef,
                            const char *inUserName,
                            const char *inUserPassword )
{
    //  Native authentication is a one step authentication scheme.
    //  Step 1
    //      Send: <length><recordname>
    //            <length><cleartextpassword>
    //   Receive: success or failure.

    tDataNodePtr anAuthType2Use = NULL;
    tDataBufferPtr anAuthDataBuf = NULL;
    tDataBufferPtr aAuthRespBuf = NULL;
    tDirStatus aDirErr = eDSNoErr;
    tContextData aContinueData = NULL;
    long aDataBufSize = 0;
    long aTempLength = 0;
    long aCurLength = 0;
    bool aResult = false;
    // First, specify the type of authentication.
    anAuthType2Use =
dsDataNodeAllocateString(inDirRef,kDSStdAuthNodeNativeClearTextOK);
    // The following is an optional method of authentication that  allows the
    // plug-in to choose the authentication method, but the client  can
    // “restrict” the authentication request to be "secure"  and not use
    // cleartext. Both authentication methods take the same buffer  arguments.
    /* anAuthType2Use = dsDataNodeAllocate(inDirRef,
kDSStdAuthNodeNativeNoClearText); */
    aDataBufSize += sizeof(long) + ::strlen(inUserName);
    aDataBufSize += sizeof(long) + ::strlen(inUserPassword);
    anAuthDataBuf = dsDataBufferAllocate(inDirRef, aDataBufSize);
    aAuthRespBuf = dsDataBufferAllocate(inDirRef, 512); // For the  response.
    // Put all of the authentication arguments into the data buffer.
    aTempLength = ::strlen(inUserName);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength,
sizeof(long));
    aCurLength += sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserName,  aTempLength);
    aCurLength += aTempLength;
    aTempLength = ::strlen(inUserPassword);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), &aTempLength,
sizeof(long));
    aCurLength += sizeof(long);
    ::memcpy(&(anAuthDataBuf->fBufferData[aCurLength]), inUserPassword,
aTempLength);
    aCurLength += aTempLength;

    anAuthBuff->fBufferLength = aDataBufSize;
    aDirErr = dsDoDirNodeAuth(inDirNodeRef, anAuthType2Use, true,  anAuthDataBuf,
  aAuthRespBuf, &aContinueData);
    switch(aDirErr)
    {
        case eDSNoErr:
            aResult = true;
            break;
        default:
            // If any other error, assume the name or password is  bad.
            aResult = false;
            break;
    }
    // Clean up allocations.
    aDirErr = dsDataBufferDeAllocate(inDirRef, anAuthDataBuf);
    anAuthDataBuf = NULL;
    // Don't need to keep the response.
    aDirErr = dsDataBufferDeAllocate(inDirRef, aAuthRespBuf);
    aAuthRespBuf = NULL;
    // Don't need the authentication type value. Build a new one  if needed
    // later.
    aDirErr = dsDataNodeDeAllocate(inDirRef, anAuthType2Use);
    anAuthType2Use = NULL;
    // Return the result of the authentication.
    return (aResult);
}
```

Working with Records

Using records is an essential part of using Open Directory. This chapter covers basic examples of how to interact with records.

Listing Records

The sample code in Listing 3-1 (page 34) demonstrates how to list all records in a node. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its `MyOpenDirNode` routine and passes to it the address of the node reference (`nodeRef`) that it has allocated. The `MyOpenDirNode` routine is described in the section "Opening and Closing a Node" (page 28).

The sample code then calls its `ListRecord` routine and passes to it the node reference (`nodeRef`) obtained by calling its `MyOpenDirNode` routine.

The `GetRecordList` routine calls `dsDataBufferAllocate` to allocate a buffer for storing the results of calling `dsGetRecordList`. It also builds three data lists for determining which records to include in the list: one for record names (`recNames`), one for record types (`recTypes`), and one for attribute types (`attrTypes`). It sets `recName` to `kDSRecordsAll` (to include all record of any name in the list), `recTypes` to `kDSStdRecordTypeUsers` to include standard user type records in the list, and sets `attrTypes` to `kDSAttributesAll` to include all attributes of the records in the list.

Then the `GetRecordList` routine calls `dsGetRecordList` to fill the data buffer with matching records and their attributes. By specifying `eDSExact` as the fourth parameter (`inPatternMatchType`), `dsGetRecordList` gets records that exactly match the requirements specified by the `recNames` parameter. By specifying `false` as the seventh parameter (`inAttributeInfoOnly`), `dsGetRecordList` gets attribute values as well as attribute information.

The `dsGetRecordList` function returns in its `recCount` parameter the count of the number of records returned in its `dataBuffer` parameter. Using `recCount` as a limit, the `GetRecordList` routine walks through the data buffer calling `dsGetRecordEntry` to get the record entry information for each record.

The record entry information contains an attribute count that `GetRecordList` uses as a limit to walk through the record's attributes, calling `dsGetAttributeEntry` for each attribute. For each attribute entry, `GetRecordList` calls `dsGetAttributeValue` and prints the attribute's value and its attribute ID.

The `GetRecordList` routine continues printing attribute values and attribute IDs until `dsGetRecordList` returns a `context` parameter that is `NULL`. It cleans up by calling `dsDeallocAttributeEntry` and `dsDeallocAttributeValueEntry` to reclaim the memory associated with the attribute entry and attribute value entry that were created by calling `dsGetAttributeEntry` and `dsGetAttributeValueEntry`. It also calls `dsDeallocRecordEntry` to reclaim the memory associated with the record entry. Before it returns, `GetRecordList` should call `dsDataListDeallocate` to reclaim the memory associated with the `recNames`, `recTypes`, and `attrTypes` data lists. It should also call `dsDataBufferDeAllocate` to deallocate the data buffer.

When the GetRecordList routine returns, the sample code in Listing 3-1 calls dsCloseDirNode to close the node that it opened in order to get the record list.

Listing 3-1 Listing records in a node

```
void main ( )
{
    long dirStatus = eDSNoErr;
    tDirNodeReference nodeRef = NULL;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            GetRecordList(nodeRef);
            dsCloseDirNode( nodeRef );
        }
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}

long GetRecordList ( const tDirNodeReference nodeRef )
{
    unsigned long i = 0;
    unsigned long j = 0;
    unsigned long k = 0;
    long dirStatus = eDSNoErr;
    unsigned long recCount = 0; // Get all records.
    tDataBufferPtr dataBuffer = NULL;
    tContextData context = NULL;
    tAttributeListRef attrListRef = NULL;
    tAttributeValueListRef valueRef = NULL;
    tRecordEntry *pRecEntry = NULL;
    tAttributeEntry *pAttrEntry = NULL;
    tAttributeValueEntry *pValueEntry = NULL;
    tDataList recNames;
    tDataList recTypes;
    tDataList attrTypes;
    dataBuffer = dsDataBufferAllocate( gDirRef, 2 * 1024 ); // allocate  a 2k
buffer
    if ( dataBuffer != NULL )
    {
        // For readability, the sample code does not check dirStatus  after
        // each call, but
        // your code should.
        dirStatus = dsBuildListFromStringsAlloc ( gDirRef, &recNames,
kDSRecordsAll, NULL );
        dirStatus = dsBuildListFromStringsAlloc ( gDirRef, &recTypes,
kDSStdRecordTypeUsers, NULL );
        dirStatus = dsBuildListFromStringsAlloc ( gDirRef, &attrTypes,
kDSAttributesAll, NULL );
        do
        {
            dirStatus = dsGetRecordList( nodeRef, dataBuffer, &recNames,
eDSExact,  &recTypes, &attrTypes, false, &recCount, &context  );
            for ( i = 1; i <= recCount; i++ )
            {
                dirStatus = dsGetRecordEntry( nodeRef, dataBuffer,  i,
&attrListRef,  &pRecEntry );
                for ( j = 1; j <= pRecEntry->fRecordAttributeCount;  j++ )
                {
                    dirStatus = dsGetAttributeEntry( nodeRef, dataBuffer,
attrListRef, j, &valueRef, &pAttrEntry );
                    for ( k = 1; k <= pAttrEntry->fAttributeValueCount;  k++ )
                    {
                        dirStatus = dsGetAttributeValue( nodeRef,  dataBuffer,
 k,  valueRef, &pValueEntry );
                        printf( "%s\t- %lu\n",
pValueEntry->fAttributeValueData.fBufferData, pValueEntry->fAttributeValueID
);
                        dirStatus = dsDeallocAttributeValueEntry(  gDirRef,
pValueEntry );
                        pValueEntry = NULL;
                        // Deallocate pAttrEntry, pValueEntry, and  pRecEntry
                        // by calling dsDeallocAttributeEntry,
                        // dsDeallocAttributeValueEntry, and
                        // dsDeallocRecordEntry, respectively.
                    }
                    dirStatus = dsCloseAttributeValueList( valueRef  );
                    valueRef = NULL;
                    dirStatus = dsDeallocAttributeEntry( gDirRef,  pAttrEntry);
                    pAttrEntry = NULL;
                }
                dirStatus = dsCloseAttributeList( attrListRef );
                attrListRef = NULL;
                dirStatus = dsDeallocRecordEntry( gDirRef, pRecEntry  );
                pRecEntry = NULL;
            }
        } while (context != NULL); // Loop until all data has been  obtained.
        // Call dsDataListDeallocate to deallocate recNames, recTypes,  and
        // attrTypes.
        // Deallocate dataBuffer by calling dsDataBufferDeAllocate.
            dsDataListDeallocate ( gDirRef, &recNames );
            dsDataListDeallocate ( gDirRef, &recTypes );
            dsDataListDeallocate ( gDirRef, &attrTypes );
            dsDataBufferDeAllocate ( gDirRef, dataBuffer );
            dataBuffer = NULL;
    }
    return dirStatus;
} // GetRecordList
```

Getting Information About a Record's Attribute

The sample code in Listing 3-2 (page 36) demonstrates how to get information about a record's attribute. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its `MyOpenDirNode` routine and passes to it the address of the node reference (`nodeRef`) that it has allocated. The `MyOpenDirNode` routine is described in the section "Opening and Closing a Node" (page 28).

The sample code then calls its GetRecInfo routine and passes to it the node reference (nodeRef) obtained by calling its MyOpenDirNode routine.

The GetRecInfo routine calls dsDataNodeAllocateString to allocate two data nodes: one named recName allocated using the string "admin" and one named recType using the constant kDSStdRecordTypeGroups. Then the GetRecInfo routine calls dsOpenRecord to open the record whose name and record type match recName and recType.

If dsOpenRecord returns successfully, the GetRecInfo routine calls dsDataNodeAllocateString to allocate a data node (attrType) containing the constant kDS1AttrPrimaryGroupID. It then calls dsGetRecordAttributeInfo using attrType to specify which attribute to get information for. The dsGetRecordAttributeInfo function stores the attribute's information in a tAttributeEntry structure.

To clean up, GetRecInfo calls dsDeallocAttributeEntry to deallocate the memory associated with pAttrInfo and call dsDataNodeDeAllocate to reclaim the memory asociated with attrType, recName, and recType.

When the GetRecInfo routine returns, the sample code in Listing 3-2 calls dsCloseDirNode to close the node that it opened in order to get the information about a record's attribute.

Listing 3-2 Getting information about a record's attribute

```
void main ( )
{
    long dirStatus = eDSNoErr;
    tDirNodeReference nodeRef = NULL;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            GetRecInfo(nodeRef);
            dsCloseDirNode( nodeRef );
        }
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}
void GetRecInfo ( const tDirNodeReference inDirNodeRef )
{
    long dirStatus = eDSNoErr;
    tRecordReference recRef = NULL;
    tAttributeEntryPtr pAttrInfo = NULL;
    tDataNodePtr recName = NULL;
    tDataNodePtr recType = NULL;
    tDataNodePtr attrType = NULL;
    recName = dsDataNodeAllocateString( gDirRef, "admin"  );
    if ( recName != NULL )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeGroups);
        if ( recType != NULL )
        {
            dirStatus = dsOpenRecord( inDirNodeRef, recType, recName,  &recRef
 );
```

```
            if ( dirStatus == eDSNoErr )
            {
                attrType = dsDataNodeAllocateString(gDirRef, 
kDS1AttrPrimaryGroupID );
                if ( attrType != NULL )
                {
                    dirStatus = dsGetRecordAttributeInfo(recRef,  attrType, 
&pAttrInfo );
                    if ( pAttrInfo != NULL )
                    {
                       dirStatus = dsDeallocAttributeEntry( gDirRef,  pAttrInfo
 );
                        pAttrInfo = NULL;
                    }
                    dirStatus = dsDataNodeDeAllocate( gDirRef, attrType  );
                    attrType = NULL;
                }
            }
            dirStatus = dsDataNodeDeAllocate( gDirRef, recType );
            recType = NULL;
        }
        dirStatus = dsDataNodeDeAllocate( gDirRef, recName );
        recName = NULL;
    }
} // GetRecInfo
```

Setting the Name of a Record

The sample code in Listing 3-3 demonstrates how to set the name of a record. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its MyOpenDirNode routine and passes to it the address of the node reference (nodeRef) that it has allocated. The MyOpenDirNode routine is described in the section "Opening and Closing a Node" (page 28).

The sample code then calls its SetRecordName routine and passes to it the node reference (nodeRef) obtained by calling its MyOpenDirNode routine.

The SetRecordName routine calls dsDataNodeAllocateString to allocate a data node (recName) containing the string "testuser". This is the current name of the record. Then the SetRecordName routine calls dsDataNodeAllocateString to allocate a data node (recType) that specifies the record type (kDSStdRecordTypeUsers) of the record whose name is to be set.

The SetRecordName routine then calls dsOpenRecord with recName and recType as parameters to specify the record to open. If SetRecordName successfully opens the record, it receives a record reference (recRef). Then SetRecordName calls dsDataNodeAllocateString to allocate a data node (newRecName) containing the string "Robert Smith" (the new name that is it be set). The SetRecordName routine then calls dsSetRecordName to set the record's new name and dsCloseRecord to close the record. It then cleans up by calling dsDataNodeDeAllocate to reclaim the memory associated with recName, recType, and newRecName.

When the SetRecordName routine returns, the sample code in Listing 3-3 calls dsCloseDirNode to close the node that it opened in order to set the record's name.

Listing 3-3 Setting the name of a record

```
void main ( )
{
    long dirStatus = eDSNoErr;
    tDirNodeReference nodeRef = NULL;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            SetRecordName( nodeRef );
            dsCloseDirNode( nodeRef );
        }
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}
void SetRecordName ( const tDirNodeReference nodeRef )
{
    long dirStatus = eDSNoErr;
    tRecordReference recRef = NULL;
    tDataNodePtr recName = NULL;
    tDataNodePtr newRecName = NULL;
    tDataNodePtr recType = NULL;
    recName = dsDataNodeAllocateString( gDirRef, "testuser"  );
    if ( recName != NULL )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers  );
        if ( recType != NULL )
        {
            dirStatus = dsOpenRecord( nodeRef, recType, recName,  &recRef );
            if ( dirStatus == eDSNoErr )
            {
                newRecName = dsDataNodeAllocateString(gDirRef, "newtestname"
);
                if ( newRecName != NULL )
                {
                    dirStatus = dsSetRecordName( recRef, newRecName  );
                    dsDataNodeDeAllocate( gDirRef, newRecName );
                    newRecName = NULL;
                }
                dirStatus = dsCloseRecord( recRef );
                recRef = NULL;
            }
            dsDataNodeDeAllocate( gDirRef, recType );
            recType = NULL;
        }
        dsDataNodeDeAllocate( gDirRef, recName );
        recName = NULL;
    }
} // SetRecordName
```

Note that for this example to work, it would have to be run by a root process on the local NetInfo domain, or by a user process that has called dsDoDirNodeAuth with the inDirNodeAuthOnlyFlag parameter set to FALSE to get permission to make this change.

Creating a Record and Adding an Attribute

The sample code in Listing 3-4 demonstrates how to create a record, open it, and add an attribute to it. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its MyOpenDirNode routine and passes to it the address of the node reference (nodeRef) that it has allocated. The MyOpenDirNode routine is described in the section "Opening and Closing a Node" (page 28).

The sample code then calls its CreateRecord routine and passes to it the node reference (nodeRef) obtained by calling its MyOpenDirNode routine.

The CreateRecord routine calls dsDataNodeAllocateString to allocate a data node (recName) containing the string "NewUserRecordName" and another data node (recType) specifying kDSStdRecordTypeUsers as the record type for the record that is to be created.

Then the CreateRecord routine then calls dsCreateRecordAndOpen, passing to it the node reference created when dsOpenDirService was called, recName, recType, and the address of a record reference value (recRef) initialized to zero. If dsCreateRecordAndOpen returns successfully, recRef will contain a record reference that the CreateRecord routine will use to add an attribute for the record.

The CreateRecord routine then calls dsDataNodeAllocateString to allocate a data node (attrName) containing kDS1AttrDistinguishedName. It also calls dsDataNodeAllocateString to allocate a data node containing the string "User Record's Display Name", which will be set as the value of the attribute.

To add the attribute and set its value, the CreateRecord routine calls dsAddAttribute. It then cleans up by calling dsCloseRecord to close the record and dsDataNodeDeAllocate to reclaim the memory associated with attrName, recType, and recName.

When the CreateRecord routine returns, the sample code in Listing 3-4 calls dsCloseDirNode to close the node that it opened in order to create and open the record.

Listing 3-4 Creating and opening a record and adding an attribute

```
void main ( )
{
    long dirStatus = eDSNoErr;
    tDirNodeReference nodeRef = NULL;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            CreateRecord( nodeRef );
            dsCloseDirNode( nodeRef );
        }
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}
void CreateRecord ( const tDirNodeReference inDirNodeRef )
{
    long dirStatus = eDSNoErr;
    tDataNodePtr recName = NULL;
    tDataNodePtr recType = NULL;
    tDataNodePtr attrName = NULL;
    tDataNodePtr attrValue = NULL;
    tRecordReference recRef = NULL;
    recName = dsDataNodeAllocateString( gDirRef, "NewUserRecordName"  );
    if ( recName != NULL )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers  );
        if ( recType != NULL )
        {
            dirStatus = dsCreateRecordAndOpen( inDirNodeRef, recType,  recName,
  &recRef );
            if ( dirStatus == eDSNoErr )
            {
                attrName = dsDataNodeAllocateString(gDirRef,
kDS1AttrDistinguishedName );
                if ( attrName != NULL )
                {
                 attrValue = dsDataNodeAllocateString( gDirRef,  "User Record's
  Display Name");
                    if ( attrValue != NULL )
                    {
                        dirStatus = dsAddAttribute(recRef, attrName,  NULL,
attrValue );
                        dirStatus = dsDataNodeDeAllocate( gDirRef,  attrValue
);
                        attrValue = NULL;
                    }
                    dirStatus = dsDataNodeDeAllocate( gDirRef, attrName  );
                    attrName = NULL;
                }
                dirStatus = dsCloseRecord( recRef );
                recRef = NULL;
            }
            dirStatus = dsDataNodeDeAllocate( gDirRef, recType );
            recType = NULL;
        }
        dirStatus = dsDataNodeDeAllocate( gDirRef, recName );
        recName = NULL;
    }
} // CreateRecord
```

Note that for this example to work, it would have to be run by a root process on the local NetInfo domain, or by a user process that has called `dsDoDirNodeAuth` with the `inDirNodeAuthOnlyFlag` parameter set to `FALSE` to get permission to make this change.

Deleting a Record

The sample code in Listing 3-5 demonstrates how to delete a record. The sample code opens an Open Directory session and gets an Open Directory reference. Then it calls its MyOpenDirNode routine and passes to it the address of the node reference (nodeRef) that it has allocated. The MyOpenDirNode routine is described in the section "Opening and Closing a Node" (page 28).

The sample code then calls its DeleteRecord routine and passes to it the node reference (nodeRef) obtained by calling its MyOpenDirNode routine.

The DeleteRecord routine calls dsDataNodeAllocateString to allocate a data node (recName) containing the string "testuser" as the name of the record that is to be deleted and another data node (recType) specifying kDSStdRecordTypeUsers as the record type of the record that is to be deleted. It then calls dsOpenRecord to open the record that is to be deleted and calls dsDeleteRecord to delete the record.

To reclaim memory associated with recType and recName, the DeleteRecord routine calls dsDataNodeDeAllocate.

The dsDeleteRecord function implicitly closes any record that it deletes. If dsDeleteRecord returns an error, indicating that the record was not deleted, the DeleteRecord routine in Listing 3-5 calls dsCloseRecord to close the record.

Listing 3-5 Deleting a record

```
void main ( )
{
    long dirStatus = eDSNoErr;
    tDirNodeReference nodeRef = NULL;
    dirStatus = dsOpenDirService( &gDirRef );
    if ( dirStatus == eDSNoErr )
    {
        dirStatus = MyOpenDirNode( &nodeRef );
        if ( dirStatus == eDSNoErr )
        {
            DeleteRecord( nodeRef );
            dsCloseDirNode( nodeRef );
        }
    }
    if ( gDirRef != NULL )
    {
        dirStatus = dsCloseDirService( gDirRef );
    }
}
void DeleteRecord ( const tDirNodeReference nodeRef )
{
    long dirStatus = eDSNoErr;
    tRecordReference recRef = NULL;
    tDataNodePtr recName = NULL;
    tDataNodePtr recType = NULL;
    recName = dsDataNodeAllocateString( gDirRef, "testuser"  );
    if ( recName != NULL )
    {
        recType = dsDataNodeAllocateString( gDirRef, kDSStdRecordTypeUsers  );
        if ( recType != NULL )
        {
            dirStatus = dsOpenRecord( nodeRef, recType, recName,  &recRef );
            if ( dirStatus == eDSNoErr )
            {
                dirStatus = dsDeleteRecord( recRef );
                if (dirStatus != eDSNoErr)
                {
                    // The record was not deleted, so close it.
                    dirStatus = dsCloseRecord( recRef );
                }
                recRef = NULL;
            }
            dsDataNodeDeAllocate( gDirRef, recType );
            recType = NULL;
        }
        dsDataNodeDeAllocate( gDirRef, recName );
        recName = NULL;
    }
} // DeleteRecord
```

Note that for this example to work, it would have to be run by a root process on the local NetInfo domain, or by a user process that has called `dsDoDirNodeAuth` with the `inDirNodeAuthOnlyFlag` parameter set to `FALSE` to get permission to make this change.

Document Revision History

This table describes the changes to *Open Directory Programming Guide*.

Date	Notes
2007-01-08	Fixed code listing in Working With Records.
2006-04-04	Moved reference documentation to become a separate document.
2005-04-29	Updated for Mac OS X v10.4. Changed "Rendezvous" to "Bonjour." Changed title from "Open Directory."