Internet-Draft
Intended Category: Standard Track
Document: draft-ietf-ldapbis-protocol-27.txt
Obsoletes: RFCs 2251, 2830, 3771

Editor: J. Sermersheim
Novell, Inc

Oct 2004

LDAP: The Protocol

Status of this Memo

Copyright Notice

Abstract

This document describes the protocol elements, along with their semantics and encodings, of the Lightweight Directory Access Protocol (LDAP). LDAP provides access to distributed directory services that act in accordance with X.500 data and service models. These protocol elements are based on those described in the X.500 Directory Access

Protocol (DAP).

Table of Contents

1. Introduction

The Directory is "a collection of open systems cooperating to provide directory services" [X.500]. A directory user, which may be a human or other entity, accesses the Directory through a client (or Directory User Agent (DUA)). The client, on behalf of the directory user, interacts with one or more servers (or Directory System Agents (DSA)). Clients interact with servers using a directory access protocol.

This document details the protocol elements of the Lightweight Directory Access Protocol (LDAP), along with their semantics. Following the description of protocol elements, it describes the way in which the protocol elements are encoded and transferred.

1.1. Relationship to Obsolete Specifications

This document is an integral part of the LDAP Technical Specification [Roadmap] which obsoletes the previously defined LDAP technical specification, RFC 3377, in its entirety.

This document obsoletes all of RFC 2251 except the following:
Sections 3.2, 3.4, 4.1.3 (last paragraph), 4.1.4, 4.1.5, 4.1.5.1, 4.1.9 (last paragraph), 5.1, 6.1, and 6.2 (last paragraph) are obsoleted by [Models].
Section 3.3 is obsoleted by [Roadmap].
Sections 4.2.1 (portions), and 4.2.2 are obsoleted by [AuthMeth].

Appendix C.1 summarizes substantive changes to the remaining sections.

This document obsoletes RFC 2830, Sections 2 and 4 in entirety. The remainder of RFC 2830 is obsoleted by [AuthMeth]. Appendix C.2 summarizes substantive changes to the remaining sections.

This document also obsoletes RFC 3771 in entirety.

2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", and "MAY" in this document are to be interpreted as described in [Keyword].

Character names in this document use the notation for code points and names from the Unicode Standard [Unicode]. For example, the letter "a" may be represented as either <U+0061> or <LATIN SMALL LETTER A>.

Note: a glossary of terms used in Unicode can be found in [Glossary]. Information on the Unicode character encoding model can be found in [CharModel].

The term "connection" refers to the underlying transport service used to carry the protocol exchange.

The term "LDAP exchange" refers to the layer where LDAP PDUs are exchanged between protocol peers.

The term "TLS layer" refers to a layer inserted between the connection and the LDAP exchange that utilizes Transport Layer Security ([TLS]) to protect the exchange of LDAP PDUs.

The term "SASL layer" refers to a layer inserted between the connection and the LDAP exchange that utilizes Simple Authentication and Security Layer ([SASL]) to protect the exchange of LDAP PDUs.

See the table in Section 5 for an illustration of these four terms.

3. Protocol Model

The general model adopted by this protocol is one of clients performing protocol operations against servers. In this model, a client transmits a protocol request describing the operation to be performed to a server. The server is then responsible for performing the necessary operation(s) in the Directory. Upon completion of an operation, the server typically returns a response containing appropriate data to the requesting client.

Protocol operations are generally independent of one another. Each operation is processed as an atomic action, leaving the directory in a consistent state.

Although servers are required to return responses whenever such responses are defined in the protocol, there is no requirement for

synchronous behavior on the part of either clients or servers. Requests and responses for multiple operations generally may be exchanged between a client and server in any order. If required, synchronous behavior may be controlled by client applications.

The core protocol operations defined in this document can be mapped to a subset of the X.500 (1993) Directory Abstract Service [X.511]. However there is not a one-to-one mapping between LDAP operations and X.500 Directory Access Protocol (DAP) operations. Server implementations acting as a gateway to X.500 directories may need to make multiple DAP requests to service a single LDAP request.

3.1 Operation and LDAP Exchange Relationship

Protocol operations are tied to an LDAP exchange. When the connection is closed, any uncompleted operations tied to the LDAP exchange are, when possible, abandoned, and when not possible, completed without transmission of the response. Also, when the connection is closed, the client MUST NOT assume that any uncompleted update operations tied to the LDAP exchange have succeeded or failed.

4. Elements of Protocol

The protocol is described using Abstract Syntax Notation One ([ASN.1]), and is transferred using a subset of ASN.1 Basic Encoding Rules ([BER]). Section 5 specifies how the protocol elements are encoded and transferred.

In order to support future extensions to this protocol, extensibility is implied where it is allowed per ASN.1 (i.e. sequence, set, choice, and enumerated types are extensible). In addition, ellipses (...) have been supplied in ASN.1 types that are explicitly extensible as discussed in [LDAPIANA]. Because of the implied extensibility, clients and servers MUST (unless otherwise specified) ignore trailing SEQUENCE components whose tags they do not recognize.

Changes to the protocol other than through the extension mechanisms described here require a different version number. A client indicates the version it is using as part of the bind request, described in Section 4.2. If a client has not sent a bind, the server MUST assume the client is using version 3 or later.

Clients may determine the protocol versions a server supports by reading the 'supportedLDAPVersion' attribute from the root DSE (DSA-Specific Entry) [Models].

4.1. Common Elements

This section describes the LDAPMessage envelope Protocol Data Unit (PDU) format, as well as data type definitions, which are used in the protocol operations.

4.1.1. Message Envelope

For the purposes of protocol exchanges, all protocol operations are encapsulated in a common envelope, the LDAPMessage, which is defined as follows:

```
     LDAPMessage ::= SEQUENCE {
          messageID       MessageID,
          protocolOp      CHOICE {
               bindRequest           BindRequest,
               bindResponse          BindResponse,
               unbindRequest         UnbindRequest,
               searchRequest         SearchRequest,
               searchResEntry        SearchResultEntry,
               searchResDone         SearchResultDone,
               searchResRef          SearchResultReference,
               modifyRequest         ModifyRequest,
               modifyResponse        ModifyResponse,
               addRequest            AddRequest,
```



```
               addResponse           AddResponse,
               delRequest            DelRequest,
               delResponse           DelResponse,
               modDNRequest          ModifyDNRequest,
               modDNResponse         ModifyDNResponse,
               compareRequest        CompareRequest,
               compareResponse       CompareResponse,
               abandonRequest        AbandonRequest,
               extendedReq           ExtendedRequest,
               extendedResp          ExtendedResponse,
               ...,
               intermediateResponse  IntermediateResponse },
          controls       [0] Controls OPTIONAL }

     MessageID ::= INTEGER (0 .. maxInt)

     maxInt INTEGER ::= 2147483647 -- (2^^31 - 1) --
```

The ASN.1 type Controls is defined in Section 4.1.11.

The function of the LDAPMessage is to provide an envelope containing

common fields required in all protocol exchanges. At this time the only common fields are the messageID and the controls.

If the server receives a PDU from the client in which the LDAPMessage SEQUENCE tag cannot be recognized, the messageID cannot be parsed, the tag of the protocolOp is not recognized as a request, or the encoding structures or lengths of data fields are found to be incorrect, then the server SHOULD return the Notice of Disconnection described in Section 4.4.1, with the resultCode set to protocolError, and MUST immediately close the connection.

In other cases where the client or server cannot parse a PDU, it SHOULD abruptly close the connection where further communication (including providing notice) would be pernicious. Otherwise, server implementations MUST return an appropriate response to the request, with the resultCode set to protocolError.

4.1.1.1. Message ID

All LDAPMessage envelopes encapsulating responses contain the messageID value of the corresponding request LDAPMessage.

The message ID of a request MUST have a non-zero value different from the the messageID of any other uncompleted requests in the LDAP exchange. The zero value is reserved for the unsolicited notification message.

Typical clients increment a counter for each request.

A client MUST NOT send a request with the same message ID as an earlier request in the same LDAP exchange unless it can be determined that the server is no longer servicing the earlier request (e.g.

after the final response is received, or a subsequent bind completes). Otherwise the behavior is undefined. For this purpose, note that abandon and abandoned operations do not send responses.

4.1.2. String Types

The LDAPString is a notational convenience to indicate that, although strings of LDAPString type encode as ASN.1 OCTET STRING types, the [ISO10646] character set (a superset of [Unicode]) is used, encoded following the [UTF-8] algorithm. Note that Unicode characters U+0000 through U+007F are the same as ASCII 0 through 127, respectively, and have the same single octet UTF-8 encoding. Other Unicode characters have a multiple octet UTF-8 encoding.

```
     LDAPString ::= OCTET STRING -- UTF-8 encoded,
                                 -- [ISO10646] characters
```

The LDAPOID is a notational convenience to indicate that the permitted value of this string is a (UTF-8 encoded) dotted-decimal representation of an OBJECT IDENTIFIER. Although an LDAPOID is encoded as an OCTET STRING, values are limited to the definition of <numericoid> given in Section 1.4 of [Models].

```
     LDAPOID ::= OCTET STRING -- Constrained to <numericoid> [Models]
```

For example,

```
     1.3.6.1.4.1.1466.1.2.3
```

4.1.3. Distinguished Name and Relative Distinguished Name

An LDAPDN is defined to be the representation of a Distinguished Name (DN) after encoding according to the specification in [LDAPDN].

```
     LDAPDN ::= LDAPString
                -- Constrained to <distinguishedName> [LDAPDN]
```

A RelativeLDAPDN is defined to be the representation of a Relative Distinguished Name (RDN) after encoding according to the specification in [LDAPDN].

```
     RelativeLDAPDN ::= LDAPString
                        -- Constrained to <name-component> [LDAPDN]
```

4.1.4. Attribute Descriptions

The definition and encoding rules for attribute descriptions are defined in Section 2.5 of [Models]. Briefly, an attribute description is an attribute type and zero or more options.

```
     AttributeDescription ::= LDAPString
```



```
                             -- Constrained to <attributedescription>
                             -- [Models]
```

4.1.5. Attribute Value

A field of type AttributeValue is an OCTET STRING containing an encoded attribute value. The attribute value is encoded according to the LDAP-specific encoding definition of its corresponding syntax.

The LDAP-specific encoding definitions for different syntaxes and attribute types may be found in other documents and in particular [Syntaxes].

```
AttributeValue ::= OCTET STRING
```

Note that there is no defined limit on the size of this encoding; thus protocol values may include multi-megabyte attribute values (e.g. photographs).

Attribute values may be defined which have arbitrary and non-printable syntax. Implementations MUST NOT display nor attempt to decode an attribute value if its syntax is not known. The implementation may attempt to discover the subschema of the source entry, and retrieve the descriptions of 'attributeTypes' from it [Models].

Clients MUST only send attribute values in a request that are valid according to the syntax defined for the attributes.

4.1.6. Attribute Value Assertion

The AttributeValueAssertion (AVA) type definition is similar to the one in the X.500 Directory standards. It contains an attribute description and a matching rule ([Models Section 4.1.3) assertion value suitable for that type. Elements of this type are typically used to assert that the value in assertionValue matches a value of an attribute.

```
AttributeValueAssertion ::= SEQUENCE {
     attributeDesc   AttributeDescription,
     assertionValue  AssertionValue }

AssertionValue ::= OCTET STRING
```

The syntax of the AssertionValue depends on the context of the LDAP operation being performed. For example, the syntax of the EQUALITY matching rule for an attribute is used when performing a Compare operation. Often this is the same syntax used for values of the attribute type, but in some cases the assertion syntax differs from the value syntax. See objectIdentiferFirstComponentMatch in [Syntaxes] for an example.

4.1.7. Attribute and PartialAttribute

Attributes and partial attributes consist of an attribute description

and attribute values. A PartialAttribute allows zero values, while Attribute requires at least one value.

```
PartialAttribute ::= SEQUENCE {
     type       AttributeDescription,
     vals       SET OF value AttributeValue }

Attribute ::= PartialAttribute(WITH COMPONENTS {
     ...,
     vals (SIZE(1..MAX))})
```

No two attribute values may be equivalent as described by Section 2.3 of [Models]. The set of attribute values is unordered. Implementations MUST NOT rely upon the ordering being repeatable.

4.1.8. Matching Rule Identifier

Matching rules are defined in Section 4.1.3 of [Models]. A matching rule is identified in the protocol by the printable representation of either its <numericoid>, or one of its short name descriptors [Models], e.g. 'caseIgnoreMatch' or '2.5.13.2'.

```
MatchingRuleId ::= LDAPString
```

4.1.9. Result Message

The LDAPResult is the construct used in this protocol to return success or failure indications from servers to clients. To various requests, servers will return responses of LDAPResult or responses containing the components of LDAPResult to indicate the final status of a protocol operation request.

```
LDAPResult ::= SEQUENCE {
     resultCode         ENUMERATED {
          success                      (0),
          operationsError              (1),
          protocolError                (2),
          timeLimitExceeded            (3),
          sizeLimitExceeded            (4),
          compareFalse                 (5),
          compareTrue                  (6),
          authMethodNotSupported       (7),
          strongAuthRequired           (8),
               -- 9 reserved --
          referral                     (10),
          adminLimitExceeded           (11),
          unavailableCriticalExtension (12),
          confidentialityRequired      (13),
          saslBindInProgress           (14),
```

```
        noSuchAttribute              (16),
        undefinedAttributeType       (17),
        inappropriateMatching        (18),
        constraintViolation          (19),
        attributeOrValueExists       (20),
        invalidAttributeSyntax       (21),
             -- 22-31 unused --
        noSuchObject                 (32),
        aliasProblem                 (33),
        invalidDNSyntax              (34),
             -- 35 reserved for undefined isLeaf --
        aliasDereferencingProblem    (36),
             -- 37-47 unused --
        inappropriateAuthentication  (48),
        invalidCredentials           (49),
        insufficientAccessRights     (50),
        busy                         (51),
        unavailable                  (52),
        unwillingToPerform           (53),
        loopDetect                   (54),
             -- 55-63 unused --
        namingViolation              (64),
        objectClassViolation         (65),
        notAllowedOnNonLeaf          (66),
        notAllowedOnRDN              (67),
        entryAlreadyExists           (68),
        objectClassModsProhibited    (69),
             -- 70 reserved for CLDAP --
        affectsMultipleDSAs          (71),
             -- 72-79 unused --
        other                        (80),
        ... },
    matchedDN          LDAPDN,
    diagnosticMessage  LDAPString,
    referral           [3] Referral OPTIONAL }
```

The resultCode enumeration is extensible as defined in Section 3.6 of [LDAPIANA]. The meanings of the listed result codes are given in Appendix A. If a server detects multiple errors for an operation, only one result code is returned. The server should return the result code that best indicates the nature of the error encountered.

The diagnosticMessage field of this construct may, at the server's option, be used to return a string containing a textual, human-readable (terminal control and page formatting characters should be avoided) diagnostic message. As this diagnostic message is not standardized, implementations MUST NOT rely on the values returned. If the server chooses not to return a textual diagnostic, the

diagnosticMessage field MUST be empty.

For certain result codes (typically, but not restricted to noSuchObject, aliasProblem, invalidDNSyntax and aliasDereferencingProblem), the matchedDN field is set (subject to access controls) to the name of the last entry (object or alias) used

in finding the target (or base) object. If no aliases were dereferenced while attempting to locate the entry, this will be a truncated form of the name provided or if aliases were dereferenced, of the resulting name, as defined in Section 12.5 of [X.511]. Otherwise the matchedDN field is empty.

4.1.10. Referral

The referral result code indicates that the contacted server cannot or will not perform the operation and that one or more other servers may be able to. Reasons for this include:

- The target entry of the request is not held locally, but the server has knowledge of its possible existence elsewhere.

- The operation is restricted on this server -- perhaps due to a read-only copy of an entry to be modified.

The referral field is present in an LDAPResult if the resultCode field value is referral, and absent with all other result codes. It contains one or more references to one or more servers or services that may be accessed via LDAP or other protocols. Referrals can be returned in response to any operation request (except unbind and abandon which do not have responses). At least one URI MUST be present in the Referral.

During a search operation, after the baseObject is located, and entries are being evaluated, the referral is not returned. Instead, continuation references, described in Section 4.5.3, are returned when other servers would need to be contacted to complete the operation.

```
Referral ::= SEQUENCE SIZE (1..MAX) OF uri URI

URI ::= LDAPString     -- limited to characters permitted in
                       -- URIs
```

If the client wishes to progress the operation, it MUST follow the referral by contacting one of the supported services. If multiple URIs are present, the client assumes that any supported URI may be used to progress the operation.

Protocol peers that follow referrals MUST ensure that they do not loop between servers. They MUST NOT repeatedly contact the same server for the same request with the same target entry name, scope and filter. Some implementations use a counter that is incremented each time referral handling occurs for an operation, and these kinds of implementations MUST be able to handle at least ten nested referrals between the root and a leaf entry.

A URI for a server implementing LDAP and accessible via [TCP]/[IP] (v4 or v6) is written as an LDAP URL according to [LDAPURL].

When an LDAP URL is used, the following instructions are followed:

- If an alias was dereferenced, the <dn> part of the URL MUST be present, with the new target object name. UTF-8 encoded characters appearing in the string representation of a DN or search filter may not be legal for URLs (e.g. spaces) and MUST be escaped using the % method in [URI].

- It is RECOMMENDED that the <dn> part be present to avoid ambiguity.

- If the <dn> part is present, the client MUST use this name in its next request to progress the operation, and if it is not present the client will use the same name as in the original request.

- Some servers (e.g. participating in distributed indexing) may provide a different filter in a URL of a referral for a search operation.

- If the <filter> part of the LDAP URL is present, the client MUST use this filter in its next request to progress this search, and if it is not present the client MUST use the same filter as it used for that search.

- For search, it is RECOMMENDED that the <scope> part be present to avoid ambiguity.

- If the <scope> part is missing, the scope of the original search is used by the client to progress the operation.

- Other aspects of the new request may be the same as or different from the request which generated the referral.

Other kinds of URIs may be returned. The syntax and semantics of such URIs is left to future specifications. Clients may ignore URIs that they do not support.

4.1.11. Controls

Controls provide a mechanism whereby the semantics and arguments of existing LDAP operations may be extended. One or more controls may be attached to a single LDAP message. A control only affects the semantics of the message it is attached to.

Controls sent by clients are termed 'request controls' and those sent by servers are termed 'response controls'.

```
     Controls ::= SEQUENCE OF control Control

     Control ::= SEQUENCE {
          controlType             LDAPOID,
          criticality             BOOLEAN DEFAULT FALSE,
          controlValue            OCTET STRING OPTIONAL }
```


The controlType field is the dotted-decimal representation of an OBJECT IDENTIFIER which uniquely identifies the control. This provides unambiguous naming of controls. Often, response control(s) solicited by a request control share controlType values with the request control.

The criticality field only has meaning in controls attached to request messages (except unbindRequest). For controls attached to response messages and the unbindRequest, the criticality field SHOULD be FALSE, and MUST be ignored by the receiving protocol peer. A value of TRUE indicates that it is unacceptable to perform the operation without applying the semantics of the control and FALSE otherwise. Specifically, the criticality field is applied as follows:

- Regardless of the value of the criticality field, if the server recognizes the control type and it is appropriate for the operation, the server is to make use of the control when performing the operation.

- If the server does not recognize the control type or it is not appropriate for the operation, and the criticality field is TRUE, the server MUST NOT perform the operation, and for operations that have a response message, MUST return unavailableCriticalExtension in the resultCode.

- If the server does not recognize the control type or it is not appropriate for the operation, and the criticality field is FALSE, the server MUST ignore the control.

The controlValue may contain information associated with the controlType. Its format is defined by the specification of the control. Implementations MUST be prepared to handle arbitrary contents of the controlValue octet string, including zero bytes. It is absent only if there is no value information which is associated with a control of its type. When a controlValue is defined in terms of ASN.1, and BER encoded according to Section 5.2, it also follows the extensibility rules in Section 4.

Servers list the controlType of request controls they recognize in the 'supportedControl' attribute in the root DSE (Section 5.1 of [Models]).

Controls SHOULD NOT be combined unless the semantics of the combination has been specified. The semantics of control combinations, if specified, are generally found in the control specification most recently published. When a combination of controls is encountered whose semantics are invalid, not specified (or not known), the message is considered to be not well-formed, thus the operation fails with protocolError. Additionally, unless order-dependent semantics are given in a specification, the order of a combination of controls in the SEQUENCE is ignored. Where the order is to be ignored but cannot be ignored by the server, the message is

considered not well-formed and the operation fails with protocolError.

This document does not specify any controls. Controls may be specified in other documents. Documents detailing control extensions are to provide for each control:

- the OBJECT IDENTIFIER assigned to the control,

- direction as to what value the sender should provide for the criticality field (note: the semantics of the criticality field are defined above should not be altered by the control's specification),

- whether information is to be present in the controlValue field, and if so, the format of the controlValue contents,

- the semantics of the control, and

- optionally, semantics regarding the combination of the control with other controls.

4.2. Bind Operation

The function of the Bind Operation is to allow authentication information to be exchanged between the client and server. The Bind operation should be thought of as the "authenticate" operation. Operational, authentication, and security-related semantics of this operation are given in [AuthMeth].

The Bind Request is defined as follows:

```
BindRequest ::= [APPLICATION 0] SEQUENCE {
     version                 INTEGER (1 .. 127),
     name                    LDAPDN,
     authentication          AuthenticationChoice }

AuthenticationChoice ::= CHOICE {
     simple                  [0] OCTET STRING,
                             -- 1 and 2 reserved
     sasl                    [3] SaslCredentials,
     ... }

SaslCredentials ::= SEQUENCE {
     mechanism               LDAPString,
     credentials             OCTET STRING OPTIONAL }
```

Fields of the Bind Request are:

- version: A version number indicating the version of the protocol to be used for the LDAP exchange. This document describes version 3 of the protocol. There is no version negotiation. The client sets this field to the version it desires. If the server does not

  support the specified version, it MUST respond with protocolError in the resultCode field of the BindResponse.

- name: If not empty, the name of the Directory object that the client wishes to bind as. This field may take on a null value (a zero length string) for the purposes of anonymous binds ([AuthMeth] Section 5.1) or when using Simple Authentication and Security Layer [SASL] authentication ([AuthMeth] Section 3.3.2). Where the server attempts to locate the named object, it SHALL NOT perform alias dereferencing.

- authentication: information used in authentication. This type is extensible as defined in Section 3.7 of [LDAPIANA]. Servers that do not support a choice supplied by a client return authMethodNotSupported in the resultCode field of the BindResponse.

Textual passwords (consisting of a character sequence with a known character set and encoding) transferred to the server using the simple AuthenticationChoice SHALL be transferred as [UTF-8] encoded [Unicode]. Prior to transfer, clients SHOULD prepare text passwords by applying the [SASLprep] profile of the [Stringprep] algorithm. Passwords consisting of other data (such as random octets) MUST NOT be altered. The determination of whether a password is textual is a local client matter.

Authorization is the decision of which access an operation has to the directory. Among other things, the process of authorization takes as input authentication information obtained during the bind operation and/or other acts of authentication (such as lower layer security services).

4.2.1. Processing of the Bind Request

Before processing a BindRequest, all uncompleted operations MUST either complete or be abandoned. The server may either wait for the uncompleted operations to complete, or abandon them. The server then proceeds to authenticate the client in either a single-step, or multi-step bind process. Each step requires the server to return a BindResponse to indicate the status of authentication.

After sending a BindRequest, clients MUST NOT send further LDAP PDUs until receiving the BindResponse. Similarly, servers SHOULD NOT process or respond to requests received while processing a BindRequest.

If the client did not bind before sending a request and receives an operationsError to that request, it may then send a Bind Request. If this also fails or the client chooses not to bind on the existing LDAP exchange, it may close the connection, reopen it and begin again by first sending a PDU with a Bind Request. This will aid in interoperating with servers implementing other versions of LDAP.

Clients may send multiple Bind Requests on an LDAP exchange to change the authentication and/or security associations or to complete a multi-stage bind process. Authentication from earlier binds is subsequently ignored.

For some SASL authentication mechanisms, it may be necessary for the client to invoke the BindRequest multiple times ([AuthMeth] Section 8.2). Clients MUST NOT invoke operations between two Bind Requests made as part of a multi-stage bind.

A client may abort a SASL bind negotiation by sending a BindRequest

with a different value in the mechanism field of SaslCredentials, or an AuthenticationChoice other than sasl.

If the client sends a BindRequest with the sasl mechanism field as an empty string, the server MUST return a BindResponse with authMethodNotSupported as the resultCode. This will allow clients to abort a negotiation if it wishes to try again with the same SASL mechanism.

4.2.2. Bind Response

The Bind Response is defined as follows.

```
BindResponse ::= [APPLICATION 1] SEQUENCE {
     COMPONENTS OF LDAPResult,
     serverSaslCreds    [7] OCTET STRING OPTIONAL }
```

BindResponse consists simply of an indication from the server of the status of the client's request for authentication.

A successful bind operation is indicated by a BindResponse with a resultCode set to success. Otherwise, an appropriate result code is set in the BindResponse. For bind, the protocolError result code may be used to indicate that the version number supplied by the client is unsupported.

If the client receives a BindResponse where the resultCode field is protocolError, it is to assume that the server does not support this version of LDAP. While the client may be able proceed with another version of this protocol (this may or may not require closing and re-establishing the connection), how to proceed with another version of this protocol is beyond the scope of this document. Clients which are unable or unwilling to proceed SHOULD close the connection.

The serverSaslCreds field is used as part of a SASL-defined bind mechanism to allow the client to authenticate the server to which it is communicating, or to perform "challenge-response" authentication. If the client bound with the simple choice, or the SASL mechanism does not require the server to return information to the client, then this field SHALL NOT be included in the BindResponse.

4.3. Unbind Operation

The function of the Unbind Operation is to terminate an LDAP exchange and close the connection. The Unbind operation is not the antithesis of the Bind operation as the name implies. The naming of these

operations is historical. The Unbind operation should be thought of as the "quit" operation.

The Unbind Operation is defined as follows:

```
UnbindRequest ::= [APPLICATION 2] NULL
```

The Unbind Operation has no response defined. Upon transmission of the UnbindRequest, each protocol peer is to consider the LDAP exchange terminated, MUST cease transmission of messages to the other peer, and MUST close the connection. Uncompleted operations are handled as specified in Section 5.1.

4.4. Unsolicited Notification

An unsolicited notification is an LDAPMessage sent from the server to the client which is not in response to any LDAPMessage received by the server. It is used to signal an extraordinary condition in the server or in the LDAP exchange or connection between the client and the server. The notification is of an advisory nature, and the server will not expect any response to be returned from the client.

The unsolicited notification is structured as an LDAPMessage in which the messageID is zero and protocolOp is of the extendedResp form (See Section 4.12). The responseName field of the ExtendedResponse always contains an LDAPOID which is unique for this notification.

One unsolicited notification (Notice of Disconnection) is defined in this document. The specification of an unsolicited notification consists of:

- the OBJECT IDENTIFIER assigned to the notification (to be specified in the responseName,

- the format of the contents (if any) of the responseValue,

- the circumstances which will cause the notification to be returned, and

- the semantics of the operation.

4.4.1. Notice of Disconnection

This notification may be used by the server to advise the client that the server is about to close the connection due to an error condition. This notification is intended to assist clients in distinguishing between an error condition and a transient network

failure. Note that this notification is not a response to an unbind requested by the client. Uncompleted operations are handled as specified in Section 5.1.

The responseName is 1.3.6.1.4.1.1466.20036, the responseValue field is absent, and the resultCode is used to indicate the reason for the disconnection.

Upon transmission of the Notice of Disconnection, the server is to consider the LDAP exchange terminated, MUST cease transmission of messages to the client, and MUST close the connection.

4.5. Search Operation

The Search Operation is used to request a server to return, subject to access controls and other restrictions, a set of entries matching a complex search criterion. This can be used to read attributes from a single entry, from entries immediately subordinate to a particular entry, or a whole subtree of entries.

4.5.1. Search Request

The Search Request is defined as follows:

```
SearchRequest ::= [APPLICATION 3] SEQUENCE {
     baseObject      LDAPDN,
     scope           ENUMERATED {
          baseObject              (0),
          singleLevel             (1),
          wholeSubtree            (2) },
     derefAliases    ENUMERATED {
          neverDerefAliases       (0),
          derefInSearching        (1),
          derefFindingBaseObj     (2),
          derefAlways             (3) },
     sizeLimit       INTEGER (0 .. maxInt),
     timeLimit       INTEGER (0 .. maxInt),
     typesOnly       BOOLEAN,
     filter          Filter,
     attributes      AttributeSelection }

AttributeSelection ::= SEQUENCE OF selector LDAPString
                -- The LDAPString is constrained to
                -- <attributeSelector> below

Filter ::= CHOICE {
     and             [0] SET OF filter Filter,
     or              [1] SET OF filter Filter,
     not             [2] Filter,
```

```
            equalityMatch   [3] AttributeValueAssertion,
            substrings      [4] SubstringFilter,
            greaterOrEqual  [5] AttributeValueAssertion,
```



```
            lessOrEqual     [6] AttributeValueAssertion,
            present         [7] AttributeDescription,
            approxMatch     [8] AttributeValueAssertion,
            extensibleMatch [9] MatchingRuleAssertion }

       SubstringFilter ::= SEQUENCE {
            type           AttributeDescription,
            -- initial and final can occur at most once
            substrings     SEQUENCE SIZE (1..MAX) OF substring CHOICE {
                 initial [0] AssertionValue,
                 any     [1] AssertionValue,
                 final   [2] AssertionValue } }

       MatchingRuleAssertion ::= SEQUENCE {
            matchingRule    [1] MatchingRuleId OPTIONAL,
            type            [2] AttributeDescription OPTIONAL,
            matchValue      [3] AssertionValue,
            dnAttributes    [4] BOOLEAN DEFAULT FALSE }
```

Fields of the Search Request are:

- baseObject: The name of the base object entry relative to which the search is to be performed.

- scope: Specifies the scope of the search to be performed. The semantics (as described in [X.511]) of the possible values of this field are:

  baseObject: The scope is constrained to the entry named by baseObject.

  singleLevel: The scope is constrained to the immediate subordinates of the entry named by baseObject.

  wholeSubtree: the scope is constrained to the entry named by the baseObject, and all its subordinates.

- derefAliases: An indicator as to how alias entries (as defined in [Models]) are to be handled in searching. The semantics of the possible values of this field are:

  neverDerefAliases: Do not dereference aliases in searching or in locating the base object of the search.

derefInSearching: While searching, dereference any alias entry subordinate to the base object which is also in the search scope. The filter is applied to the dereferenced object(s). If the search scope is wholeSubtree, the search continues in the subtree of any dereferenced object. Aliases in that subtree are also dereferenced. Servers SHOULD eliminate duplicate entries that arise due to alias dereferencing while searching.

derefFindingBaseObj: Dereference aliases in locating the base object of the search, but not when searching subordinates of the base object.

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

Servers MUST detect looping while dereferencing aliases in order to prevent denial of service attacks of this nature.

- sizeLimit: A size limit that restricts the maximum number of entries to be returned as a result of the search. A value of zero in this field indicates that no client-requested size limit restrictions are in effect for the search. Servers may also enforce a maximum number of entries to return.

- timeLimit: A time limit that restricts the maximum time (in seconds) allowed for a search. A value of zero in this field indicates that no client-requested time limit restrictions are in effect for the search. Servers may also enforce a maximum time limit for the search.

- typesOnly: An indicator as to whether search results are to contain both attribute descriptions and values, or just attribute descriptions. Setting this field to TRUE causes only attribute descriptions (no values) to be returned. Setting this field to FALSE causes both attribute descriptions and values to be returned.

- filter: A filter that defines the conditions that must be fulfilled in order for the search to match a given entry.

  The 'and', 'or' and 'not' choices can be used to form combinations of filters. At least one filter element MUST be present in an 'and' or 'or' choice. The others match against individual attribute values of entries in the scope of the search. (Implementor's note: the 'not' filter is an example of a tagged choice in an implicitly-tagged module. In BER this is treated as if the tag was explicit.)

A server MUST evaluate filters according to the three-valued logic of X.511 (1993) Section 7.8.1. In summary, a filter is evaluated to either "TRUE", "FALSE" or "Undefined". If the filter evaluates to TRUE for a particular entry, then the attributes of that entry are returned as part of the search result (subject to any applicable access control restrictions). If the filter evaluates to FALSE or Undefined, then the entry is ignored for the search.

A filter of the "and" choice is TRUE if all the filters in the SET OF evaluate to TRUE, FALSE if at least one filter is FALSE, and otherwise Undefined. A filter of the "or" choice is FALSE if all of the filters in the SET OF evaluate to FALSE, TRUE if at least one filter is TRUE, and Undefined otherwise. A filter of the 'not' choice is TRUE if the filter being negated is FALSE, FALSE if it is TRUE, and Undefined if it is Undefined.

The present match evaluates to TRUE where there is an attribute or subtype of the specified attribute description present in an entry, and FALSE otherwise (including a presence test with an unrecognized attribute description.)

The matching rule for equalityMatch filter items is defined by the EQUALITY matching rule for the attribute type.

There SHALL be at most one 'initial', and at most one 'final' in the 'substrings' of a SubstringFilter. If 'initial' is present, it SHALL be the first element of 'substrings'. If 'final' is present, it SHALL be the last element of 'substrings'.
The matching rule for AssertionValues in a substrings filter item is defined by the SUBSTR matching rule for the attribute type. Note that the AssertionValue in a substrings filter item conforms to the assertion syntax of the EQUALITY matching rule for the attribute type rather than the assertion syntax of the SUBSTR matching rule for the attribute type. Conceptually, the entire SubstringFilter is converted into an assertion value of the substrings matching rule prior to applying the rule.

The matching rule for the greaterOrEqual filter item is defined by the ORDERING and EQUALITY matching rules for the attribute type.

The matching rule for the lessOrEqual filter item is defined by the ORDERING matching rule for the attribute type.

An approxMatch filter item evaluates to TRUE when there is a value of the attribute or subtype for which some locally-defined approximate matching algorithm (e.g. spelling variations, phonetic match, etc.) returns TRUE. If an item matches for equality, it

also satisfies an approximate match. If approximate matching is not supported for the attribute, this filter item should be treated as an equalityMatch.

An extensibleMatch filter item is evaluated as follows:

If the matchingRule field is absent, the type field MUST be present, and an equality match is performed for that type.

If the type field is absent and the matchingRule is present, the matchValue is compared against all attributes in an entry which support that matchingRule. The matchingRule determines the syntax for the assertion value. The filter item evaluates to TRUE if it matches with at least one attribute in the entry, FALSE if it does not match any attribute in the entry, and Undefined if the matchingRule is not recognized or the assertionValue is invalid.

If the type field is present and the matchingRule is present, the matchValue is compared against entry attributes of the specified type. In this case, the matchingRule MUST be one

suitable for use with the specified type (see [Syntaxes]), otherwise the filter item is Undefined.

If the dnAttributes field is set to TRUE, the match is additionally applied against all the AttributeValueAssertions in an entry's distinguished name, and evaluates to TRUE if there is at least one attribute in the distinguished name for which the filter item evaluates to TRUE. The dnAttributes field is present to alleviate the need for multiple versions of generic matching rules (such as word matching), where one applies to entries and another applies to entries and dn attributes as well.

A filter item evaluates to Undefined when the server would not be able to determine whether the assertion value matches an entry. Examples include:

- An attribute description in an equalityMatch, substrings, greaterOrEqual, lessOrEqual, approxMatch or extensibleMatch filter is not recognized by the server.

- The attribute type does not define the appropriate matching rule.

- A MatchingRuleId in the extensibleMatch is not recognized by the server or is not valid for the attribute type.

- The type of filtering requested is not implemented.

- The assertion value is invalid.

For example, if a server did not recognize the attribute type shoeSize, a filter of (shoeSize=*) would evaluate to FALSE, and the filters (shoeSize=12), (shoeSize>=12) and (shoeSize<=12) would each evaluate to Undefined.

Servers MUST NOT return errors if attribute descriptions or matching rule ids are not recognized, assertion values are invalid, or the assertion syntax is not supported. More details of filter processing are given in Section 7.8 of [X.511].

- attributes: A selection list of the attributes to be returned from each entry which matches the search filter. LDAPString values of this field are constrained to the following Augmented Backus-Naur Form ([ABNF]):

attributeSelector = attributedescription / selectorpecial

selectorspecial = noattrs / alluserattrs

noattrs = %x31.2E.31 ; "1.1"

alluserattrs = %x2A ; asterisk ("*")

The <attributedescription> production is defined in Section 2.5 of [Models].

There are three special cases which may appear in the attributes selection list:

- an empty list with no attributes,

- a list containing "*" (with zero or more attribute descriptions), and

- a list containing only "1.1".

An empty list requests the return of all user attributes.

A list containing "*" requests the return of all user attributes in addition to other listed (operational) attributes.

A list containing only the OID "1.1" indicates that no attributes

are to be returned. If "1.1" is provided with other attributeSelector values, the "1.1" attributeSelector is ignored. This OID was chosen because it does not (and can not) correspond to any attribute in use.

Client implementors should note that even if all user attributes are requested, some attributes and/or attribute values of the entry may not be included in search results due to access controls or other restrictions. Furthermore, servers will not return operational attributes, such as objectClasses or attributeTypes, unless they are listed by name. Operational attributes are described in [Models].

Attributes are returned at most once in an entry. If an attribute description is named more than once in the list, the subsequent names are ignored. If an attribute description in the list is not recognized, it is ignored by the server.

Note that an X.500 "list"-like operation can be emulated by the client requesting a one-level LDAP search operation with a filter checking for the presence of the 'objectClass' attribute, and that an X.500 "read"-like operation can be emulated by a base object LDAP search operation with the same filter. A server which provides a gateway to X.500 is not required to use the Read or List operations, although it may choose to do so, and if it does, it must provide the same semantics as the X.500 search operation.

4.5.2. Search Result

The results of the search operation are returned as zero or more searchResultEntry messages, zero or more SearchResultReference messages, followed by a single searchResultDone message.

```
        SearchResultEntry ::= [APPLICATION 4] SEQUENCE {
```



```
             objectName      LDAPDN,
             attributes      PartialAttributeList }

        PartialAttributeList ::= SEQUENCE OF
                                 partialAttribute PartialAttribute
        -- Note that the PartialAttributeList may hold zero elements.
        -- This may happen when none of the attributes of an entry
        -- were requested, or could be returned.
        -- Note also that the partialAttribute vals set may hold zero
        -- elements. This may happen when typesOnly is requested, access
        -- controls prevent the return of values, or other reasons.

        SearchResultReference ::= [APPLICATION 19] SEQUENCE
```

```
                                SIZE (1..MAX) OF uri URI

     SearchResultDone ::= [APPLICATION 5] LDAPResult
```

Each SearchResultEntry represents an entry found during the search. Each SearchResultReference represents an area not yet explored during the search. The SearchResultEntry and SearchResultReference PDUs may come in any order. Following all the SearchResultReference and SearchResultEntry responses, the server returns a SearchResultDone response, which contains an indication of success, or detailing any errors that have occurred.

Each entry returned in a SearchResultEntry will contain all appropriate attributes as specified in the attributes field of the Search Request. Return of attributes is subject to access control and other administrative policy.

Some attributes may be constructed by the server and appear in a SearchResultEntry attribute list, although they are not stored attributes of an entry. Clients SHOULD NOT assume that all attributes can be modified, even if permitted by access control.

If the server's schema defines short names [Models] for an attribute type then the server SHOULD use one of those names in attribute descriptions for that attribute type (in preference to using the <numericoid> [Models] format of the attribute type's object identifier). The server SHOULD NOT use the short name if that name is known by the server to be ambiguous, or otherwise likely to cause interoperability problems.

4.5.3. Continuation References in the Search Result

If the server was able to locate the entry referred to by the baseObject but was unable to search one or more non-local entries, the server may return one or more SearchResultReference entries, each containing a reference to another set of servers for continuing the operation. A server MUST NOT return any SearchResultReference if it has not located the baseObject and thus has not searched any entries; in this case it would return a SearchResultDone containing either a

referral or noSuchObject result code (depending on the server's knowledge of the entry named in the baseObject).

If a server holds a copy or partial copy of the subordinate naming context [Section 5 of Models], it may use the search filter to determine whether or not to return a SearchResultReference response.

Otherwise SearchResultReference responses are always returned when in scope.

The SearchResultReference is of the same data type as the Referral.

A URI for a server implementing LDAP and accessible via [TCP]/[IP] (v4 or v6) is written as an LDAP URL according to [LDAPURL].

In order to complete the search, the client issues a new search operation for each SearchResultReference that is returned. Note that the abandon operation described in Section 4.11 applies only to a particular operation sent on the LDAP exchange between a client and server. The client must abandon subsequent search operations it wishes to individually.

Clients that follow search continuation references MUST ensure that they do not loop between servers. They MUST NOT repeatedly contact the same server for the same request with the same target entry name, scope and filter. Some clients use a counter that is incremented each time search result reference handling occurs for an operation, and these kinds of clients MUST be able to handle at least ten nested search result references between the root and a leaf entry.

When an LDAP URL is used, the following instructions are followed:

- The <dn> part of the URL MUST be present, with the new target object name. The client MUST use this name when following the reference. UTF-8 encoded characters appearing in the string representation of a DN or search filter may not be legal for URLs (e.g. spaces) and MUST be escaped using the % method in [URI].

- Some servers (e.g. participating in distributed indexing) may provide a different filter in a URL of a SearchResultReference.

- If the <filter> part of the URL is present, the client MUST use this filter in its next request to progress this search, and if it is not present the client MUST use the same filter as it used for that search.

- If the originating search scope was singleLevel, the <scope> part of the URL will be "base".

- It is RECOMMENDED that the <scope> part be present to avoid ambiguity.

- Other aspects of the new search request may be the same as or different from the search request which generated the SearchResultReference.

- The name of an unexplored subtree in a SearchResultReference need not be subordinate to the base object.

Other kinds of URIs may be returned. The syntax and semantics of such URIs is left to future specifications. Clients may ignore URIs that they do not support.

4.5.3.1. Examples

For example, suppose the contacted server (hosta) holds the entry <DC=Example,DC=NET> and the entry <CN=Manager,DC=Example,DC=NET>. It knows that either LDAP-capable servers (hostb) or (hostc) hold <OU=People,DC=Example,DC=NET> (one is the master and the other server a shadow), and that LDAP-capable server (hostd) holds the subtree <OU=Roles,DC=Example,DC=NET>. If a wholeSubtree search of <DC=Example,DC=NET> is requested to the contacted server, it may return the following:

```
SearchResultEntry for DC=Example,DC=NET
SearchResultEntry for CN=Manager,DC=Example,DC=NET
SearchResultReference {
  ldap://hostb/OU=People,DC=Example,DC=NET??sub
  ldap://hostc/OU=People,DC=Example,DC=NET??sub }
SearchResultReference {
  ldap://hostd/OU=Roles,DC=Example,DC=NET??sub }
SearchResultDone (success)
```

Client implementors should note that when following a SearchResultReference, additional SearchResultReference may be generated. Continuing the example, if the client contacted the server (hostb) and issued the search for the subtree <OU=People,DC=Example,DC=NET>, the server might respond as follows:

```
SearchResultEntry for OU=People,DC=Example,DC=NET
SearchResultReference {
  ldap://hoste/OU=Managers,OU=People,DC=Example,DC=NET??sub }
SearchResultReference {
  ldap://hostf/OU=Consultants,OU=People,DC=Example,DC=NET??sub }
SearchResultDone (success)
```

Similarly, if a singleLevel search of <DC=Example,DC=NET> is requested to the contacted server, it may return the following:

```
SearchResultEntry for CN=Manager,DC=Example,DC=NET
SearchResultReference {
  ldap://hostb/OU=People,DC=Example,DC=NET??base
  ldap://hostc/OU=People,DC=Example,DC=NET??base }
SearchResultReference {
  ldap://hostd/OU=Roles,DC=Example,DC=NET??base }
SearchResultDone (success)
```

If the contacted server does not hold the base object for the search, but has knowledge of its possible location, then it may return a referral to the client. In this case, if the client requests a subtree search of <DC=Example,DC=ORG> to hosta, the server returns a SearchResultDone containing a referral.

```
SearchResultDone (referral) {
  ldap://hostg/DC=Example,DC=ORG??sub }
```

4.6. Modify Operation

The Modify Operation allows a client to request that a modification of an entry be performed on its behalf by a server. The Modify Request is defined as follows:

```
ModifyRequest ::= [APPLICATION 6] SEQUENCE {
     object          LDAPDN,
     changes         SEQUENCE OF change SEQUENCE {
          operation       ENUMERATED {
               add     (0),
               delete  (1),
               replace (2) },
          modification    PartialAttribute } }
```

Fields of the Modify Request are:

- object: The name of the object to be modified. The value of this field contains the DN of the entry to be modified. The server SHALL NOT perform any alias dereferencing in determining the object to be modified.

- changes: A list of modifications to be performed on the entry. The entire list of modifications MUST be performed in the order they are listed as a single atomic operation. While individual modifications may violate certain aspects of the directory schema (such as the object class definition and DIT content rule), the resulting entry after the entire list of modifications is performed MUST conform to the requirements of the directory model and controlling schema [Models].

  - operation: Used to specify the type of modification being performed. Each operation type acts on the following modification. The values of this field have the following semantics respectively:

add: add values listed to the modification attribute, creating the attribute if necessary;

delete: delete values listed from the modification attribute, removing the entire attribute if no values are listed, or if all current values of the attribute are listed for deletion;

replace: replace all existing values of the modification attribute with the new values listed, creating the attribute if it did not already exist. A replace with no value will delete the entire attribute if it exists, and is ignored if the attribute does not exist.

- modification: A PartialAttribute (which may have an empty SET of vals) used to hold the attribute type or attribute type and values being modified.

Upon receipt of a Modify Request, the server attempts to perform the necessary modifications to the DIT and returns the result in a Modify Response, defined as follows:

```
ModifyResponse ::= [APPLICATION 7] LDAPResult
```

The server will return to the client a single Modify Response indicating either the successful completion of the DIT modification, or the reason that the modification failed. Due to the requirement for atomicity in applying the list of modifications in the Modify Request, the client may expect that no modifications of the DIT have been performed if the Modify Response received indicates any sort of error, and that all requested modifications have been performed if the Modify Response indicates successful completion of the Modify Operation. The result of the modification is indeterminate if the Modify Response is not received (e.g. the LDA exchange is terminated or the Modify Operation is abandoned).

The Modify Operation cannot be used to remove from an entry any of its distinguished values, i.e. those values which form the entry's relative distinguished name. An attempt to do so will result in the server returning the notAllowedOnRDN result code. The Modify DN Operation described in Section 4.9 is used to rename an entry.

Note that due to the simplifications made in LDAP, there is not a direct mapping of the changes in an LDAP ModifyRequest onto the changes of a DAP ModifyEntry operation, and different implementations of LDAP-DAP gateways may use different means of representing the

change. If successful, the final effect of the operations on the entry MUST be identical.

4.7. Add Operation

The Add Operation allows a client to request the addition of an entry into the Directory. The Add Request is defined as follows:

```
AddRequest ::= [APPLICATION 8] SEQUENCE {
     entry           LDAPDN,
     attributes      AttributeList }

AttributeList ::= SEQUENCE OF attribute Attribute
```

Fields of the Add Request are:

- entry: the name of the entry to be added. The server SHALL NOT dereference any aliases in locating the entry to be added.

- attributes: the list of attributes that, along with those from the RDN, make up the content of the entry being added. Clients MAY or MAY NOT include the RDN attribute in this list. Clients MUST include the 'objectClass' attribute, and values of any mandatory attributes of the listed object classes. Clients MUST NOT supply NO-USER-MODIFICATION attributes such as the createTimestamp or creatorsName attributes, since the server maintains these automatically.

The entry named in the entry field of the AddRequest MUST NOT exist for the AddRequest to succeed. The immediate superior (parent) of an object or alias entry to be added MUST exist. For example, if the client attempted to add <CN=JS,DC=Example,DC=NET>, the <DC=Example,DC=NET> entry did not exist, and the <DC=NET> entry did exist, then the server would return the noSuchObject result code with the matchedDN field containing <DC=NET>.

Server implementations SHOULD NOT restrict where entries can be located in the Directory unless DIT structure rules are in place. Some servers allow the administrator to restrict the classes of entries which can be added to the Directory.

Upon receipt of an Add Request, a server will attempt to add the requested entry. The result of the add attempt will be returned to the client in the Add Response, defined as follows:

```
AddResponse ::= [APPLICATION 9] LDAPResult
```

A response of success indicates that the new entry has been added to the Directory.

4.8. Delete Operation

The Delete Operation allows a client to request the removal of an entry from the Directory. The Delete Request is defined as follows:

```
DelRequest ::= [APPLICATION 10] LDAPDN
```

The Delete Request consists of the name of the entry to be deleted. The server SHALL NOT dereference aliases while resolving the name of the target entry to be removed.

Only leaf entries (those with no subordinate entries) can be deleted with this operation.

Upon receipt of a Delete Request, a server will attempt to perform the entry removal requested and return the result in the Delete Response defined as follows:


```
DelResponse ::= [APPLICATION 11] LDAPResult
```

4.9. Modify DN Operation

The Modify DN Operation allows a client to change the Relative Distinguished Name (RDN) of an entry in the Directory, and/or to move a subtree of entries to a new location in the Directory. The Modify DN Request is defined as follows:

```
ModifyDNRequest ::= [APPLICATION 12] SEQUENCE {
     entry           LDAPDN,
     newrdn          RelativeLDAPDN,
     deleteoldrdn    BOOLEAN,
     newSuperior     [0] LDAPDN OPTIONAL }
```

Fields of the Modify DN Request are:

- entry: the name of the entry to be changed. This entry may or may not have subordinate entries.

- newrdn: the new RDN of the entry. If the operation moves the entry to a new superior without changing its RDN, the value of the old RDN is supplied for this parameter.
  Attribute values of the new RDN not matching any attribute value of the entry are added to the entry and an appropriate error is

returned if this fails.

- deleteoldrdn: a boolean field that controls whether the old RDN attribute values are to be retained as attributes of the entry, or deleted from the entry.

- newSuperior: if present, this is the name of an existing object entry which becomes the immediate superior (parent) of the existing entry.

The server SHALL NOT dereference any aliases in locating the objects named in entry or newSuperior.

Upon receipt of a ModifyDNRequest, a server will attempt to perform the name change and return the result in the Modify DN Response, defined as follows:

```
ModifyDNResponse ::= [APPLICATION 13] LDAPResult
```

For example, if the entry named in the entry field was <cn=John Smith,c=US>, the newrdn field was <cn=John Cougar Smith>, and the newSuperior field was absent, then this operation would attempt to rename the entry to be <cn=John Cougar Smith,c=US>. If there was already an entry with that name, the operation would fail with the entryAlreadyExists result code.

The object named in newSuperior MUST exist. For example, if the client attempted to add <CN=JS,DC=Example,DC=NET>, the

<DC=Example,DC=NET> entry did not exist, and the <DC=NET> entry did exist, then the server would return the noSuchObject result code with the matchedDN field containing <DC=NET>.

If the deleteoldrdn field is TRUE, the attribute values forming the old RDN but not the new RDN are deleted from the entry. If the deleteoldrdn field is FALSE, the attribute values forming the old RDN will be retained as non-distinguished attribute values of the entry. The server MUST fail the operation and return an error in the result code if the setting of the deleteoldrdn field would cause a schema inconsistency in the entry.

Note that X.500 restricts the ModifyDN operation to only affect entries that are contained within a single server. If the LDAP server is mapped onto DAP, then this restriction will apply, and the affectsMultipleDSAs result code will be returned if this error occurred. In general, clients MUST NOT expect to be able to perform arbitrary movements of entries and subtrees between servers or between naming contexts.

4.10. Compare Operation

The Compare Operation allows a client to compare an assertion value with the values of a particular attribute in a particular entry in the Directory. The Compare Request is defined as follows:

```
CompareRequest ::= [APPLICATION 14] SEQUENCE {
     entry           LDAPDN,
     ava             AttributeValueAssertion }
```

Fields of the Compare Request are:

- entry: the name of the entry to be compared. The server SHALL NOT dereference any aliases in locating the entry to be compared.

- ava: holds the attribute value assertion to be compared.

Upon receipt of a Compare Request, a server will attempt to perform the requested comparison and return the result in the Compare Response, defined as follows:

```
CompareResponse ::= [APPLICATION 15] LDAPResult
```

The resultCode field is set to compareTrue, compareFalse, or an appropriate error. compareTrue indicates that the assertion value in the ava field matches a value of the attribute or subtype according to the attribute's EQUALITY matching rule. compareFalse indicates that the assertion value in the ava field and the values of the attribute or subtype did not match. Other result codes indicate either that the result of the comparison was Undefined (Section 4.5.1), or that some error occurred.

Note that some directory systems may establish access controls which permit the values of certain attributes (such as userPassword) to be compared but not interrogated by other means.

4.11. Abandon Operation

The function of the Abandon Operation is to allow a client to request that the server abandon an uncompleted operation. The Abandon Request is defined as follows:

```
AbandonRequest ::= [APPLICATION 16] MessageID
```

The MessageID is that of an operation which was requested earlier in this LDAP exchange. The abandon request itself has its own MessageID. This is distinct from the MessageID of the earlier operation being abandoned.

There is no response defined in the Abandon operation. Upon receipt of an AbandonRequest, the server MAY abandon the operation identified by the MessageID. Since the client cannot tell the difference between a successfully abandoned operation and an uncompleted operation, the application of the Abandon operation is limited to uses where the client does not require an indication of its outcome.

Abandon, Bind, Unbind, and StartTLS operations cannot be abandoned.

In the event that a server receives an Abandon Request on a Search Operation in the midst of transmitting responses to the search, that server MUST cease transmitting entry responses to the abandoned request immediately, and MUST NOT send the SearchResponseDone. Of course, the server MUST ensure that only properly encoded LDAPMessage PDUs are transmitted.

The ability to abandon other (particularly update) operations is at the discretion of the server.

Clients should not send abandon requests for the same operation multiple times, and MUST also be prepared to receive results from operations it has abandoned (since these may have been in transit when the abandon was requested, or are not able to be abandoned).

Servers MUST discard abandon requests for message IDs they do not recognize, for operations which cannot be abandoned, and for operations which have already been abandoned.

4.12. Extended Operation

The extended operation allows additional operations to be defined for services not already available in the protocol. For example, to add operations to install transport layer security (see Section 4.14).

The extended operation allows clients to make requests and receive responses with predefined syntaxes and semantics. These may be defined in RFCs or be private to particular implementations.

Each extended operation consists of an extended request and an extended response.

```
ExtendedRequest ::= [APPLICATION 23] SEQUENCE {
     requestName      [0] LDAPOID,
     requestValue     [1] OCTET STRING OPTIONAL }
```

The requestName is a dotted-decimal representation of the unique OBJECT IDENTIFIER corresponding to the request. The requestValue is information in a form defined by that request, encapsulated inside an OCTET STRING.

The server will respond to this with an LDAPMessage containing an ExtendedResponse.

```
ExtendedResponse ::= [APPLICATION 24] SEQUENCE {
     COMPONENTS OF LDAPResult,
     responseName     [10] LDAPOID OPTIONAL,
     responseValue    [11] OCTET STRING OPTIONAL }
```

The responseName is typically not required to be present as the syntax and semantics of the response (including the format of the responseValue) is implicitly known and associated with the request by the messageID.

If the extended operation associated with the requestName is not supported by the server, the server MUST NOT provide a responseName nor a responseValue and MUST return a resultCode of protocolError.

The requestValue and responseValue fields contain any information associated with the operation. The format of these fields is defined by the specification of the extended operation. Implementations MUST be prepared to handle arbitrary contents of these fields, including zero bytes. Values that are defined in terms of ASN.1 and BER encoded according to Section 5.2, also follow the extensibility rules in Section 4.

Servers list the requestName of Extended Requests they recognize in the 'supportedExtension' attribute in the root DSE (Section 5.1 of [Models]).

Extended operations may be specified in other documents. The specification of an extended operation consists of:

- the OBJECT IDENTIFIER assigned to the requestName,

- the OBJECT IDENTIFIER (if any) assigned to the responseName (note that the same OBJECT IDENTIFIER my be used for both the requestName and responseName),

- the format of the contents of the requestValue and responseValue (if any), and

- the semantics of the operation.

4.13. IntermediateResponse Message

While the Search operation provides a mechanism to return multiple response messages for a single search request, other operations, by nature, do not provide for multiple response messages.

The IntermediateResponse message provides a general mechanism for defining single-request/multiple-response operations in LDAP. This message is intended to be used in conjunction with the extended operation to define new single-request/multiple-response operations or in conjunction with a control when extending existing LDAP operations in a way that requires them to return intermediate response information.

It is intended that the definitions and descriptions of extended operations and controls that make use of the IntermediateResponse message will define the circumstances when an IntermediateResponse message can be sent by a server and the associated meaning of an IntermediateResponse message sent in a particular circumstance.

```
IntermediateResponse ::= [APPLICATION 25] SEQUENCE {
        responseName     [0] LDAPOID OPTIONAL,
        responseValue    [1] OCTET STRING OPTIONAL }
```

IntermediateResponse messages SHALL NOT be returned to the client unless the client issues a request that specifically solicits their return. This document defines two forms of solicitation: extended operation and request control. IntermediateResponse messages are specified in documents describing the manner in which they are solicited (i.e. in the extended operation or request control specification that uses them). These specifications include:

- the OBJECT IDENTIFIER (if any) assigned to the responseName,

- the format of the contents of the responseValue, and

- the semantics associated with the IntermediateResponse message.

Extensions that allow the return of multiple types of IntermediateResponse messages SHALL identify those types using unique responseName values (note that one of these may specify no value).

Sections 4.13.1 and 4.13.2 describe additional requirements on the inclusion of responseName and responseValue in IntermediateResponse messages.

4.13.1. Usage with LDAP ExtendedRequest and ExtendedResponse

A single-request/multiple-response operation may be defined using a single ExtendedRequest message to solicit zero or more IntermediateResponse messages of one or more kinds followed by an ExtendedResponse message.

4.13.2. Usage with LDAP Request Controls

A control's semantics may include the return of zero or more IntermediateResponse messages prior to returning the final result code for the operation. One or more kinds of IntermediateResponse messages may be sent in response to a request control.

All IntermediateResponse messages associated with request controls SHALL include a responseName. This requirement ensures that the client can correctly identify the source of IntermediateResponse messages when:

- two or more controls using IntermediateResponse messages are included in a request for any LDAP operation or

- one or more controls using IntermediateResponse messages are included in a request with an LDAP extended operation that uses IntermediateResponse messages.

4.14. StartTLS Operation

The Start Transport Layer Security (StartTLS) operationÆs purpose is to initiate installation of a TLS layer. The StartTLS operation is defined using the extended operation mechanism described in Section 4.12.

4.14.1. StartTLS Request

A client requests TLS establishment by transmitting a StartTLS request PDU to the server. The StartTLS request is defined in terms of an ExtendedRequest. The requestName is "1.3.6.1.4.1.1466.20037", and the requestValue field is always absent.

The client MUST NOT send any PDUs on this LDAP exchange following this request until it receives a StartTLS extended response and, in the case of a successful response, completes TLS negotiations.

Sequencing problems (particularly those detailed in Section 3.1.1 of [AuthMeth] result in an operationsError being returned in the resultCode.

If the server does not support TLS (whether by design or by current configuration), it returns the protocolError resultCode as described in Section 4.12.

4.14.2. StartTLS Response

When a StartTLS request is made, servers supporting the operation MUST return a StartTLS response PDU to the requestor. The responseName, if present, is also "1.3.6.1.4.1.1466.20037". The responseValue is absent.

If the server is willing and able to negotiate TLS, it returns a success resultCode. Refer to Section 4 of [AuthMeth] for details.

If the server is otherwise unwilling or unable to perform this operation, the server is to return an appropriate result code indicating the nature of the problem. For example, if the TLS subsystem is not presently available, the server may return indicate so by returning the unavailable resultCode.

4.14.3. Removal of the TLS Layer

Two forms of TLS layer removal -- graceful and abrupt -- are provided. These do not involve LDAP PDUs, but are preformed at the underlying layers.

If the connection is closed, uncompleted operations are handled as specified in Section 5.1.

4.14.3.1. Graceful Removal

Either the client or server MAY remove the TLS layer and leave the LDAP exchange intact by sending and receiving a TLS closure alert.

The initiating protocol peer sends the TLS closure alert. If it wishes to leave the LDAP exchange intact, it then MUST cease to send further PDUs and MUST ignore any received LDAP PDUs until it receives a TLS closure alert from the other peer.

Once the initiating protocol peer receives a TLS closure alert from the other peer it MAY send and receive LDAP PDUs.

When a protocol peer receives the initial TLS closure alert, it may choose to allow the LDAP exchange to remain intact. In this case, it MUST immediately transmit a TLS closure alert. Following this, it MAY send and receive LDAP PDUs.

Protocol peers MAY close the connection after sending or receiving a TLS closure alert.

After the TLS layer has been removed, the server MUST NOT send responses to any request message received before the TLS closure

alert. Thus, clients wishing to receive responses to messages sent while the TLS layer is intact MUST wait for those message responses before sending the TLS closure alert.

4.14.3.2. Abrupt Removal

Either the client or server MAY abruptly remove the TLS layer by closing the connection. In this circumstance, a server MAY send the client a Notice of Disconnection before closing the connection.

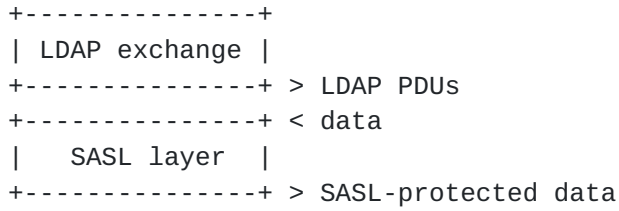
5. Protocol Encoding, Connection, and Transfer

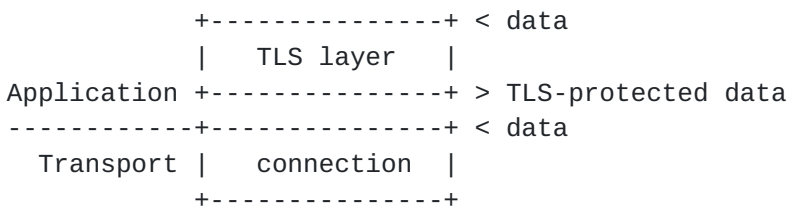
This protocol is designed to run over connection-oriented, reliable transports, where the data stream is divided into octets (8-bit units), with each octet and each bit being significant.

One underlying service, LDAP over TCP, is defined in Section 5.3. This service is generally applicable to applications providing or consuming X.500-based directory services on the Internet. This specification was generally written with the TCP mapping in mind. Specifications detailing other mappings may encounter various obstacles.

Implementations of LDAP over TCP MUST implement the mapping as described in Section 5.3.

This table illustrates the relationship between the different layers involved in an exchange between two protocol peers:





5.2. Protocol Encoding

The protocol elements of LDAP SHALL be encoded for exchange using the Basic Encoding Rules [BER] of [ASN.1] with the following restrictions:

- Only the definite form of length encoding is used.

- OCTET STRING values are encoded in the primitive form only.

- If the value of a BOOLEAN type is true, the encoding of the value octet is set to hex "FF".

- If a value of a type is its default value, it is absent. Only some BOOLEAN and INTEGER types have default values in this protocol definition.

These restrictions are meant to ease the overhead of encoding and decoding certain elements in BER.

These restrictions do not apply to ASN.1 types encapsulated inside of OCTET STRING values, such as attribute values, unless otherwise stated.

5.3. Transmission Control Protocol (TCP)

The encoded LDAPMessage PDUs are mapped directly onto the [TCP] bytestream using the BER-based encoding described in Section 5.2. It is recommended that server implementations running over the TCP provide a protocol listener on the Internet Assigned Numbers Authority (IANA)-assigned LDAP port, 389 [PortReg]. Servers may instead provide a listener on a different port number. Clients MUST support contacting servers on any valid TCP port.

6. Security Considerations

This version of the protocol provides facilities for simple authentication using a cleartext password, as well as any [SASL]

mechanism. Installing SASL layers can provide integrity and other data security services.

It is also permitted that the server can return its credentials to the client, if it chooses to do so.

Use of cleartext password is strongly discouraged where the underlying transport service cannot guarantee confidentiality and may result in disclosure of the password to unauthorized parties.

Servers are encouraged to prevent directory modifications by clients that have authenticated anonymously [AuthMeth].

Security considerations for authentication methods, SASL mechanisms, and TLS are described in [AuthMeth].

It should be noted that SASL authentication exchanges do not provide data confidentiality nor integrity protection for the version or name fields of the bind request nor the resultCode, diagnosticMessage, or referral fields of the bind response nor of any information contained in controls attached to bind request or responses. Thus information contained in these fields SHOULD NOT be relied on unless otherwise

protected (such as by establishing protections at the transport layer).

Server implementors should plan for the possibility of (protocol or external) events which alter the information used to establish security factors (e.g., credentials, authorization identities, access controls) during the course of the LDAP exchange, and even during the performance of a particular operation, and should take steps to avoid insecure side effects of these changes. The ways in which these issues are addressed are application and/or implementation specific.

Implementations which cache attributes and entries obtained via LDAP MUST ensure that access controls are maintained if that information is to be provided to multiple clients, since servers may have access control policies which prevent the return of entries or attributes in search results except to particular authenticated clients. For example, caches could serve result information only to the client whose request caused it to be in the cache.

Servers may return referrals or search result references which redirect clients to peer servers. It is possible for a rogue application to inject such referrals into the data stream in an attempt to redirect a client to a rogue server. Clients are advised to be aware of this, and possibly reject referrals when confidentiality measures are not in place. Clients are advised to reject referrals from the StartTLS operation.

The matchedDN and diagnosticMessage fields, as well as some resultCode values (e.g., attributeOrValueExists and entryAlreadyExists), could disclose the presence the specific data in the directory which is subject to access and other administrative controls. Server implementations should restrict access to protected information equally under both normal and error conditions.

Protocol peers MUST be prepared to handle invalid and arbitrary length protocol encodings. Invalid protocol encodings include: BER encoding exceptions, format string and UTF-8 encoding exceptions, overflow exceptions, integer value exceptions, and binary mode on/off flag exceptions. The LDAPv3 PROTOS [PROTOS-LDAP] test suite provides excellent examples of these exceptions and test cases used to discover flaws.

7. Acknowledgements

This document is based on RFC 2251 by Mark Wahl, Tim Howes, and Steve Kille. RFC 2251 was a product of the IETF ASID Working Group.

It is also based on RFC 2830 by Jeff Hodges, RL "Bob" Morgan, and Mark Wahl. RFC 2830 was a product of the IETF LDAPEXT Working Group.

It is also based on RFC 3771 by Roger Harrison, and Kurt Zeilenga. RFC 3771 was an individual submission to the IETF.

This document is a product of the IETF LDAPBIS Working Group. Significant contributors of technical review and content include Kurt Zeilenga, Steven Legg, and Hallvard Furuseth.

8. Normative References

[ABNF] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[ASN.1] ITU-T Recommendation X.680 (07/2002) | ISO/IEC 8824-1:2002 "Information Technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation"

[AuthMeth] Harrison, R., "LDAP: Authentication Methods and Connection Level Security Mechanisms", draft-ietf-ldapbis-authmeth-xx.txt, (a work in progress).

[BER] ITU-T Rec. X.690 (07/2002) | ISO/IEC 8825-1:2002, "Information technology - ASN.1 encoding rules:

Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", 2002.

[IP] Postel, J., "Internet Protocol", STD5 and RFC 791, September 1981

[ISO10646] Universal Multiple-Octet Coded Character Set (UCS) - Architecture and Basic Multilingual Plane, ISO/IEC 10646-1 : 1993.

[Keyword] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.

[LDAPDN] Zeilenga, K., "LDAP: String Representation of Distinguished Names", draft-ietf-ldapbis-dn-xx.txt, (a work in progress).

[LDAPIANA] Zeilenga, K., "IANA Considerations for LDAP", draft-ietf-ldapbis-bcp64-xx.txt, (a work in progress).

[LDAPURL] Smith, M., "LDAP: Uniform Resource Locator", draft-ietf-ldapbis-url-xx.txt, (a work in progress).

[Models] Zeilenga, K., "LDAP: Directory Information Models", draft-ietf-ldapbis-models-xx.txt (a work in progress).

[Roadmap] Zeilenga, K., "LDAP: Technical Specification Road Map", draft-ietf-ldapbis-roadmap-xx.txt (a work in progress).

[SASL] Melnikov, A., "Simple Authentication and Security Layer", draft-ietf-sasl-rfc2222bis-xx.txt (a work in progress).

[SASLPrep] Zeilenga, K., "Stringprep profile for user names and passwords", draft-ietf-sasl-saslprep-xx.txt, (a work in progress).

[StringPrep] Hoffman P. and M. Blanchet, "Preparation of Internationalized Strings ('stringprep')", draft-hoffman-rfc3454bis-xx.txt, a work in progress.

[Syntaxes] Legg, S., and K. Dally, "LDAP: Syntaxes and Matching Rules", draft-ietf-ldapbis-syntaxes-xx.txt, (a work in progress).

[TCP] Postel, J., "Transmission Control Protocol", STD7 and RFC

793, September 1981

[TLS] Dierks, T. and C. Allen. "The TLS Protocol Version 1.1", draft-ietf-tls-rfc2246-bis-xx.txt, a work in progress.

[Unicode] The Unicode Consortium, "The Unicode Standard, Version 3.2.0" is defined by "The Unicode Standard, Version 3.0" (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5), as amended by the "Unicode Standard Annex #27: Unicode 3.1" (http://www.unicode.org/reports/tr27/) and by the "Unicode Standard Annex #28: Unicode 3.2" (http://www.unicode.org/reports/tr28/).

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD63 and RFC3629, November 2003.

[X.500] ITU-T Rec. X.500, "The Directory: Overview of Concepts, Models and Service", 1993.

[X.501] ITU-T Rec. X.501, "The Directory: Models", 1993.

[X.511] ITU-T Rec. X.511, "The Directory: Abstract Service Definition", 1993.

9. Informative References

[Glossary] The Unicode Consortium, "Unicode Glossary", <http://www.unicode.org/glossary/>.

[CharModel] Whistler, K. and M. Davis, "Unicode Technical Report #17, Character Encoding Model", UTR17, <http://www.unicode.org/unicode/reports/tr17/>, August 2000.

[PROTOS-LDAP] University of Oulu, "PROTOS Test-Suite: c06-ldapv3" <http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/ldapv3/>

[PortReg] IANA, "Port Numbers", http://www.iana.org/assignments/port-numbers

10. IANA Considerations

It is requested that the Internet Assigned Numbers Authority (IANA) update the LDAP result code registry to indicate that this document provides the definitive technical specification for result codes 0-36, 48-54, 64-70, 80-90.

It is requested that the IANA update the LDAP Protocol Mechanism registry to indicate that this document and [AuthMeth] provides the definitive technical specification for the StartTLS (1.3.6.1.4.1.1466.20037) extended operation.

It is requested that the IANA update the occurrence of "RFC XXXX" in Appendix B with this RFC number at publication.

11. Editor's Address

Jim Sermersheim
Novell, Inc.
1800 South Novell Place
Provo, Utah 84606, USA
jimse@novell.com
+1 801 861-3088

This normative appendix details additional considerations regarding LDAP result codes and provides a brief, general description of each LDAP result code enumerated in Section 4.1.9.

Additional result codes MAY be defined for use with extensions [LDAPIANA]. Client implementations SHALL treat any result code which they do not recognize as an unknown error condition.

A.1 Non-Error Result Codes

These result codes (called "non-error" result codes) do not indicate an error condition:

success (0),
compareFalse (5),
compareTrue (6),
referral (10), and
saslBindInProgress (14).

The success, compareTrue, and compareFalse result codes indicate successful completion (and, hence, are referred to as "successful" result codes).

The referral and saslBindInProgress result codes indicate the client is required to take additional action to complete the operation.

A.2 Result Codes

Existing LDAP result codes are described as follows:

success (0)
: Indicates the successful completion of an operation. Note: this code is not used with the compare operation. See compareFalse (5) and compareTrue (6).

operationsError (1)
: Indicates that the operation is not properly sequenced with relation to other operations (of same or different type).

  For example, this code is returned if the client attempts to StartTLS [TLS] while there are other uncompleted operations or if a TLS layer was already installed.

protocolError (2)
: Indicates the server received data which is not well-formed.

  For bind operation only, this code is also used to indicate that the server does not support the requested protocol version.

For extended operations only, this code indicates that the server does not support (by design or configuration) the extended operation associated with the requestName.

For request operations specifying multiple controls, this may be used to indicate that the server cannot ignore the order of the controls as specified, or that the combination of the specified controls is invalid or unspecified.

timeLimitExceeded (3)
Indicates that the time limit specified by the client was exceeded before the operation could be completed.

sizeLimitExceeded (4)
Indicates that the size limit specified by the client was exceeded before the operation could be completed.

compareFalse (5)
Indicates that the compare operation has successfully completed and the assertion has evaluated to FALSE or Undefined.

compareTrue (6)
Indicates that the compare operation has successfully completed and the assertion has evaluated to TRUE.

authMethodNotSupported (7)
Indicates that the authentication method or mechanism is not supported.

strongAuthRequired (8)
Indicates that the server has detected that an established security association between the client and server has unexpectedly failed or been compromised, or that the server now requires the client to authenticate using a strong(er) mechanism.

referral (10)
Indicates that a referral needs to be chased to complete the operation (see Section 4.1.10).

adminLimitExceeded (11)
Indicates that an administrative limit has been exceeded.

unavailableCriticalExtension (12)
Indicates a critical control is unrecognized (see Section

4.1.11).

confidentialityRequired (13)
Indicates that data confidentiality protections are required.

saslBindInProgress (14)

Indicates the server requires the client to send a new bind request, with the same SASL mechanism, to continue the authentication process (see Section 4.2).

noSuchAttribute (16)
Indicates that the named entry does not contain the specified attribute or attribute value.

undefinedAttributeType (17)
Indicates that a request field contains an unrecognized attribute description.

inappropriateMatching (18)
Indicates that an attempt was made (e.g. in an assertion) to use a matching rule not defined for the attribute type concerned.

constraintViolation (19)
Indicates that the client supplied an attribute value which does not conform to the constraints placed upon it by the data model.

For example, this code is returned when multiple values are supplied to an attribute which has a SINGLE-VALUE constraint.

attributeOrValueExists (20)
Indicates that the client supplied an attribute or value to be added to an entry, but the attribute or value already exists.

invalidAttributeSyntax (21)
Indicates that a purported attribute value does not conform to the syntax of the attribute.

noSuchObject (32)
Indicates that the object does not exist in the DIT.

aliasProblem (33)
Indicates that an alias problem has occurred. For example,

the code may used to indicate an alias has been dereferenced which names no object.

invalidDNSyntax (34)
Indicates that an LDAPDN or RelativeLDAPDN field (e.g. search base, target entry, ModifyDN newrdn, etc.) of a request does not conform to the required syntax or contains attribute values which do not conform to the syntax of the attribute's type.

aliasDereferencingProblem (36)
Indicates that a problem occurred while dereferencing an alias. Typically an alias was encountered in a situation where it was not allowed or where access was denied.

inappropriateAuthentication (48)
Indicates the server requires the client which had attempted to bind anonymously or without supplying credentials to provide some form of credentials.

invalidCredentials (49)
Indicates that the provided credentials (e.g. the user's name and password) are invalid.

insufficientAccessRights (50)
Indicates that the client does not have sufficient access rights to perform the operation.

busy (51)
Indicates that the server is too busy to service the operation.

unavailable (52)
Indicates that the server is shutting down or a subsystem necessary to complete the operation is offline.

unwillingToPerform (53)
Indicates that the server is unwilling to perform the operation.

loopDetect (54)
Indicates that the server has detected an internal loop (e.g. while dereferencing aliases or chaining an operation).

namingViolation (64)
Indicates that the entry's name violates naming restrictions.

objectClassViolation (65)

Indicates that the entry violates object class restrictions.

notAllowedOnNonLeaf (66)
Indicates that the operation is inappropriately acting upon a non-leaf entry.

notAllowedOnRDN (67)
Indicates that the operation is inappropriately attempting to remove a value which forms the entry's relative distinguished name.

entryAlreadyExists (68)
Indicates that the request cannot be fulfilled (added, moved, or renamed) as the target entry already exists.

objectClassModsProhibited (69)
Indicates that an attempt to modify the object class(es) of an entry's 'objectClass' attribute is prohibited.

For example, this code is returned when a client attempts to modify the structural object class of an entry.

affectsMultipleDSAs (71)
Indicates that the operation cannot be performed as it would affect multiple servers (DSAs).

other (80)
Indicates the server has encountered an internal error.

Appendix B - Complete ASN.1 Definition

This appendix is normative.

```
Lightweight-Directory-Access-Protocol-V3
-- Copyright (C) The Internet Society (2004). This version of
-- this ASN.1 module is part of RFC XXXX; see the RFC itself
-- for full legal notices.
DEFINITIONS
IMPLICIT TAGS
EXTENSIBILITY IMPLIED ::=

BEGIN

LDAPMessage ::= SEQUENCE {
     messageID       MessageID,
     protocolOp      CHOICE {
          bindRequest           BindRequest,
          bindResponse          BindResponse,
          unbindRequest         UnbindRequest,
          searchRequest         SearchRequest,
          searchResEntry        SearchResultEntry,
          searchResDone         SearchResultDone,
```

```
                searchResRef          SearchResultReference,
                modifyRequest         ModifyRequest,
                modifyResponse        ModifyResponse,
                addRequest            AddRequest,
                addResponse           AddResponse,
                delRequest            DelRequest,
                delResponse           DelResponse,
                modDNRequest          ModifyDNRequest,
                modDNResponse         ModifyDNResponse,
                compareRequest        CompareRequest,
                compareResponse       CompareResponse,
                abandonRequest        AbandonRequest,
                extendedReq           ExtendedRequest,
                extendedResp          ExtendedResponse,
                ...,
                intermediateResponse  IntermediateResponse },
           controls       [0] Controls OPTIONAL }

      MessageID ::= INTEGER (0 .. maxInt)

      maxInt INTEGER ::= 2147483647 -- (2^^31 - 1) --

      LDAPString ::= OCTET STRING -- UTF-8 encoded,
                                  -- [ISO10646] characters

      LDAPOID ::= OCTET STRING -- Constrained to <numericoid> [Models]

      LDAPDN ::= LDAPString -- Constrained to <distinguishedName>
                            -- [LDAPDN]

      RelativeLDAPDN ::= LDAPString -- Constrained to <name-component>
```



```
                                  -- [LDAPDN]

      AttributeDescription ::= LDAPString
                               -- Constrained to <attributedescription>
                               -- [Models]

      AttributeValue ::= OCTET STRING

      AttributeValueAssertion ::= SEQUENCE {
           attributeDesc   AttributeDescription,
           assertionValue  AssertionValue }

      AssertionValue ::= OCTET STRING

      PartialAttribute ::= SEQUENCE {
           type       AttributeDescription,
           vals       SET OF value AttributeValue }
```

```
        Attribute ::= PartialAttribute(WITH COMPONENTS {
             ...,
             vals (SIZE(1..MAX))})

        MatchingRuleId ::= LDAPString

        LDAPResult ::= SEQUENCE {
             resultCode         ENUMERATED {
                  success                      (0),
                  operationsError              (1),
                  protocolError                (2),
                  timeLimitExceeded            (3),
                  sizeLimitExceeded            (4),
                  compareFalse                 (5),
                  compareTrue                  (6),
                  authMethodNotSupported       (7),
                  strongAuthRequired           (8),
                       -- 9 reserved --
                  referral                     (10),
                  adminLimitExceeded           (11),
                  unavailableCriticalExtension (12),
                  confidentialityRequired      (13),
                  saslBindInProgress           (14),
                  noSuchAttribute              (16),
                  undefinedAttributeType       (17),
                  inappropriateMatching        (18),
                  constraintViolation          (19),
                  attributeOrValueExists       (20),
                  invalidAttributeSyntax       (21),
                       -- 22-31 unused --
                  noSuchObject                 (32),
                  aliasProblem                 (33),
                  invalidDNSyntax              (34),
                       -- 35 reserved for undefined isLeaf --
                  aliasDereferencingProblem    (36),
                       -- 37-47 unused --
```



```
                  inappropriateAuthentication  (48),
                  invalidCredentials           (49),
                  insufficientAccessRights     (50),
                  busy                         (51),
                  unavailable                  (52),
                  unwillingToPerform           (53),
                  loopDetect                   (54),
                       -- 55-63 unused --
                  namingViolation              (64),
                  objectClassViolation         (65),
```

```
               notAllowedOnNonLeaf          (66),
               notAllowedOnRDN              (67),
               entryAlreadyExists           (68),
               objectClassModsProhibited    (69),
                    -- 70 reserved for CLDAP --
               affectsMultipleDSAs          (71),
                    -- 72-79 unused --
               other                        (80),
               ... },
          matchedDN          LDAPDN,
          diagnosticMessage  LDAPString,
          referral           [3] Referral OPTIONAL }

     Referral ::= SEQUENCE SIZE (1..MAX) OF uri URI

     URI ::= LDAPString     -- limited to characters permitted in
                            -- URIs

     Controls ::= SEQUENCE OF control Control

     Control ::= SEQUENCE {
          controlType             LDAPOID,
          criticality             BOOLEAN DEFAULT FALSE,
          controlValue            OCTET STRING OPTIONAL }

     BindRequest ::= [APPLICATION 0] SEQUENCE {
          version                 INTEGER (1 .. 127),
          name                    LDAPDN,
          authentication          AuthenticationChoice }

     AuthenticationChoice ::= CHOICE {
          simple                  [0] OCTET STRING,
                                  -- 1 and 2 reserved
          sasl                    [3] SaslCredentials,
          ... }

     SaslCredentials ::= SEQUENCE {
          mechanism               LDAPString,
          credentials             OCTET STRING OPTIONAL }

     BindResponse ::= [APPLICATION 1] SEQUENCE {
          COMPONENTS OF LDAPResult,
          serverSaslCreds    [7] OCTET STRING OPTIONAL }
```



```
     UnbindRequest ::= [APPLICATION 2] NULL

     SearchRequest ::= [APPLICATION 3] SEQUENCE {
          baseObject      LDAPDN,
```

```
     scope           ENUMERATED {
          baseObject              (0),
          singleLevel             (1),
          wholeSubtree            (2) },
     derefAliases    ENUMERATED {
          neverDerefAliases       (0),
          derefInSearching        (1),
          derefFindingBaseObj     (2),
          derefAlways             (3) },
     sizeLimit       INTEGER (0 .. maxInt),
     timeLimit       INTEGER (0 .. maxInt),
     typesOnly       BOOLEAN,
     filter          Filter,
     attributes      AttributeSelection }

AttributeSelection ::= SEQUENCE OF selector LDAPString
               -- The LDAPString is constrained to
               -- <attributeSelection> in Section 4.5.1

Filter ::= CHOICE {
     and             [0] SET OF filter Filter,
     or              [1] SET OF filter Filter,
     not             [2] Filter,
     equalityMatch   [3] AttributeValueAssertion,
     substrings      [4] SubstringFilter,
     greaterOrEqual  [5] AttributeValueAssertion,
     lessOrEqual     [6] AttributeValueAssertion,
     present         [7] AttributeDescription,
     approxMatch     [8] AttributeValueAssertion,
     extensibleMatch [9] MatchingRuleAssertion }

SubstringFilter ::= SEQUENCE {
     type           AttributeDescription,
     -- at least one must be present,
     -- initial and final can occur at most once
     substrings     SEQUENCE SIZE (1..MAX) OF substring CHOICE {
          initial [0] AssertionValue,
          any     [1] AssertionValue,
          final   [2] AssertionValue } }

MatchingRuleAssertion ::= SEQUENCE {
     matchingRule    [1] MatchingRuleId OPTIONAL,
     type            [2] AttributeDescription OPTIONAL,
     matchValue      [3] AssertionValue,
     dnAttributes    [4] BOOLEAN DEFAULT FALSE }

SearchResultEntry ::= [APPLICATION 4] SEQUENCE {
     objectName      LDAPDN,
     attributes      PartialAttributeList }
```

```
PartialAttributeList ::= SEQUENCE OF
                         partialAttribute PartialAttribute

SearchResultReference ::= [APPLICATION 19] SEQUENCE
                            SIZE (1..MAX) OF uri URI

SearchResultDone ::= [APPLICATION 5] LDAPResult

ModifyRequest ::= [APPLICATION 6] SEQUENCE {
     object          LDAPDN,
     changes         SEQUENCE OF change SEQUENCE {
          operation       ENUMERATED {
               add     (0),
               delete  (1),
               replace (2) },
          modification    PartialAttribute } }

ModifyResponse ::= [APPLICATION 7] LDAPResult

AddRequest ::= [APPLICATION 8] SEQUENCE {
     entry           LDAPDN,
     attributes      AttributeList }

AttributeList ::= SEQUENCE OF attribute Attribute

AddResponse ::= [APPLICATION 9] LDAPResult

DelRequest ::= [APPLICATION 10] LDAPDN

DelResponse ::= [APPLICATION 11] LDAPResult

ModifyDNRequest ::= [APPLICATION 12] SEQUENCE {
     entry           LDAPDN,
     newrdn          RelativeLDAPDN,
     deleteoldrdn    BOOLEAN,
     newSuperior     [0] LDAPDN OPTIONAL }

ModifyDNResponse ::= [APPLICATION 13] LDAPResult

CompareRequest ::= [APPLICATION 14] SEQUENCE {
     entry           LDAPDN,
     ava             AttributeValueAssertion }

CompareResponse ::= [APPLICATION 15] LDAPResult

AbandonRequest ::= [APPLICATION 16] MessageID

ExtendedRequest ::= [APPLICATION 23] SEQUENCE {
     requestName      [0] LDAPOID,
```

```
             requestValue     [1] OCTET STRING OPTIONAL }

         ExtendedResponse ::= [APPLICATION 24] SEQUENCE {
              COMPONENTS OF LDAPResult,
              responseName     [10] LDAPOID OPTIONAL,
```



```
              responseValue    [11] OCTET STRING OPTIONAL }

         IntermediateResponse ::= [APPLICATION 25] SEQUENCE {
              responseName     [0] LDAPOID OPTIONAL,
              responseValue    [1] OCTET STRING OPTIONAL }

         END
```

Appendix C - Changes

This appendix is non-normative.

This appendix summarizes substantive changes made to RFC 2251 and RFC 2830.

C.1 Changes made to RFC 2251:

This section summarizes the substantive changes made to Sections 1, 2, 3.1, and 4 through the remainder of RFC 2251. Readers should consult [Models] and [AuthMeth] for summaries of changes to other sections.

C.1.1 Section 1

- Removed IESG note. Post publication of RFC 2251, mandatory LDAP authentication mechanisms have been standardized which are sufficient to remove this note. See [AuthMeth] for authentication mechanisms.

C.1.2 Section 3.1 and others

- Removed notes giving history between LDAP v1, v2 and v3. Instead, added sufficient language so that this document can stand on its own.

C.1.3 Section 4

- Clarified where the extensibility features of ASN.1 apply to the protocol. This change also affected various ASN.1 types.
- Removed the requirement that servers which implement version 3 or later MUST provide the 'supportedLDAPVersion' attribute. This statement provided no interoperability advantages.

C.1.4 Section 4.1.1

- There was a mandatory requirement for the server to return a Notice of Disconnection and drop the connection when a PDU is malformed in a certain way. This has been clarified such that the server SHOULD return the Notice of Disconnection, and MUST drop the connection.

C.1.5 Section 4.1.1.1

- Clarified that the messageID of requests MUST be non-zero.

- Clarified when it is and isn't appropriate to return an already used message id. RFC 2251 accidentally imposed synchronous server behavior in its wording of this.

C.1.6 Section 4.1.2

- Stated that LDAPOID is constrained to <numericoid> from [Models].

C.1.7 Section 4.1.5.1 and others

- Removed the Binary Option from the specification. There are numerous interoperability problems associated with this method of alternate attribute type encoding. Work to specify a suitable replacement is ongoing.

C.1.8 Section 4.1.8

- Combined the definitions of PartialAttribute and Attribute here, and defined Attribute in terms of PartialAttribute.

C.1.9 Section 4.1.10

- Renamed "errorMessage" to "diagnosticMessage" as it is allowed to be sent for non-error results.
- Moved some language into Appendix A, and refer the reader there.
- Allowed matchedDN to be present for other result codes than those listed in RFC 2251.

C.1.10 Section 4.1.11

- Defined referrals in terms of URIs rather than URLs.
- Removed the requirement that all referral URIs MUST be equally capable of progressing the operation. The statement was ambiguous and provided no instructions on how to carry it out.
- Added the requirement that clients MUST NOT loop between servers.
- Clarified the instructions for using LDAPURLs in referrals, and in doing so added a recommendation that the scope part be present.

C.1.11 Section 4.1.12

- Specified how control values defined in terms of ASN.1 are to be encoded.
- Noted that the criticality field is only applied to request messages (except unbindRequest), and must be ignored when present on response messages and unbindRequest.
- Added language regarding combinations of controls and the ordering of controls on a message.

- Specified that when the semantics of the combination of controls is undefined or unknown, it results in a protocolError.
- Changed "The server MUST be prepared" to "Implementations MUST be prepared" in the eighth paragraph to reflect that both client and server implementations must be able to handle this (as both parse controls).

C.1.12 Section 4.2

- Mandated that servers return protocolError when the version is not supported.
- Clarified behavior when the simple authentication is used, the name is empty and the password is non-empty.
- Required servers to not dereference aliases for bind. This was added for consistency with other operations and to help ensure data consistency.
- Required that textual passwords be transferred as UTF-8 encoded Unicode, and added recommendations on string preparation. This was to help ensure interoperability of passwords being sent from different clients.

C.1.13 Section 4.2.1

- This section was largely reorganized for readability and language was added to clarify the authentication state of failed and

abandoned bind operations.
- Removed: "If a SASL transfer encryption or integrity mechanism has been negotiated, that mechanism does not support the changing of credentials from one identity to another, then the client MUST instead establish a new connection."
  Each SASL negotiation is, generally, independent of other SASL negotiations. If there were dependencies between multiple negotiations of a particular mechanism, the mechanism technical specification should detail how applications are to deal with them. LDAP should not require any special handling. And if an LDAP client had used such a mechanism, it would have the option of using another mechanism.
- Dropped MUST imperative in paragraph 3 to align with [Keywords].
- Mandated that clients not send non-bind operations while a bind is in progress, and suggested that servers not process them if they are received. This is needed to ensure proper sequencing of the bind in relationship to other operations.

C.1.14 Section 4.2.3

- Moved most error-related text to Appendix A, and added text regarding certain errors used in conjunction with the bind operation.
- Prohibited the server from specifying serverSaslCreds when not appropriate.

C.1.15 Section 4.3

- Required both peers to cease transmission and close the LDAP exchange for the unbind operation.

C.1.16 Section 4.4

- Added instructions for future specifications of Unsolicited Notifications.

C.1.17 Section 4.5.1

- SearchRequest attributes is now defined as an AttributeSelection type rather than AttributeDescriptionList, and an ABNF is provided.
- SearchRequest attributes may contain duplicate attribute descriptions. This was previously prohibited. Now servers are instructed to ignore subsequent names when they are duplicated.

This was relaxed in order to allow different short names and also OIDs to be requested for an attribute.
- The Filter choice SubstringFilter substrings type is now defined with a lower bound of 1.
- The SubstringFilter substrings 'initial, 'any', and 'final' types are now AssertionValue rather than LDAPString. Also, added imperatives stating that 'initial' (if present) must be listed first, and 'final' (if present) must be listed last.
- Clarified the semantics of the derefAliases choices.
- Added instructions for equalityMatch, substrings, greaterOrEqual, lessOrEqual, and approxMatch.

C.1.18 Section 4.5.2

- Recommended that servers not use attribute short names when it knows they are ambiguous or may cause interoperability problems.
- Removed all mention of ExtendedResponse due to lack of implementation.

C.1.19 Section 4.5.3

- Made changes similar to those made to Section 4.1.11.

C.1.20 Section 4.5.3.1

- Fixed examples to adhere to changes made to Section 4.5.3.

C.1.21 Section 4.6

- Removed restriction that required an EQUALITY matching rule in order to perform value delete modifications. It is sufficiently documented that in absence of an equality matching rule, octet equality is used.
- Replaced AttributeTypeAndValues with Attribute as they are equivalent.
- Clarified what type of modification changes might temporarily violate schema.

C.1.22 Section 4.7

- Aligned Add operation with X.511 in that the attributes of the RDN are used in conjunction with the listed attributes to create the entry. Previously, Add required that the distinguished values be

present in the listed attributes.

C.1.23 Section 4.9

- Required servers to not dereference aliases for modify DN. This was added for consistency with other operations and to help ensure data consistency.
- Allow modify DN to fail when moving between naming contexts.
- Specified what happens when the attributes of the newrdn are no present on the entry.

C.1.24 Section 4.10

- Clarified that compareFalse means that the compare took place and the result is false. There was confusion which lead people to believe that an Undefined match resulted in compareFalse.
- Required servers to not dereference aliases for compare. This was added for consistency with other operations and to help ensure data consistency.

C.1.25 Section 4.11

- Explained that since abandon returns no response, clients should not use it if they need to know the outcome.
- Specified that Abandon and Unbind cannot be abandoned.

C.1.26 Section 4.12

- Specified how values of extended operations defined in terms of ASN.1 are to be encoded.
- Added instructions on what extended operation specifications consist of.
- Added a recommendation that servers advertise supported extended operations.

C.1.27 Section 5.2

- Moved referral-specific instructions into referral-related sections.

C.1.28 Section 7

- Reworded notes regarding SASL not protecting certain aspects of

the LDAP bind PDU.
- Noted that Servers are encouraged to prevent directory modifications by clients that have authenticated anonymously [AuthMeth].
- Added a note regarding the scenario where an identity is changed (deleted, privileges or credentials modified, etc.).
- Warned against following referrals that may have been injected in the data stream.
- Noted that servers should protect information equally, whether in an error condition or not, and mentioned specifically; matchedDN, diagnosticMessage, and resultCodes.
- Added a note regarding malformed and long encodings.

C.1.29 Appendix A

- Added "EXTENSIBILITY IMPLIED" to ASN.1 definition.
- Removed AttributeType. It is not used.

C.2 Changes made to RFC 2830:

This section summarizes the substantive changes made to Sections of RFC 2830. Readers should consult [AuthMeth] for summaries of changes to other sections.

C.2.1 Section 2.3

- Removed wording indicating that referrals can be returned from StartTLS
- Removed requirement that only a narrow set of result codes can be returned. Some result codes are required in certain scenarios, but any other may be returned if appropriate.

C.2.1 Section 4.13.3.1

- Reworded most of this section and added the requirement that after the TLS connection has been closed, the server MUST NOT send responses to any request message received before the TLS closure.

C.3 Changes made to RFC 3771:

- In general, all technical language was transferred in whole. Supporting and background language seen as redundant due to its presence in this document was omitted.

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.