The drawings contained in this Recommendation have been done in AUTOCAD

In both cases (chaining result and referral) an administrative authority through its DSA may elect to ignore the request for returning cross references.

10.4.2 *Knowledge inconsistencies*

The Directory has to support consistency-checking mechanisms to guarantee a certain degree of knowledge consistency.

10.4.2.1 *Detection of knowledge inconsistencies*

The kind of inconsistency and its detection varies for the different types of knowledge references.

- Cross and subordinate references:

This type of reference is invalid if the referenced DSA does not have a local naming context with the context prefix contained in the reference. This inconsistency will be detected during the determination of the initial naming context of the name resolution process by the operation progress and reference type components of the **ChainingArgument**.

- Non-specific Subordinate-references:

This type of reference is invalid if the referenced DSA does not have a local naming context whose immediately superior context prefix is contained in the reference, i.e. the reference contains that DSA's local context prefix minus the last RDN. The consistency check is applied as above.

- Superior references:

An invalid superior reference is one which does not form part of a reference path to the root. The maintenance of superior references must be done by external means and is outside the scope of this Recommendation.

Note - It is not always possible to detect an invalid superior reference.

10.4.2.2 *Reporting of knowledge inconsistencies*

If chaining is used in performing a Directory request, all knowledge inconsistencies will be detected by the DSA which holds the invalid knowledge reference, through receiving a **ServiceError** with problem of **invalidReference**.

If a DSA returns a referral which is based on an invalid knowledge reference, the requestor will be returned a **ServiceError** with problem of **invalidReference** if it uses the referral. How the error condition will be propagated to the DSA which stores the invalid reference is not within the scope of this Recommendation.

10.4.2.3 *Treatment of inconsistent knowledge references*

After a DSA has detected an invalid reference it should try to re-establish knowledge consistency. For example, this can be done by simply deleting an invalid cross reference or by replacing it with a correct one which can be obtained using the **requestCrossReferences** mechanisms.

The way in which a DSA actually handles invalid references is a local matter, and outside the scope of this Recommendation.

SECTION 4 - *DSA abstract service*

11 Overview of DSA abstract service

11.1 The abstract service of the directory is fully described in Recommendation X.511. When such a service is provided in a distributed environment, as modelled in 7 of this Recommendation, it can be regarded as being provided by means of a set of DSAs. This is illustrated in Figure 1/X.518.

11.2 To describe this model, the refinement of the **directory** object into its component **dsa** objects can be expressed as:

```
DirectoryRefinement ::= REFINE directory AS
        dsa             RECURRING
            readPort                [S] VISIBLE
            searchPort              [S] VISIBLE
            modifyPort              [S] VISIBLE
            chainedReadPort         PAIRED with dsa
            chainedSearchPort       PAIRED with dsa
            chainedModifyPort       PAIRED with dsa
```

11.3 The **dsa** object itself can be defined as follows:

```
dsa OBJECT
        PORTS { readPort            [S],
                searchPort          [S],
                modifyPort          [S],
                chainedReadPort,
                chainedSearchPort,
                chainedModifyPort}
::= id-ot-dsa
```

The DSA supplies Read, Search and Modify ports, thus making visible those services to the users of the directory object, namely the DUAs. In addition, a DSA supports "chained" versions of these ports, namely Chained Read, Chained Search, and Chained Modify, which allow DSAs to propagate requests for those services to other DSAs.

11.4 The ports cited from 11.2 and 11.3 (excluding those which are defined in Recommen- dation X.511) are defined as follows:

```
chainedReadPort     PORT
        ABSTRACT OPERATIONS {
                ChainedRead, ChainedCompare,
                ChainedAbandon}
        ::=     id-pt-chained-read
chainedSearchPort   PORT
        ABSTRACT OPERATIONS {
                ChainedList, ChainedSearch}
        ::=     id-pt-chained-search
chainedModifyPort   PORT
        ABSTRACT OPERATIONS {
                ChainedAddEntry,
                ChainedRemoveEntry,
                ChainedModifyEntry,
                ChainedModifyRDN}
        ::= id-pt-chained-modify
```

12 Information types

12.1 *Introduction*

12.1.1 This paragraph identifies, and in some cases defines, a number of information types which are subsequently used in the definition of various of the operations of the DSA abstract service. The information types concerned are those which are common to more than one operation, are likely to be in the future, or which are sufficiently complex or self-contained as to merit being defined separately from the operation which uses them.

12.1.2 Several of the information types used in the definition of the DSA abstract service are actually defined elsewhere. 12.2 identifies these types and indicates the source of their definition. Each of the remaining (12.3 to 12.9) identifies and defines an information type.

12.2 *Information types defined elsewhere*

12.2.1 The following information types are defined in Recommendation X.501:

a) **aliasedObjectName;**

b) **DistinguishedName;**

c) **Name;**

d) **RelativeDistinguishedName.**

12.2.2 The following information types are defined in Recommendation X.511:

(Abstract-bind)

a) **DirectoryBind;**

(Abstract-operations)

b) **Abandon;**

(Abstract-errors)

c) **Abandoned;**

d) **AttributeError;**

e) **NameError;**

f) **SecurityError;**

g) **ServiceError;**

h) **UpdateError;**

(Macro)

i) **OPTIONALLY-SIGNED;**

(Data Type)

j) **SecurityParameters.**

12.2.3 The following information type is defined in Recommendation X.520:

a) **PresentationAddress.**

12.3 *Chaining arguments*

12.3.1 The **ChainingArguments** are present in each Chained abstract-operation, to convey to a DSA the information needed to successfully perform its part of the overall task:

```
ChainingArguments ::= SET {
        originator              [0]  DistinguishedName OPTIONAL,
        targetObject            [1]  DistinguishedName OPTIONAL,
        operationProgress       [2]  OperationProgress DEFAULT {notStarted},
        traceInformation        [3]  TraceInformation,
        aliasDereferenced       [4]  BOOLEAN DEFAULT FALSE,
        aliasedRDNs             [5]  INTEGER OPTIONAL,
                -- absent unless aliasDereferenced is TRUE
        returnCrossRefs         [6]  BOOLEAN DEFAULT FALSE,
        referenceType           [7]  ReferenceType DEFAULT superior,
        Info                    [8]  DomainInfo OPTIONAL,
        timeLimit               [9]  UTCTime OPTIONAL,
                                [10] SecurityParameters DEFAULT {}}
```

12.3.2 The omponents have the meanings as defined in 12.3.2.1 to 12.3.2.11.

12.3.2.1 The **originator** component conveys the name of the (ultimate) originator of the request, unless already specified in the security parameters. If **requestor** is present in **CommonArguments**, this argument may be omitted.

12.3.2.2 The **targetObject** component conveys the name of the object whose directory entry is being routed to. The role of this object depends on the particular abstract-operation concerned: it may be the object whose entry is to be operated on, or which is to be the base object for a request or sub- request involving multiple objects (e.g. **ChainedList** or **ChainedSearch**). This component may be omitted only if it would have had the same value as the base object parameter in **XArgument** (see 14.3.1), in which case its implied value is that value.

12.3.2.3 The **operationProgress** component is used to inform the DSA of the progress of the operation, and hence of the role which it is expected to play in its overall performance. The information conveyed in this component is specified in 12.5.

12.3.2.4 The **traceInformation** component is used to prevent looping among DSAs when chaining is in operation. A DSA adds a new element to trace information prior to chaining an operation to another DSA. On being requested to perform an operation, a DSA checks, by examination of the trace information, that the operation has not formed a loop. The information conveyed in this component is specified in 12.6.

12.3.2.5 The **aliasDereferenced** component is a Boolean value which is used to indicate whether or not one or more alias entries have so far been encountered and dereferenced during the course of distributed name resolution. The default value of **FALSE** indicates that no alias entry has been dereferenced.

12.3.2.6 The **aliasedRDNs** component indicates how many of the RDNs in the **targetObject Name** have been generated from the **aliasedObjectName** attributes of one (or more) alias entries. The integer value is set whenever an alias entry is encountered and dereferenced. This component shall be present if and only if the **aliasDereferenced** component is **TRUE**.

12.3.2.7 The **returnCrossRefs** component is a Boolean value which indicates whether or not knowledge references, used during the course of performing a distributed operation, are requested to be passed back to the initial DSA, as cross references, along with a result or referral. The default value of **FALSE** indicates that such knowledge references are not to be returned.

12.3.2.8 The **referenceType** component indicates, to the DSA being asked to perform the abstract- operation, what type of knowledge was used to route the request to it. The DSA may therefore be able to detect errors in the knowledge held by the invoker. If such an error is detected it shall be indicated by a **ServiceError** with the **invalidReference** problem. **ReferenceType** is described fully in 12.7.

Note - If the **referenceType** is missing, then the value **superior** shall be assumed.

12.3.2.9 The **info** component is used to convey DMD-specific information among DSAs which are involved in the processing of a common request. This component is of type **DomainInfo**, which is of unrestricted type:

DomainInfo ::= ANY

12.3.2.10 The **timeLimit** component, if present, indicates the time by which the operation is to be completed.

12.3.2.11 The **SecurityParameters** component is specified in Recommendation X.511. Its absence is deemed equivalent to there being an empty set of security parameters.

12.4 *Chaining results*

12.4.1 The **ChainingResults** are present in the result of each abstract-operation and provide feedback to the DSA which invoked the abstract-operation.

```
ChainingResults ::= SET {
    Info            [0]  DomainInfo  OPTIONAL,
    crossReferences [1]  SEQUENCE OF CrossReference OPTIONAL,
                    [2]  SecurityParameters DEFAULT {}}
```

12.4.2 The various components have the meanings as defined in 12.4.2.1 to 12.4.2.3.

12.4.2.1 The **info** component is used to convey DMD-specific information among DSAs which are involved in the processing of a common request. This component is of type **DomainInfo**, which is of unrestricted type.

12.4.2.2 The **crossReferences** component is not present in the **ChainingResults** unless the **returnCrossRefs** component of the corresponding request had the value **TRUE**. This component consists of a sequence of **CrossReference** items, each of which contains a **contextPrefix** and an **accessPoint** descriptor (see 12.8).

```
CrossReference ::= SET{
    contextPrefix [0]  DistinguishedName,
    accessPoint   [1]  AccessPoint}
```

A **CrossReference** may be added by a DSA when it matches part of the **targetObject** argument of an abstract-operation with one of its context prefixes. The administrative authority of a DSA may have a policy not to return such knowledge, and will in this case not add an item to the sequence.

12.4.2.3 The **SecurityParameters** component is specified in Recommendation X.511. Its absence is deemed equivalent to there being an empty set of security parameters.

12.5 *Operation progress*

12.5.1 An **OperationProgress** value describes the state of progress in the performance of an abstract-operation which several DSAs must participate in.

```
OperationProgress ::= SET {
        nameResolutionPhase [0]
                ENUMERATED {
                        notStarted      (1),
                        proceeding      (2),
                        completed       (3)},
        nextRDNToBeResolved [1]
                INTEGER OPTIONAL}
```

12.5.2 The various components have the meanings as defined in 12.5.2.1 and 12.5.2.2.

12.5.2.1 The **nameResolutionPhase** component indicates what phase has been reached in handling the **targetObject** name of an operation. Where this indicates that name resolution has **notStarted**, then a DSA has not hitherto been reached with a naming context containing the initial RDN(s) of the name. If name resolution is **proceeding**, then the initial part of the name has been recognized, though the DSA holding the target object has not yet been reached. The **nextRDNToBeResolved** indicates how much of the name has already been recognized (12.5.2.2). If name resolution is **completed**, then the DSA holding the target object has been reached, and performance of the operation proper is proceeding.

12.5.2.2 The **nextRDNToBeResolved** indicates to the DSA which of the RDNs in the **targetObject** name is the next to be resolved. It takes the form of an integer in the range one to the number of RDNs in the name. This component is only present if the **nameResolutionPhase** component has the value **proceeding**.

12.6 *Trace information*

12.6.1 A **TraceInformation** value carries forward a record of the DSAs which have been involved in the performance of an operation. It is used to detect the existence of, or avoid, loops which might arise from inconsistent knowledge or from the presence of alias loops in the DIT.

```
TraceInformation        ::=     SEQUENCE OF TraceItem
TraceItem                       ::=     SET {
        dsa                             [0]  Name,
        targetObject            [1]  Name OPTIONAL,
        operationProgress       [2]  OperationProgress }
```

12.6.2 Each DSA which is propagating an operation to another adds a new item to the trace information. Each such **TraceItem** contains:

a) the **Name** of the dsa which is adding the item;

b) the **targetObject Name** which the DSA adding the item received on the incoming request. This parameter is omitted if the query being chained came from a DUA (in which case its implied value is the **object** or **baseObject** in **XOperation**), or if its value is the same as the (actual or implied) **targetObject** in the **ChainingArgument** of the outgoing request;

c) the **operationProgress** which the DSA adding the item received on the incoming request.

12.7 *Reference type*

12.7.1 A **ReferenceType** value indicates one of the various kinds of reference defined in 10.

```
ReferenceType ::=

        ENUMERATED {
                superior                (1),
                subordinate             (2),
                cross                   (3),
                nonSpecificSubordinate  (4)}
```

12.8 *Access point*

12.8.1 An **AccessPoint** value identifies a particular point at which access to the Directory, specifically to a DSA, can occur. The access point has a **Name**, that of the DSA concerned, and a **PresentationAddress**, to be used in OSI communications to that DSA.

```
AccessPoint     ::=     SET {
                ae-title    [0]  Name,
                address     [1]  PresentationAddress }
```

12.9 *Continuation reference*

12.9.1 A **ContinuationReference** describes how the performance of all or part of an abstract- operation can be continued at a different DSA or DSAs. It is typically returned as a referral when the DSA involved is unable or unwilling to propagate the request itself.

```
ContinuationReference     ::=   SET {
    targetObject          [0]     Name,
    aliasedRDNs           [1]     INTEGER OPTIONAL,
    operationProgress     [2]     OperationProgress,
    rdnsResolved          [3]     INTEGER OPTIONAL,
    referenceType         [4]     ReferenceType OPTIONAL,
    -- only present in the DSP
    accessPoints          [5]     SET OF AccessPoint}
```

12.9.2 The various components have the meanings as defined in 12.9.2.1 to 12.9.2.6.

12.9.2.1 The **targetObject Name** which is proposed to be used in continuing the operation. This might be different from the **targetObject Name** received on the incoming request if, for example, an alias has been dereferenced, or the base object in a search has been located.

12.9.2.2 The **aliasedRDNs** component indicates how many (if any) of the RDNs in the target object name have been produced by dereferencing an alias. The argument is only present if an alias has been dereferenced.

12.9.2.3 The **operationProgress** which has been achieved, and which will govern the further performance of the abstract-operation by the DSAs named, should the DSA or DUA receiving the **ContinuationReference** follow it up.

12.9.2.4 The **rdnsResolved** component value, (which need only be present if some of the RDNs in the name have not been the subject of full name resolution, but have been assumed to be correct from a cross reference) indicates how many RDNs have actually been resolved, using internal references only.

12.9.2.5 The **referenceType** component, which is only present in the DSA abstract service, indicates what type of knowledge was used in generating this continuation.

12.9.2.6 The **accessPoints** component indicates the access points which are to be followed up to achieve this continuation. Where Nonspecific Subordinate References are involved there may be more than one **AccessPoint** listed, and each should be followed up, e.g. by multicasting.

13 Abstract-bind and abstract-unbind

DSABind and **DSAUnbind**, respectively, are used by a DSA at the beginning and at the end of a period accessing another DSA.

13.1 *DSA bind*

13.1.1 A **DSABind** abstract-bind-operation is used by a DSA to bind its **chainedRead**, **chainedSearch**, and **chainedModify** ports to those of another DSA.

```
DSABind         ::=     ABSTRACT-BIND
    TO                  {chainedRead,
                         chainedSearch,
                         chainedModify}
    DirectoryBind
```

13.1.2 The components of the **DSABind** are identical to their counterparts in the **DirectoryBind** (see Recommendation X.511) with the following differences.

13.1.2.1 The **Credentials** of the **DirectoryBindArgument** allows information identifying the AE-Title of the initiating DSA to be sent to the responding DSA. The AE-Title must be in the form of a Directory Distinguished Name.

13.1.2.2 The **Credentials** of the **DirectoryBindResult** allows information identifying the AE-Title of the responding DSA to be sent to the initiating DSA. The AE-Title must be in the form of Distinguished Name.

13.2 *DSA unbind*

13.2.1 A **DSAUnbind** operation is used to unbind the Chained Read, Chained Search and Chained Modify ports of a pair of DSAs.

```
DSAUnbind    ::=    ABSTRACT-UNBIND
    FROM            {chainedRead,
                     chainedSearch,
                     chainedModify}
```

13.2.2 There are no arguments, results or errors.

14 Chained abstract-operations

14.1 Corresponding to each of the ports of the Directory abstract service is a port of the DSA which allows the abstract service to be provided by cooperating DSAs. The abstract-operations in the corresponding ports are also in one-to-one correspondence. The names of the ports and the abstract- operations have been chosen to reflect this correspondence, with the port or abstract-operation in the DSA abstract service being formed from that of the Directory abstract service by prefixing the word "Chained". The resulting ports and abstract-operations are as follows:

```
ChainedReadPort:        ChainedRead,
                        ChainedCompare,
                        ChainedAbandon

ChainedSearchPort:      ChainedList,
                        ChainedSearch

ChainedModifyPort:
                        ChainedAddEntry,
                        ChainedRemoveEntry,
                        ChainedModifyEntry,
                        ChainedModifyRDN
```

14.2 The arguments, results, and errors of the chained abstract-operation are, with one exception, formed systematically from the arguments, results, and errors of the corresponding abstract- operations in the Directory abstract service (as described in 14.3). The one exception is the **ChainedAbandon** abstract-operation, which is syntactically equivalent to its Directory abstract- service counterpart (described in 14.4).

14.3 A **ChainedX** abstract-operation is used to propagate between DSAs a request which (normally) originated as a DUA invoking an **X** abstract-operation at a DSA, that DSA having elected to chain it. The arguments of the abstract-operation may optionally be signed by the invoker, and, if so requested, the performing DSA may sign the results.

14.3.1 The systematic derivation of a Chained abstract-operation **ChainedX** from its counterpart **X** is as follows:

given:

```
X    ::=
     ABSTRACT-OPERATION
          ARGUMENT      XArgument
          RESULT        XResult
          ERRORS  {...,  Referral,...}
```

the Chained abstract-operation is derived as:

```
ChainedX        ::=
    ABSTRACT-OPERATION
            ARGUMENT OPTIONALLY-SIGNED SET{
                    ChainingArgument,
                    [0]  XArgument}
    RESULT  OPTIONALLY-SIGNED SET{
                    ChainingResult,
                    [0] XResult}
    ERRORS {...,DsaReferral,...}
```

Note - The definitive specification of the DSA abstract service in Annex A applies this derivation in full to the Chained abstract-operations.

14.3.2 The arguments of the derived abstract-operation have the meanings as described in 14.3.2.1 and 14.3.2.2.

14.3.2.1 The **ChainingArgument** contains that information, over and above the original DUA-supplied arguments, which is needed in order for the performing DSA to carry out the operation. This information type is defined in 12.3.

14.3.2.2 The **XArgument** contains the original DUA-supplied arguments, as specified in the appropriate clause of Recommendation X.511.

14.3.3 Should the request succeed, the result will be returned. The result parameters have the meanings as described in 14.3.3.1 and 14.3.3.2.

14.3.3.1 The **ChainingResult** contains the information, over and above that to be supplied to the originating DUA, which may be needed by previous DSAs in a chain. This information type is defined in 12.4.

14.3.3.2 The **XResult** contains the result which is being returned by the performer of this abstract- operation, and which is intended to be passed back in the result to the originating DUA. This information is as specified in the appropriate clause of Recommendation X.511.

14.3.4 Should the request fail, one of the listed errors will be returned. The set of errors which may be reported are as described for the corresponding abstract-operation in Recommendation X.511, except that **DSAReferral** is returned instead of **Referral**. The various errors are defined or referenced in 15.

14.4 A **ChainedAbandon** abstract-operation is used by one DSA to indicate to another that it is no longer interested in having a previously invoked chained operation performed. This may be for any of a number of reasons, of which the following are examples:

a) the operation which led to the DSA originally chaining has itself been abandoned, or has implicitly been aborted by the breakdown of an association;

b) the DSA has obtained the necessary information in another way, e.g. from a faster responding DSA involved in a multicast.

A DSA is never obliged to issue a **ChainedAbandon**, or indeed to actually abandon an operation if requested to do so.

If **ChainedAbandon** actually succeeds in stopping the performance of an operation, then a result will be returned, and the subject operation will return an **Abandoned** abstract-error. If the **ChainedAbandon** does not succeed in stopping the operation, then it itself will return an **AbandonFailed** error.

15 Chained abstract-errors

15.1 *Introduction*

15.1.1 For the most part, the same abstract-errors can be returned in the DSA abstract service which can be returned in the Directory abstract-service. The exceptions are that the **DSAReferral** "error" is returned (see 15.2), instead of **Referral**, and the following service problems have the same abstract syntax but different semantics.

a) **invalidReference**.

b) **loopDetected.**

15.1.2 The precedence of the abstract-errors which may occur is as for their precedence in the Directory abstract service, as specified in Recommendation X.511.

15.2 *DSA Referral*

15.2.1 The **DSAReferral** abstract-error is generated by a DSA when, for whatever reason, it doesn't wish to continue performing an abstract-operation by chaining or multicasting the abstract-operation to one or more other DSAs. The circumstances where it may return a referral are described in 8.4.

```
DSAReferral    ::=
    ABSTRACT-ERROR
        PARAMETER SET{
            [0]  ContinuationReference,
            contextPrefix [1] DistinguishedName OPTIONAL }
```

15.2.2 The various parameters have the meanings as described in 15.2.2.1 and 15.2.2.2.

15.2.2.1 The **ContinuationReference** contains the information needed by the invoker to propagate an appropriate further request, perhaps to another DSA. This information type is specified in 12.9.

15.2.2.2 If the **returnCrossRefs** component of the **ChainingArguments** for this abstract-operation had the value **TRUE**, and the referral is being based upon a subordinate or cross-reference, then the **contextPrefix** parameter may optionally be included. The administrative authority of any DSA will decide which knowledge references, if any, can be returned in this manner (the others, for example, may be confidential to that DSA).

SECTION 5 - *Distributed operations procedures*

16 Introduction

16.1 *Scope and limits*

This paragraph specifies the procedures for distributed operation of the Directory which are performed by DSAs. Each DSA individually performs the procedures described below: the collective action of all DSAs produces the full set of services provided to users by the Directory.

The description of procedures for a single DSA is based on the models in 7 to 10 of this Recommendation.

It should be noted that the model and procedures are included for expositional purposes only and are not intended to constrain or govern the implementation of an actual DSA.

This paragraph is divided into three sub-paragraphs: this introduction, a conceptual model for describing directory behaviour and an introduction of both DSA-Centred and Operation-Centred models of DSA operations.

16.2 *Conceptual model*

The complexity of the Directory's distributed operation gives rise to a need for conceptual modelling using both narrative and pictorial descriptive techniques. However, neither the narrative nor graphic diagrams should be construed as a formal description of distributed directory operation.

16.3 *Individual and cooperative operation of DSAs*

The model views DSA operation from two separate perspectives, which, taken together, provide a complete, operational picture of the Directory.

a) DSA-Centred Perspective. In this perspective the set of procedures that support the directory is described from the viewpoint of a single DSA. This makes it possible to provide a definitive specification of each procedure and to fully account for their interrelationships and overall control structure. 18 describes the DSA procedures from a DSA-centred perspective.

b) Operation-Centred Perspective. The DSA-centred view provides complete detail but makes it difficult to understand the structure of individual operations, which may undergo processing by multiple DSAs. Consequently 17 adopts a primarily operation-centred view to introduce the processing phases applicable to each.

To support the distributed operation of the directory, each DSA must perform actions needed to realize the intent of each operation and additional actions needed to distribute that realization across multiple DSAs. 17 explores the distinction between these two kinds of actions. In 18 both kinds of actions are specified in detail.

17 Distributed directory behaviour

17.1 *Cooperative fulfillment of operations*

Each DSA is equipped with procedures capable of completely fulfilling all Directory operations. In the case that a DSA contains the entire DIB all operations are, in fact, completely carried out within that DSA. In the case that the DIB is distributed across multiple DSAs the completion of a typical operation is fragmented, with just a portion of that operation carried out in each of potentially many cooperating DSAs.

In the distributed environment, the typical DSA sees each operation as a transitory event; the operation is invoked by a DUA or some other DSA; the DSA carries out processing on the object and then directs it toward another DSA for further processing.

An alternate view considers the total processing experienced by an operation during its fulfillment by multiple, cooperating DSAs. This perspective reveals the common processing phases that apply to all operations.

17.2 *Phases of operation processing*

Every Directory operation may be thought of as comprising three distinct phases:

a) the Name Resolution phase - in which the name of the object on whose entry a particular operation is to be performed is used to locate the DSA which holds the entry;

b) the Evaluation phase - in which the operation specified by a particular directory request (e.g. read) is actually performed;

c) the Results Merging phase - in which the results of a specified operation are returned to the requesting DUA. If a chaining mode of interaction was chosen, the Results Merging phase may involve several DSAs, each of which chained the original request or sub-request (as defined in 17.3.1 Request Decomposition) to another DSA during either or both of the preceding phases.

In the case of the operations **Read**, **Compare**, **List**, **Search**, and **ModifyEntry**, name resolution takes place on the object name provided in the argument of the operation. In the case of **AddEntry**, **RemoveEntry**, and **ModifyRDN**, name resolution takes place on the name of the immediately superior object (derived by removing the final RDN from the name provided in the operation argument).

An operation on a particular entry may initially be directed at any DSA in the Directory. That DSA used its knowledge, possibly in conjunction with other DSAs to process the operation through the three phases.

17.2.1 *Name resolution phase*

Name Resolution is the process of sequentially matching each RDN in a purported Name to an arc (or vertex) of the DIT, beginning logically at the Root and progressing downwards in the DIT. However, because the DIT is distributed between arbitrarily many DSAs, each DSA may only be able to perform a fraction of the name resolution process. A given DSA performs its part of the Name Resolution process by traversing its local knowledge. This process is described in 18.6 and the accompanying diagrams (Figures 11/X.518 to 13/X.518). When a DSA reaches the border of its naming context, it will know from the knowledge information contained therein, whether the resolution can be continued by another DSA or whether the name is erroneous.

17.2.2 *Evaluation phase*

When the name resolution phase has been completed, the actual operation required (e.g. read or search) is performed.

Operations that involve a single entry - **Read**, **Compare**, **AddEntry**, **RemoveEntry**, **ModifyRDN** and **ModifyEntry** - can be carried out entirely within the DSA in which that entry has been located. **AddEntry**, **RemoveEntry** and **ModifyRDN** may affect knowledge in more than one DSA. See 18.7.1.

Operations that involve multiple entries - **List** and **Search** - need to locate subordinates of the target, which may or may not reside in the same DSA. If they do not all reside in the same DSA, operations need to be directed to the DSAs specified in the subordinate references to complete the evaluation process.

17.2.3 *Results merging phase*

The results merging phase is entered once some of the results of the evaluation phase are available.

In those cases where the operation affected only a single entry, the result of the operation can simply be returned to the requesting DUA. In those cases where the operation has affected multiple entries on multiple DSAs, results need to be combined.

The permissible responses returned to a requestor after results merging include:

a) a complete result of the operation;

b) a result which is not complete because some parts of the DIT remain unexplored (applies to **List** and **Search** only). Such a partial result may include continuation references for those parts of the DIT not explored;

c) an error (a referral being a special case);

d) and if the requestor was a DSA, a ChainingResult.

17.3 *Managing distributed operations*

Information is included in the argument of each abstract-operation which a DSA may be asked to perform indicating the progress of each operation as it traverses various of the DSAs of the Directory. This makes it possible for each DSA to perform the appropriate aspect of the processing required, and to record the completion of that aspect before directing the operation outward toward further DSAs.

Additional procedures are included in the DSA to physically distribute the operations and support other needs arising from their distribution.

17.3.1 *Request decomposition*

Request decomposition is a process performed internally by a DSA prior to communication with one or more other DSAs. A request is decomposed into several sub-requests such that each of the latter accomplishes a part of the original task. Request decomposition can be used, for example, in the search operation, after the base object has been found. After decomposition, each of the sub- requests may then be chained or multicast to other DSAs, to continue the task.

17.3.2 *DSA as Request responder*

A DSA that receives a request can check the progress of that request using the Operation Progress parameter. This will determine whether the operation is still in the name resolution phase or has reached the evaluation phase, and what portion of the operation the DSA should attempt to satisfy. If the DSA cannot fully satisfy the request it must either pass the operation on to one or more DSAs which can help to fulfill the request (by chaining or multicasting) or return a referral to another DSA or terminate the request with an error.

17.3.3 *Completion of operations*

Each DSA that has initiated an operation or propagated an operation to one or more other DSAs must keep track of that operation's existence until each of the other DSAs has returned a result or error, or the operation's maximum time limit has expired. This requirement applies to all operations, propagation modes and processing phases. It ensures the orderly closing down of distributed operations that have propagated out into the Directory.

17.4 *Other considerations for distributed operation*

17.4.1 *Request validation*

On receipt of a directory operation a DSA must initially validate the operation to ensure that it can be progressed. Circumstances such as loops within the DIT caused by inappropriate use of aliases or the use of erroneous knowledge may cause operations to be sent to DSAs that cannot be processed.

In the simple case these erroneous circumstances are adequately handled by name resolution procedures as described in 18. However, where circumstances cause operations to loop (as described in 17.4.3) name resolution alone is inadequate.

The request validation actions ensures that a loop is detected before any attempt is made to progress an operation through the erroneous data caused by the loop. The detection process is carried out by the loop detection procedure specified in 18.5.1.

Where security procedures are in force request validation also verifies the identity of the requesting DSA or DUA, and the validity of the request.

17.4.2 *State and trace information*

The progression of an operation within the directory and the presence of loop conditions are determined by an operation's "state", where state is defined to be the following:

- the name of the DSA currently processing the operation;
- the name of the **targetObject** as contained within the argument of the operation;
- the **operationProgress** as contained within the argument of the operation and as defined in 12.5.

In addition to the current state of an operation, a DSA also needs to know all previous states for that operation. These are recorded in the **traceInformation** argument and conveyed with the operation.

The **traceInformation** argument forms the basis of loop avoidance/detection strategies as specified in 17.4.3.

17.4.3 *Looping*

Within the context of a particular directory operation a loop occurs if at any time the operation returns to a previous state (as defined above). Looping is managed using the **traceInformation** argument. Two strategies are defined to handle loops. In loop detection a DSA determines whether a loop has occurred in an incoming operation and, if so returns an error. In loop avoidance a DSA determines whether an operation, if forwarded, would yield a loop.

17.4.4 *Service controls*

Some service controls need special consideration in the distributed environment in order that the operation is processed the way that was requested.

a) **chainingProhibited**: A DSA consults this service control when determining the mode of propagation of an operation. If it is set then the DSA always uses referral mode. If, however, it is not set, the DSA can choose whether to use chaining or referral depending on its capabilities.

b) **timeLimit**: A DSA needs to take account of this service control to ensure that the time limit is not exceeded in that DSA. A DSA requested to perform an operation by a DUA, initially heeds the **timeLimit** expressed by the DUA as the available elapsed time in seconds for completion of the operation. If chaining is required, the **timeLimit** is included in the chaining argument to be passed to the next DSA(s). In this case the same value of the limit is used for each chained request, and is the (UTC) time by which the operation must be completed to meet the originally specified constraint. On receiving a chaining argument with a **timeLimit** specified, the receiving DSA respects this limit.

c) **sizeLimit**: A DSA needs to take account of this service control to ensure that the list of results does not exceed the size specified. The limit, as included in the common argument of the original request, is conveyed unchanged as the request is chained/multicast. If request decomposition is required, the same value is included in the argument to be passed to the next DSA: that is, the full limit is used for each sub-request. When the results are returned the requestor DSA resolves the multiple results and applies the limit to the total to ensure that only the requested numbers are returned. If the limit has been exceeded, this is indicated in the reply.

d) **Priority**: In all modes of propagation, each DSA is responsible for ensuring that the processing of operations is ordered so as to support this service control if present.

e) **localScope**: The operation is limited to a locally defined scope and cannot be propagated by any of the modes.

f) **scopeOfReferral**: If the DSA returns a referral or partial result to a **List** or **Search** operation, then the embedded **ContinuationReference**s shall be within the requested scope.

All other service controls need to be respected, but their use does not require any special consideration in the distributed environment.

17.4.5 *Extensions*

17.4.5.1 If a DSA encounters an extended abstract-operation in the name resolution phase of processing and determines that the abstract-operation should be chained to one or more other DSAs, it shall include unchanged in the chained abstract-operation any extensions present.

Note - An Administrative Authority may determine that it is appropriate to return a **ServiceError** with problem **unwillingToPerform** if it does not wish to propagate an extension.

17.4.5.2 If a DSA encounters an extension in the execution phase of processing, two possibilities may arise. If the extension is not critical, the DSA shall ignore the extension. If the extension is critical, the DSA shall return a **ServiceError** with problem **unavailableCriticalExtension**.

A critical extension to a multiple object operation may result in both results and service errors of this variety. A DSA merging such results and errors shall discard these service errors and employ the **unavailableCriticalExtension** component of **PartialOutcomeQualifier** as described in 10.1.1 of Recommendation X.511.

17.4.6 *Alias Dereferencing*

Alias dereferencing is the process of creating a new target object name, by replacing the alias entry distinguished name part of the original target object name with the Aliased Object Name attribute value from the alias entry. The object name in the operation is not affected by alias dereferencing.

17.5 *Authentication of distributed operations*

Users of the Directory together with administrative authorities that provide directory services may, at their discretion, require that directory operations be authenticated. For any particular directory operation the nature of the authentication process will depend upon the security policy in force.

Two sets of authentication procedures are available which collectively enable a range of authentication requirements to be met. One set of procedures are those provided by Bind: these facilitate authentication between two directory application-entities for the purposes of establishing an association. The Bind procedures accommodate a range of authentication exchanges from a simple exchange of identities to strong authentication.

In addition to the peer entity authentication of an association as provided by Bind, additional procedures are defined within the directory to enable individual operations to be authenticated. Two distinct sets of directory authentication procedures are defined. One facilitates originator authentication services, which address the authentication, by a DSA, of the initiator of the original service request. The second set facilitates results authentication services which address the authentication, by an initiator, of any results that are returned.

For originator authentication two procedures are defined, one based upon a simple exchange of identities, termed identity based authentication, and one based upon digital signature techniques, termed signature based authentication. The former of these procedures is rudimentary in nature since the identity exchange is based upon the exchange of distinguished names which are transmitted in the clear.

For authentication of results a single results authentication procedure is defined, based upon digital signature techniques; due to the generally complex nature of results collation a simpler, identity-based procedure is not defined.

Authentication of error responses is not supported by these procedures.

The services described above are to be considered as augmenting those provided by the Bind service; Bind procedures are assumed to have been effected successfully prior to authentication of directory operations.

The procedures to be effected by a DSA in providing originator and results authentication are specified in 18.9.

18 DSA behaviour

18.1 *Introduction*

Corresponding to each operation invoked by a requestor (e.g. DUA or DSA) the performing DSA must behave in accordance with well-defined procedures so that an appropriate response will be returned deterministically. This paragraph specifies the allowed behaviour by modelling a DSA in terms of processes implementing a particular collection of procedures. It is important to realize that a DSA need conform only to the externally visible behaviour implied by these procedures, and not to the procedures themselves.

18.2 *Overview of the DSA behaviour*

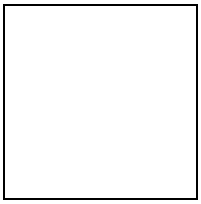
The behaviour of the distributed Directory as a whole is the sum of the behaviour of its cooperating DSAs. Each of these DSAs can be viewed as a process, supported internally by a set of procedures.

Figure 6/X.518 illustrates the internal view of the DSA behaviour.

The Operation Dispatcher is the main controlling procedure in a DSA. It guides each operation through the three phases of processing described in 17.2.

The procedures which support the Operation Dispatcher are: Name Resolution, Find Naming Context, Local Name Resolution, Evaluation, Single Object Evaluation, Multiple Object Evaluation, and Result Merging. The relationships among these procedures are shown graphically in Figure 6/X.518.

FIGURE 6/X.518 - T0704560-88



18.2.1 *The operation dispatcher*

Upon initially receiving an operation, the Operation Dispatcher validates it, checking for loop or authentication errors. If none is found, it calls Name Resolution, which returns either a Found indication, a Reference, or an error indication. References are handled by a referral or by a Chain or Multicast action, Found indications by calling the Evaluation procedure, which actually performs the intended operation. Once returned, internal or external results are collated by Results Merging, and, in the absence of errors, returned to the calling DUA or DSA.

18.2.2 *Name resolution*

Name Resolution calls Find Naming Context. If the returned context is local, then Local Name resolution is called, otherwise Name Resolution returns a reference or an error and terminates. If Local Name Resolution encounters an alias, it is dereferenced (if permitted) and Name Resolution repeats the analysis from the beginning. Otherwise Local Name Resolution returns a Found indication, an error or a Referral, which is passed back to the Operation Dispatcher.

18.2.3 *Find naming context*

Find Naming Context attempts to match the Purported Name against Context Prefixes. If none matches, then Find Naming Context attempts to identify a cross or superior reference. If a context prefix is matched, Find Naming Context returns a cross reference relating downwards in the DIT, or an indication that a suitable naming context was found locally, and sets NameResolutionPhase to "proceeding".

18.2.4 *Local Name Resolution*

The Local Name Resolution procedure attempts to match RDNs in the Purported Name internally until it can return a Found indication. If unable to match all RDNs internally, it attempts to identify first specific, then non-specific subordinate references, and return these to Name Resolution. If an alias is encountered, and dereferencing is allowed by the service controls, a dereferenced alias indication is returned. If dereferencing is not allowed, a Found indication is returned if and only if all RDNs had matched at the time the alias was encountered, otherwise a **nameError** is returned.

18.2.5 *Evaluation*

The Evaluation procedure actually executes the requested Directory operation against the target object. Depending on the type of operation, Single Object Evaluation or Multiple Object Evaluation is invoked.

18.2.6 *Single object evaluation*

Single object evaluation is invoked for **Read**, **Compare**, **AddEntry**, **RemoveEntry**, **ModifyEntry**, and **ModifyRDN**. It is in this procedure that attributes are actually retrieved, checked, or changed.

18.2.7 *Multiple object evaluation*

The Multiple Object Evaluation procedure is invoked for the **Search** and **List** operations to check filters, retrieve results, and if necessary, dispatch sub-requests.

18.2.8 *Result merging*

The Results Merging procedure collates results or errors received from other DSAs with locally retrieved results.

18.3 *Specific operations*

The operations fall into three categories of operations (in each case the operation and its Chained counterpart are both in the same category).

a) Single-Object Operations: Read, Compare, AddEntry, ModifyEntry, ModifyRDN, RemoveEntry.

b) Multiple-Object Operations: List, Search.

c) Abandon Operation, i.e. Abandon.

The handling of these categories are described in 18.3.1 to 18.3.3 respectively. Since there is considerable similarity between the way that a DSA behaves in performing an operation of a service-port and in performing its counterpart chained operation of a chained service-port, there is a single description applying to both, with exceptions to this rule being noted.

18.3.1 *Single-object operations*

Single-object operations are those which affect a single entry, and which therefore can be carried out entirely within the DSA which contains the entry on which the operation is to be performed. Such operations can be commonly described by the following sequence of events:

1) Activate the Operation Dispatcher.

2) Perform Name Resolution to locate the object whose name was specified as the argument of the operation.

3) Perform the single-object evaluation procedure.

4) Service controls, such as time limit, should be checked during the course of the operation to enforce the constraints specified by the user.

5) Return the results to the DUA or DSA which forwarded the request.

18.3.2 *Multiple-object operations*

Multiple-object operations are those which affect several entries which may or may not be co- located in the same DSA. Such operations may thus entail a cooperative effort by several DSAs to locate and operate on all the entries affected by the requested operation. The common behaviour of such operations can be summarized as follows:

1) Activate the Operation Dispatcher.

2) Perform the Name Resolution procedures to locate the object whose name was specified as the argument of operation.

3) Once the target object of the operation has been located, perform the multiple-object evaluation procedures.

4) If request decomposition has taken place in one of the multiple-object evaluation procedures and sub-requests have been chained/multicasted, the Operation Dispatcher maintains the current local results, waits for chained responses, and activates Results Merging.

5) Service Controls such as time limit, size limit should be checked during the course of the operation to remain within the constraints specified in the common argument.

6) Return the results or errors to the DUA or DSA which forwarded the request.

18.3.3 *Abandon operation*

On receipt of an abandon operation, a DSA determines whether it can abandon the specified operation, and, if so, abandons it and returns a result (the operation that was abandoned returns an **Abandoned** error). If it cannot abandon the specified operation, it returns an **AbandonFailed** error.

The following specifies the procedure specific to the **Abandon** operation.

1) Locate the operation whose invoke identifier is specified as the argument of the **Abandon** operation.

2) Optionally compose request(s) with the proper invoke-id to abandon any outstanding chained/multicast operations to other DSAs.

3) Optionally, the abandon operation is performed locally as defined in Recommendation X.511.

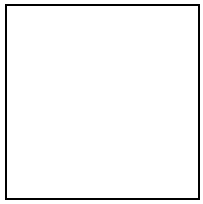
4) Return result or error to the DUA or DSA which forwarded the request.

18.4 *Operation dispatcher*

18.4.1 *Introduction*

The Operation Dispatcher utilizes the Name Resolution described in 18.6 of this Recommendation and all the interactions (i.e. DSA to DSA or DUA to DSA) necessary to locate target entries in a distributed directory environment. Figure 7/X.518 shows a detailed diagram describing the Operation Dispatcher. The algorithm is summarized below.

FIGURE 7/X.518 - T0704571-88



18.4.2 *Implicit actions*

18.4.2.1 *Security*

It should be noted that although the checking of signatures is not explicitly included in this algorithm, this action is always the first step when a signed operation, result or error arrives to the DSA.

Note - This does not include embedded signatures.

Should the signature be invalid, or absent in a case when it should be present, a **SecurityError** is returned. All processing of the operation is terminated and the operation dispatcher goes to its idle state.

The signing of an operation result if required is likewise an implicit last step before sending it off.

18.4.2.2 *ServiceControls*

Although the **ServiceControls** are not explicitly mentioned, they are respected. For example, the checking of the **timeLimit** of an arriving operation and the checking of **sizeLimit** before sending a result are regarded as mandatory. These are discussed in 17.4.4.

18.4.2.3 *TraceInformation*

TraceInformation is always updated with the state it arrived to the DSA in, before including it in the **ChainingArguments**. That is not explicitly stated in the text below.

18.4.3 *Arguments*

Chaining arguments for the particular operation.

18.4.4 *Results*

Chaining results for the particular operation.

18.4.5 *Errors*

Any error defined in this Recommendation.

18.4.6 *Algorithm*

1) Receive operation.

   If the operation originates from another DSA it will comprise the chaining arguments, including: **operationProgress**, **aliasDereferenced**, **aliasedRDNs**, **targetObject Name** and **traceInformation** as well as the parameters contained in the original operation.

   If the operation originates from a DUA it will not contain the **aliasDereferenced** indication: thus adopt the value of **FALSE**. The argument also does not include any **TraceInformation**, so no loop checking needs to be performed. Set **targetObject Name** to the name of the target object for the operation (see 17.2). Other chaining arguments are set according to the parameters in the DAP operation. **Originator** is set to the name of the user.

2) If the operation came from a DSA, check the trace information for loops (activate Loop Detection). If a loop is detected, return **ServiceError** with a problem of **loopDetected** and terminate the processing.

3) Perform security checks to the operation (originating either from a DUA or a DSA). If there is a violation, a **SecurityError** is returned. Otherwise, set **operationProgress** and **aliasDereferenced** according to the operation argument or by default.

4) Perform the Name Resolution Procedure.

   The Name Resolution Procedure will return a found indication, a remote reference, or an error indication.

5) One of the following errors may be raised:

   ServiceError (**UnableToProceed**) - if a DSA determines that it was forwarded an operation pertaining to information which it does not hold.

   ServiceError (**invalidReference**) - if a DSA determines that an invalid knowledge reference was used.

   NameError (**noSuchObject**) - if the purported name specified in the operation request is determined to be invalid.

   NameError (**aliasProblem**) - if an alias has been dereferenced which names no object.

   Name Error (**aliasDereferencingProblem**) - if an alias was encountered in a situation where it is not allowed.

   On receipt of any one of these errors, the Operation Dispatcher terminates and an error is returned to the DSA or DUA which originated the distributed operation.

6) If Found is returned, activate the Evaluation Procedure.

7) If a remote reference is returned (whether from Name Resolution or Evaluation) it may be any one of the following: a cross reference, a subordinate reference, a superior reference or a non-specific subordinate reference.

   If any such reference is returned it signifies that the Name Resolution or Evaluation cannot be completed in this DSA, but must involve the DSA identified in the reference.

   The Operation Dispatcher then checks for referral or chaining mode.

8) If the referral mode or interaction has been selected, then, subject to **scopeOfReferral**, either the information contained in the returned reference will be returned to the originating DUA or DSA as a referral, or **outOfScope ServiceError** will be returned. The processing of this operation will then terminate.

   Note - If **returnCrossRefs** is true and reference is not a non-specific subordinate reference or superior reference and, in addition, the administrative authority is willing to provide knowledge, then the context prefix in the referral can be set.

9) If the chaining mode of interaction has been selected, the operation is forwarded to the DSA specified in the reference. In the case of a non-specific subordinate reference, the operation must be forwarded to each DSA whose name was attained as part of a non-specific subordinate reference. Such forwarding may be accomplished either by multicasting or by sequentially chaining the operation.

10) Perform Loop Avoidance for each operation to be sent. If the avoidance turns out to be not applicable or no loop is detected, assign values to the chaining arguments, including an updated version of traceInformation, and send the operations.

    If no operations were sent (because of looping problems), return a serviceError (with problem of loopDetected) and terminate the processing of this operation.

Note - If the decomposed operation was aborted because of loop avoidance in this step it is a local matter whether to return a partial result or to abort the whole operation and return an error. If the latter is chosen then return **ServiceError** (with problem **loopDetected**) and terminate processing.

11) Wait for the responses then perform the Results Merging procedure.

18.5 *Looping*

Within the context of a particular directory operation a loop occurs if at any time the operation returns to a previous state (as defined in 17.4.2). This does not mean that an operation cannot be processed multiple times by a particular DSA. However, it does mean that the DSA will not process the same operation in the same state multiple times.

Looping is managed using the traceInformation argument as defined in 12.6. Two strategies are defined to determine loops: loop detection and loop avoidance, described in 18.5.1 and 18.5.2 respectively.

18.5.1 *Loop detection*

Loop detection requires that a DSA, when receiving an incoming operation, determines whether the current state of the operation appears in the sequence of previous states recorded in the **traceInformation** argument for that operation. If it does, the operation is looping and a **ServiceError** (with problem of **loopDetected**) is returned. Otherwise the DSA continues processing the operation according to the procedures specified in 18.4.

18.5.2 *Loop avoidance*

Loop avoidance requires that a DSA, immediately prior to forwarding an operation to another DSA (as part of a chaining, multicasting, or request decomposition procedure), determines whether the consequential state of the operation (if known) appears on the sequence of previous states recorded in the trace-information argument for the original incoming operation. The consequential state is the value of **TraceItem** which will be added to **TraceInformation** by the receiving DSA.

In the event that the original incoming operation was to a service-port (rather than a chained- service-port) there will be no trace information and the loop avoidance procedure will not be relevant.

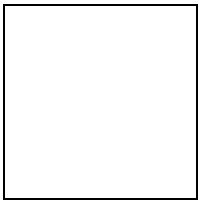
If the consequential state of the operation is known and does appear within the **traceInformation**, the operation, if invoked, would cause a loop. Under this circumstance the appropriate response to the original operation is a **ServiceError** (with problem of **loopDetected**).

18.6 *Name resolution procedure*

This paragraph describes in detail the Name Resolution procedure, its input and output parameters, and its possible error conditions. Figure 7/X.518 shows the overall procedure in the form of a diagram. The Name Resolution procedure calls two component procedures:

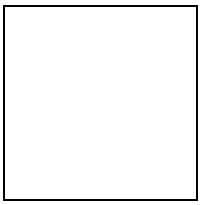
1) Find Naming Context (Figure 8/X.518).

FIGURE 8/X.518 - T0704581-88



2) Local Name Resolution (Figure 9/X.518).

FIGURE 9/X.518 - T0704590-88



The Name Resolution procedure conveys back to the Operation Dispatcher the results of the above mentioned component procedures, except in the following two cases. The first one is when the Find Naming Context procedure identifies a suitable context which has to be further examined, and returns the local naming context. The second case is when the Local Name Resolution procedure indicates that it has dereferenced an alias. In the former case, the Name Resolution procedure calls the Local Name Resolution procedure. In the latter case, the Name Resolution procedure is reactivated with the new target object name.

18.6.1 *Arguments*

The procedure makes use of the following arguments:

- the target object name (the purported name);
- operation progress;
- the value of the **dontDereferenceAliases** service control;
- the value of the **aliasedRDNs** parameter;
- the value of the **aliasDereferenced** parameter.

18.6.2 *Results*

There are two cases of successful outcome.

The first of these returns:

- a reference;
- operation progress (updated appropriately);
- **aliasDereferenced** indication and, optionally, **aliasedRDNs**.

The second of these returns:

- an indication that the naming context was found (together with the local pointer to the entry);
- operation progress (updated appropriately);
- **aliasDereferenced** indication and, optionally, **aliasedRDNs**.

18.6.3 *Errors*

One of the following errors may be returned:

- **ServiceError (unableToProceed);**
- **ServiceError (invalidReference);**
- **NameError (aliasProblem, noSuchObject or aliasDereferencingProblem).**

18.6.4 *Procedure*

1) Activate the Find Naming Context procedure.
2) Wait for response from Find Naming Context procedure.
3) Receive returned results or error, i.e. Local Naming Context Found, Remote Reference, Unable to Proceed Error, Name Error, or invalidReference.

4) Perform functions based on returned results or error.

a) If the local naming context has been found, activate the Local Name Resolution procedure. This procedure may return an Internal Reference Found, a Remote Reference, an Alias Dereference, or a NameError. Each of these causes the Name Resolution to be terminated with the outcome reported, except that if an alias has been dereferenced, the procedure is restarted at step 1).

b) Any other outcome is passed back to the Operation Dispatcher.