Open Directory Plug-in Programming Guide

Networking > Mac OS X Server

2006-05-23

Contents

Figures, Tables, and Listings

Introduction

An Open Directory plug-in is a Mac OS X dynamically loaded library that responds to requests for directory service from applications that are clients of Open Directory.

This book describes the runtime environment for Open Directory plug-ins and how to build and configure an Open Directory plug-in. It also describes the entry points that an Open Directory plug-in must provide, the requests that an Open Directory plug-in must be prepared to respond to, and the Open Directory callback routines that the plug-in can call to register and unregister nodes and to write in log files.

Organization of This Document

This book contains the following chapters:

- "Runtime Environment" (page 9) describes how Open Directory plug-ins are loaded.
- "Required Entry Points" (page 11) defines the required entry points for an Open Directory plug-in
- "Processing Open Directory Requests" (page 13) defines the requests an Open Directory plug-in must be prepared to handle.
- "Processing Concurrent Requests" (page 15) explains the asynchronous nature of Open Directory plug-ins
- "Open Directory Callbacks" (page 17) defines the callback routines provided by Open Directory.
- "Calling Mac OS X Functions" (page 19) explains the advantages and disadvantages of using Mac OS X functions in an Open Directory plug-in.
- "Managing References" (page 21) describes how Open Directory plug-ins interact with object references.
- "Standard Record and Attribute Types" (page 23) explains what information needs to be maintained from record and attribute types.
- "Authentication" (page 25) defines what authentication methods an Open Directory plug-in needs to support.
- "Property List for an Open Directory Plug-in" (page 27) explains the keys used in an Open Directory plug-in property list and how those keys are utilized.
- "Configuring an Open Directory Plug-in" (page 29) describes the variety of configurations for setting up an Open Directory plug-in.
- "Client Side Buffer Parsing" (page 35) explains how to use a `tDataBuffer` object in an Open Directory plug-in.

See Also

Refer to the following reference document for Open Directory plug-ins:

- *Open Directory Reference*

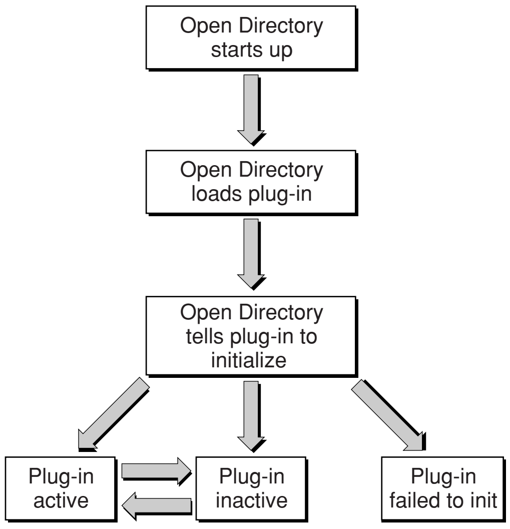
For more information about Open Directory client programming, and administration, see:

- *Open Directory Programming Guide*
- Mac OS X Server Open Directory Administration

Runtime Environment

Figure 1-1 (page 9) summarizes the flow of events that occur with regard to plug-ins when Open Directory starts up.

Figure 1-1 Open Directory startup and plug-in states



When Open Directory starts up, it uses the CFBundle mechanism to load into memory each plug-in that it finds in the following directories:

- `/System/Library/Frameworks/DirectoryService.framework/Resources/Plugins`
- `/Library/DirectoryServices/PlugIns`

The `/Library/DirectoryServices/PlugIns` directory is the recommended location for your plug-in.

After a plug-in loads, it is in the "loaded but not initialized" state. For each successfully loaded plug-in, Open Directory calls the plug-in's Initialize entry point. If a plug-in fails to initialize itself, it is in the "failed to initialize" state. When a plug-in successfully initializes itself, it enters the "active" state. In response to settings in the Directory Access application, Open Directory may tell an active plug-in to become inactive or an inactive plug-in to become active at any time.

Loading of plug-ins that are not configured to be loaded at startup is deferred until loading the plug-in becomes absolutely necessary when, for example an application opens a node for which the as-yet-unloaded plug-in is responsible. Search requests from clients such as the automounter can also cause a plug-in to be loaded. This type of deferred plug-in loading is know as **lazy loading**.

Prior to Mac OS X v10.4, plug-ins that were disabled by the Directory Access application were loaded if an event occurred to trigger lazy loading. Starting with Mac OS X v10.4, plug-ins that have been disabled by the Directory Access application are not longer subject to lazy loading. This change allows disabled plug-ins to be configured without the risk of them being inadvertently loaded.

A plug-in that is in the active or inactive state can only be called through certain entry points:

- In the active state, the plug-in can be called through its periodic task, process request, shutdown, and set plug-in state entry points.
- In the inactive state, the plug-in can be called through its periodic task, set plug-in state, and shutdown entry points.

In three special cases, an inactive plug-in can be called through its process entry point:

- when a node having the same name as the plug-in is opened in order to configure the plug-in. For example, when inactive, the LDAPv3 plug-in's process entry point is called when an application opens the node `/LDAPv3` and calls `dsDoPluginCustomCall` to configure the plug-in.
- after the plug-in is loaded and initialized in order to receive the `sHeader` structure. The `fContextData` field of that structure contains the DirectoryService daemon's current run loop, which your plug-in can use to set timers.
- after the plug-in is loaded and initialized in order to receive the Kerberos mutex.

Entry points are described in the next chapter, "Required Entry Points" (page 11)

Required Entry Points

Every Open Directory plug-in must provide the entry points described in this section. The entry points are listed below in the order in which they are typically called.

- `Initialize`, called by Open Directory so that the plug-in can initialize itself.
- `Validate`, called by Open Directory when plug-ins are loaded in order to pass to each plug-in a unique value that the plug-in uses to identify itself when it calls Open Directory callback routines in order to register and unregister directory nodes or to write information in an Open Directory log file.
- `SetPluginState`, called by Open Directory to notify the plug-in of a change in state. For example, this entry point would be called to enable or disable the plug-in.
- `PeriodicTask`, called by Open Directory on a regular basis so that the plug-in can perform periodic tasks.
- `ProcessRequest`, called by Open Directory to pass requests from Open Directory clients.
- `Shutdown`, called by Open Directory to tell the plug-in that Open Directory is shutting down. For example, this entry point would be called when the system shuts down. The plug-in should release memory and perform any other tasks to prepare itself for shutdown.

Processing Open Directory Requests

Open Directory passes to the appropriate Open Directory plug-in certain requests from Open Directory clients. The requests correspond to a subset of the Open Directory function calls described in *Open Directory Programming Guide*. Every Open Directory plug-in must be prepared to process each of the requests described in this section even if only to respond that the requested service is not implemented (`eNotYetImplemented`) or not handled (`eNotHandledByThisNode`). To indicate the outcome of processing a request, the plug-in should return a result code from the list of result codes documented in *Open Directory Programming Guide*.

The plug-in must be prepared to process requests for each of the Open Directory functions described in this section.

Table 3-1 Open Directory functions that cause the `ProcessRequest` entry point to be called

`dsAddAttribute`	`dsGetDirNodeInfo`
`dsAddAttributeValue`	`dsGetRecordAttributeInfo`
`dsCloseAttributeList`	`dsGetRecordAttributeValueByID`
`dsCloseAttributeValueList`	`dsGetRecordAttributeValueByValue`
`dsCloseDirNode`	`dsGetRecordEntry`
`dsCloseRecord`	`dsGetRecordList`
`dsCreateRecord`	`dsGetRecordReferenceInfo`
`dsCreateRecordAndOpen`	`dsOpenDirNode`
`dsDeleteRecord`	`dsOpenRecord`
`dsDoAttributeValueSearch`	`dsRemoveAttribute`
`dsDoAttributeValueSearchWithData`	`dsRemoveAttributeValue`
`dsDoDirNodeAuth`	`dsSetAttributeAccess`
`dsDoPluginCustomCall`	`dsSetAttributeFlags`
`dsDoMultipleAttributeValueSearch`	`dsSetAttributeValue`
`dsDoMultipleAttributeValueSearchWithData`	`dsSetAttributeValues`
`dsFlushRecord`	`dsSetRecordAccess`
`dsGetAttributeEntry`	`dsSetRecordFlags`
`dsGetAttributeValue`	`dsSetRecordName`

As an alternative to processing `dsCloseAttributeList`, `dsCloseAttributeValueList`, `dsGetRecordEntry`, `dsGetAttributeEntry`, and `dsGetAttributeValue` requests in the plug-in, applications can use client-side buffer parsing to process these requests. For information, see the chapter "Client Side Buffer Parsing" (page 35).

See "Runtime Environment" (page 9) for information on three special cases in which the `ProcessRequest` entry point of an inactive plug-in is called.

Processing Concurrent Requests

Open Directory is multi-threaded, so plug-ins must be thread-safe. Plug-ins may be called multiple times by multiple applications. For example, the following requests may occur at the same time:

- Application A makes a request that takes a long length of time to complete.
- Application B makes a request that takes a short length of time to complete.
- Application C makes a request that takes a medium length of time to complete.

Open Directory passes requests to the responsible plug-in as the requests come in and does not manage or serialize requests in any way. The plug-in is responsible for handling multiple concurrent requests in any way that it deems appropriate. It may choose to process Application A's request first and Application B's request last, process the requests serially, or use some other algorithm for determining the order in which to process concurrent requests.

Open Directory Callbacks

Open Directory provides three callback routines for plug-ins to call:

- `DSDebugLog`. Writes an entry in the Open Directory log file. All records written by all Open Directory plug-ins are written to the same log file in the order by which Open Directory receives them.
- `DSRegisterNode`. Registers a node so that it is available for use by applications that make Open Directory calls.
- `DSUnregisterNode`. Unregisters a node that was previously registered.

The Open Directory callback routines are described in detail in the section Open Directory Callbacks" in the Reference section.

Calling Mac OS X Functions

Open Directory plug-ins can call any Mac OS X function but to reduce memory usage and make porting to other platforms easier, Open Directory plug-ins should restrict themselves to the System and CoreFoundation frameworks and use other frameworks only when there is a compelling reason to do so. (For example, the Open Directory LDAP plug-in uses the LDAP library.)

Open Directory plug-ins themselves should not display any human interface (HI). Only a plug-in's separate configuration application or Directory Access plug-in should display HI.

Managing References

Open Directory allocates Directory Service references, such as Open Directory node references, open record references, and attribute list value references, and passes them to the appropriate plug-in as part of a process request. Plug-ins can use these references to keep track of their own data. When a reference becomes invalid, such as when an Open Directory node is closed, the plug-in must free any memory that is associated with the now invalid reference.

Standard Record and Attribute Types

Plug-ins should support the standard record and attribute types described in “Record Type Constants” in the Reference chapter as appropriate for the data that the directory system provides and in accordance with the needs of Mac OS X. Plug-ins should also honor the meta types described in that section. (Meta types are types that are created dynamically, such as a user’s current location.) Plug-ins should map the standard record and attribute types to the plug-in’s native record and attribute types.

Plug-ins can support as many native record and attribute types as they want.

Authentication

If an Open Directory plug-in is going to support authentication, at minimum, it should support node native, change password, and set password as root methods. Other authentication methods are optional. It is up to the plug-in to determine which node native authentication allows cleartext authentication.

Property List for an Open Directory Plug-in

An Open Directory plug-in is a standard Mac OS X bundle and follows the guidelines defined for Mac OS X packages.

Open Directory plug-ins are loaded from the following directories:

- `/System/Library/Frameworks/DirectoryService.framework/Resources/Plugins` (which may be read-only)
- `/Library/DirectoryServices/PlugIns` (which is always writable)

or from other directories that may be defined later by Mac OS X.

Open Directory loads Open Directory plug-ins using the CFBundle load mechanism.

No special linker commands are required to build an Open Directory plug-in, but you should use the `-bundle_loader /usr/sbin/DirectoryService` option if you want to call APIs from the DirectoryService framework. Do no link to the DirectoryService framework, but use the DirectoryService framework to locate header files. The benefit of this approach is that calls can be directly dispatched without the overhead of a Mach message.

You must provide a property list file for your Open Directory plug-in. Here is the property list file for a plug-in named `SamplePlugin`:

Listing 10-1 Property list for a sample plug-in

```
{
    "CFBundleExecutable" = "SamplePlugin";
    "CFBundleName" = "DirectoryServiceSamplePlugIn";
    "CFBundleIdentifier" = "com.apple.DirectoryService.SamplePlugin";
    "CFBundleVersion" = "1.0";
    "CFBundleShortVersionString" = "1.0";
    "CFBundlePackageType" = "dspi";
    "CFBundlePackageSignature" = "adss";
    "CFPlugInDynamicRegistration" = "NO";
    "CFPlugInFactories" = {
        "D970D52E-D515-11D3-9FF9-000502C1C736" = "ModuleFactory";
    };
    "CFPlugInTypes" = {"697B5D6C-87A1-1226-89CA-000502C1C736"  =
        ("D970D52E-D515-11D3-9FF9-000502C1C736");
    };
    "DSServerSignature" = "Samp";
}
```

In Listing 10-1 (page 27), you are responsible for setting the following values:

- `CFBundleExecutable` — the name of the Open Directory plug-in
- `CFBundleName` — the name of the bundle

- `CFBundleIdentifier` — the bundle identifier
- `CFBundleVersion` — the bundle's version number
- `CFBundleShortVersionString` — the bundle's short version number
- `CFPluginFactories` — a UNIX unique identifier (UUID) that is unique among all plug-ins generated by the `uuidgen` utility
- `CFPlugInTypes` — the first value must be `697B5D6C-87A1-1226-89CA-000502C1C736;` the second value must be your plug-in's UUID
- `DSServerSignature` — your server signature

The following values must be set as shown in Listing 10-1 (page 27):

- `CFBundlePackageType` — must be `dspi`
- `CFBundlePackageSignature` — must be `adss`
- `CFPlugInDynamicRegistration` — must be `NO`
- `CFPlugInTypes` (first value) — must be `697B5D6C-87A1-1226-89CA-00502C1C736`

See the section "Local Configuration" (page 29) for information about the `CFBundleConfigAvail` property, which is used to specify a configuration application for your Open Directory plug-in Mac OS X v10.1.

Configuring an Open Directory Plug-in

In Mac OS X v10.1, Open Directory plug-ins are configured through the Directory Setup application, which supports local configuration only. The Directory Setup application launches your plug-in configuration application when an administrator selects your plug-in for configuration. The configuration application to launch is specified in the property list file. Local configuration in Mac OS X v10.1 is described in the section "Local Configuration" (page 29).

In Mac OS X v10.2 and later, administrators use the Directory Access application to select Open Directory plug-ins for configuration. If you provide a Directory Access plug-in that uses custom calls to communicate with your Open Directory plug-in, your Open Directory plug-in can be configured locally and remotely. Local and remote configuration in Mac OS X v10.2 and later is described in the section "Remote Configuration" (page 30).

Local Configuration

In Mac OS X v10.1, the property list file is used to specify a configuration application that is to be launched when an administrator selects a plug-in for configuration. The `CFBundleConfigAvail` property specifies the full pathname, including filename, for the configuration application to launch. For example:

```
"CFBundleConfigAvail" = "/System/Library/Frameworks/DirectoryService.framework/
 Resources/YourPlugInConfig.app"
```

You can also specify the name of the configuration file to open when your configuration application is launched:

```
"CFBundleConfigFile" = "/Library/Preferences/DirectoryService/
YourPlugInConfig.xml"
```

In this example, the configuration file is in XML format, as indicated by the extension `.xml`.

If your plug-in does not have a configuration application, it can set `CFBundleConfigAvail` to `Not Available`. If the `CFBundleConfigFile` property is missing from your property list file or is set to `Not Available`, but `CFBundleConfigFile` is set to a file name, the file is opened with its default application. If `CFBundleConfigAvail` is set, but `CFBundleConfigFile` is missing or set to `Not Available`, the configuration application is launched without a file. If both properties are missing or set to `Not Available`, the Configure button is dimmed when your plug-in is selected.

Remote Configuration

Mac OS X v10.2 and later supports remote configuration. To configure your Open Directory plug-in remotely, you need to create a Directory Access plug-in having the same name as your Open Directory plug-in but ending with `.daplug` instead of `.dsplug`. You can place your Directory Access plug-in in the Directory Access application bundle:

```
/Applications/Utilities/Directory Access.app/Contents/PlugIns/myplug.daplug
```

Your Directory Access plug-in should implement at least two functions:

- handle the Configure button click
- accept a PluginAPIImplementor object that provides your Directory Access plug-in with the information necessary to handle local and remote connections

You should use custom calls to your plug-in so you can support remote configuration as well as local configuration.If you need to support configuration of your Open Directory plug-in in Mac OS X v10.1 as well as Mac OS X v10.2 and later, you'll need to use the `CFBundleConfigAvail` property in your property list file for Mac OS X v10.1 and have a Directory Access plug-in for Mac OS X v10.2 and later. If both types of configuration are installed on the same system running Mac OS X v10.2 or later, your Directory Access plug-in will override the launching of the configuration application specified by the `CFBundleConfigAvail` property.

DirectoryAccess Plug-ins

Starting with Mac OS 10.2, the Directory Access application, used by administrators to configure Open Directory, supports DirectoryAccess plug-ins. DirectoryAccess plug-ins are stored in the `Contents/PlugIns` directory inside the Directory Access application directory. A DirectoryAccess plug-in must be an NSBundle, have a name that matches the Open Directory plug-in it configures and have the extension `.daplug`. For Mac OS 10.3 and later, you can use the `pluginName` method to override the name showing in the list.

The bundle's main class should support the DirectoryAccessPlugin interface, described below:

```
- (void)setPluginAPIImplementor:(id)implementor;
- (BOOL)saveChanges; // Return FALSE if can't save for some reason.
- (BOOL)revertChanges;
- (BOOL)applicationWillQuitSavingChanges:(BOOL)save;
- (BOOL)applicationWillLockSavingChanges:(BOOL)save;
- (BOOL)isDirty;
- (void)configureButtonClicked:(id)sender;
- (void)setEnabled:(BOOL)enabled forLocation:(NSString*)location;
- (BOOL)isEnabledForLocation:(NSString*)location;
```

Table 11-1 describes each method.

Table 11-1 DirectoryAccessPlugin interface methods

Method	Description
`setPluginAPIImplementor:`	Provides the API implementor used for callbacks.

Method	Description
`saveChanges:`	Called when the Apply button is clicked.
`revertChanges:`	Called when the Revert button is clicked.
`applicationWillQuit-SavingChanges:`	Called before the application quits, with the `save` parameter indicating whether to save changes. If appropriate, `saveChanges` will be called after `applicationWillQuitSavingChanges`.
`applicationWillLock-SavingChanges:`	Called before the application locks, with the `save` parameter indicating whether to save changes. If appropriate, `saveChanges` or `revertChanges` will be called after `applicationWillLock-SavingChanges`.
`isDirty:`	Indicates whether the DirectoryAccess plug-in has any unsaved changes. Return `YES` from `isDirty` to ensure that the user is prompted to save changes when quitting or locking.
`configureButtonClicked:`	Called when your DirectoryAccess plug-in is selected when the Configure button is clicked or if your plug-in is double-clicked.
`setEnabled`	Used to override the state of the Enabled checkbox.
`isEnabledForLocation`	Used to override the state of the Enabled for Location checkbox.

Your bundle's main class is not required to implement all of the methods in `DirectoryAccessPlugin.h`; instead, it should implement only those methods that your DirectoryAccess plug-in actually needs. The most likely methods that you will need to implement are `setPluginAPIImplementor:` and `configureButtonClicked:`.

Starting with Mac OS X v10.4, new DirectoryAccess methods are available:

```
// PluginAPIImplementor new methods available in Mac OS X v10.4  and later
- (BOOL)canSetEnabledForLocation:(NSString*)location;
- (NSString*)hostName;
- (BOOL)saveSearchPolicies;
- (BOOL)isNodeOnAuthSearchPolicy:(NSString*)nodeName;
- (BOOL)isNodeOnContactSearchPolicy:(NSString*)nodeName;
- (void)addNodeToAuthSearchPolicy:(NSString*)nodeName;
- (void)addNodeToContactsSearchPolicy:(NSString*)nodeName;
- (void)deleteNodeFromAuthSearchPolicy:(NSString *)nodeName;
- (void)deleteNodeFromContactsSearchPolicy:(NSString *)nodeName;
- (void)deleteNodesFromAuthSearchPolicyWithPrefix:(NSString *)prefix;
- (void)deleteNodesFromContactsSearchPolicyWithPrefix:(NSString  *)prefix;
- (void)nodeNameChangedFrom:(NSString*)oldNodeNameto:(NSString*)newNodeName;
- (NSString *)pluginVersionWithPrefix:(NSString *)prefix;
- (BOOL)isPluginEnabled:(NSString*)prefix;
- (void)setPlugin:(NSString*)prefix enabled:(BOOL)enabled;
```

Table 11-2 describes the new DirectoryAccessPlugin interface methods.

Table 11-2 New DirectoryAccessPlugin interface methods

Method	Description
`canSetEnabledForLocation:`	Used to determine whether to enable or disable the checkbox in the Services list.
hostName	Returns the host name of the machine being configured; useful when configuring a plug-in on a remote system.
`saveSearchPolicies`	Causes any pending search policy changes to be saved; can be used in conjunction with other methods that manipulate search policy lists.
`isNodeOnAuthSearchPolicy:`	Returns `YES` if the specified node name is on the authentication search policy; otherwise, returns `NO`.
isNodeonContactsSearchPolicy:	Returns `YES` if the specified node name is on the contacts search policy; otherwise, returns `NO`.
addNodeToAuthSeachPolicy:	Adds the specified node name to the authentication search policy and changes the authentication search policy to a custom search policy if necessary.
`addNodeToContactsSearchPolicy:`	Adds the specified node name to the contacts search policy and changes the contacts search policy to a custom search policy if necessary.
`deleteNodeFromAuthSearchPolicy:`	Removes the specified node name from the authentication search policy.
deleteNodeFromContactsSearchPolicy:	Removes the specified node name from the contacts search policy.
`deleteNodesFromAuthSearchPolicy-WithPrefix`	Removes all node names that have the specified prefix from the authentication search policy. Useful when disabling a plug-in.
`deleteNodesFrom-ContactsSearchPolcyWithPrefix`	Removes all node names that have the specified prefix from the contacts search policy. Useful when disabling a plug-in.
`nodeNameChangedFrom: to:`	Renames any nodes on either search policy from the old name to the new name. Useful when a configuration change in your plug-in changes a node name.
`plug-inVersionWithPrefix:`	Returns the version number from the named plug-in's `Info.plist`. Useful when using one configuration plug-in to configure different versions of an Open Directory plug-in.
`isPluginEnabled:`	Returns `YES` if the Open Directory plug-in is set to be enabled; otherwise returns `NO`.
`setPlugin: enabled:`	Sets the specified plug-in enabled state as specified. Useful if you want to enable your plug-in when configuring it to be added to the search policy from within your custom configuration sheet.

The file `PluginAPIImplementor.h` defines the following object and methods:

```
@interface PluginAPIImplementor : NSObject {
}

- (BOOL)preflightDSRef;
- (BOOL)preflightAuthRights;
- (OSStatus)makeAuthExternalForm:(AuthorizationExternalForm*)authExtForm;
- (tDirReference)dsRef;
- (AuthorizationRef)authorizationRef;
- (void)pluginEnableStateChanged:(NSString*)pluginName to:(BOOL)newstate;
- (void)reloadSearchPolicies;
- (void)markDirty:(id)sender;
```

The `PluginAPIImplementor` object provides callbacks to DirectoryAccess plug-ins and is passed using `setPluginAPIImplementor:` after the plug-in is loaded and initialized. If you want your Open Directory plug-in to be configurable remotely, or if you need to call any Open Directory functions, you must use the API reference provided by `dsRef`.

The recommended strategy is to have a specially named node, such as `/MyPlugin` for a DirectoryAccess plug-in named `MyPlugin`, that you open and call `dsDoPluginCustomCall` to read and write. To make changes, you can use `makeAuthExternalForm:` to put an externalized `AuthorizationRef` into the buffer so that your DirectoryAccess plug-in can verify that the user is authorized to make changes. Directory Access requires the system.services.directory.configure authorization right, which you can check from your DirectoryAccess plug-in using `AuthorizationCopyRights`. This prevents a malicious user from reconfiguring your Open Directory plug-in without first providing an administrator name and password.

Before performing a read operation, you should call `preflightDSRef`. This ensures that the connection is still established and reconnects if it isn't. If this method returns `NO`, your DirectoryAccess plug-in should also return `NO` to indicate the operation failed.

Before performing a write operation, you should call `preflightAuthRights` to ensure that the `AuthorizationRef` is still valid. If `preflightAuthRights` returns `NO`, your DirectoryAccess plug-in should also return `NO` to indicate to the DirectoryAccess application that the `saveChanges` call failed.

Client Side Buffer Parsing

Open Directory provides standard data buffer parsing for a `tDataBuffer` returned by the plug-in serviced functions `dsGetRecordList`, `dsDoAttributeValueSearch`, and `dsDoAttributeValueSearchWithData` if and only if the buffer was built using the standard format, known as Client Side Buffer Parsing (CSBP) described in this section.

CSBP reduces round trip Mach and TCP message traffic between the client and the server that would otherwise require the passing of the entire `tDataBuffer`. Thus, CSBP leads to a considerable performance improvement in servicing calls to `dsGetRecordEntry`, `dsGetAttributeEntry`, and `dsGetAttributeValue`, which have to extract particular data points such as attribute values out of a `tDataBuffer`.

Note: At this time, CSBP is not used on a `tDataBuffer` created by `dsGetDirNodeInfo` even though the `dsGetAttributeEntry` and `dsGetAttributeValue` functions are used to extract data from the buffer.

There are two CSBP standards: StdA and StdB. StdB was the first format and it is still supported. StdA allows for larger attributes and is the recommended format.

Table 12-1 lists the order of the overall data block holding *x* number of records.

Table 12-1 Format of a StdA and StdB data block

Number of Bytes	Description
4	ulong:: tag describing the data block; value is `STDA` or `STDB`
4	ulong:: count of records contained in this data block
4	ulong:: offset in bytes to the start of the first record block
4	ulong:: offset in bytes to the start of the second record block
...	
4	ulong:: offset in bytes to the start of the last record block
4	ulong:: tag describing end of offsets; value is `ENDT`
...	empty space until the start of the last record block; that is, record blocks are packed into the buffer starting at the end of the data block
4	ulong:: length in bytes of last record block
#	last record block
...	
4	ulong:: length in bytes of first record block

Number of Bytes	Description
#	first record block

Table 12-2 lists the order of a StdA single record block holding *y* complete attribute blocks.

Table 12-2 Format of a StdA single record block

Number of Bytes	Description
2	ushort:: length of record type string
n	UTF-8[n]:: record type string
2	ushort:: length of record name string
m	UTF-8[m]:: record name string
2	ushort:: number of attributes in this record block
4	ulong:: length in bytes of first attribute block
#	first attribute block for this record block
....	
4	ulong:: length in bytes of last attribute block
#	last attribute block for this record block

Table 12-3 lists the order of a StdA single attribute block holding *z* values (that is, all of the attribute's values).

Table 12-3 Format of a StdA single attribute block

Number of Bytes	Description
2	ushort:: length of attribute name
r	UTF-8[r]:: attribute name string
2	ushort:: number of attribute values in this attribute block
4	ulong:: length of first attribute value
s	byte[s]:: first attribute value for this attribute type
....	
4	ulong:: length of last attribute value
t	byte[t]:: last attribute value for this attribute type

Table 12-4 lists the order of a StdB single record block holding *y* complete attribute blocks.

Table 12-4 Format of a StdB single record block

Number of Bytes	Description
2	ushort:: length of record type string
n	UTF-8[n]:: record type string
2	ushort:: length of record name string
m	UTF-8[m]:: record name string
2	ushort:: number of attributes in this record block
2	ushort:: length in bytes of first attribute block
#	first attribute block for this record block
....	
2	ushort:: length in bytes of last attribute block
#	last attribute block for this record block

Table 12-5 lists the order of a StdB single attribute block holding *z* values (that is, all of the attribute's values).

Table 12-5 Format of a StdB single attribute block

Number of Bytes	Description
2	ushort:: length of attribute name
r	UTF-8[r]:: attribute name string
2	ushort:: number of attribute values in this attribute block
2	ushort:: length of first attribute value
s	byte[s]:: first attribute value for this attribute type
....	
2	ushort:: length of last attribute value
t	byte[t]:: last attribute value for this attribute type

Document Revision History

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

Date	Notes
2006-05-23	Moved reference information to the new document "Open Directory Reference."
2005-04-29	Updated for Mac OS X v10.4. Changed "Rendezvous" to "Bonjour." Changed title from "Open Directory Plug-ins."