CHAPTER 3

An Introduction to LDAP

In Chapter 2, "A Brief History of Directories," we talked about the history of directories and how LDAP was born. In this chapter, we take a much closer look at LDAP, both in its role as a network protocol and as a set of models that guide you in constructing and accessing your directory. We'll also examine two other important aspects of LDAP: the LDAP application programming interfaces (APIs), which you can use to develop LDAP applications; and the LDAP Data Interchange Format (LDIF), which is a common, text-based format for exchanging directory data between systems.

What Is LDAP?

At its core, LDAP is a standard, extensible directory access protocol—a common language that LDAP clients and servers use to communicate with each other. Standardization of the protocol has the benefit that client and server software from different vendors can interoperate. When you buy an LDAP-enabled program, you can expect that it will work with any standards-compatible LDAP server. This has many advantages, but we'll discuss those later in this chapter.

LDAP is a "lightweight" protocol, which means that it is efficient, straightforward, and easy to implement, while still being highly functional. Contrast this with a "heavyweight" protocol, such as the X.500 Directory Access Protocol (DAP). X.500 DAP uses complex encoding methods and requires use of the OSI network protocol stack—a networking system that has failed to gain wide acceptance.

LDAP, on the other hand, uses a simplified set of encoding methods and runs directly on top of TCP/IP. Every major desktop and server computing platform currently available (Microsoft Windows, DOS, UNIX, and the Apple Macintosh) either ships with a TCP/IP implementation or can be easily equipped with one. OSI networking, on the other hand, is not universally available, and it is almost always an extra-cost option. LDAP, by virtue of its light weight, removes significant barriers to implementation and deployment.

As mentioned in Chapter 2, there have been two major revisions of the LDAP protocol. The first widely available version was LDAP version 2, defined in RFCs 1777 and 1778. As of this writing, LDAP version 3 is a Proposed Internet Standard, defined in RFCs 2251 through 2256. Because it is so new, not all vendors completely support LDAPv3 yet. As we discuss LDAP in this chapter, we will focus our discussion on LDAPv3. However, we will point out new features found only in LDAPv3 so that you can understand the limitations you will encounter if you are using LDAPv2.

In addition to its role as a network protocol, the LDAP standards also define four models that guide you in your use of the directory. These models promote interoperability between directory installations while still allowing you the flexibility to tailor the directory to your specific needs. The models borrow concepts from X.500, but they generally lack many of the restrictions that the X.500 models include. The four LDAP models are as follows:

- The LDAP information model, which defines the kind of data you can put into the directory.
- The LDAP naming model, which defines how you organize and refer to your directory data.
- The LDAP functional model, which defines how you access and update the information in your directory.
- The LDAP security model, which defines how information in the directory can be protected from unauthorized access.

In addition to guiding you in the use of your directory, the LDAP models guide directory developers when designing and implementing LDAP server and client software. The LDAP models are discussed in detail later in the chapter.

There are several LDAP APIs, the oldest of which is for the C programming language. The C API is supported by several freely available software development kits (SDKs), including one available in binary and source code format from Netscape Communications Corporation. In addition to the C API, Netscape's freely available Java SDK (also available in binary and source code formats) supports all LDAPv3 features. Netscape also provides PerLDAP, a toolkit for the Perl language that allows you to access LDAP directories.

SunSoft's JNDI is a proprietary, unified directory access API that supports access to multiple types of directory services (NIS+, LDAP, and others). Microsoft offers its own proprietary unified directory access API, known as Active Directory Services Interface (ADSI). These APIs and their various strengths are covered later in this chapter and in Chapter 20, "Developing New Applications."

LDAP also defines the LDAP Data Interchange Format (LDIF), a common, text-based format for describing directory information. LDIF can describe a set of directory entries or a set of updates to be applied to a directory. Directory data can also be exported from one directory and into another using LDIF. Most of the commonly available command-line utilities also read and write LDIF. The LDIF format is discussed in more detail later in this chapter.

What Can LDAP Do for You?

If you are a system administrator, then LDAP

- Makes it possible for you to centrally manage users, groups, devices, and other data.
- Helps you do away with the headache of managing separate application-specific directories (such as LAN-based electronic mail software).

If you are a IT decision maker, then LDAP

- Allows you to avoid tying yourself exclusively to a single vendor and/or operating system platform.
- Helps you decrease the total cost of ownership by reducing the number of distinct directories your staff needs to manage.

If you are a software developer, then LDAP

- Allows you to avoid tying your software exclusively to a single vendor and/or operating system platform.
- Helps you save development time by avoiding the need to construct your own user and group management database.

For example, before the advent of LDAP, each of your applications probably had its own directory of user information. If you had a LAN-based email package, there was probably an interface you used to create and manage users and groups. If you had a LAN-based collaboration package from a different vendor, it most likely had its own directory and method of accessing that directory. And, often, if you used LAN-based file server software, it probably had its own distinct user directory.

Some vendors offered packaged solutions, which made it possible to manage users and groups in one place—as long as you used only their software suite. However, if you needed to mix and match software from multiple vendors, you probably ended up developing an elaborate procedure for ensuring that new employees were added to each package's proprietary directory.

With LDAP comes the promise of eliminating this management nightmare. Instead of creating an account for each user in each system he or she needs to access, you will be able to simply create a single directory entry for the user—and all directory-enabled applications will simply refer to the user's entry in the LDAP directory. When an employee is terminated, access to all systems can be revoked by removing the user's directory entry instead of hunting down all accounts granted to the person and disabling each one. Users benefit as well because they need to remember and manage only a single password instead of one for each system.

In addition to consolidating the management of your users' access privileges, LDAP directories allow easier sharing of directory information with trading partners in an extranet environment. If you have established business relationships with other companies, you can use the directory to share common user information between your two organizations. This allows you to set up workflow processes that cross company boundaries, making both organizations more efficient.

LDAP directories also can be used to build entirely new applications. An Internet service provider, for example, might create an LDAP directory that contains information about all its subscribers and the special add-on services it may have purchased. The directory can be consulted each time the user wants to access a given service. If the user has appropriate permissions, as registered in the directory, the application grants the user access; otherwise, access is denied. Management of all the value-added services is handled by updating the directory.

Not all software vendors have made the transition to LDAP-based management, but LDAP has tremendous momentum in the software industry.

Over time, more and more applications will be directory-enabled with the net benefit of reducing the total cost of ownership of your applications.

How Does LDAP Work?

In this section, we'll delve into the LDAP protocol in detail. We'll start with an overview of LDAP as a client/server protocol. We'll then discuss the individual LDAP protocol operations and show how clients can use them to perform useful tasks such as sending secure email. We'll also discuss LDAP extensibility, and we'll conclude by showing you how LDAP works "on the wire" by discussing the actual wire protocol.

A *client/server protocol* is a protocol model in which a client program running on one computer constructs a request and sends it over the network to a computer (possibly the same computer) running a server program. The server program receives the request, takes some action, and returns a result to the client program. Examples of other client/server protocols are Hypertext Transfer Protocol (HTTP), which is typically used to serve Web pages; and Internet Message Access Protocol (IMAP), a protocol used to access electronic mail messages.

The basic idea behind a client/server protocol is that it allows work to be assigned to computers that are optimized for the task at hand. For example, a typical LDAP server computer will probably have a lot of RAM that it uses for caching the directory contents for fast performance. It will also probably have very fast disks and a fast processor, but it probably doesn't need a large-screen monitor and expensive graphics support. A client computer, on the other hand, might be on an employee's desk, probably optimized for the type of work that the employee does. Rather than putting a copy of the corporate directory on every employee's workstation, it's a better idea to maintain the directory centrally on a server (or replicated set of servers).

The LDAP protocol is a message-oriented protocol. The client constructs an LDAP message containing a request and sends it to the server. The server processes the request and sends the result or results back to the client as a series of LDAP messages.

For example, when an LDAP client searches the directory for a specific entry, it constructs an LDAP search request message and sends it to the server. The server retrieves the entry from its database and sends it to the client in an LDAP message. It also returns a result code to the client in a separate LDAP message. This interaction is shown in Figure 3.1.

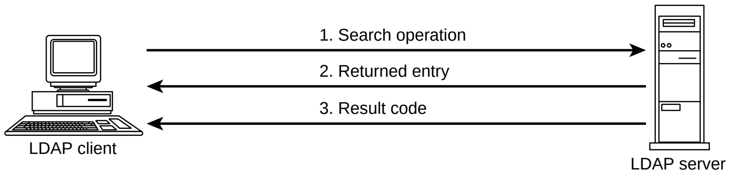


FIGURE 3.1 *A client retrieves a single entry from the directory.*

If the client searches the directory and multiple matching entries are found, the entries are sent to the client in a series of LDAP messages, one for each entry. The results are terminated with a result message, which contains an overall result for the search operation, as shown in Figure 3.2.

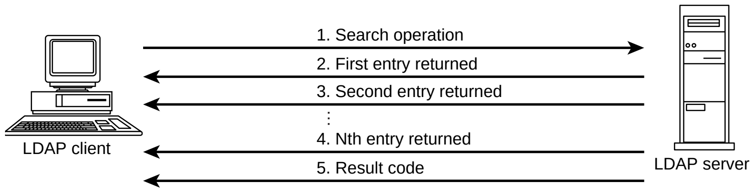


FIGURE 3.2 *A client searches the directory, and multiple entries are returned.*

Because the LDAP protocol is message-based, it also allows the client to issue multiple requests at once. Suppose, for example, a client might issue two search requests simultaneously. In LDAP, the client would generate a unique message ID for each request; returned results for specific request would be tagged with its message ID, allowing the client to sort out multiple responses to different requests arriving out of order or at the same time. In Figure 3.3, the client has issued two search requests simultaneously. The server processes both operations and returns the results to the client.

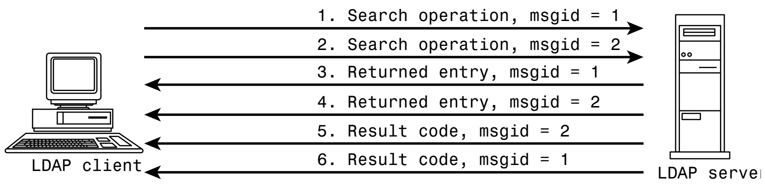

1. Search operation, msgid = 1
2. Search operation, msgid = 2
3. Returned entry, msgid = 1
4. Returned entry, msgid = 2
5. Result code, msgid = 2
6. Result code, msgid = 1
LDAP client
LDAP serve

FIGURE 3.3 *A client issues multiple LDAP search requests simultaneously.*

Notice that in Figure 3.3 the server sends the final result code of message ID 2 to the client before it sends the final result code from message ID 1. This is perfectly acceptable and happens quite frequently. These details are typically hidden from the programmer by an LDAP SDK. Programmers writing an LDAP application don't need to be concerned with sorting out these results; the SDKs take care of this automatically.

Allowing multiple concurrent requests "in flight" allows LDAP to be more flexible and efficient than protocols that operate in a "lock-step" fashion (for example, HTTP). With a lock-step protocol, each client request must be answered by the server before another may be sent. For example, an HTTP client program—such as a Web browser that wants to download multiple files concurrently—must open one connection for each file. LDAP, on the other hand, can manage multiple operations on a single connection, reducing the maximum number of concurrent connections a server must be prepared to handle.

The LDAP Protocol Operations

LDAP has nine basic protocol operations, which can be divided into three categories:

- *Interrogation operations: search, compare.* These two operations allow you to ask questions of the directory.
- *Update operations: add, delete, modify, modify DN (rename).* These operations allow you to update information in the directory.
- *Authentication and control operations: bind, unbind, abandon.* The bind operation allows a client to identify itself to the directory by providing an identity and authentication credentials; the unbind operation allows the client to terminate a session; and the abandon operation allows a client to indicate that it is no longer interested in the results of an operation it had previously submitted.

We will discuss each of the individual protocol operations when we describe the LDAP functional model later in this chapter.

A typical complete LDAP client/server exchange might proceed as depicted in Figure 3.4.

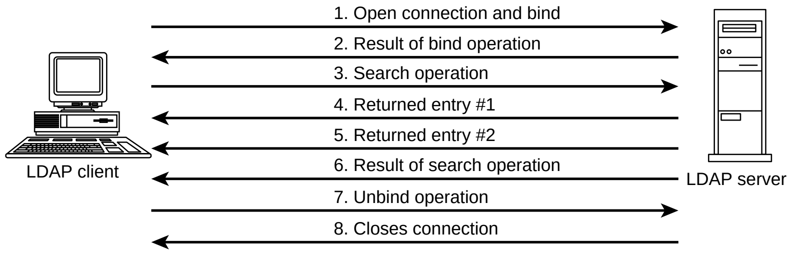


FIGURE 3.4 *A typical LDAP exchange.*

In Figure 3.4, an LDAP client and server perform the following steps:

Step 1: The client opens a TCP connection to an LDAP server and submits a bind operation. This bind operation includes the name of the directory entry the client wants to authenticate as, along with the credentials to be used when authenticating. Credentials are often simple passwords, but they might also be digital certificates used to authenticate the client.

Step 2: After the directory has verified the bind credentials, it returns a success result to the client.

Step 3: The client issues a search request.

Steps 4 and 5: The server processes this request, which results in two matching entries.

Step 6: The server sends a result message.

Step 7: The client then issues an unbind request, which indicates to the server that the client wants to disconnect.

Step 8: The server obliges by closing the connection.

By combining a number of these simple LDAP operations, directory-enabled clients can perform complex tasks that are useful to their users. For example, as shown in Figure 3.5, an electronic mail client such as Netscape Communicator can look up mail recipients in a directory, helping a user address an email message. It can also use a digital certificate stored in the directory to digitally sign and encrypt an outgoing message. Behind the scenes, the user’s email program performs a number of directory operations that allow the mail to be addressed, signed, and encrypted. But from the user’s point of view, it is all taken care of automatically.

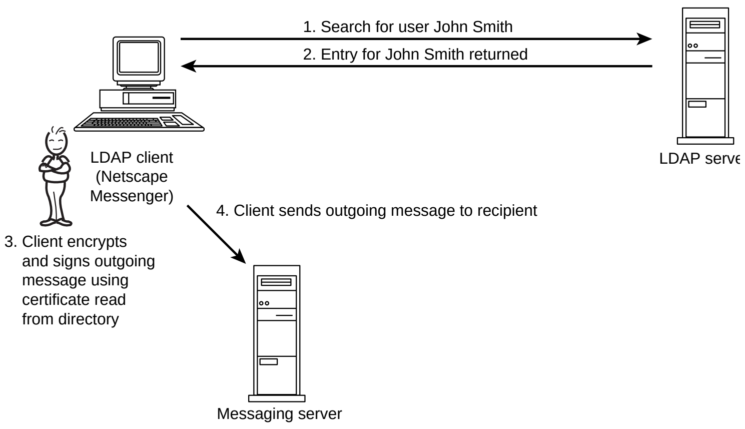

1. Search for user John Smith
2. Entry for John Smith returned
LDAP client (Netscape Messenger)
LDAP serve
4. Client sends outgoing message to recipient
3. Client encrypts and signs outgoing message using certificate read from directory
Messaging server

FIGURE 3.5 *A directory-enabled application performing a complex task.*

Although end-user applications can certainly be directory enabled, they are not the only kind of directory enabled applications. Server-based applications often benefit from being directory enabled, too. For example, the Netscape Messaging Server can use an LDAP directory when routing incoming electronic mail, as shown in Figure 3.6.

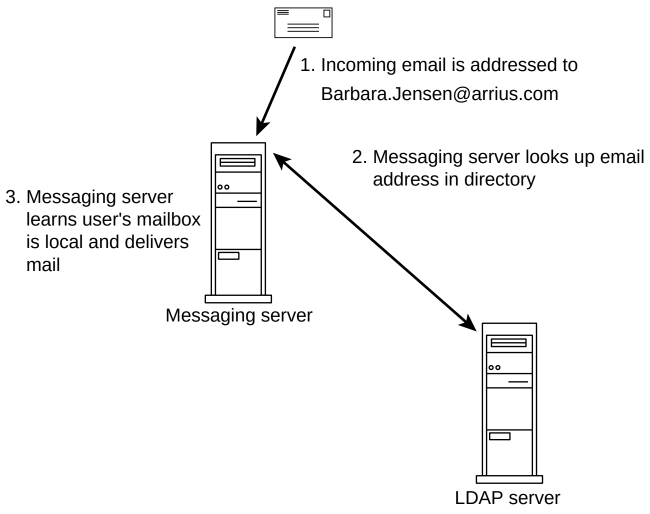

1. Incoming email is addressed to Barbara.Jensen@arrius.com
2. Messaging server looks up email address in directory
3. Messaging server learns user's mailbox is local and delivers mail
Messaging server
LDAP server

FIGURE 3.6 *A directory-enabled server application.*

These are just two examples of how directory-enabled applications can leverage the power of the directory to add functionality and ease of management. We will see more and more of this in the future.

In addition to providing the nine basic protocol operations, LDAP version 3 is designed to be extensible via three methods:

- *LDAP extended operations*—A new protocol operation, like the nine basic LDAP operations discussed earlier. If in the future there is a need for a new operation, it can be defined and made standard without requiring changes to the core LDAP protocol. An example of an extended operation is `StartTLS`, which indicates to the server that the client wants to begin using transport layer security (TLS) to encrypt and optionally authenticate the connection.
- *LDAP controls*—Extra pieces of information carried along with existing LDAP operations, altering the behavior of the operation. For example, the `manageDSAIT` control is sent along with a modify operation when the client wants to manipulate certain types of meta-information stored in the directory (this meta-information is normally hidden from users of the directory). In the future, additional controls may be defined that alter the behavior of existing LDAP operations in useful ways.
- *Simple Authentication and Security Layer (SASL)*—A framework for supporting multiple authentication methods. By using the SASL framework to implement authentication, LDAP can easily be adapted to support new, stronger authentication methods. SASL also supports a framework for clients and servers to negotiate lower-layer security mechanisms, such as encryption of all client/server traffic. SASL is not specific to LDAP, though; its general framework can be adapted to a wide range of Internet protocols.

How does an LDAP client know whether a particular LDAP extended operation, LDAP control, or SASL mechanism is supported by the server it is in contact with? LDAP version 3 servers are required to advertise the extended operations, controls, and SASL mechanisms they support in a special directory entry called the root DSE. The root DSE contains a number of attributes that describe the capabilities and configuration of the particular LDAP server.

Standardization of LDAP Extensions

How does an enhancement such as a new extended operation, LDAP control, or SASL authentication method become a standard? It goes through a standardization process in the Internet Engineering Task Force (IETF).

First, the enhancement is described in a document called an Internet Draft. The draft is reviewed by participants in the IETF, changes and improvements are made, and revised drafts are submitted by the authors. Once there is consensus in the IETF that the enhancement is a good idea, and is soundly designed, the document becomes a Proposed Internet Standard. It then goes through more peer review, becomes a Draft Standard, and finally becomes a full Internet Standard. However, multiple interoperable implementations are required before a document makes it all the way through the standards process to prove that implementation is feasible.

At all times during this process, the document is freely available on the Internet for anyone to download, read, comment on, and implement. The whole process is designed to encourage open development of standards and thorough peer review, without bogging it down with a complex standardization process. This approach has worked quite well historically; it's how the Internet was designed and built!

The LDAP Protocol on the Wire

What information is actually transmitted back and forth between LDAP clients and servers? We won't go into a great deal of detail here because this book isn't about protocol design, but we do feel that there are a few things you might want to know about the LDAP wire protocol.

LDAP uses a simplified version of the Basic Encoding Rules (BER). BER is a set of rules for encoding various data types, such as integers and strings, in a system-independent fashion. It also defines ways of combining these primitive data types into useful structures such as sets and sequences. The simplified BER that LDAP uses is often referred to as lightweight BER (LBER). LBER does away with many of the more esoteric data types that BER can represent, and instead it represents most items as simple strings.

Because LDAP is not a simple string-based protocol like HTTP, you can't simply telnet to the LDAP port on your server and start typing commands. The LDAP protocol primitives are not simple strings, so it's difficult, if not impossible, to converse with an LDAP server by typing at it. If you are familiar with text-based Internet protocols such as POP, IMAP, and SMTP, this may seem like an unfortunate limitation. On the other hand, DNS, a very successful distributed system, uses a protocol that has nontextual protocol primitives. The presence of universal implementations of client libraries for both DNS and LDAP makes this limitation less problematic.

The LDAP Models

LDAP defines four basic models that fully describe its operation, what data can be stored in LDAP directories, and what can be done with that data. These models are described in the following sections.

The LDAP Information Model

The LDAP information model defines the types of data and basic units of information you can store in your directory. In other words, the LDAP information model describes the building blocks you can use to create your directory.

The basic unit of information in the directory is the *entry*, a collection of information about an object. Often, the information in an entry describes some real-world object such as a person, but this is not required by the model. If you look at a typical directory, you'll find thousands of entries that correspond to people, departments, servers, printers, and other real-world objects in the organization served by the directory. Figure 3.7 shows a portion of a typical directory, with objects corresponding to some of the real-world objects in the organization.

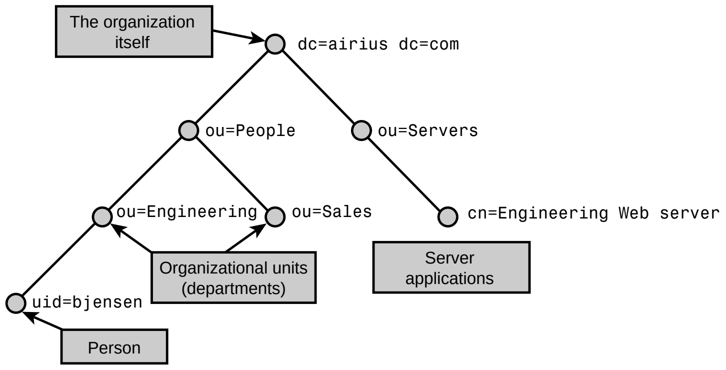
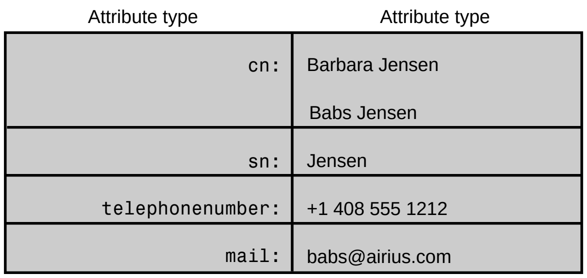


FIGURE 3.7 *Part of a typical directory.*

An entry is composed of a set of *attributes*, each of which describes one particular trait of the object. Each attribute has a *type* and one or more *values*. The type describes the kind of information contained in the attribute, and the value contains the actual data. For example, Figure 3.8 zooms in on an entry describing a person, with attributes for the person's full name, surname (last name), telephone number, and email address.



Attribute type	Attribute type
cn:	Barbara Jensen Babs Jensen
sn:	Jensen
telephonenumber:	+1 408 555 1212
mail:	babs@airius.com

FIGURE 3.8 *A directory entry showing attribute types and values.*

LDIF

Throughout this book you'll see directory entries shown in the LDIF text format. This is a standard way of representing directory data in a textual format that is used when exporting data from and importing data into a directory server. LDIF files consist solely of ASCII text, making it possible to pass them through email systems that are not 8-bit clean. Because it's a legible and text-based format, LDIF is used in this book when we want to represent a directory entry.

Let's look at a typical directory entry represented in LDIF:

```
dn: uid=bjensen, dc=airius, dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Barbara Jensen
cn: Babs Jensen
sn: Jensen
mail: bjensen@airius.com
telephoneNumber: +1 408 555 1212
description: A big sailing fan.
```

An LDIF entry consists of a series of lines. It begins with dn:, followed by the distinguished name of the entry, all on one line. After this come the attributes of the entry, with one attribute value per line.

continues

continued

Each attribute value is preceded by the attribute type and a colon (`:`). The order of the attribute values is not important; however, it makes the entry more readable if you place all the `objectclass` *values first and keep all the attribute values of a given attribute type together.*

There are other, more-sophisticated things you can do with LDIF, including representing modifications to be applied to directory entries. We'll cover these more-sophisticated uses of LDIF later in this chapter.

Attribute types also have an associated *syntax*, which describes the types of data that may be placed in attribute values of that type. It also defines how the directory compares values when searching. For example, the `caseIgnoreString` syntax specifies that strings are ordered lexicographically and that case is not significant when searching or comparing values. Hence, the values `Smith` and `smith` are considered equivalent if the syntax is `caseIgnoreString`. The `caseExactString` syntax, by contrast, specifies that case *is* significant when comparing values. Thus, `Smith` and `smith` are not equivalent values if the syntax is `caseExactString`.

With both `caseIgnoreString` and `caseExactString` syntaxes, trailing and leading spaces are not significant, and multiple spaces are treated as a single space when searching or comparing. The rules for how attribute values of a particular syntax are compared are referred to as *matching rules*.

X.500 servers typically support a number of different syntaxes that are either some primitive type (such as a string, integer, or Boolean value) or some complex data type built from sets or sequences of the primitive types. LDAP servers typically avoid this complicated abstraction layer and support only the primitive types. The Netscape Directory Server, for example, supports the case-ignore and case-exact string, distinguished name, integer, and binary syntaxes. However, a plug-in interface allows new syntaxes to be defined.

Attributes are also classified broadly in two categories: user and operational. *User attributes*, the "normal" attributes of an entry, may be modified by the users of the directory (with appropriate permissions). *Operational attributes* are special attributes that either modify the operation of the directory server or reflect the operational status of the directory. An example of an operational attribute is the `modifytimestamp` attribute, which is automatically maintained by the directory and reflects the time that the entry was last modified. When an entry is sent to a client, operational attributes are not included unless the client requests them by name.

Attribute values can also have additional constraints placed on them. Some server software allows the administrator to declare whether a given attribute type may hold multiple values or if only a single attribute value may be stored.

For example, the `givenName` attribute is typically multivalued, for when a person may want to include more than one given name (e.g., Jim and James). On the other hand, an attribute holding an employee ID number is likely to be single-valued.

The other type of attribute constraint is the size of the attribute. Some server software allows the administrator to set the maximum size value that a given attribute may hold. This can be used to prevent users of the directory from using unreasonable amounts of storage.

Maintaining Order: Directory Schemas

Any entry in the directory has a set of attribute types that are required and a set of attribute types that are allowed. For example, an entry describing a person is required to have a `cn` (common name) attribute and an `sn` (surname) attribute. A number of other attributes are allowed, but not required, for person entries. Any other attribute type not explicitly required or allowed is prohibited.

The collections of all information about required and allowed attributes are called the *directory schemas*. Directory schemas, which are discussed in detail in Chapter 7, "Schema Design," allow you to retain control and maintain order over the types of information stored in your directory.

In summary, the LDAP information model describes *entries*, which are the basic building blocks of your directory. Entries are composed of attributes, which are composed of an attribute type and one or more values. Attributes may have constraints that limit the type and length of data placed in attribute values. The directory schemas place restrictions on the attribute types that must be or are allowed to be contained in an entry.

However, building blocks aren't very interesting unless you can actually use them to build something. The rules that govern how you arrange entries in a directory information tree are what comprise the LDAP naming model.

The LDAP Naming Model

The LDAP naming model defines how you organize and refer to your data. In other words, it describes the types of structures you can build out of your individual building blocks, which are the directory entries. After you've arranged your entries into a logical structure, the naming model also tells you how you can refer to any particular directory entry within that structure.

The flexibility afforded by the LDAP naming model allows you to place your data in the directory in a way that is easy for you to manage. For example, you might choose to create one container that holds all the entries describing people in your organization, and another container that holds all your groups. Or, you might choose to arrange your directory in a way that reflects the hierarchy of your organizational structure. Chapter 8, "Namespace Design," guides you in making good choices when you design your directory hierarchy or namespace.

The LDAP naming model specifies that entries are arranged in an inverted tree structure, as shown in Figure 3.9.

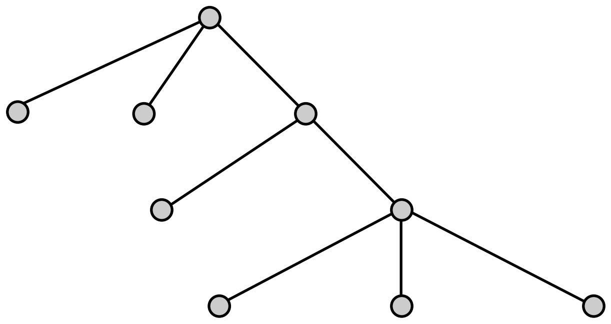


FIGURE 3.9 *A directory tree.*

Readers familiar with the hierarchical file system used by UNIX systems will note its similarities to this directory structure. Such a file system consists of a set of directories and files; each directory may have zero or more files or directories beneath it. Part of a typical UNIX file system is shown in Figure 3.10.

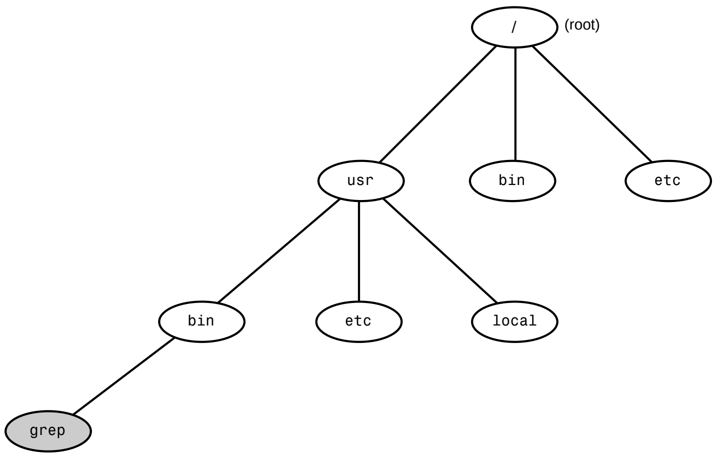


FIGURE 3.10 *Part of a typical UNIX file system.*

There are three significant differences between the UNIX file system hierarchy and the LDAP directory hierarchy, however.

The first major difference between the two models is that there isn't really a root entry in the LDAP model. A file system, of course, has a root directory, which is the common ancestor of all files or directories in the file system hierarchy. In an LDAP directory hierarchy, on the other hand, the root entry is conceptual—it doesn't exist as an entry you can place data into. There is a special entry called the root DSE that contains server-specific information, but it is not a normal directory entry.

The second major difference is that in an LDAP directory every node contains data, and any node can be a container. This means that any LDAP entry may have child nodes underneath it. Contrast this with a file system, in which a given node is either a file or a directory, but not both. In the file system, only directories may have children, and only files may contain data.

Another way of thinking of this is that an entry in a directory may be both a file and a directory simultaneously. The directory tree shown in Figure 3.11 illustrates this concept. Notice how the entries `dc=airius`, `dc=com`, `ou=People`, and `ou=Devices` all contain data (attributes) but are also containers with child nodes beneath them.

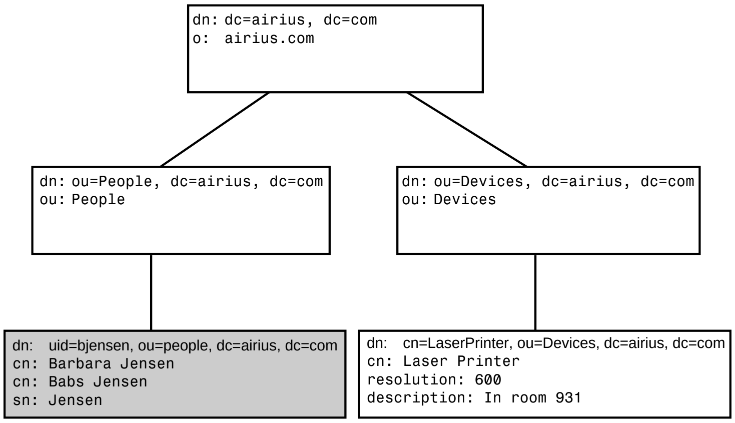

dn: dc=airius, dc=com
o: airius.com
dn: ou=People, dc=airius, dc=com
ou: People
dn: ou=Devices, dc=airius, dc=com
ou: Devices
dn: uid=bjensen, ou=people, dc=airius, dc=com
cn: Barbara Jensen
cn: Babs Jensen
sn: Jensen
dn: cn=LaserPrinter, ou=Devices, dc=airius, dc=com
cn: Laser Printer
resolution: 600
description: In room 931

FIGURE 3.11 *Part of a typical LDAP directory.*

The third and final difference between the file system hierarchy and LDAP hierarchy is how individual nodes in the tree are named. LDAP names are backward relative to file system names. To illustrate this, let's consider the names of the shaded nodes in Figures 3.10 and 3.11. In Figure 3.10, the shaded node is a file with a complete filename of `/usr/bin/grep`. Notice that if you read the filename from left to right, you move from the top of the tree (`/`) down to the specific file being named.

Contrast this with the name of the shaded directory entry in Figure 3.11. Its name is `uid=bjensen, ou=people, dc=airius, dc=com`. Notice that, if you read from left to right, you move from the specific entry being named back up toward the top of the tree.

As you've seen, LDAP supports a hierarchical arrangement of directory entries. It does not, however, mandate any particular type of hierarchy. Just as you're free to arrange your file system in a way that makes sense to you and is easy for you to manage, you're free to construct any type of directory hierarchy you desire. Of course, some directory structures are better than others, depending on your particular situation; we'll cover the topic of designing your directory namespace in Chapter 8.

The one exception to this freedom is if your LDAP directory service is actually a front end to an X.500 service. The X.500 naming model is much more restrictive than the LDAP naming model. In the X.500 1993 standard, directory structure rules limit the types of hierarchies you can create. The standard accomplishes this by specifying what types of "objectclasses" may be direct children of an entry. For example, in the X.500 model, only entries representing countries, localities, or organizations may be placed at the root of the directory tree. The LDAP naming model, on the other hand, does not limit the tree structure in any way; any type of entry may be placed anywhere in the tree.

In addition to specifying how you arrange your directory entries into hierarchical structures, the LDAP naming model describes how you refer to individual entries in the directory. We mentioned this briefly when we were discussing the similarities and differences between file system hierarchy and LDAP directory hierarchy. Now let's go into more detail about naming.

Why Is Naming Important?

A naming model is needed so that you can give a unique name to any entry in the directory, allowing you to refer to any entry unambiguously. In LDAP, distinguished names (DNs) are how you refer to entries.

Like file system pathnames, the name of an LDAP entry is formed by connecting in a series all the individual names of the parent entries back to the root. For example, look back at the directory tree shown in Figure 3.11. The shaded entry's name is `uid=bjensen, ou=People, dc=airius, dc=com`. Reading this name from left to right, you can trace the path from the entry itself back to the root of the directory tree. The individual components of the name are separated by commas. Spaces after the commas are optional, so the following two distinguished names are equivalent:

```
uid=bjensen, ou=People, dc=airius, dc=com
uid=bjensen,ou=People,dc=airius,dc=com
```

In any entry's DN, the leftmost component is called the relative distinguished name (RDN). Among a set of peer entries (those which share a common immediate parent), each RDN must be unique. This rule, when recursively applied to the entire directory tree, ensures that no two entries may have the same DN. If you attempt to add two entries with the same name, the directory server will reject the attempt to add the second entry; this is similar to a UNIX host, which will reject an attempt to create a file with the same name as an existing file within a directory.

Note that RDNs have to be unique only if they share a common immediate parent. Look at the tree in Figure 3.12. Even though there are two entries with the RDN `cn=John Smith` in the directory, they are in different subtrees—making the tree completely legal. Whether this is a good way to construct your directory is another matter, one we will address in Chapter 8.

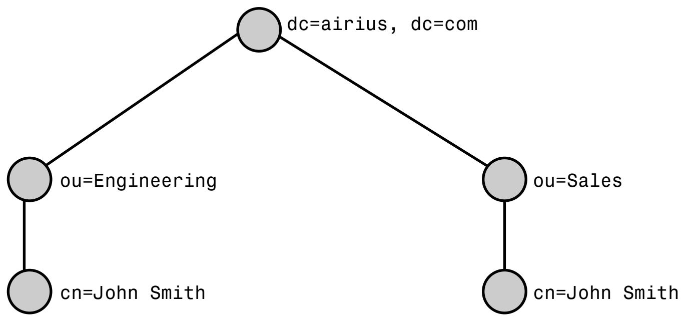


FIGURE 3.12 *Entries with the same RDNs are permitted if they are in different parts of the tree.*

Messy RDN Topics: Multivalued RDNs and Quoting

You've probably noticed that each RDN we've shown is composed of two parts: an attribute name and a value, separated by an equal sign (=). It's also possible for an RDN to contain more than one such name / value pair. Such a construction, called a *multivalued RDN,* looks like the following:

```
cn=John Smith + mail=jsmith@airius.com
```

The RDN for this entry consists of two `attribute=value` pairs: `cn=John Smith` and `mail=jsmith@airius.com`.

Multivalued RDNs can be used to distinguish RDNs that would otherwise be the same. For example, if there is more than one John Smith in the same container, a multivalued RDN would allow you to assign unique RDNs to each entry. However, you should generally avoid using multivalued RDNs in your directory. They tend to clutter your namespace, and there are better ways to arrive at unique names for your entries. (Approaches for uniquely naming your entries are discussed in Chapter 8.)

Recall that the individual RDNs in a DN are separated by commas. You may be curious how to proceed if an RDN contains a comma. How do you tell which commas are contained in RDNs, and which commas separate the individual RDN components? For example, what if you have an entry named `o=United Widgets,Ltd.` in your directory?

If you have DNs like this in your directory, you must escape all literal commas (those within an RDN) with a backslash. In our example, then, the DN would be

```
o=United Widgets\, Ltd., c=GB
```

Certain other characters must also be quoted when they appear within a component of a DN. Table 3.1 shows all the characters that must be quoted, according to the LDAPv3 specification.

TABLE 3.1 CHARACTERS REQUIRING QUOTING WHEN CONTAINED IN DISTINGUISHED NAMES

Character	Decimal Value	Escaped as
Space at the beginning or end of a DN or RDN	32	\<space>
Octothorpe (#) character at the beginning of a DN or RDN	35	\#
Comma (,)	44	\,
Plus sign (+)	43	\+
Double-quote (")	34	\"
Backslash (\)	92	\\
Less-than symbol (<)	60	\<
Greater-than symbol (>)	62	\>
Semicolon (;)	59	\;

Aliases

Alias entries in the LDAP directory allow one entry to point to another one, which means you can devise structures that are not strictly hierarchical. Alias entries perform a function like symbolic links in the UNIX file system or shortcuts in the Windows 95/NT file system. In Figure 3.13, the dotted entry is an alias entry pointing to the "real" entry.

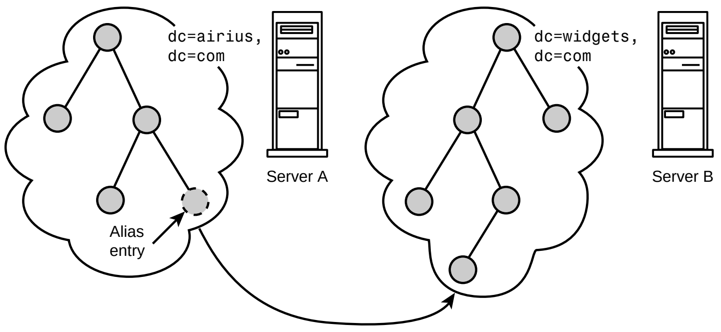


FIGURE 3.13 *An alias entry points to another directory entry.*

To create an alias entry in the directory, you must first create an entry with the object class `alias` and an attribute named `aliasedObjectName`. The value of the `aliasedObjectName` attribute must be the DN of the entry you want this alias to point to.

Not all LDAP directory servers support aliases. Because aliases can point to any directory entry, even one that is on a different server, aliases may exact a severe performance penalty. Consider the directory trees shown in Figure 3.13. Alias entries in one of the trees point to entries in the other tree, which is housed in another server. To support searching across the entire `ou=Marketing, dc=airius,dc=com` tree, Server A must contact Server B each time an alias entry is encountered while servicing the search operation. This can significantly slow down searches, which is the main reason certain software does not support aliases.

Often, the goals you are trying to achieve by using aliases can be met by using referrals, or by placing LDAP URLs in entries that clients can use to chase down the referred-to information. More information on using referrals can be found in Chapter 9, “Topology Design.”

The LDAP Functional Model

Now that you understand the LDAP information and naming models, you need some way to actually access the data stored in the directory tree. The LDAP functional model describes the operations that you can perform on the directory using the LDAP protocol.

The LDAP functional model consists of a set of operations divided into three groups. The *interrogation operations* allow you to search the directory and retrieve directory data. The *update operations* allow you to add, delete, rename, and change directory entries. The *authentication and control operations* allow clients to identify themselves to the directory and control certain aspects of a session.

In addition to these three main groups of operations, version 3 of the LDAP protocol defines a framework for adding new operations to the protocol via LDAP *extended operations*. Extended operations allow the protocol to be extended in an orderly fashion to meet new marketplace needs as they emerge. Extended operations were described earlier in this chapter.

The LDAP Interrogation Operations

The two LDAP interrogation operations allow LDAP clients to search the directory and retrieve directory data.

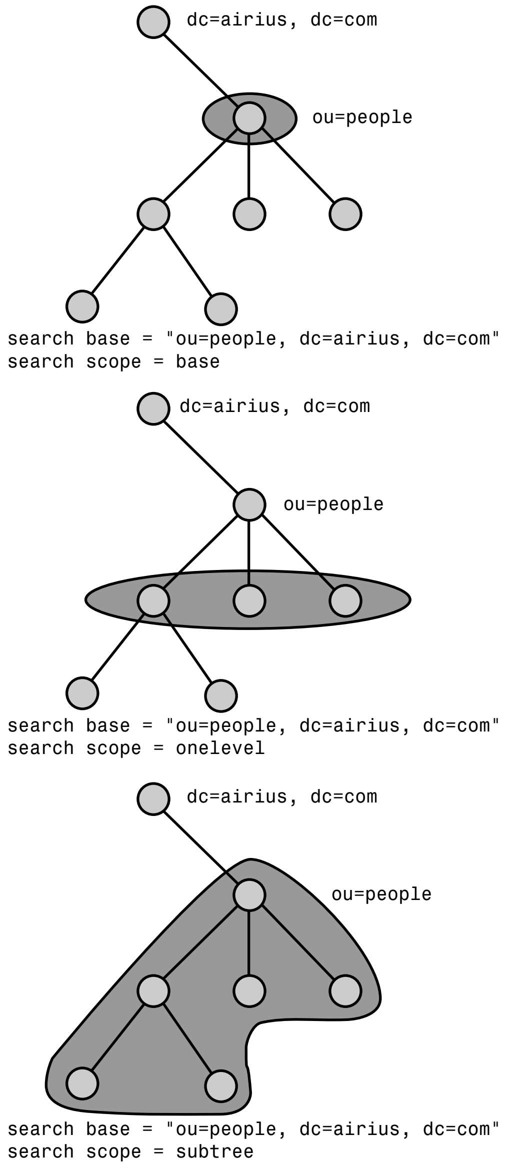
The LDAP search operation is used to search the directory for entries and retrieve individual directory entries. There is no LDAP read operation. When you want to read a particular entry, you must use a form of the search operation in which you restrict your search to just the entry you want to retrieve. Later in the chapter we'll discuss how to search the directory and retrieve specific entries, as well as how to list all the entries at a particular location in the tree.

The LDAP search operation requires eight parameters. The first parameter is the base object for the search. This parameter, expressed as a DN, indicates the top of the tree you want to search.

The second parameter is the scope. There are three types of scope. A scope of `subtree` indicates that you want to search the entire subtree from the base object all the way down to the leaves of the tree. A scope of `onelevel` indicates that you want to search only the immediate children of the entry at the top of the search base. A scope of `base` indicates that you want to limit your search to just the base object; this is used to retrieve one particular entry from the directory. Figure 3.14 depicts the three types of search scope.

The third search parameter, `derefAliases`, tells the server whether aliases should be dereferenced when performing the search. There are four possible values for this parameter's value:

- `neverDerefAliases`—Do not dereference aliases in searching or in locating the base object of the search.
- `derefInSearching`—Dereference aliases in subordinates of the base object in searching, but not in locating the base object of the search.
- `derefFindingBaseObject`—Dereference aliases in locating the base object of the search, but not when searching subordinates of the base object.
- `derefAlways`—Dereference aliases both in searching and in locating the base object of the search.



```
search base = "ou=people, dc=airius, dc=com"
search scope = base

search base = "ou=people, dc=airius, dc=com"
search scope = onelevel

search base = "ou=people, dc=airius, dc=com"
search scope = subtree
```

FIGURE 3.14 *The three types of search scope.*

The fourth search parameter is the size limit. This parameter tells the server that the client is interested in receiving only a certain number of entries. For example, if the client passes a size limit of 100, but the server locates 500 matching entries, only the first 100 will be returned to the client, along with a result code of `LDAP_SIZELIMIT_EXCEEDED`. A size limit of 0 means that the client wants to receive all matching entries. (Note that servers may impose a maximum size limit that cannot be overridden by unprivileged clients.)

The fifth search parameter is the time limit. This parameter tells the server the maximum time in seconds that it should spend trying to honor a search request. If the time limit is exceeded, the server will stop processing the request and send a result code of `LDAP_TIMELIMIT_EXCEEDED` to the client. A time limit of 0 indicates that no limit should be in effect. (Note that servers may impose a maximum time limit that cannot be overridden by unprivileged clients.)

The sixth search parameter, the `attrsOnly` parameter, is a Boolean parameter. If it is set to true, the server will send only the attribute types to the client; attribute values will not be sent. This can be used if the client is interested in finding out which attributes are contained in an entry but not in receiving the actual values. If this parameter is set to false, attribute types and values are returned.

The seventh search parameter is the search filter, an expression that describes the types of entries to be returned. The filter expressions used in LDAP search operations are very flexible, and are discussed in detail in the next section.

The eighth and final search parameter is a list of attributes to be returned for each matching entry. You can specify that all attributes should be returned, or you can request that only specific attributes of an entry be returned. We'll focus on how to request specific attributes later in this chapter. First, though, let's look at the different types of LDAP filters you can use when searching the directory.

An LDAP filter is a Boolean combination of attribute-value assertions. An attribute value assertion consists of two parts: an attribute name and a value assertion, which you can think of as a value with wildcards allowed. The following sections look at the various types of search filters.

Equality Filters

An equality filter allows you to look for entries that exactly match some value. Here's an example:

```
(sn=smith)
```

This filter matches entries in which the sn (surname) attribute contains a value that is exactly smith. Because the syntax of the sn attribute is a case-ignore string, the case of the attribute and the filter is not important when locating matching entries.

Substring Filters

When you use wildcards in filters, they are called substring filters. Here's an example:

```
(sn=smith*)
```

This filter matches any entry that has an sn attribute value that begins with smith. Entries with a surname of Smith, Smithers, Smithsonian, and so on will be returned.

Wildcards may appear anywhere in the filter expression, so the filter

```
(sn=*smith)
```

matches entries in which the surname ends with smith (e.g., Blacksmith). The filter

```
(sn=smi*th)
```

matches entries in which the surname begins with smi and ends with th, and the filter

```
(sn=*smith*)
```

matches entries that contain the string smith in the surname attribute. Note that the wildcard character matches zero or more instances of any character, so the filter (sn=*smith*) would match the entry with the surname Smith as well as any surnames in which the string smith is embedded.

Approximate Filters

In addition to the equality and substring filters, servers support an approximate filter. For example, on most directory servers, the filter

```
(sn~=jensen)
```

returns entries in which the surname attribute has a value that sounds like jensen (for example, jenson). Exactly how the server implements this is particular to each vendor and the languages supported by the server. The Netscape Directory Server, for example, uses the metaphone algorithm to locate entries when an approximate filter is used. Internationalization also throws an interesting wrinkle into the concept of approximate matching; each language may need its own particular sounds-like algorithm.

”Greater Than or Equal To” and “Less Than or Equal To” Filters

LDAP servers also support “greater than or equal to” and “less than or equal to” filters on attributes that have some inherent ordering. For example, the filter

```
(sn<=Smith)
```

returns all entries in which the surname is less than or equal to `Smith` lexicographically. The ordering used depends on the ordering rules for the syntax of any particular attribute. The `sn` attribute, which has the case-ignore string syntax, is ordered lexicographically without respect to case. An attribute that has integer syntax would be ordered numerically. Attributes that have no inherent ordering, such as JPEG photos, cannot be searched for with this type of filter.

If you find that you need a greater than or less than filter (without the equals part), note that “greater than” is the complement of “less than or equal to” and “less than” is the complement of “greater than or equal to.” In other words,

```
(age>21)
```

which is equivalent to

```
(!(age<=21))
```

Similarly, the filter

```
(age<21)
```

which is also not a valid LDAP filter, is equivalent to

```
(!(age>=21))
```

In these cases, `!` is the negation operator, which we will discuss in more detail shortly.

Presence Filters

Another type of search filter is the presence filter. It matches any entry that has at least one value for the attribute. For example, the filter

```
(telephoneNumber=*)
```

matches all entries that have a telephone number.

Extensible Matching

The last type of search filter is the extensible match filter. It is only supported by LDAPv3 servers.

The purpose of an extensible match filter is to allow new matching rules to be implemented in servers and used by clients. Recall our earlier example involving the `caseIgnoreString` and `caseExactString` syntaxes. Each syntax has an associated method for comparing values, depending on whether case is to be considered significant when comparing values. When new attribute syntaxes are developed, it may also be necessary to define a new way of comparing values for equality. Extensible matching also allows language-specific matching rules to be defined so that values in languages other than English can be meaningfully compared.

As an added benefit, extensible matching allows you to specify that the attributes that make up the DN of the entry should be searched. So, for example, using extensible matching you can locate all the entries in the directory that contain the attribute value assertion `ou=Engineering` anywhere in their DN.

To see the usefulness of this feature, consider an entry named `cn=Babs Jensen, ou=Engineering, dc=bigco, dc=com`. Suppose you're interested in finding all the Babs Jensens in the engineering department, and you search from the top of your subtree using a search filter like `(&(cn=Babs Jensen)(ou=engineering))`. Normally, this filter would not find Babs's entry unless it explicitly contained an `ou` attribute with a value of `engineering`. Using extensible matching feature, you can treat the attribute values contained in a DN as attribute values of the entry that can match the search.

The syntax of an extensible matching filter is a bit complicated. It consists of five parts, three of which are optional. These parts are

- An attribute name. If omitted, any attribute type that supports the given matching rule is compared against the value.
- The optional string `:dn`, which indicates that the attributes forming the entry's DN are to be treated as attributes of the entry during the search.
- An optional colon and matching rule identifier that identifies the particular matching rule to be used. If no matching rule is provided, the default matching rule for the attribute being searched should be used. If the attribute name is omitted, the colon and matching rule must be present.
- The string `":="`.
- An attribute value to be compared against.

Formally, the grammar for the extensible search filter is

```
attr [":dn"] [":" matchingrule] ":=" value
```

The elements of this syntax are as follows:

`attr` is an attribute name.

`matchingrule` is usually given by an Object Identifier (OID), although if a descriptive name has been assigned to the matching rule, that may be used as well. The OIDs of the matching rules supported by your directory server will be given in its documentation.

`value` is an attribute value to be used for comparison.

Object Identifiers

Object Identifiers, commonly referred to as OIDs, are unique identifiers assigned to objects. They are used to uniquely identify many different types of things, such as X.500 directory object and attribute types. In fact, just about everything in the X.500 directory system is identified by an OID. OIDs are also used to uniquely identify objects in other protocols, such as the Simple Network Management Protocol (SNMP).

OIDs are written as strings of dotted decimal numbers. Each part of an OID represents a node in a hierarchical OID tree. This hierarchy allows an arbitrarily large number of objects to be named, and it supports delegation of the namespace. For example, all the user attribute types defined by the X.500 standards begin with 2.5.4. The `cn` attribute is assigned the OID 2.5.4.3, and the `sn` attribute is assigned the OID 2.5.4.4.

An individual subtree of the OID tree is called an *arc*. Individual arcs may be assigned to organizations, which can then further divide the arc into *subarcs*, if so desired. For example, Netscape Communications has been assigned an arc of the OID namespace for its own use. Internally, it has divided that arc into a number of subarcs for use by the various product teams. By delegating the management of the OID namespace in this fashion, conflicts can be avoided.

The X.500 protocol makes extensive use of OIDs to uniquely identify various protocol elements. LDAP, on the other hand, favors short, textual names for things: `cn` to describe the common name attribute and `person` to identify the `person` object class, for example. To maintain compatibility with X.500, LDAP allows a string representation of an OID to be used interchangeably with the short name for the item. For example, the search filters `(cn=Barbara Jensen)` and `(2.5.4.3=Barbara Jensen)` are equivalent. Unless you are working with an LDAP-based gateway into an X.500 system, you should generally avoid using OIDs in your directory-enabled applications.

Although LDAP largely does away with the mandatory use of OIDs, you will see them from time to time, especially if you use extensible matching rules or if you design your own schema extensions. The topic of extending your directory schema is discussed in Chapter 7.

Let's look at some examples of extensible matching filters.

The following filter specifies that the all entries in which the `cn` attribute matches the value `Barbara Jensen` should be returned:

```
(cn:1.2.3.4.5.6:=Barbara Jensen)
```

When comparing values, the matching rule given by the OID 1.2.3.4.5.6 should be used.

The following filter specifies that all entries that contain the string `jensen` in the surname should be returned:

```
(sn:dn:1.2.3.4.5.7:=jensen)
```

The `sn` attributes within the DN are also searched. When comparing values, the matching rule given by the OID 1.2.3.4.5.7 should be used.

The following filter returns any entries in which the `o` (organization) attribute exactly matches `Airius` and any entries in which `o=Airius` is one of the components of the DN:

```
(o:dn:=Airius)
```

The following filter returns any entries in which a DN component with a syntax appropriate to the given matching rule matches `Airius`:

```
(:dn:1.2.3.4.5.8:=Airius)
```

The matching rule given by the OID 1.2.3.4.5.8 should be used.

Negation

Any search element can be negated by preceding the filter with an exclamation point (`!`). For example, the filter

```
(!(sn=Smith))
```

matches all entries in which the `sn` attribute does not contain the value `smith`, including entries with no `sn` attribute at all.

Combining Filter Terms

Filters can also be combined using AND and OR operators. The AND operator is signified by an ampersand (`&`) symbol, and the OR operator is signified by the vertical bar (`¦`) symbol. When combining search filters, you use *prefix* *notation*, in which the operator precedes its arguments. Those familiar with the "reverse polish notation" common on Hewlett-Packard calculators will be familiar with this concept (although reverse polish is a postfix notation, not a prefix notation like that used in LDAP search filters).

Let's look at some examples of combinations of LDAP search filters. The filter

```
(&(sn=Smith)(l=Mountain View))
```

matches all entries with a surname of `smith` that also have an `l` (locality) attribute of `Mountain View`. In other words, this filter will find everyone named Smith in the Mountain View location.

The filter

```
(¦(sn=Smith)(sn=Jones))
```

matches everyone with a surname of `Smith` or `Jones`.

You use parentheses to group more-complex filters to make the meaning of the filter unambiguous. For example, if you want to search the directory for all entries that have an email address but do not have a telephone number, you would use the filter

```
(&(mail=*)(!(telephoneNumber=*)))
```

Note that the parentheses bind the negation operator to the presence filter for telephone number.

Technically speaking, parentheses are always required, even if the filter consists of only a single term. Some LDAP software allows you to omit the enclosing parentheses and inserts them for you before sending the search request to the server. However, if you are developing your own software using one of the available SDKs, you need to include the enclosing parentheses.

Table 3.2 summarizes the six types of search filters and the three Boolean operators.

TABLE 3.2 TYPES OF LDAP SEARCH FILTERS

Filter Type	Format	Example	Matches
Equality	*(attr=value)*	`(sn=jensen)`	Surnames exactly equal to `jensen`
Substring	*(attr=[leading] *[any]*[trailing]*	`(sn=*jensen*)`	Surnames contain ing the string `jensen`
		`(sn=jensen*)`	Surnames starting with the string `jensen`
		`(sn=*jensen)`	Surnames ending with the string `jensen`
		`(sn=jen*s*en)`	Surnames starting with `jen`, containing an `s`, and ending with `en`
Approximate	*(attr~=value)*	`(sn~=jensin)`	Surnames approximately equal to `Jensin` (for example, surnames that sound like Jensin—note the misspelling)
Greater than or equal to	*(attr>=value)*	`(sn>=Jensen)`	Surnames lexicographically greater than or equal to `Jensen`
Less than or equal to	*(attr<=value)*	`(sn<=Jensen)`	Surnames lexicographically less than or equal to `Jensen`
Presence	*(attr=*)*	`(sn=*)`	All surnames
AND	*(&(filter1) (filter2)...))*	`(&(sn=Jensen) (objectclass=person))`	Entries with an object class of person and surname exactly equal to `Jensen`
OR	*(¦(filter1) (filter2)...))*	`(¦(sn~=Jensin) (sn=*jensin))`	Entries with a sur– name approximately equal to `Jensin` or common name ending in `jensin`
NOT	*(!(filter)*	`(!(mail=*))`	All entities without a mail attribute

Quoting in Search Filters

If you need to search for an attribute value that contains one of five specific characters, you need to substitute the character with an escape sequence consisting of a backslash and a two-digit hexadecimal sequence representing the character's value. Table 3.3 shows the characters that must be escaped, along with the escape sequence you should use for each.

TABLE 3.3 CHARACTERS THAT MUST BE ESCAPED IF USED IN A SEARCH FILTER

Character	Value (Decimal)	Value (Hex)	Escape Sequence
`*` (asterisk)	42	0x2A	`\2A`
`(` (left parenthesis)	40	0x28	`\28`
`)` (right parenthesis)	41	0x29	`\29`
`\` (backslash)	92	0x5C	`\5C`
`NUL` (the null byte)	0	0x00	`\00`

For example, if you want to search for all entries in which the `cn` attribute exactly matches the value `A*Star`, you would use the filter `(cn=A\2AStar)`.

Readers should note that the rules for quoting search filters and the rules for quoting distinguished names are different and not interchangeable.

Specifying Which Attributes Are to Be Returned

As previously mentioned, the last search parameter is a list of attributes to be returned for each matching entry. If this list is empty, all user attributes are returned. The special value `*` also means that all user attributes are to be returned, but it allows you to specify additional nonuser (operational) attributes that should be returned. (Without this special value, there would be no way to request all user attributes plus some operational attributes.)

If you want to retrieve no attributes at all, you should specify the attribute name `1.1` (there is no such attribute OID, so no attributes can be returned). Table 3.4 provides some examples of attribute lists and the attributes that are returned by the server.

TABLE 3.4 EXAMPLES OF ATTRIBUTE LISTS AND CORRESPONDING ATTRIBUTES RETURNED BY THE SERVER

Attribute List	Attributes Returned
`cn, sn, givenname`	`cn`, `sn`, and `givenname` only
`*`	All user attributes
`1.1`	No attributes
`modifiersname`	`modifiersname` only (an operational attribute)
`*, modifiersname`	All user attributes plus `modifiersname`

Common Types of Searches

Although the LDAP search operation is extremely flexible, there are some types of searches that you'll probably use more frequently than others:

- *Retrieving a single entry*—To retrieve a particular directory entry, you use a scope of `base`, a search base equal to the DN of the entry you want to retrieve, and a filter of `(objectclass=*)`. The filter, which is a presence filter on the `objectclass` attribute, will match any entry that contains at least one value in its `objectclass` attribute. Because every entry in the directory must have an `objectclass` attribute, this filter is guaranteed to match any directory entry. And because you've specified a scope of `base`, only one entry will be returned by the search (if the entry exists at all). This is how you use the search operation to read a particular entry.
- *Listing all entries directly below an entry*—To list all the directory entries at a particular level in the tree, you use the same filter `(objectclass=*)` as when retrieving a particular entry; but you use a scope of `onelevel` and a search base equal to the DN just above the level you want to list. All the entries immediately below the `searchbase` entry are returned. The search base entry itself is not returned in a `onelevel` search. (The search base entry *is* returned in a `base` or `subtree` search if it matches the search filter.)
- *Searching for matching entries within a subtree*—Another common search operation occurs within a subtree of the directory for all entries that match some search criteria. To perform this type of search, use a filter that selects the entries you are interested in retrieving—or `(objectclass=*)` if you want all entries—along with a scope of `subtree` and a search base equal to the DN of the entry at the top of the tree you want to search.

Hiding LDAP Filters from Users

You might justifiably be thinking that your users will never be able to understand LDAP filter syntax. The prefix notation it uses is hardly intuitive, after all! Bear in mind, though, that any good directory access GUI will hide the details of filter construction from end users.

Instead of requiring users to type raw LDAP filters, a set of pop-up menus and text boxes is typically used to allow the user to specify the search criteria, and the GUI client constructs the filter for the user. For example, in Figure 3.15, Netscape Communicator's Search window uses the provided information to construct the filter `(&(cn=*smith*)(l=*Dearborn*))`.

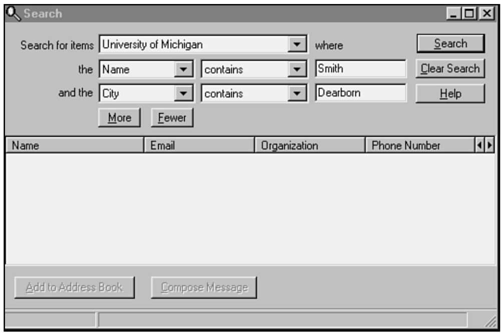


FIGURE 3.15 *A GUI interface for searching the directory.*

If you are a directory administrator, it's a good idea to become familiar with LDAP filter syntax. You can use this knowledge to provide complex "canned" queries for your end users, for example. Filter syntax also crops up in LDAP URLs and configuration files. Spending a little time understanding filter syntax is well worth the effort.

The Compare Operation

The second of the two interrogation operations, the LDAP compare operation, is used to check whether a particular entry contains a particular attribute value. The client submits a compare request to the server, supplying a DN, an attribute name, and a value. The server returns an affirmative response to the client if the entry named by the DN contains the given value in the given attribute type. If not, a negative response is returned.

It may seem odd that the compare operation even exists. After all, if you want to determine whether a particular entry contains a particular attribute value, you can just perform a search with a search base equal to the DN of the entry, a scope of `base`, and a filter expressing the test you want to make. If the entry is returned, the test was successful; if no entry is returned, the test was not successful.

The reasons that the compare operation exists are historical and related to LDAP's roots in X.500. There is only one case in which the compare and search operations behave differently. If a comparison is attempted on an attribute, but the attribute is not present in the entry, the compare operation will return a special indication to the client that the attribute does not exist. The search operation, on the other hand, would simply not return the entry. This ability to distinguish between "the entry has the attribute but contains no matching value" and "the entry does not have the attribute at all" may be convenient in some situations. The other advantage of the compare operation is that it is more compact in terms of the number of protocol bytes exchanged between the client and the server.

The LDAP Update Operations

There are four LDAP update operations: add, delete, rename (modify DN), and modify. These four operations define the ways that you can manipulate the data in your directory.

The add operation allows you to create new directory entries. It has two parameters: the distinguished name of the entry to be created and a set of attributes and attribute values that will comprise the new entry. In order for the add operation to complete successfully, four conditions must be met:

- The parent of the new entry must already exist in the directory.
- There must not be an entry of the same name.
- The new entry must conform to the schema in effect.
- Access control must permit the operation.

If all these conditions are met, the new entry is added to the directory.

The delete operation removes an entry from the directory. It has a single parameter: the DN of the entry to be deleted. In order for the delete operation to complete successfully, three conditions must be met:

- The entry to be deleted must exist.
- It must have no children.
- Access control must permit the entry to be deleted.

If these conditions are all met, the entry is removed from the directory.

The rename, or modify DN operation, is used to rename and/or move entries in the directory. It has four parameters: the DN of the entry to be renamed, the new RDN for the entry, an optional argument giving the new parent of the entry, and the delete-old-RDN flag. In order for the modify DN operation to succeed, the following conditions must be met:

- The entry being renamed must exist.
- The new name for the entry must not already be in use by another entry.
- Access control must permit the operation.

If all these conditions are met, the entry is renamed and/or moved.

If the entry is to be renamed but will still have the same parent entry, the new parent argument is left blank. Otherwise, the new parent argument gives the DN of the container where the entry is to be moved. The delete-old-RDN flag is a Boolean flag that specifies whether the old RDN of the entry is to be retained as an attribute of the entry or removed. Figures 3.16 through 3.20 show the various combinations of renaming and moving entries that can be performed with the modify DN operation.

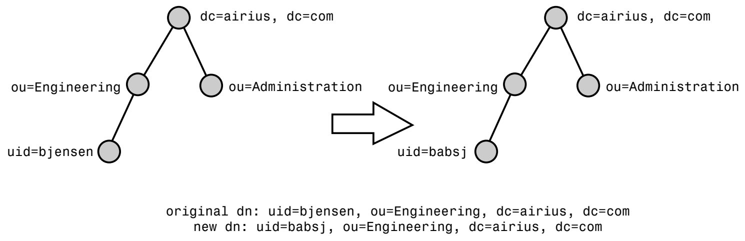


FIGURE 3.16 *Renaming an entry without moving it.*

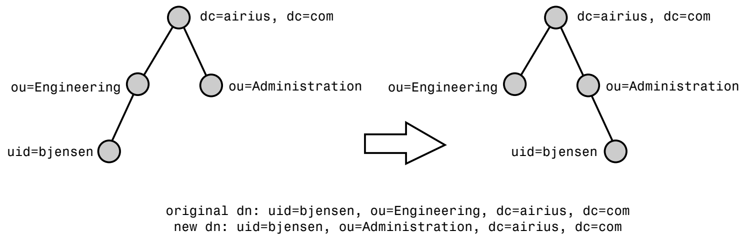
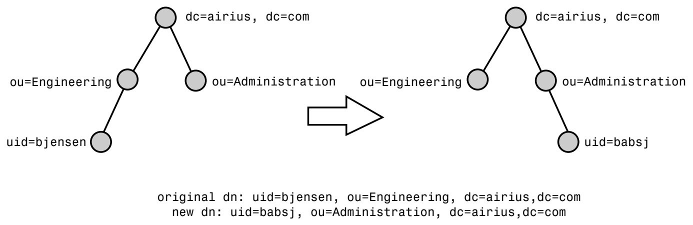
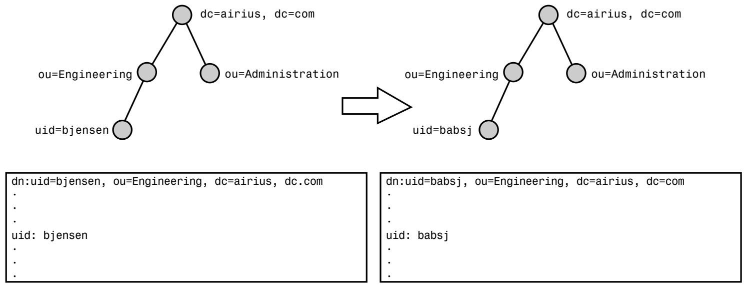


FIGURE 3.17 *Moving an entry without changing its RDN.*



```
original dn: uid=bjensen, ou=Engineering, dc=airius,dc=com
     new dn: uid=babsj, ou=Administration, dc=airius,dc=com
```

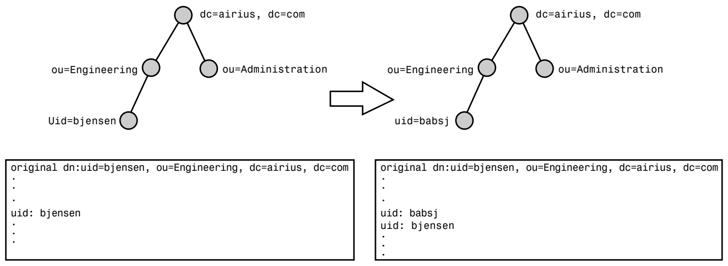
FIGURE 3.18 *Moving an entry and changing its RDN simultaneously.*



```
dn:uid=bjensen, ou=Engineering, dc=airius, dc.com
.
.
.
uid: bjensen
.
.
.
```

```
dn:uid=babsj, ou=Engineering, dc=airius, dc=com
.
.
.
uid: babsj
.
.
.
```

FIGURE 3.19 *Renaming an entry,* `deleteoldrdn=true`.



```
original dn:uid=bjensen, ou=Engineering, dc=airius, dc=com
.
.
.
uid: bjensen
.
.
.
```

```
original dn:uid=bjensen, ou=Engineering, dc=airius, dc=com
.
.
.
uid: babsj
uid: bjensen
.
.
.
```

FIGURE 3.20 *Renaming an entry,* `deleteoldrdn=false`.

LDAPv2 did not have a modify DN operation—it had only a modify RDN operation. As the name implies, modify RDN allows only the RDN of an entry to be changed. This means that an LDAPv2 server may rename an entry but may not move it to a new location in the tree. To accomplish a move with LDAPv2, you must copy the entry, along with any child entries underneath it, to the new location in the tree and delete the original entry or entries.

The modify operation allows you to update an existing directory entry. It takes two parameters: the DN of the entry to be modified and a set of modifications to be applied. These modifications can specify that new attribute values are to be added to the entry, that specific attribute values are to be deleted from the entry, or that all attribute values for a given attribute are to be replaced with a new set of attribute values. The modify request can include as many attribute modifications as needed.

In order for the modify operation to succeed, the following conditions must be met:

- The entry to be modified must exist.
- All of the attribute modifications must succeed.
- The resulting entry must obey the schema in effect.
- Access control must allow the update.

If all these conditions are met, the entry is modified. Note that all the modifications must succeed, or else the entire operation fails and the entry is not modified. This prevents inconsistencies that might arise from half-completed modify operations.

This last point raises one additional but very important topic about the LDAP update operations: Each operation is *atomic*, meaning that the whole operation is processed as a single unit of work. This unit either completely succeeds or no modifications are performed. For example, a modify request that affects multiple attributes within an entry cannot half-succeed, with certain attributes updated and others not updated. If the client receives a success result from the server, then all the modifications were applied to the entry. If the server returns an error to the client, then none of the modifications were applied.

The LDAP Authentication and Control Operations

There are two LDAP authentication operations, bind and unbind, and one control operation, abandon.

The bind operation is how a client authenticates itself to the directory. It does so by providing a distinguished name and a set of credentials. The server checks whether the credentials are correct for the given DN and, if they are, notes that the client is authenticated as long as the connection remains open or until the client re-authenticates. The server can grant privileges to the client based on its identity.

There are several different types of bind methods. In a simple bind, the client presents a DN and a password in cleartext to the LDAP server. The server verifies that the password matches the password value stored in the `userpassword` attribute of the entry and, if so, returns a success code to the client.

The simple bind does send the password over the network to the server in the clear. However, you can protect against eavesdroppers intercepting passwords by encrypting the connections using secure sockets layer (SSL) or TLS, which are discussed in the next section. In the future, LDAPv3 will include a digest-based authentication method that does not require that a cleartext password be sent to the server.

LDAPv3 also includes a new type of bind operation, the SASL bind. SASL is an extensible, protocol-independent framework for performing authentication and negotiation of security parameters. With SASL, the client specifies the type of authentication protocol it wants to use. If the server supports the authentication protocol, the client and server perform the agreed-upon authentication protocol.

For example, the client could specify that it wants to authenticate using the Kerberos protocol. If the server knows how to speak the Kerberos protocol, it indicates this to the client that sends a service ticket for the LDAP service. The server verifies the service ticket and returns a mutual authentication token to the client. Whenever the Kerberos authentication completes, the SASL bind is complete and the server returns a success code to the client. SASL can also support multistep authentication protocols such as S/KEY.

Incorporation of SASL into LDAPv3 means that new authentication methods, such as smart cards or biometric authentication, can be easily implemented for LDAP without requiring a revision of the protocol.

The second authentication operation is the unbind operation. The unbind operation has no parameters. When a client issues an unbind operation, the server discards any authentication information it has associated with the client's connection, terminates any outstanding LDAP operations, and disconnects from the client, thus closing the TCP connection.

The abandon operation has a single parameter: the message ID of the LDAP operation to abandon. The client issues an abandon operation when it is no longer interested in obtaining the results of a previously initiated operation. Upon receiving an abandon request, the server terminates processing of the operation that corresponds to the message ID. The abandon request, typically used by GUI clients, is sent when the user cancels a long-running search request.

Note that it's possible for the abandon request (coming from the client) and the results of the abandoned operation (going to the client) to pass each other in flight. The client needs to be prepared to receive (and discard) results from operations it has abandoned but the server sent anyway. If you are using an LDAP SDK, however, you don't need to worry about this; the SDK takes care of this for you.

The LDAP Security Model

We've discussed three of the four LDAP models so far. We have a set of directory entries, which are arranged into a hierarchy, and a set of protocol operations that allow us to authenticate to, search, and update the directory. All that remains is to provide a framework for protecting the information in the directory from unauthorized access. This is the purpose of the LDAP security model.

The security model relies on the fact that LDAP is a connection-oriented protocol. In other words, an LDAP client opens a connection to an LDAP server and performs a number of protocol operations on the same connection. The LDAP client may authenticate to the directory server at some point during the lifetime of the connection, at which point it may be granted additional (or fewer) privileges. For example, a client might authenticate as a particular identity that has been granted read-write access to all the entries in the directory. Before this authentication, it has some limited set of privileges (usually a default set of privileges extended to all users of the directory). After it authenticates, however, it is granted expanded privileges as long as the connection remains open.

What exactly is authentication? From the client's perspective, it is the process of proving to the server that the client is some entity. In other words, the client asserts that it has some identity and provides some credentials to prove this assertion. From the server's perspective, the process of authentication involves accepting the identity and credentials provided by the client and checking whether they prove that the client is who it claims to be.

To illustrate this abstract concept with a concrete example, let's examine how LDAP simple authentication works. In simple authentication, an LDAP client provides to an LDAP server a distinguished name and a password, which are sent to the server in the clear (not hashed or encrypted in any way). The server locates the entry in the directory corresponding to the DN provided by the client and checks whether the password presented by the client matches the value stored in the `userpassword` attribute of the entry. If it does, the client is authenticated; if it does not, the authentication operation fails and an error code is returned to the client.

The process of authenticating to the directory is called *binding*. An identity is bound to the connection when a successful authentication occurs via the bind operation we introduced in the previous section. If a client does not authenticate, or if it authenticates without providing any credentials, the client is bound anonymously. In other words, the server has no idea who the client is, so it grants some default set of privileges to the client. Usually, this default set of privileges is very minimal. In some instances, the default set of privileges is completely restrictive—no part of the directory may be read or searched. How you treat anonymously bound clients is up to you and depends on the security policy appropriate to your organization. You can find more information on security and privacy in Chapter 11, "Privacy and Security Design."

There are many different types of authentication systems available that are independent of LDAP. LDAP version 2 supported only simple authentication, in which a DN and password are transmitted in the clear from the client to the server.

Note

The previous statement is not completely correct because LDAPv2 also supported Kerberos version 4 authentication, which does not require that passwords be sent in the clear. However, KerberosV4 was not commercially successful and has been superseded by Kerberos version 5. Kerberos support was therefore dropped from the core LDAPv3 protocol, although it's entirely feasible to support it via an SASL mechanism.

Acknowledging the need to support many different authentication methods, LDAPv3 has adopted the SASL framework. SASL provides a standard way for multiple authentication protocols to be supported by LDAPv3. Each type of authentication system corresponds to a particular SASL mechanism. An SASL mechanism is an identifier that describes the type of authentication protocol being supported.

Note

The IETF's Internet Engineering Steering Group (IESG) has requested that the LDAPv3 specification be altered to mandate that all clients and servers implement some authentication method more secure than sending cleartext passwords over the wire. The intent is to raise the bar for interoperability so that people using LDAPv3 clients and servers can be assured that their authentication credentials are not susceptible to network eavesdropping. As of September 1998, the details about the new mandatory-to-implement authentication methods were still to be worked out within the IETF. If you are considering purchase of LDAP software, you should ask your vendor about support for the final version of the LDAPv3 standard.

After the server has verified the identity of the client, it can choose to grant additional privileges based on some site-specific policy. For example, you might have a policy that, when authenticated, users may search the directory, but that they may not modify their own directory entries. Or, you might have a more permissive policy that allows some authenticated users to modify certain attributes of their own entries whereas other users (your administrative staff) may modify any attribute of any entry. The way you describe the access rights, the entities to which those rights are granted, and the directory entries to which those rights apply is called *access control*.

Access Control Models

It may come as somewhat of a disappointment to learn that LDAP does not currently define a standard access control model. However, this does not mean that individual LDAP server implementations have no access control model. In fact, any commercially successful server software must have such a model.

The Netscape Directory Server, for example, has a rich access control model. The model works by describing what a given identity can do to some set of entries, with granularity down to the attribute level. For example, with the Netscape Server it is possible to specify an access control item (ACI) that allows a person to modify only the `description` attribute of his or her own entry. Or, the model can allow you to grant complete rights to the directory to all persons who are in a particular group. This allows easy creation of a set of directory administrators; a given person's rights can be easily revoked by removing them from the group. The model is fully documented in the *Netscape Directory Server Administrator's Guide*.

Work has begun in the IETF on defining a standard access control model and a standard syntax for representing access control rights. The promise for the future is that you, as a directory deployer, will be able to deploy directory servers from several vendors and implement a consistent security policy across those servers—whether they cooperate to serve a distributed directory or they are replicas of each other. Unfortunately, that is not the case today. You would be wise to document your access control policy in plain language so that you can adapt it to whatever model and syntax emerge from the standards bodies in the future.

SSL and TLS

SSL and TLS are new security technologies that encrypt all the data flowing between a client and a server. SSL, the older of the two technologies, has been a successful technology for the World Wide Web, securing electronic commerce and other transactions that depend on transmission of data being hidden from eavesdroppers. TLS, the follow-up to SSL, is an emerging Internet standard. LDAP offers a standard way for clients to begin encrypting all data flowing to and from LDAP on the connection using TLS.

Just as SSL enabled a new class of applications on the Web, TLS will enable new uses of directory technology. For example, two companies in a trading partner relationship can allow directory queries from their trading partners to travel over the Internet. Because TLS encrypts these queries and the results, each company can rest assured that the directory data is protected while in transit over the Internet. Figure 3.21 depicts this scenario.

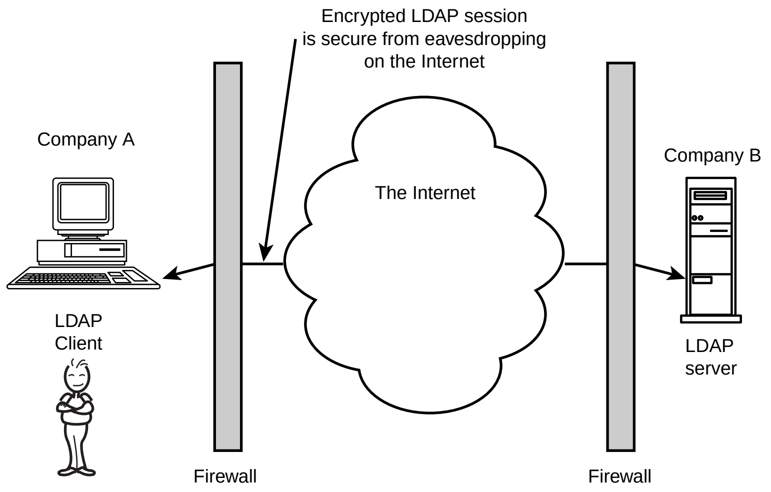


FIGURE 3.21 *TLS allows secure transmission of directory data over the Internet.*

In addition to allowing bulk encryption of all data flowing between clients and servers, SSL and TLS both allow mutual authentication using strong cryptography. Using X.509-based certificates, clients may prove their identity to servers while in turn verifying the identity of the server to which they are connected.

This technology is already widely deployed on the World Wide Web. The Hypertext Transfer Protocol over SSL (HTTPS) is used to provide secure Web access and client authentication. In a similar fashion, LDAP over SSL (LDAPS) allows secure LDAP client access and client authentication facilities today. In the future, LDAPv3 clients can use the `startTLS` extended operation to begin encrypting an LDAP connection, and TLS to prove their identity to the server, as well as verify the server's identity.

LDAP APIs

Early on, the developers of LDAP realized that the creation of directory-enabled applications would happen much more quickly if there existed a standard API for accessing and updating the directory. The original LDAP distribution from the University of Michigan (often referred to as the U-M LDAP release; refer to Chapter 2) included a C programming library and several sample client programs built on this library. For quite a while, the C API included in the U-M distribution was the only API/SDK available. With the current industry momentum behind LDAP, however, the number of SDKs is increasing, and additional SDKs are becoming available. (We will discuss these additional SDKs later in this section and in Chapter 20.) Figure 3.22 shows how the LDAP SDK fits into a directory-enabled client application.

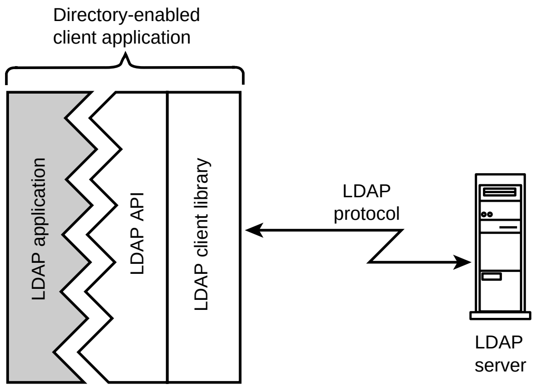


FIGURE 3.22 *The LDAP API provides a common interface to an LDAP client library SDK.*

The LDAP C API for LDAP version 2 is documented in RFC 1823, and a proposed C API for LDAP version 3 is in draft form at this time (available from the IETF Web site at `http://www.ietf.org`). The C API document simply defines the API calls and their semantics.

To obtain an actual SDK, you need to download one from one of a number of sources:

- The original University of Michigan SDK, which supports LDAPv2, is available in source code form from `http://www.umich.edu/~dirsvcs/ldap/`.
- An updated C SDK that supports LDAPv2 and LDAPv3 is available free of charge in binary form from Netscape at `http://developer.netscape.com`.

- Source code for the Netscape SDK is publicly available from mozilla.org at `http://www.mozilla.org`.
- Another LDAPv2/LDAPv3 SDK is available from Innosoft at `http://www.innosoft.com`.

All of the C SDKs can, of course, be used from a C++ program.

An Overview of the C LDAP API

The LDAP C API defines a set of core functions that map almost one-to-one onto the LDAP protocol operations. Those core functions are shown in Table 3.5.

TABLE 3.5 THE MAIN LDAP C API FUNCTIONS

Function	Description
`ldap_search()`	Searches for directory entries
`ldap_compare()`	Sees whether an entry contains a given attribute value
`ldap_bind()`	Authenticates (proves your identity) to a directory server
`ldap_unbind()`	Terminates an LDAP session
`ldap_modify()`	Makes changes to an existing directory entry
`ldap_add()`	Adds a new directory entry
`ldap_delete()`	Deletes an existing directory entry
`ldap_rename()`	Renames an existing directory entry (this call is named `ldap_modrdn()` in LDAPv2-only SDKs)
`ldap_result()`	Retrieves the results of one of the previous operations

The APIs listed in Table 3.5 provide an asynchronous interface to the directory; that is, the calls are used to initiate a protocol operation to the server, and the `ldap_result()` call is used later to collect results from the previously initiated operations. This allows your client to issue multiple protocol requests or perform other work, such as updating window contents, while the operation is in progress on the server.

The API also provides a synchronous interface, in which the API calls are blocked until all results are returned from the server. The synchronous calls are generally simpler to use and are appropriate for simple command-line clients and multithreaded applications.

In addition to the API calls listed in Table 3.5 and their synchronous counterparts, the LDAP API defines a set of utility routines that can be used to parse returned results from the server; iterates over sets of entries, attributes, and attribute values; and performs other useful operations. For a complete description of the various API calls available in the SDK you are using, consult the documentation.

A useful reference book that covers the C API in detail and offers general advice on building directory-enabled applications was written by two of the authors of this book. It is called *LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*, by Tim Howes and Mark Smith, published by Macmillan Technical Publishing.

Other LDAP APIs

In addition the various implementations of the C API, four other APIs are available:

- Netscape has developed an LDAPv2 and LDAPv3 Java API that, like the C API, has a close mapping onto the LDAP protocol. The Java API specification, currently in draft form, is available from the IETF Web site at `http://www.ietf.org`. An SDK that implements the draft API is available from `http://developer.netscape.com/` and, like the C SDK, is available in source code form at `http://www.mozilla.org`. Online documentation is also available. The Java classes that implement the Netscape SDK are also included with versions of Netscape Communicator currently being shipped.
- Perl fans can use PerLDAP, available from `http://www.mozilla.org`.
- JavaSoft has developed the proprietary Java Naming and Directory Interface (JNDI). This API/SDK defines a common interface for accessing a number of different directory systems from a Java application or applet. Additional types of directory systems and protocols can be supported by developing additional service provider interfaces (SPIs) for JNDI. This allows a JNDI client to access a number of distinct directory services, such as NIS, DNS, LDAP, NDS, or X.500. JNDI is available from JavaSoft at `http://www.javasoft.com/`.
- Microsoft also has a proprietary, object-oriented SDK, called ADSI, for accessing multiple directory systems. ADSI APIs are available for Visual Basic, C, and C++. For more information on ADSI, see `http://www.microsoft.com`.

These "directory-agnostic" access APIs (APIs that can access a number of different directory systems) can be useful if you are writing client software that must simultaneously access multiple directory services running incompatible protocols. However, because they present a single API across all the different directory protocols they support, these tools may not have sufficient fidelity for your needs. In other words, some features supported by the underlying protocol may not be available in the unified API.

In order to support these new features, the unified API must be revised to expose the new features. If the new feature exposes functionality in some protocol you aren't using, this is unnecessary clutter and overhead. LDAP-only APIs don't suffer from this problem.

LDIF

LDIF is a standard text-based format for describing directory entries. LDIF allows you to export your directory data and import it into another directory server, even if the two servers use different internal database formats. In the database/spreadsheet world, the tab-delimited format performs a similar function: It provides a simple format that virtually all spreadsheets and databases can import and export.

There are two different types of LDIF files. The first form describes a set of directory entries, such as your entire corporate directory, or perhaps a subset of it. The other type of LDIF file is a series of LDIF update statements that describes changes to be applied to directory entries. In the following sections we'll look at both formats in detail.

LDIF Representation of Directory Entries

Listing 3.1 represents two directory entries in LDIF format.

LISTING 3.1 A TYPICAL LDIF FILE

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Barbara Jensen
cn: Babs Jensen
givenname: Barbara
sn: Jensen
uid: bjensen
mail: bjensen@airius.com
telephoneNumber: +1 408 555 1212
description: Manager, switching products division

dn: uid=ssmith, ou=people, dc=airius, dc=com
```

```
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Steve Smith
cn: Stephen Smith
givenname: Stephen
sn: Smith
uid:ssmith
mail: ssmith@airius.com
telephoneNumber: +1 650 555 1212
description: Member of Technical Staff.
```

An individual entry expressed in LDIF format consists of two parts: a distinguished name and a list of attribute values. The DN, which must be the first line of the entry, is composed of the string `dn` followed by a colon (:) and the distinguished name of the entry. After the DN comes the attributes of the entry. Each attribute value is composed of an attribute type, a colon (:), and the attribute value. Attribute values may appear in any order; for readability, however, we suggest that you list the `objectclass` values for the entry first and group multiple values for the same attribute type together, as in Listing 3.1.

Any line in an LDIF file may be folded into multiple lines, which is typically done when an individual line is extremely long. To fold a line, insert a newline character and a space character into the value. Folding is not required, but some editors do not handle extremely long lines. Listing 3.2 shows an entry with a folded line; note how the `description` attribute is folded into four lines.

LISTING 3.2 AN LDIF FILE WITH A FOLDED ATTRIBUTE VALUE

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Barbara Jensen
cn: Babs Jensen
givenname: Barbara
sn: Jensen
uid: bjensen
mail: bjensen@airius.com
telephoneNumber: +1 408 555 1212
description: I will be out of the
 office from August 12, 1998, to September 10, 1998. If you need
 assistance with the Ostrich project, please contact Steve Smith
 at extension 7226.
```

If an LDIF file contains an attribute value or a distinguished name that is not ASCII, that value or DN must be encoded in a special format called *base 64*. This encoding method can represent any arbitrary data as a series of printable characters. When an attribute is base 64–encoded, the attribute type and value are separated by two colons, instead of a single colon. Listing 3.3 shows an entry in LDIF format that contains a base 64–encoded binary attribute (`jpegPhoto`). Notice how, in addition to being base 64–encoded, the attribute is folded.

LISTING 3.3 AN ENTRY IN LDIF FORMAT CONTAINING A BASE 64–ENCODED ATTRIBUTE VALUE

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Barbara Jensen
cn: Babs Jensen
givenname: Barbara
sn: Jensen
uid: bjensen
mail: bjensen@airius.com
telephoneNumber: +1 408 555 1212
jpegPhoto:: /9j/4AAQSkZJRgABAAAAAQABAAD/2wBDABALDA4MChAODQ4
 SERATGCgaGBYWGDEjJR0oOjM9PDkzODdASFxOQERXRTc4UG1RV19iZ2hnP
 k1xeXBkeFxlZ2P/2wBDARESEhgVGC8aGi9jQjhCY2NjY2NjY2NjY2NjY2N
 jY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2P/wAARCACcA
 LgDASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQo
 L/8QAtRAAAgEDAwIEAwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBk
 aEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUp
 TVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjp
 KWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6er
 x8vP09fb3+
```

More formally, the syntax of an entry represented in LDIF format is

```
("dn:" <DN of entry> ¦ "dn::" <base 64-encoded DN of entry>)
<attribute type> (":" <attribute value> ¦"::" <base 64 attribute value>)
...
```

A complete formal definition of the LDIF syntax is available from the IETF Web site at `http://www.ietf.org`.

LDIF Update Statements

The second type of LDIF file describes a set of changes to be applied to one or more directory entries. An individual LDIF update statement consists of a DN, a change type, and possibly a set of modifications. Typically, you will use this type of LDIF file as input to a command-line utility such as the `ldapmodify` program, which is included with the Netscape Directory Server and Netscape LDAP SDK. The `ldapmodify` program reads the update statements, converts those statements to LDAP protocol operations, and sends them to a server for processing.

There are four types of changes that can be described by an LDIF update statement. These change types correspond exactly to the types of update operations that can be performed over the LDAP protocol: add a new entry, delete an existing entry, modify an existing entry, and rename an existing entry. Although the examples in the following sections do not show either folding or base 64-encoding, both are permitted in LDIF update statements.

Adding a New Entry

The `add changetype` statement indicates that an entry is to be added to the directory. The syntax of this update statement is

```
dn: <dn of entry to be added>
changetype: add
<attribute type>: value
...
```

For example, you would use the following to add a new entry to the directory:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: add
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Barbara Jensen
cn: Babs Jensen
givenname: Barbara
sn: Jensen
uid: bjensen
mail: bjensen@airius.com
telephoneNumber: +1 408 555 1212
```

Deleting an Entry

The `delete changetype` statement indicates that an entry is to be removed from the directory. The syntax of this type of update statement is

```
dn: <dn of entry to be deleted>
changetype: delete
```

For example, you would use the following to delete an entry from the directory:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: delete
```

Modifying an Entry

The `modify changetype` statement indicates that an existing entry is to be modified. It also allows you to add new attribute values, remove specific attribute values, remove an attribute entirely, or replace all attribute values with a new set of values. The syntax of the modify update statement is

```
dn: <dn of entry to be modified>
changetype: modify
<modifytype> <attribute type>
[<attribute type>: <attribute value>]
-
...
```

Note that there is an additional operator: `modifytype`. This is either `add`, `delete`, or `replace`, and is interpreted as follows.

To add one or more new attribute values, use a `modifytype` of `add` and include the attribute values you want to add. The following example adds two new values to the `telephoneNumber` attribute; if there are already existing values for this attribute, they are unaffected:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: modify
add: telephoneNumber
telephoneNumber: +1 216 555 1212
telephoneNumber: +1 408 555 1212
```

To delete one or more specific attribute values, use a `modifytype` of `delete` and include the values you want to delete. The following example removes the value `+1 216 555 1212` from the `telephoneNumber` attribute; any other `telephoneNumber` attribute values are unaffected:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: modify
delete: telephoneNumber
telephoneNumber: +1 216 555 1212
```

To completely remove an attribute, use a `modifytype` of `delete`, but do not include any specific attribute value to be deleted. The following example completely removes the `telephoneNumber` attribute from the entry:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: modify
delete: telephoneNumber
```

To replace an attribute with a new set of values, use a `modifytype` of `replace` and include the values that should replace any existing attribute values. The following example replaces any existing values of the `telephoneNumber` attribute with the two given values:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: modify
replace: telephoneNumber
telephoneNumber: +1 216 555 1212
telephoneNumber: +1 405 555 1212
```

Multiple modifytypes can be combined into a single update statement. Each set of lines comprising one modifytype must be separated by a line that contains only a single dash. For example, the following update statement adds a new value to the mail attribute, removes a specific value from the telephoneNumber attribute, completely removes the description attribute, and replaces the givenname attribute with a new set of values:

```
dn: uid=bjensen, ou=people, dc=airius, dc=com
changetype: modify
add: mail
mail: bjensen@airius.com
-
delete: telephoneNumber
telephoneNumber: +1 216 555 1212
-
delete: description
-
replace: givenname
givenname: Barbara
givenname: Babs
-
```

When multiple modifications are included in a single LDIF update statement and the ldapmodify program sends the corresponding LDAP operations to an LDAP server, the server performs the update only if all the individual attribute modifications succeed. In the last example, if the entry did not contain the telephone number attribute value +1 216 555 1212, it would not be possible to delete that specific value. The server treats each update statement as a single unit, so none of the attribute modifications would be made, and an error would be returned to the client.

Renaming and/or Moving an Entry

The moddn changetype statement indicates that an existing entry is to be renamed and optionally moved to a new location in the directory tree. The syntax of the moddn update statement is

```
dn: <dn of entry to be modified>
changetype: moddn
[newsuperior: <dn of new parent>]
[deleteoldrdn: ( 0 ¦ 1 )]
[newrdn: <new relative distinguished name for the entry>]
```

If an entry's RDN is to be changed, the newrdn and deleteoldrdn parameters must be provided. If an entry is to be moved to a new location in the tree, the newsuperior parameter must be provided. Both operations can be performed at once; that is, an entry can have its RDN changed at the same time it is moved to a new location in the tree.

For example, to change an entry's name without moving it to a new location in the tree, you'd use the following:

```
dn: uid=bjensen, ou=People, dc=airius, dc=com
changetype: moddn
newrdn: uid=babsj
deleteoldrdn: 0
```

After this update is performed on the server, the entry would look like this:

```
dn: uid=babsj, ou=People, dc=airius, dc=com
[other attributes omitted for brevity]
uid: babsj
uid: bjensen
```

Notice how the old RDN, `uid: bjensen`, is still present in the entry. When `0` is provided for the `deleteoldrdn` flag, the old RDN is retained as an attribute of the entry. If you want the old RDN to be removed from the entry, include `deleteoldrdn: 1` in your `moddn` update statement. If this were done, the entry would look like this after being renamed:

```
dn: uid=babsj, ou=People, dc=airius, dc=com
[other attributes omitted for brevity]
uid: babsj
```

If you want to move an entry to a new location in the tree, you can use the `newsuperior` parameter to specify the DN of the entry you would like the entry to be moved to. For example, if you want to move Babs's entry under the Terminated Employees organizational unit, you would use the following LDIF update statement:

```
dn: uid=bjensen, ou=People, dc=airius, dc=com
changetype: moddn
newsuperior: ou=Terminated Employees, dc=airius, dc=com
```

The `moddn` `changetype` statement may behave differently depending on whether the server supports LDAPv3. If the server supports only LDAPv2, the `newsuperior` parameter may not be used; LDAPv2 does not support moving an entry to a new location in the tree.

LDAP and Internationalization

Directory services, by their very nature, span language boundaries. Multinational companies might have offices in dozens of countries, each with a distinct language. To address this growing need, LDAPv3 has been designed so that it can easily support multiple languages.

LDAPv3 uses the UTF-8 (Unicode Transformation Format-8) character set for all textual attribute values and distinguished names. UTF-8 is a standard character coding system that can represent text in virtually all written languages in use today. It is defined and developed by the Unicode Consortium, an industry group.

There are two important points to understand about UTF-8. First, because of the way UTF-8 is designed, ASCII data is also valid UTF-8 data. This has the benefit of being highly compatible with existing English-language directory data; no work needs to be done to transform the data into valid UTF-8.

The second point is that when you use UTF-8, it becomes unnecessary to declare an attribute value to be in a particular character set. In other systems, values must be tagged with their character set (e.g., Latin-1, Shift-JIS) so that the data may be correctly interpreted. However, because the UTF-8 character set contains codes for the glyphs of virtually all languages, this is unnecessary. It's even possible to use multiple languages within a single attribute value.

Because LDAPv3 servers can store text in multiple languages, it is useful to have some way to store and access attributes by language type. For example, in an international corporation with offices in the United States and Japan, it may be desirable to store several representations of a Japanese employee's name in the directory, including a version in Japanese and a version in English. The LDAP Extensions Working Group in the IETF has proposed a method for accomplishing this through the use of language codes.

A language code is an option on an LDAP attribute name. Separated from the base attribute name with a semicolon, it gives the particular language for the attribute in a standard format. For example, the attribute type `cn;lang-fr` refers to a common name in the French language, and the attribute type `sn;lang-ja` refers to a surname in the Japanese language. All language names are represented by a two-character code defined in ISO Standard 639, "Code for the representation of names of languages."

The LDAP language code standard also allows for names to be represented in a particular regional dialect or usage of a particular language. For example, there are some minor differences in how the English language is written in the United States and the United Kingdom. The language code `lang-en-US` identifies an attribute in the U.S. dialect, whereas the language code `lang-en-GB` indicates the British dialect. The country codes used to specify the region are defined in ISO Standard 3166, "Codes for the representation of names of countries."

An LDAP client may use language codes in search filters and attribute lists. In other words, an LDAP client may limit its search to only those attributes in the specific language it is interested in, and it may request that only specific languages be returned by specifying language codes in the list of attributes to be returned. For example, a client could search the French common name attribute with the filter `(cn;lang-fr=Jules)` and specify that the French common name and `description` attributes be returned by including only `cn;lang-fr` and `description;lang-fr` in the list of attributes to be returned.

Note that there is no way to retrieve all dialects of a particular language code. For example, the attribute type `cn;lang-en` is not the same as the attribute type `cn;lang-en-US`. Each dialect must be specifically requested. In general, avoid the use of dialects unless necessary. However, attributes with language codes are treated as subtypes of attributes without language codes. So, for example, the attribute `cn;lang-en` is a subtype of the attribute `cn`. Requesting the `cn` attribute will retrieve all language code variations of the `cn` attribute.

Language codes are a relatively new development, and not all servers support them at this time. Check with your software vendor to see if language codes are supported.

LDAP Overview Checklist

The term *LDAP* has come to mean four things:

- ☐ A set of models that guides you in your use of the directory; a data model that describes what you can put in the directory; a naming model that describes how you arrange and refer to directory data; a functional model that describes what you can do with directory data; and a security model that describes how directory data can be protected from unauthorized access.
- ☐ The LDAP protocol itself.
- ☐ An API for developing directory-enabled applications.
- ☐ LDIF, a standard interchange format for directory data.

Further Reading

A Summary of the X.500(96) User Schema for use with LDAPv3 (RFC 2256). M. Wahl, 1997. Available on the World Wide Web at `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2256.txt`.

Active Directory Services Interface (ADSI). Available on Microsoft's SDK World Wide Web site at `http://www.microsoft.com/msdn/sdk/`.

Code for the representation of names of languages, ISO Standard 639. The International Organization for Standardization, 1st edition, 1988.

Codes for the representation of names of countries, ISO Standard 3166. The International Organization for Standardization, 3rd edition, 1988.

Java Naming and Directory Interface (JNDI). Available on JavaSoft's JNDI Web site at `http://java.sun.com/products/jndi/`.

LDAP Data Interchange Format: Technical Specification, Internet Draft. G. Good, 1998. Available on the World Wide Web at `http://www.ietf.org`.

Lightweight Directory Access Protocol (v3) (RFC 2251). M. Wahl, T. Howes, S. Kille, 1998. Available on the World Wide Web at `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2251.txt`.

Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions (RFC 2252). M. Wahl, A. Coulbeck, T. Howes, S. Kille, 1998. Available on the World Wide Web at `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2252.txt`.

Lightweight Directory Access Protocol (v3): The String Representation of LDAP Search Filters (RFC 2254). M. Wahl, T. Howes, S. Kille, 1998. Available on the World Wide Web at `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2254.txt`.

Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names (RFC 2253). M. Wahl, T. Howes, S. Kille, 1998. Available on the World Wide Web at `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2253.txt`.

PerLDAP: an Object-Oriented LDAP Perl Module for Perl5. Available on Netscape's Directory Developer Central Web site at `http://developer.netscape.com/tech/directory/`.

Programming Directory-Enabled Applications with Lightweight Directory Access Protocol. T. Howes, M. Smith, Macmillan Technical Publishing, 1997.

The C LDAP Application Program Interface, Internet Draft. M. Smith, T. Howes, A. Herron, C. Weider, M. Wahl, A. Anantha, 1998. Available on the World Wide Web at `http://www.ietf.org`.

The Java LDAP Application Program Interface, Internet Draft. R. Weltman, T. Howes, M. Smith, 1998. Available on the World Wide Web at `http://www.ietf.org`.

The Unicode Standard, Version 2.0. The Unicode Consortium, Addison-Wesley, 1996.

Understanding X.500: The Directory. D. Chadwick, International Thomson Computer Press, 1996. Now out of print; selected portions available on the World Wide Web at `http://www.salford.ac.uk/its024/X500.htm`.

Looking Ahead

In Part I of this book, "An Introduction to Directory Services and LDAP," we've laid the groundwork for the rest of the book by giving you an overview and history of directory services and an introduction to the LDAP protocol and models. In Part II, "Designing Your Directory Service," we'll focus on designing your directory service from the ground up. We'll discuss the life cycle of a directory system and how you should assess your directory needs. Then we'll cover each major directory design topic in detail.