INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

X.518

(11/1988)

SERIES X: DATA COMMUNICATION NETWORKS: TRANSMISSION, SIGNALLING AND SWITCHING, NETWORK ASPECTS, MAINTENANCE AND ADMINISTRATIVE ARRANGEMENTS

Data communication networks – Transmission, signalling and switching

THE DIRECTORY – PROCEDURES FOR DISTRIBUTED OPERATION

Reedition of CCITT Recommendation X.518 published in the Blue Book, Fascicle VIII.8 (1988)

NOTES

1 CCITT Recommendation X.518 was published in Fascicle VIII.8 of the *Blue Book*. This file is an extract from the *Blue Book*. While the presentation and layout of the text might be slightly different from the *Blue Book* version, the contents of the file are identical to the *Blue Book* version and copyright conditions remain unchanged (see below).

2 In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Recommendation X.518

THE DIRECTORY – PROCEDURES FOR DISTRIBUTED OPERATION [1]

(Melbourne, 1988)

CONTENTS

[1] Recommendation X.518 and ISO 9594-4, Information Processing Systems – Open Systems Interconnection – The Directory – Procedures for Distributed Operation, were developed in close collaboration and are technically aligned.

0 Introduction

0.1 This document, together with the others of the series, has been produced to facilitate the interconnection of information processing systems to provide directory services. The set of all such systems, together with the directory information which they hold, can be viewed as an integrated whole, called the Directory. The information held by the Directory, collectively known as the Directory Information Base (DIB), is typically used to facilitate communication between, with or about objects such as OSI application entities, people, terminals, and distribution lists.

0.2 The Directory plays a significant role in Open Systems Interconnection, whose aim is to allow, with a minimum of technical agreement outside of the interconnection standards themselves, the interconnection of information processing systems:

– from different manufacturers;

– under different managements;

– of different levels of complexity; and

– of different ages.

0.3 This Recommendation specifies the procedures by which the distributed components of the Directory interwork in order to provide a consistent service to its users.

1 Scope and field of application

1.1 This Recommendation specifies the behaviour of DSAs taking part in the distributed Directory application. The allowed behaviour has been designed so as to ensure a consistent service given a wide distribution of the DIB across many DSAs.

1.2 The Directory is not intended to be a general purpose database system, although it may be built on such systems. It is assumed that there is a considerably higher frequency of queries than of updates.

2 References

Recommendation X.200 – Open Systems Interconnection – Basic Reference Model

Recommendation X.208 – Open Systems Interconnection – Specification of Abstract Syntax Notation (ASN.1)

Recommendation X.500 – The Directory – Overview of Concepts, Models and Services

Recommendation X.501 – The Directory – Models

Recommendation X.511 – The Directory – Abstract Service Definition

Recommendation X.519 – The Directory – Protocol Specifications

Recommendation X.520 – The Directory – Selected Attribute Types

Recommendation X.521 – The Directory – Selected Object Classes

Recommendation X.407 – Message Handling Systems – Abstract Service Definition Conventions

3 Definitions

The definitions contained in this paragraph make use of the abbreviations defined in § 4.

3.1 *OSI Reference Model Definitions*

This Recommendation makes use of the following term defined in X.200:

a) *application entity title*.

3.2 *Basic Directory Definitions*

This Recommendation makes use of the following terms defined in Recommendation X.500:

a) *(the) Directory*;

b) *Directory Information Base*.

3.3 *Directory Model Definitions*

This Recommendation makes use of the following terms defined in Recommendation X.501:

a) *access point*;

b) *alias*;

c) *distinguished name*;

d) *Directory Information Tree*;

e) *Directory System Agent*;

f) *Directory User Agent*;

g) *relative distinguished name*.

3.4 *Abstract Service Definition Conventions*

This Recommendation makes use of the following terms defined in X.407:

a) *abstract error*;

b) *abstract operation*;

c) *result*.

3.5 *Distributed Operation Definitions*

This Recommendation makes use of the following terms, as defined here:

a) *chaining:* a mode of interaction optionally used by a DSA which cannot perform an operation itself. The DSA chains by invoking an operation of another DSA and then relaying the outcome to the original requestor;

b) *context prefix:* the sequence of RDNs leading from the Root of the DIT to the initial vertex of a naming context, corresponds to the distinguished name of that vertex;

c) *cross reference:* a knowledge reference containing information about the DSA that holds an entry. This is used for optimisation. The entry need have no superior or subordinate relationship;

d) *DIB fragment:* the portion of the DIB that is held by one DSA, comprising one or more naming contexts;

e) *distributed name resolution:* the process by which name resolution is performed in more than one DSA;

f) *internal reference:* a knowledge reference containing an internal pointer to an entry held in the same DSA;

g) *knowledge information:* the information which a particular DSA has about the entries it holds and how to locate other entries in the directory;

h) *knowledge reference:* knowledge which associates, either directly or indirectly, a DIT entry with the DSA in which it is located;

i) *knowledge tree:* the conceptual model of the knowledge information that a DSA holds to enable it to perform distributed name resolution;

j) *multicasting:* a mode of interaction which may optionally be used by a DSA which cannot perform an operation itself. The DSA multicasts the operation, i.e. invokes the same operation of several other DSAs (in series or in parallel) and passes an appropriate outcome to the original requestor;

k) *name resolution:* the process of locating an entry by sequentially matching each RDN in a purported name to a vertex of the DIT;

l) *naming context:* a partial sub-tree of the DIT which starts at a vertex and extends downwards to leaf and/or non-leaf vertices. Such vertices constitute the border of the naming context. Non-leaf vertices belonging to the border denote the start of further naming contexts;

m) *non-specific subordinate reference:* a knowledge reference that holds information about the DSA that holds one or more unspecified subordinate entries;

n) *operation progress:* a set of values which denotes the extent to which name resolution has taken place;

o) *reference path:* a continuous sequence of knowledge references;

p) *referral:* an outcome which can be returned by a DSA which cannot perform an operation itself, and which identifies one or more other DSAs more able to perform the operation;

q) *request decomposition:* decomposition of a request into subrequests each accomplishing a part of the distributed operation;

r) *root context:* the naming context for the vertex whose name comprises the empty sequence of RDNs;

s) *subordinate reference:* a knowledge reference containing information about the DSA that holds a specific subordinate entry;

t) *subrequest:* a request generated by request decomposition;

u) *superior reference:* a knowledge reference containing information about the DSA that holds a superior entry.

4 Abbreviations

The following abbreviations are used in this Recommendation:

DIB	Directory Information Base
DIT	Directory Information Tree
DSA	Directory System Agent
DUA	Directory User Agent
RDN	Relative Distinguished Name

5 Notation

The notation used in this paragraph is defined as follows:

a) the data syntax notation, encoding and macro notation are defined in Recommendation X.208;

b) the notations for abstract models and abstract services are defined in Recommendation X.407.

SECTION 2 – *Overview*

6 Overview

The Directory Abstract Service allows the interrogation, retrieval and modification of Directory information in the DIB. This service is described in terms of the abstract Directory object as specified in Recommendation X.511.

Necessarily, the specification of the abstract Directory object does not in any way address the physical realization of the Directory, in particular it does not address the specification of Directory System Agents (DSA) within which the DIB is stored and managed, and through which the service is provided. Furthermore, it does not consider whether the DIB is centralized, i.e. contained within a single DSA, or distributed over a number of DSAs. Consequently, the requirements for DSAs to have knowledge of, navigate to, and cooperate with other DSAs, in order to support the abstract service in a distributed environment, is also not covered by the service description.

This Recommendation specifies the refinement of the abstract Directory object, the refinement being expressed in terms of a set of one or more DSA objects which collectively constitute the distributed directory service. Inherent in this is the identification and specification of the DSA ports that are internal to the Directory object. For each such port, this Recommendation specifies the associated abstract services and its procedures.

In addition, this Recommendation specifies the permissible ways in which the DIB may be distributed over one or more DSAs. For the limiting case where the DIB is contained within a single DSA, the Directory is in fact centralized; for the case where the DIB is distributed over two or more DSAs, knowledge and navigation mechanisms are specified which ensure that the whole of the DIB is potentially accessible from all DSAs that hold constituent entries.

Additionally, request handling interactions are specified that enable particular operational characteristics of the Directory to be controlled by its users. In particular, the user has control over whether a DSA, responding to a directory enquiry pertaining to information held in other DSA(s), has the option of interrogating the other DSA(s) directly (chaining/multicasting) or, whether it should respond with information about other DSA(s) which could further progress the enquiry (referral).

Generally, the decision by a DSA to chain/multicast or refer is determined by the service controls set by the user, and by the DSA's own administrative, operational, or technical circumstances.

Recognizing that, in general, the Directory will be distributed, that directory enquiries will be satisfied by an arbitrary number of cooperating DSAs which may arbitrarily chain/multicast or refer according to the above criteria, this Recommendation specifies the appropriate procedures to be effected by DSAs in responding to distributed directory enquiries. These procedures will ensure that users of the distributed Directory service perceive it to be both user-friendly and consistent.

SECTION 3 – *Distributed directory models*

7 Distributed directory system model

The Directory abstract service as defined in Recommendation X.511 models the directory as an object which provides a set of directory services to its users. The services of the directory are modelled in terms of ports, where each port provides a particular set of directory services. Users of the directory access its services through an access point. The directory may have one or more access points and each access point is characterized by the services it provides and the mode of interaction used to provide these services.

This paragraph addresses the internal structure of the directory object, identifying its constituent objects and their ports, and thereby facilitates the specification of a distributed directory service.

Figure 1/X.518 illustrates the distributed directory which will be used as the basis for specifying the distributed aspects of the directory. It illustrates the directory object as comprising a set of one or more DSA-objects.

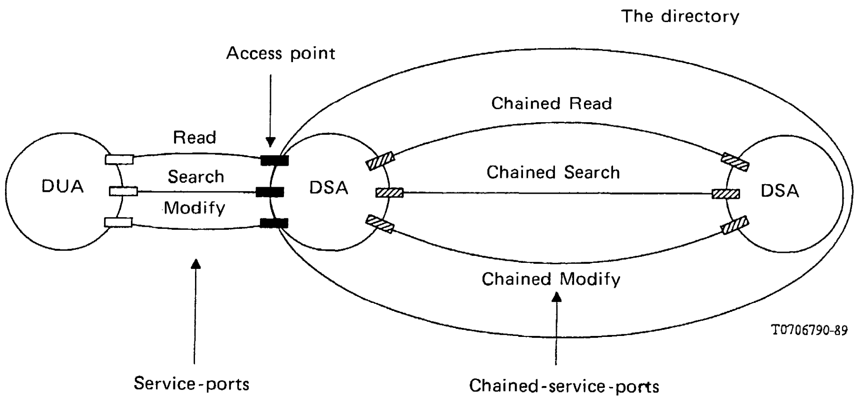


FIGURE 1/X.518

Objects of the distributed directory model

DSA objects are specified in detail in the subsequent clauses of this Recommendation. This clause merely states a number of their characteristics in order to serve as an introduction and to establish the relationship between this Recommendation and other Recommendations.

DSA objects are defined in order that distribution of the DIB can be accommodated and that a number of physically distributed DSAs can interact in a prescribed, cooperative manner to provide directory services to the users of the directory (DUAs).

DSA objects, like the Directory object, are characterized by their externally visible ports. The ports associated with a DSA-object are of two types: service-ports and chained-service-ports.

The service-ports of a DSA object are identical to those of the Directory object, namely, **read**, **search** and **modify**. Figure 1/X.518 illustrates that the service-ports associated with a DSA object constitute an access-point through which directory services are made available.

The detailed specification of the **read**, **search**, and **modify** service-ports of the DSA object can be found in Recommendation X.511. (The protocol specification for the corresponding OSI application service elements, as derived from these port definitions, can be found in Recommendation X.519.)

In addition to the service-ports of the DSA object which accommodate access to the Directory object, a second set of ports are defined, the chained-service-ports. These permit inter-DSA communication in order that the Directory abstract service can be realized in a distributed environment.

The chained-service-ports and the operations provided through them are in direct correspondence to the similarly named service-ports, and are, respectively, **chainedRead**, **chainedSearch**, and **chainedModify**.

The process of specifying the constituent objects of a more abstract object is termed "refinement". The specification of the refinement of the Directory object into its component parts (the DSAs), and the specification of the abstract service provided by each of them (the DSA Abstract Service) is contained in Section Four of this Recommendation. The protocol specification of the corresponding OSI application service elements, as derived from the chained port definitions, can be found in Recommendation X.519.

8 DSA interactions model

A basic characteristic of the Directory is that, given a distributed DIB, a user should potentially be able to have any service request satisfied (subject to security, access control and administrator policies) irrespective of the access point at which the request originates. In accommodating this requirement it is necessary that any DSA involved in satisfying a particular service request have some knowledge (as specified in § 10 of this Recommendation) of where the requested information is located and either return this knowledge to the requestor or attempt to have the request satisfied on its behalf. (The requestor may either be a DUA or another DSA: in the latter case both DSAs must have a chained port.)

Three modes of DSA interaction are defined to meet these requirements, namely "chaining", "multicasting", and "referral". "Chaining" and "multicasting" are defined to meet the latter of the above requirements whilst referrals address the former.

8.1 *Chaining*

This mode of interaction (depicted in Figure 2/X.518) may be used by one DSA, to pass on a request to another DSA when the former has knowledge about naming contexts held by the latter. Chaining may be used to contact a single DSA pointed to in a cross reference, a subordinate reference, or a superior reference. Multicasting is a form of chaining, described in § 8.2.

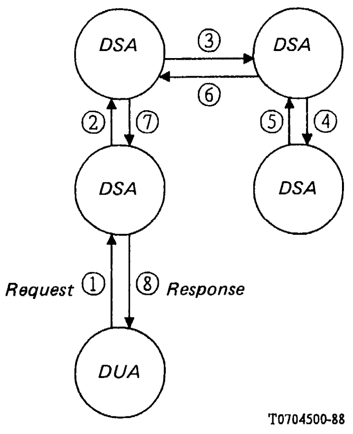


FIGURE 2/X.518

Chaining mode

Note – In Figure 2/X.518, the order of interactions is defined by the numbers associated with the interaction lines.

8.2 *Multicasting*

This mode of interaction (depicted in Figures 3a/X.518 and 3b/X.518) may be used by a DSA, to chain an identical request in parallel (a) or sequential (b) to one or more DSAs, when the former does not know the complete naming contexts held by the other DSAs. Multicasting is only used by a DSA to contact other DSAs pointed to in a non-specific subordinate reference. Each of the DSAs is passed the identical request. Normally, during name resolution, only one of the DSAs will be able to continue processing the remote operation, all of the others returning the **unableToProceed ServiceError**. However, during the evaluation phase of search and list operations, all DSAs in a non-specific subordinate reference should be able to continue processing the request.

Note – In Figures 3a/X.518 and 3b/X.518, the order of interactions is defined by the numbers associated with the interaction lines.

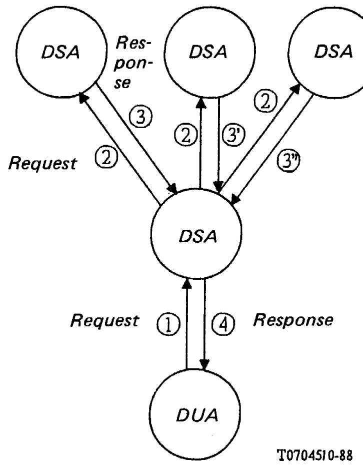


FIGURE 3a/X.518

Multicasting mode

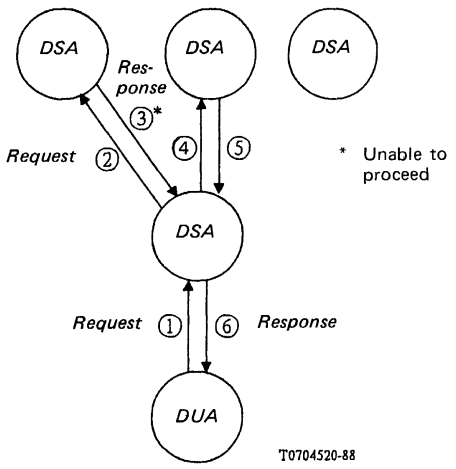


FIGURE 3b/X.518

Multicasting mode

8.3 *Referral*

A referral (depicted in Figures 4a/X.518 and 4b/X.518) is returned by a DSA in its response to a request which it had been requested to perform, either by a DUA, or by another DSA (in which case both DSAs must have a chained-service port). The referral may constitute the whole response (in which case it is categorized as an error) or just part of the response. The referral contains a knowledge reference, which may be either a superior, subordinate, cross or non-specific subordinate reference.

The DSA (Figure 4a/X.518) receiving the referral may use the knowledge reference contained therein, to subsequently chain or multicast (depending upon the type of reference) the original operation to other DSAs. Alternatively, a DSA receiving a referral, may in turn pass the referral back in its response. A DUA (Figure 4b/X.518) receiving a referral may use it to contact one or more other DSAs to progress the request.

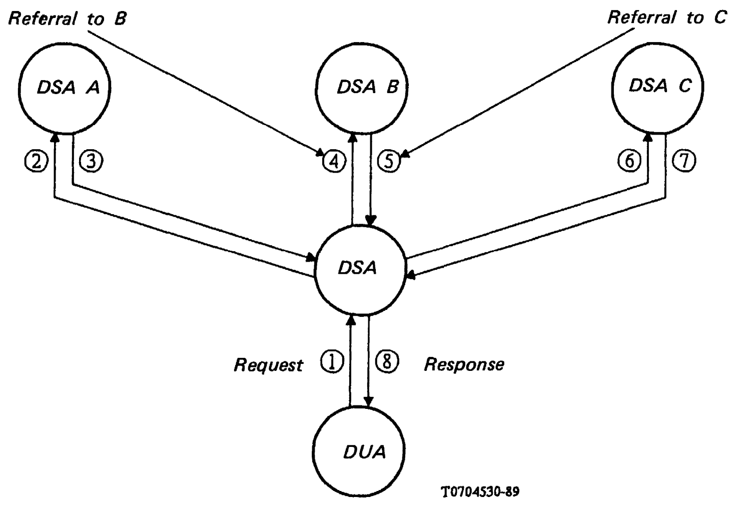


FIGURE 4a/X.518

Referral mode – DSA with chained port

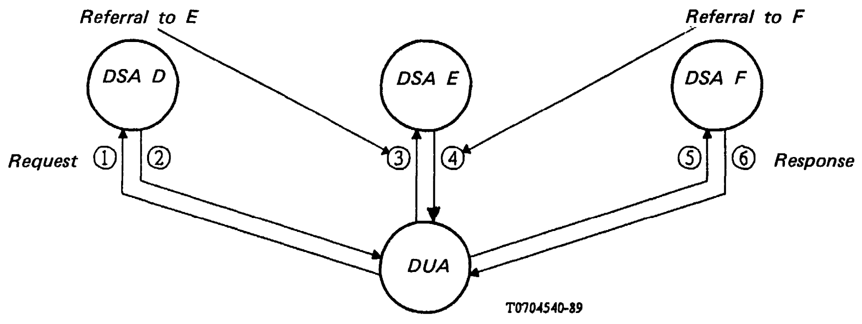


FIGURE 4b/X.518

Referral mode – DUA requests DSAs with no chained ports

Note – In Figures 4a/X.518 and 4b/X.518, the order of interactions is defined by the numbers associated with the interaction lines.

8.4 *Mode determination*

If a DSA cannot itself fully resolve a request, it must chain/multicast the request (or a request formed by decomposing the original one), to another DSA, unless:

a) chaining is prohibited by the user via the service controls, in which case the DSA must return a referral or a **chainingRequired ServiceError** (at its choice), or

b) the DSA has administrative, operational, or technical reasons for preferring not to chain, in which case the DSA must return a referral.

Note 1 – A "technical reason" for not chaining/multicasting is that the DSA identified in the knowledge reference has no chained service ports.

Note 2 – If the **localScope** service control is set, then the DSA (or DMD) must either resolve the request or return an error.

Note 3 – If the user prefers referrals, the user should set **chainingProhibited**.

9 Directory distribution

This paragraph defines the principles according to which the DIB can be distributed.

Each entry within the DIB is administered by one, and only one, DSA's Administrator who is said to have administrative authority for that entry. Maintenance and management of an entry must take place in a DSA administered by the administrative authority for the entry.

Although the Directory does not provide any support for the replication of entries, it is nevertheless possible to realize replication in two ways:

– Copies of an entry may be stored in other DSA(s) through bilateral agreement. The means by which these copies are maintained and managed is a function of the bilateral agreement and is not defined in this Recommendation.

– Copies of an entry may be acquired by storing (locally and dynamically) a copy of an entry which results from a request.

Note – The acquisition of cache entries is subject to access control.

The originator of the request is informed (via **fromCopy**) as to whether information returned in response to a request is from a replicated entry or not. A service control, **dontUseCopy**, is defined which allows the user to prohibit the use of replicated entries.

Each DSA within the Directory holds a fragment of the DIB. The DIB fragment held by a DSA is described in terms of the DIT and comprises one or more naming contexts. A naming context is a partial subtree of the DIT defined as starting at a vertex and extending downwards to leaf and/or non-leaf vertices. Such vertices constitute the border of the naming context. Subordinates of the non-leaf vertices belonging to the border denote the start of further naming contexts.

It is possible for a DSA's administrator to have administrative authority for several disjoint naming contexts. For every naming context for which a DSA has administrative authority, it must logically hold the sequence of RDNs which lead from the root of the DIT to the initial vertex of the subtree comprising the naming context. This sequence of RDNs is called the context prefix.

A DSA's administrator may delegate administrative authority for any immediate subordinates of any entry held locally to another DSA. A DSA that delegated authority is called a superior DSA and the context that holds the superior entry of one for which the administrative authority was delegated, is called the superior naming context. Delegation of administrative authority begins with the root and proceeds downwards in the DIT; that is, it can only occur from an entry to its subordinates.

Figure 5/X.518 illustrates a hypothetical DIT logically partitioned into five naming contexts (named A, B, C, D and E), which are physically distributed over three DSAs (DSA1, DSA2, and DSA3).

From the example it can be seen that the naming contexts held by particular DSAs may be configured so as to meet a wide range of operational requirements. Certain DSAs may be configured to hold those entries that represent higher level naming domains within some logical part(s) of the DIB, the organizational structure of a large company say, but not necessarily all the subordinate entries. Alternatively, DSAs may be configured to hold only those naming contexts representing primarily leaf entries.

From the above definitions, the limiting case for a naming context can be either a single entry or the whole of the DIT.

Whilst the logical to physical mapping of the DIT onto DSAs is potentially arbitrary, the task of information location and management is simplified if the DSAs are configured to hold a small number of naming contexts.

In order for a DUA to begin processing a request it must hold some information, specifically the presentation address, about at least one DSA that it can contact initially. How it acquires and holds this information is a local matter.

During the process of modification of entries it is possible that the directory may become inconsistent. This will be particularly likely if modification involves aliases or aliased objects which may be in different DSAs. The inconsistency must be corrected by specific administrator action, for example to delete aliases if the corresponding aliased objects have been deleted. The Directory continues to operate during this period of inconsistency.

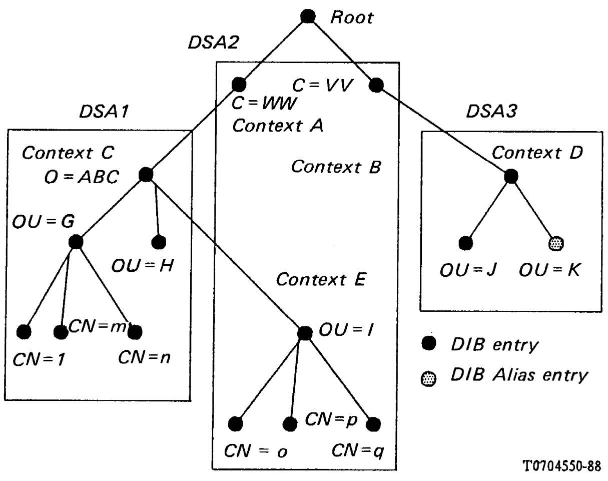


FIGURE 5/X.518

Hypothetical DIT

Note – The Root is not held by any DSA, however some indication must exist at the local level to distinguish those vertices (e.g. C = VV, C = WW) which are immediate subordinates of the Root.

10 Knowledge

The DIB is potentially distributed across multiple DSAs with each DSA holding a DIB fragment; the principles that govern distribution of the DIB are specified in § 9 of this Recommendation.

It is a requirement of the Directory that, for particular modes of user interaction, the distribution of the directory be rendered transparent, thereby giving the effect that the whole of the DIB appears to be within each and every DSA.

In order to support the operational requirements described above, it is necessary that each DSA holding a fragment of the DIB be able to identify and optionally interact with other fragments of the DIB held by other DSAs.

This paragraph defines knowledge as the basis for the mapping of a name to its location within a fragment of the DIT.

Conceptually DSAs hold two types of information:

a) Directory Information;

b) Knowledge Information.

Directory Information is the collection of entries comprising the Naming Context(s) for which the Administrator of a particular DSA has Administrative Authority.

Knowledge Information embodies the Naming Context(s) held by a particular DSA and denotes how these fit into the overall DIT hierarchy. Name Resolution, the process of locating the DSA which has Administrative Authority for a particular entry given that entry's name, is based on knowledge information.

A *Context Prefix* is the sequence of RDNs leading from the Root of the DIT to the initial vertex of a naming context and corresponds to the distinguished name of that vertex.

A *Naming Context* comprises a collection of knowledge references and a Context Prefix. A Naming Context must contain exactly the following knowledge references:

– All the internal references which define the internal structure of the portion of the DIT included in the Naming Context.

– All the subordinate and non-specific subordinate references to other Naming Contexts.

10.1 *Minimal knowledge references*

It is a property of the Directory that each entry can be accessed independently of where a request is generated.

To accomplish this, each DSA shall at least maintain the following knowledge references:

– *subordinate references* as defined in § 10.3.2 and/or *non-specific subordinate references* as defined in § 10.3.5; and

– *superior references* as defined in § 10.3.3.

It is then possible to establish a *reference path*, as a continuous sequence of knowledge references, to all naming contexts within the Directory.

Optionally, *cross references*, as defined in § 10.3.4 may form part of a reference path to optimize performance.

10.2 *Root context*

Because of the autonomy of the different countries or global organizations, there is likely to be no "single" DSA which holds the root context. The functionality of a "root-DSA" concerning the name resolution process has to be provided by those DSAs which have administrative authority for naming contexts that are immediately subordinate to the root. These DSAs are called First Level DSAs. Each First Level DSA must be able to simulate the functionality of the "root-DSA". This requires full knowledge about the root naming context. The root context is replicated onto each *First Level DSA* and therefore has to be administered commonly by the autonomous first level administrative authorities. Administration procedures have to be determined by multilateral agreements outside the scope of this Recommendation.

– Each first level DSA shall hold the root context, which implies a reference path to each other first level DSA.

– Each non-first level DSA shall have a superior reference, which implies a reference path to any arbitrary first level DSA.

10.3 Knowledge references

The knowledge possessed by a DSA is defined in terms of a set of one or more knowledge references where each reference associates, either directly or indirectly, entries of the DIB with DSAs which hold those entries.

To be able to fulfill the requirements to reach every DIB entry from any DSA, every DSA is required to have knowledge about the entries which it itself holds, and about subordinates and possibly superiors thereof. This gives rise to the following types of knowledge references:

– Internal references

– Subordinate references

– Superior references

– Non-specific subordinate references.

Additionally, for optimization purposes the following type of optional reference is defined:

– Cross references

In the event that the set of knowledge references associated with a particular DSA contain only internal references, the DSA has no knowledge of other DSAs and the DIB is therefore centralized.

10.3.1 *Internal references*

An *internal reference* consists of:

– the RDN corresponding to a DIB entry;

– an internal pointer to where the entry is stored in the local DIB. (The specification of this pointer is outside the scope of this Recommendation.)

All entries for which a particular DSA has Administrative Authority are represented by internal references in the knowledge information of that DSA.

10.3.2 *Subordinate references*

A *subordinate reference* consists of:

– an RDN corresponding to an immediate subordinate DIB entry;

– the Access Point of the DSA to which Administrative Authority for that entry was delegated.

All subordinate entries held by another DSA to which this DSA has delegated Administrative Authority, must be represented by subordinate references (or non-specific subordinate references as described in § 10.3.5).

10.3.3 *Superior references*

A *superior reference* consists of:

– the Access Point of a DSA.

Each non-first level DSA maintains precisely one superior reference. The superior reference shall form part of a reference path to the root. Unless some method outside of the standard is employed to ensure this, for example within a DMD, this shall be accomplished by referring to a DSA which holds a naming context whose context prefix has less RDNs than the context prefix with fewest RDNs held by this DSA.

If a new non-first level DSA is introduced, it must have a minimal initial knowledge, which is represented by the superior reference. Any further knowledge will be added by subordinate references or cross references (as described in § 10.3.4). If a new first level DSA is introduced, it must acquire the root context and advise all other first level DSAs. How this is accomplished is outside the scope of this Recommendation.

10.3.4 *Cross reference*

A *cross reference* consists of:

– a Context Prefix;

– the Access Point of a DSA which has Administrative Authority for that Naming Context.

This type of reference is optional and serves to optimize Name Resolution. A DSA may hold any number (including zero) of cross references.

10.3.5 *Non-specific subordinate references*

A *non-specific subordinate* reference consists of:

– The Access Point of a DSA which holds one or more immediately subordinate Naming Contexts.

This type of reference is optional, to allow for the case in which a DSA is known to contain some subordinate entries but the specific RDNs of those entries is not known.

For each naming context which it holds, a DSA may hold any number (including zero) of non-specific subordinate references, which will be evaluated if all specific internal and subordinate references have been pursued. DSAs accessed via a non-specific reference must be able to resolve the request directly (either success or failure). In the event of failure a **ServiceError** reporting a problem of **unableToProceed** is returned to the requestor.

10.4 *Knowledge administration*

To operate a widely distributed Directory with an acceptable degree of consistency and performance, procedures are required to maintain and extend the knowledge held by each DSA. The same procedures are appropriate for creating initial knowledge.

Knowledge can be maintained by:

a) The DSA or its administrative authority propagating changes of knowledge to those DSAs holding all kinds of references to it, whenever changes at that DSA cause the references to become invalid. This is the only way superior, subordinate and non-specific subordinate references can be maintained.

b) DSAs requesting and obtaining cross references to improve the performance through ordinary directory operations.

This Recommendation does not define any procedures for propagating knowledge changes as described in a). Bilateral agreements must be established locally for this.

10.4.1 *Requesting cross reference*

To improve the performance of the Directory System, the local set of cross references can be expanded using ordinary Directory operations. If a DSA has a chained port it may request another DSA (which also must have a chained port) to return those knowledge references which contain information about the location of naming contexts related to the target object name of an ordinary Directory operation.

If the **returnCrossReference** component of the **ChainingArgument** is set to **TRUE**, the **crossReference** component of the **ChainingResult** may be present, consisting of a sequence of cross reference items.

If a DSA is not able to chain a request to the next DSA a referral is returned to the originating DSA. If the **returnCrossReference** component of the chaining argument was **TRUE**, the referral may contain additionally the context prefix of the naming context which the referral refers to. The **contextPrefix** component is absent if the referral is based on a non-specific subordinate reference. The cross reference returned by a referral is only based on knowledge held by the DSA which generated the referral.

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 Recommendation 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 various components 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) he 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) he Evaluation phase – in which the operation specified by a particular directory request (e.g. read) is actually performed;

c) he 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) 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.

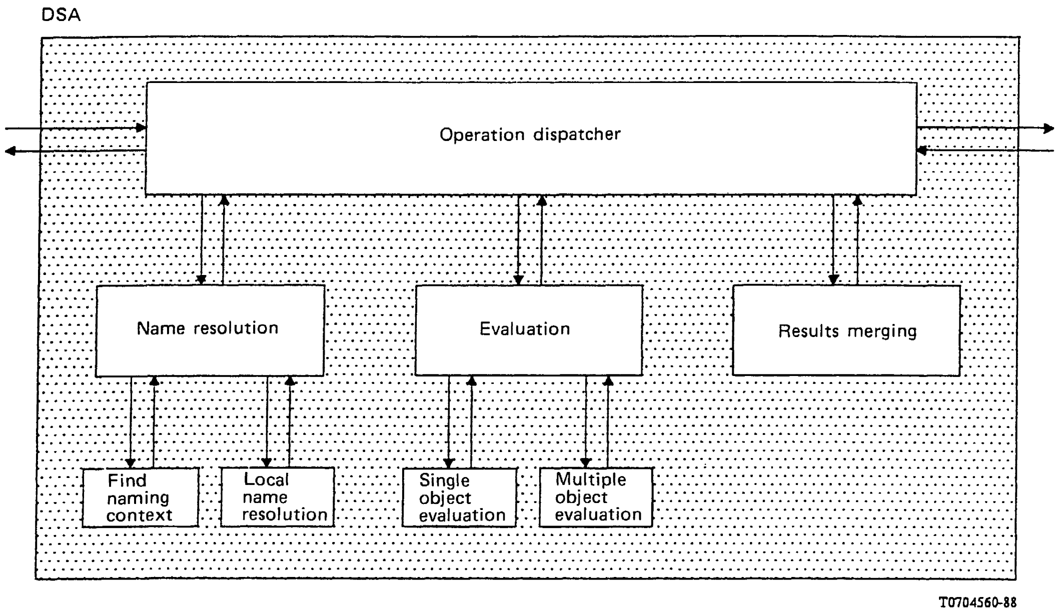


FIGURE 6/X.518

DSA Behaviour – Internal view

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.

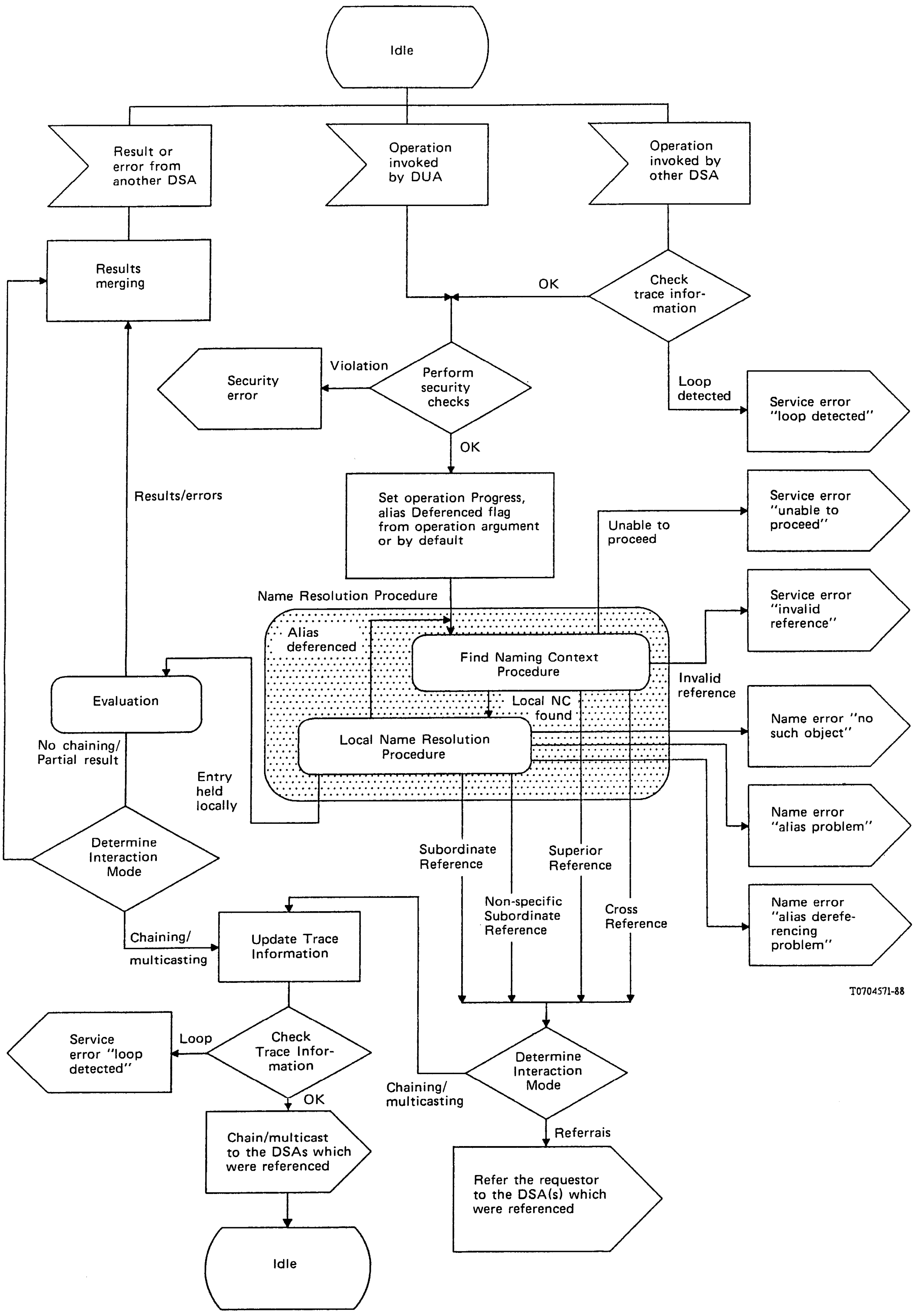

Idle
Result or error from another DSA
Operation invoked by DUA
Operation invoked by other DSA
Results merging
OK
Check trace information
Security error
Violation
Perform security checks
Loop detected
Service error "loop detected"
OK
Results/errors
Set operation Progress, alias Deferenced flag from operation argument or by default
Service error "unable to proceed"
Unable to proceed
Name Resolution Procedure
Service error "invalid reference"
Alias deferenced
Find Naming Context Procedure
Invalid reference
Evaluation
Local NC found
Name error "no such object"
Local Name Resolution Procedure
No chaining/ Partial result
Entry held locally
Name error "alias problem"
Determine Interaction Mode
Subordinate Reference
Superior Reference
Non-specific Subordinate Reference
Cross Reference
Name error "alias dereferencing problem"
Chaining/ multicasting
Update Trace Information
T0704571-88
Service error "loop detected"
Loop
Check Trace Information
Determine Interaction Mode
Chaining/ multicasting
OK
Chain/multicast to the DSAs which were referenced
Referrals
Refer the requestor to the DSA(s) which were referenced
Idle

FIGURE 7/X.518

Operation Dispatcher

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).

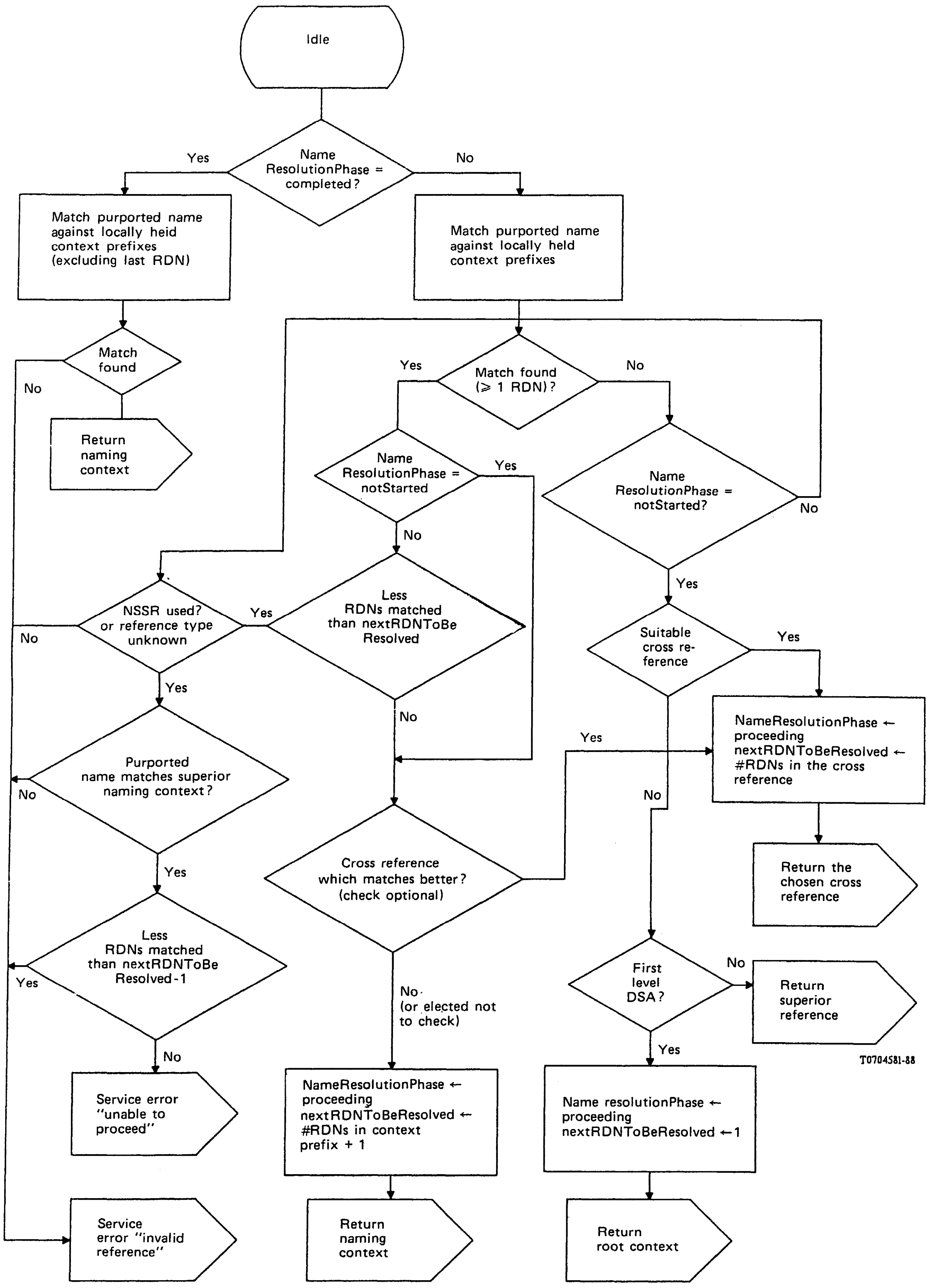
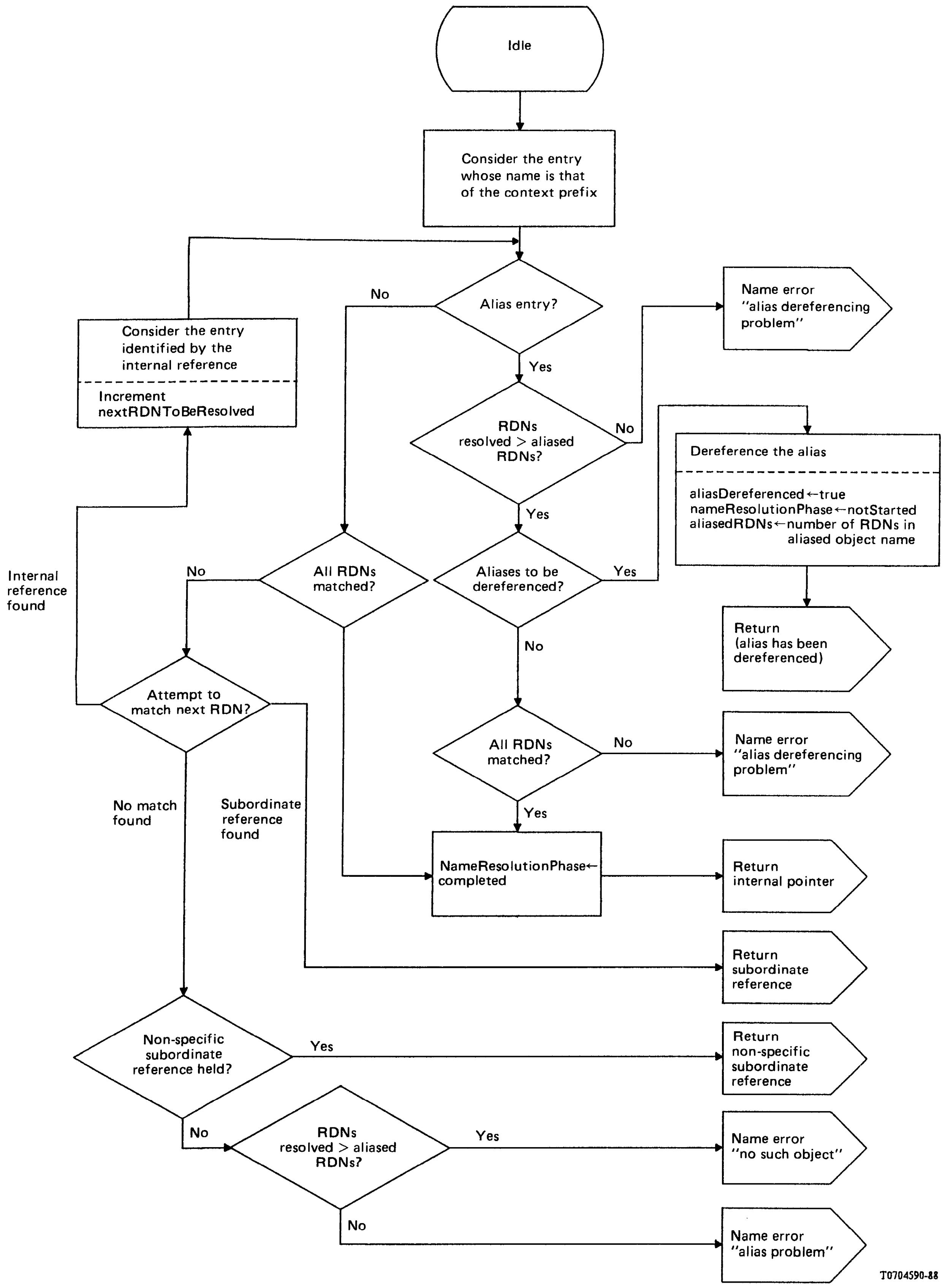

Idle
Name ResolutionPhase = completed?
Yes
No
Match purported name against locally heid context prefixes (excluding last RDN)
Match purported name against locally held context prefixes
Match found
No
Return naming context
Yes
Match found (≥ 1 RDN)?
No
Name ResolutionPhase = notStarted
Yes
Name ResolutionPhase = notStarted?
No
No
Less RDNs matched than nextRDNToBe Resolved
Yes
NSSR used? or reference type unknown
No
Yes
Suitable cross re-ference
Yes
NameResolutionPhase ← proceeding nextRDNToBeResolved ← #RDNs in the cross reference
Yes
Purported name matches superior naming context?
No
No
No
Yes
Cross reference which matches better? (check optional)
Return the chosen cross reference
Less RDNs matched than nextRDNToBe Resolved-1
Yes
No (or elected not to check)
First level DSA?
No
Return superior reference
Yes
No
Service error "unable to proceed"
NameResolutionPhase ← proceeding nextRDNToBeResolved ← #RDNs in context prefix + 1
Name resolutionPhase ← proceeding nextRDNToBeResolved ← 1
Service error "invalid reference"
Return naming context
Return root context
T0704581-88

FIGURE 8/X.518

Find Naming Context

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



Idle
Consider the entry whose name is that of the context prefix
Alias entry?
No
Yes
Name error "alias dereferencing problem"
Consider the entry identified by the internal reference
Increment nextRDNToBeResolved
RDNs resolved > aliased RDNs?
No
Yes
Dereference the alias
aliasDereferenced←true
nameResolutionPhase←notStarted
aliasedRDNs←number of RDNs in aliased object name
Internal reference found
No
All RDNs matched?
Aliases to be dereferenced?
Yes
No
Return (alias has been dereferenced)
Attempt to match next RDN?
All RDNs matched?
No
Yes
Name error "alias dereferencing problem"
No match found
Subordinate reference found
NameResolutionPhase← completed
Return internal pointer
Return subordinate reference
Non-specific subordinate reference held?
Yes
Return non-specific subordinate reference
No
RDNs resolved > aliased RDNs?
Yes
Name error "no such object"
No
Name error "alias problem"
T0704590-88

FIGURE 9/X.518

Local Name Resolution

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.

18.6.5 *Find naming context procedure*

18.6.5.1 *Introduction*

Figure 8/X.518 shows this procedure in the form of a diagram. Below is a textual description. In this it is assumed that the current value of Operation Progress is always returned upon exit of the procedure.

18.6.5.2 *Arguments*

The procedure makes use of the following arguments:

– the target object name (the purported name);

– operation progress.

18.6.5.3 *Results*

There are two cases of successful outcome.

The first of these returns:

– a reference;

– operation progress (updated appropriately).

The second of these returns:

– an indication that a suitable naming context was found locally;

– operation progress (updated appropriately).

18.6.5.4 *Errors*

One of the following errors may be returned:

– **ServiceError** (**unableToProceed**);

– **ServiceError** (**invalidReference**).

18.6.5.5 *Procedure*

1) If **nameResolutionPhase** is set to **completed** on entry, attempt to match the purported name against the context prefixes of the superior naming contexts of all the locally held naming contexts. If a match is found, return all the appropriate locally held naming contexts. If no match is found, return an **invalidReference ServiceError**.

2) If **nameResolutionPhase** is not set to **completed**, attempt to match context prefixes against a sequence of one or more RDNs in the initial portion of the purported name. For a match to be found, all RDNs in a context prefix must be matched. The context prefixes used are those of Naming Contexts for which this DSA has administrative authority. In case of multiple matches the one with the maximum number of matched RDNs is chosen.

   If a match is found, execute (3).

   If a match is not found, execute (5).

3) If **nameResolutionPhase** is **notStarted**, execute (4). If the number of RDNs in the initial portion of the purported name, matched as described in (2) above, is greater or equal to the **nextRDNToBeResolved** component of **OperationProgress**, then execute (4), otherwise execute (9).

4) The **nextRDNToBeResolved** is set to the number of matched RDNs plus 1 and the **nameResolutionPhase** is set to **Proceeding**. The context is returned and this procedure terminated.

   As a performance enhancement, the DSA may optionally match the purported name against the cross references held by the DSA. If more RDNs are matched against a cross reference than against the locally held context prefixes, then execute step (7).

   Note – The Name Resolution procedure will in case of this outcome call the Local Name Resolution.

5) If no match was found, the value of the **nameResolutionPhase** is checked. If the **nameResolutionPhase** is **notStarted**, execute (6).

   If the **nameResolutionPhase** is **proceeding** or **completed**, then execute (9).

6) Using Cross Reference context prefixes, attempt to match against a sequence of one or more RDNs in the initial portion of the purported name. In case of multiple matches, the one with the maximum number of matched RDNs is chosen.

7) If a match was found to a cross reference, set the **nextRDNToBeResolved** to the number of RDNs in the chosen cross reference. The cross reference is returned and this procedure is terminated.

8) If no match was found to a cross reference, determine if the DSA is a first level DSA. If not, it will have a superior reference. Return this and terminate the procedure.

   If the DSA is a first level DSA, set **nextRDNToBeResolved** to one, and **nameResolutionPhase** to **proceeding**. Return the root naming context and terminate the procedure.

9) Check the value of the **referenceType** component of the **ChainingArgument**. If a non-specific subordinate reference was used, or the request came from a DUA, execute (10); otherwise, return **ServiceError** with **invalidReference** problem and terminate the procedure.

10) Compare the initial portion of the purported name to the context prefixes (minus their last RDN) of the locally held naming contexts. This effectively is a comparison to some of the naming contexts of the immediate superior to this DSA.

   If there is no match, return **ServiceError** with **invalidReference** problem and terminate the procedure.

   If a match is found, and the number of RDNs matched is less than in **nextRDNToBeResolved – 1**, return **ServiceError** with **invalidReference** problem; otherwise, return **ServiceError unableToProceed** problem. Terminate the procedure.

18.6.6 *Local Name Resolution*

18.6.6.1 *Introduction*

The Local Name Resolution matches RDNs in the purported name against internal knowledge references. It returns Found, Remote Reference, Alias Dereferenced, or Error indication.

Figure 9/X.518 shows this procedure in the form of a diagram. Below is a textual description.

18.6.6.2 *Arguments*

The procedure makes use of the following arguments:

– internal reference to naming context (with pointer to the entry whose name is the same as the context prefix);

– 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.

18.6.6.3 *Results*

There are three cases of successful outcome.

The first of these returns:

– a reference;

– operation progress (updated appropriately).

The second of these returns:

– an indication that the entry was found locally;

– operation progress (updated appropriately).

The third of these returns:

– an indication that an alias was dereferenced;

– operation progress (set back to "not started").

18.6.6.4 *Errors*

One of the following errors may be returned:

– name error.

18.6.6.5 *Procedure*

The naming context returned by FindNaming Context will point to the entry of the root of the subtree. In the case of the root context, the entry is only a null entry.

1) If the internal reference is for an alias entry, execute step (7), otherwise step (2).

2) If all the RDNs in the purported name have been matched, then the target entry has been found. Set **nameResolutionPhase** to **completed**. An internal pointer is returned and the procedure terminated.

   Otherwise step (3) should be executed.

   Note – The matching could be attained with the context prefix on its own, or with the context prefix plus successive RDNs contained in internal references in the knowledge tree.

3) If an internal reference entry is found subordinate to the current entry in the knowledge tree which matches the next RDN in the purported name, then increment the **nextRDNToBeResolved**, set current entry to subordinate entry, and execute step (1) of this procedure again.

4) If the current entry has a subordinate reference whose RDN matches the next one in the purported name, return it and terminate the procedure.

5) If there are any non-specific subordinate references, subordinate to the current entry in the knowledge tree, return them as references and terminate the procedure.

6) If an internal reference, subordinate reference, or non-specific subordinate reference is not found, then check the number of RDNs in the purported name that have been matched. If more RDNs have been matched than in the **aliasedRDNs** component of **ChainingArgument**, then return **NameError** with **noSuchObject** problem. If less RDNs have been matched, then return **NameError** with **aliasProblem**.

7) If the number of RDNs in the purported name that have been matched is less than or equal to the **aliasedRDNs** component of **ChainingArgument** (if any), then the previous alias that was dereferenced (if any) points to another alias. If so, return **NameError** with **aliasDereferencingProblem**.

8) If the **aliasedRDNs** component is missing, or if the number of RDNs matched is greater than **aliasedRDNs** component of **ChainingArgument**, then check the **dontDereferenceAlias** service control. If aliases can be dereferenced, then execute step (9), otherwise step (10).

9) Dereference the alias. Set **nameResolutionPhase** of **OperationProgress** to **notStarted**. Set **aliasDereferenced** component of **ChainingArgument** to **TRUE**, and **aliasedRDNs** to the number of RDNs in the **aliasedObjectName** attribute of the alias entry. Set **targetObject** to the new name. Terminate the procedure. (The process of Name Resolution will be restarted.)

10) If all the RDNs in the purported name have been matched, execute step (2). Otherwise, return **NameError** with **aliasDereferencingProblem**.

18.7 *Object evaluation procedures*

The object evaluation procedures specified comprise two categories of procedures:

a) single-object evaluation procedure;

b) multiple-object evaluation procedures.

Figure 10/X.518 shows the object evaluation procedure.

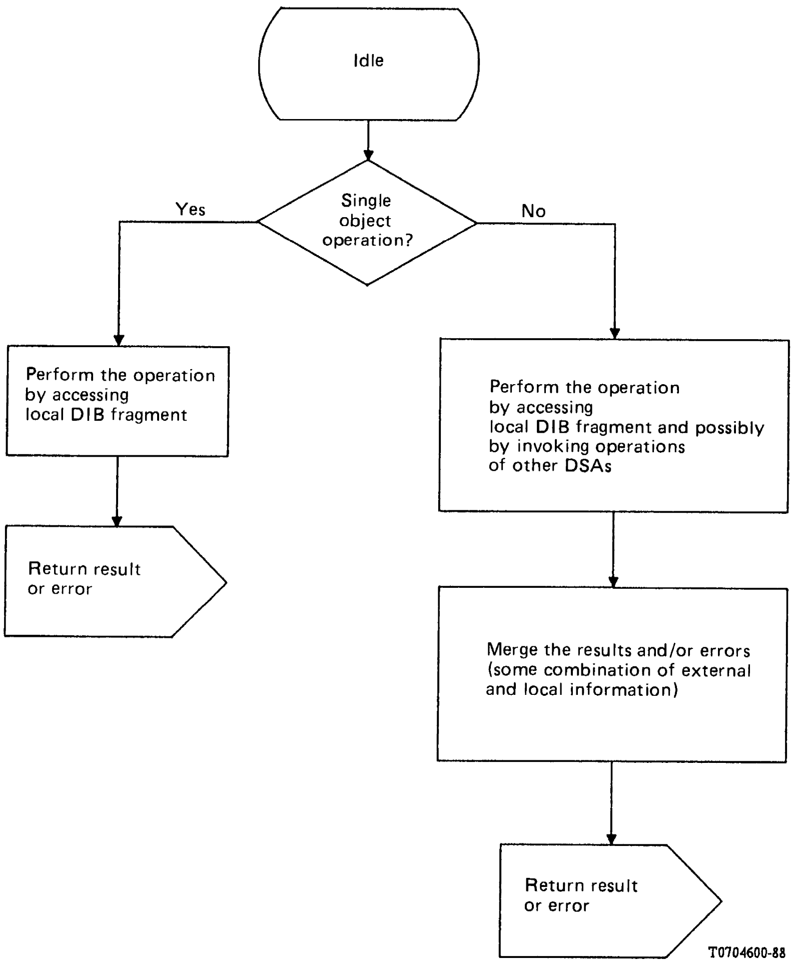

Idle
Single object operation?
Yes
No
Perform the operation by accessing local DIB fragment
Perform the operation by accessing local DIB fragment and possibly by invoking operations of other DSAs
Return result or error
Merge the results and/or errors (some combination of external and local information)
Return result or error
T0704600-88

FIGURE 10/X.518

Evaluation and result merging

18.7.1 *Single-object evaluation procedures*

Single-object evaluation procedures, which are common to the class of operations concerned with accessing a single object are carried out directly, with the result or error being returned to the invoker.

These operations comprise **Read**, **Compare**, **AddEntry**, **RemoveEntry**, **ModifyEntry** and **ModifyRDN**, and their Chained counterparts.

The action required on the entry is as described in the appropriate paragraph of Recommendation X.511.

AddEntry, **RemoveEntry**, and **ModifyRDN** operations affect knowledge. If the immediate superior of the entry is in a different DSA, correct external knowledge references shall be maintained. How this is done is outside the scope of this Recommendation.

How the DSA is chosen to contain the entry created by **AddEntry** is outside the scope of this Recommendation.

If the immediate superior of an entry to be created by **AddEntry** or modified by **ModifyRDN** has non-specific subordinate references, procedures outside the scope of this Recommendation shall be followed to ensure that no two entries have the same distinguished name.

Requests which cannot be satisfied under these conditions shall fail with an **UpdateError** with problem **affectsMultipleDSAs**.

18.7.2 *Multiple-object evaluation procedures*

Multiple-object evaluation procedures, which are common to the class of operations concerned with accessing multiple objects, are specified in the following subparagraphs.

These operations comprise **List** and **Search**, and their Chained counterparts.

18.7.2.1 *List*

This paragraph specifies the evaluation procedure specific to **List** and **ChainedList**. (In what follows the term "List" applies to both.)

18.7.2.1.1 *List procedure (I)*

This procedure applies where the List request has n**ameResolutionPhase** component of **OperationProgress** set to **notStarted** or **proceeding** and where the DSA, after performing Name Resolution The base object will be denoted by "e".

1) Get each locally held immediate subordinate of e to form a local set of results. Set **aliasEntry** and **fromEntry** in **ListResult** as appropriate.

2) Get the set of non-specific subordinate references and subordinate references to DSAs which hold immediate subordinates of "e".

3) Pass the subrequest with base object = e, and **OperationProgress** set to **completed** to the Operation Dispatcher which subsequently forwards it to each DSA which holds immediate subordinates of e.

Note – If the DSA holds subordinate references with an indication of whether or not the subordinate entry are aliases, and the **dontUseCopy** is **FALSE**, then this step can be omitted for those entries. The information about the subordinates is available directly.

18.7.2.1.2 *List Procedure (II)*

This procedure applies to a List request with the **nameResolutionPhase** component of **OperationProgress** set to **completed**.

The base object will be denoted by "e".

1) Get each locally held immediate subordinate of e to form a local set of results. Set **aliasEntry** and **fromEntry** in **ListResult** as appropriate.

2) Pass the results to the Operation Dispatcher which forwards them to the requesting DUA or DSA.

18.7.2.2 *Search*

This paragraph specifies the evaluation procedure specific to **Search** and **ChainedSearch**. (In what follows the term "Search" applies to both.)

Note that two circumstances exist, requiring two separate procedures. The first procedure (18.7.2.2.1) applies when the DSA executing the Search contains the **targetObject** as a local entry. The second procedure (18.7.2.2.2) applies when the DSA executing the Search does not hold the **targetObject**, but only subordinates of the **targetObject**.

18.7.2.2.1 *Search procedure (I)*

This procedure applies to a Search request with the **nameResolutionPhase** component of **OperationProgress** set to **notStarted** or **proceeding** and where the DSA, after performing Name Resolution, determines that it holds the target object.

The base object will be denoted by "e".

1) If the **subset** argument is **baseObject** or **wholeSubtree**, then apply the filter argument specified in the Search to the entry e, to form a set of local results. Return the results for Results Merging. If the **subset** argument is **baseObject**, terminate the procedure, otherwise continue at (2).

2) If the **subset** argument is **oneLevel** or **wholeSubtree** form a set E from the locally-held immediate subordinates of e, except that:

   If aliases are to be dereferenced, i.e. the **searchAliases** parameter is **TRUE**, then any alias entries that are found are handled in paragraph 5) below and do not contribute to these results.

   Apply the filter arguments to E to give a filtered subset E′ gE; return this set E′ of local results for Results Merging.

3) Other subordinates of e may reside in other DSAs, and if so will be referenced as subordinate or non-specific subordinate references. For each DSA which is so referenced, prepare a new Search with targetObject = e, and with **nameResolutionPhase** of **OperationProgress** set to **completed**. Return each Search subrequest to the Operation Dispatcher for forwarding. If any error result is returned from a subrequest, it is ignored, as if no subrequest had been sent.

4) If the **subset** argument is **oneLevel**, the Search is now complete so terminate the procedure.

   If the **subset** argument is **wholeSubtree**, then:

   if the set E from paragraph (2) is empty, then the whole subtree held in this DSA has been searched, so terminate the procedure;

   otherwise continue processing as follows:

   let each entry that was in set E be denoted by e. Repeat the Search procedure from paragraph (2), for each entry e.

5) If aliases are to be dereferenced, any alias entries found in step (2) are placed in set D. For each entry d in D, dereference the alias, and formulate a new Search with **nameResolutionPhase** set to **notStarted**, and **targetObject** created from the **aliasedObjectName** attribute and the old **targetObject** name.

   If the **subset** argument was **oneLevel**, set it to **baseObject** in the new subrequest, otherwise set it to **wholeSubtree**.

   If any error result is returned from the subrequest, it is ignored, as if no subrequest had been made.

18.7.2.2.2 *Search Procedure (II)*

This procedure applies to a Search request with the **nameResolutionPhase** component of **OperationProgress** set to **completed**.

The target object will be denoted by "e".

For each locally held immediate subordinate e′ of e, formulate a new request with targetObject = e′. If the **subset** argument was **oneLevel,** set it to **baseObject,** otherwise leave it as **wholeSubtree**. Now carry out the procedure defined in steps (1) to (5) in § 18.7.2.2.1. If there are no such subordinates, return **unableToProceed ServiceError**.

18.8 *Result merging procedure*

This procedure is called when external results and/or errors are present. There might also be one internal result. All results and errors are assumed to be held within the DSA until the procedure completes.

The external information could be due to chaining, multicasting or request decomposition.

In the case of chaining there will be a single result or error. In the case of multicasting there might be either no result, one result or several identical results. In addition, there may be some errors. If there is more than one result, all but one of them are arbitrarily discarded. A result is always returned in preference to an error. If there are no results, an error is returned, with the following exceptions:

i) If **invalidReference** was returned, the reference is marked as such, and the DSA may either use an appropriate alternate external reference to continue the request, or return **ditError** to the requestor. (The handling of invalid external references is beyond the scope of this Recommendation.)

ii) In the case of multicasting, **unableToProceed** errors should be ignored, unless all responses are of this type in which case **NameError noSuchObject** should be returned to the responder. If at least one result is returned, then all errors can be ignored.

iii) In the case of referrals, these need not be treated as errors, and may be acted upon.

   If the merging is required due to a request decomposition, the merging amounts to forming the union of the results.

   In the case of decomposition, when there are both results and errors to be merged, an incomplete result is returned to the requestor.

A DSA might at this stage choose to extract referrals from the incoming results and errors that should be merged. It might then decide to explore all or some of these further, in which case operations are chained. The old result will have to be saved and later merged with the results or errors produced by the chaining.

The handling of signatures which may be present with the results being returned is specified in § 18.9.2 below.

18.9 *Procedures for distributed authentication*

This paragraph specifies the procedures necessary to support the directory distributed authentication services. These services, and hence the procedures, are categorized as:

– originator authentication, which is supported in either an unprotected (simple identity based) or secure (based upon digital signatures) form; and

– results authentication which is similarly protected (again based upon digital signatures).

18.9.1 *Originator authentication*

18.9.1.1 *Identity based authentication*

The identity based authentication service enables DSAs to authenticate the original requestor of information for the purpose of effecting local access controls. DSAs wishing to exploit this service must adopt the following procedure:

– for a DSA requiring to authenticate a DAP request, the DSA acquires the distinguished name of the requestor through the Bind procedures at the time a DUA association (DUA or DSA) is established. Successful conclusion of these procedures does not in any way prejudice the level of authentication that may subsequently be required for processing operations using that association;

– the DSA with which the DUA association exists must insert the requestor's distinguished name in the initiator field of the ChainingArgument for all subsequent chained operations to other DSAs;

– a DSA, on receiving a chained-operation, may satisfy that operation, or not, depending upon the determination of access rights (a locally defined mechanism). If the outcome is not satisfactory a **SecurityError** may be returned with **SecurityProblem** set to **insufficientAccessRights**.

18.9.1.2 *Signature-based originator authentication*

This signature-based originator authentication service enables a DSA to authenticate (in a secure manner) the originator of a particular service request. The procedures to be effected by a DSA in realizing this service are described in this paragraph.

The signature-based authentication service is invoked by a DUA using the **SIGNED** variant of an optionally-signed service request.

A DSA, on receiving a signed request from another DSA, shall remove that DSA's signature prior to processing the operation. Assuming the result of any signature verification proves to be satisfactory, the DSA will continue to progress the operation. If, during processing, the DSA requires to perform chaining, multicasting or request decomposition, the argument set for each associated chained operation shall be constructed as follows:

– the DSA forms an argument set which may be optionally signed; the argument set comprises the incoming signed argument set together with a modified **ChainingArgument**.

In the event that the DSA is able to contribute information to the response, originator authentication, based upon the signed service request, may be used for the determination of access rights to that information.

If a DSA receives an unsigned service request for information which will only be released subject to originator authentication, a **SecurityError** will be returned with **SecurityProblem** set to **protectionRequired**.

18.9.2 *Results authentication*

This service is provided to enable requestors of directory operations (either DUA or DSAs) to verify (in a secure manner using digital signature techniques) the source of results. The results authentication service may be requested irrespective of whether originator authentication is to be used.

The results authentication service is initiated using the **signed** value of the **protectionRequest** component as contained within the argument set of directory operations; a DSA receiving an operation with this option selected may then optionally sign any subsequent results. The **signed** option in the **protectionRequest** serves as an indication, to the DSA, of the requestor's preference; the DSA may, or may not, actually sign any subsequent results.

In the case where a DSA performs chaining, multicasting or request decomposition of such a request, the DSA has a number of options in terms of the form of results sent back to the requestor, namely:

a) return a composite response (signed or unsigned) to the requestor;

b) return a set of two or more uncollated partial responses (signed or unsigned) to the requestor; within this set zero or more members may be signed and zero or one unsigned. In the event that an unsigned partial result is present, this member may in fact be a collation of one or more unsigned partial responses which have been received from other DSAs, contributed by this DSA, or both.

ANNEX A

(to Recommendation X.518)

ASN.1 for distributed operations

This Annex is part of the Recommendation.

This Annex includes all of the ASN.1 type, value and macro definitions contained in this Recommendation in the form of the ASN.1 module **Distributed Operations**.

```
DistributedOperations {joint-iso-ccitt  ds(5)  modules(1)  distributedOperations(3)}
DEFINITIONS ::=
BEGIN
EXPORTS
        DirectoryRefinement, chainedReadPort, chainedSearchPort, chainedModifyPort,
        DSABind, DSABindArgument,
        DSAUnbind,
        ChainedRead, ChainedCompare, ChainedAbandon,
        ChainedList, ChainedSearch,
        ChainedAddEntry, ChainedRemoveEntry,
        ChainedModifyEntry, ChainedModifyRDN,
        DsaReferral, ContinuationReference;
IMPORTS
        InformationFramework, abstractService, distributedOperations,
        directoryObjectIdentifiers, selectedAttributeTypes
              FROM      UsefulDefinitions  {joint-iso-ccitt ds(5) modules(1)  usefulDefinitions(0)}

        DistinguishedName, Name, RelativeDistinguishedName
              FROM      InformationFramework informationFramework

        id-ot-dsa, id-pt-chained-read, id-pt-chained-search, id-pt-chained-modify
              FROM      DistributedDirectoryObjectIdentifiers, distributedDirectoryObjectIdentifiers

        PresentationAddress
              FROM      SelectedAttributeTypes selectedAttributeTypes

        directory, readPort, searchPort, modifyPort
        DirectoryBind,
        ReadArgument, ReadResult,
        CompareArgument, CompareResult,
        Abandon
        ListArgument, ListResult,
        SearchArgument, SearchResult,
        AddEntryArgument, AddEntryResult,
        RemoveEntryArgument, RemoveEntryResult,
        ModifyEntryArgument, ModifyEntryResult,
        ModifyRDNArgument, ModifyRDNResult,
        Abandoned, AttributeError, NameError, ServiceError, SecurityError, UpdateError
        OPTIONALLY-SIGNED, SecurityParameters
              FROM      DirectoryAbstractService directoryAbstractService

-- objects and ports --

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
```

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

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

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

DSAUnbind::=   UNBIND
     FROM           {chainedRead,
                    chainedSearch,
                    chainedModify}

-- operations, arguments and results --

ChainedRead      ::=
     ABSTRACT-OPERATION
          ARGUMENT        OPTIONALLY-SIGNED     SET{
                                                ChainingArgument,
                                                [0]  ReadArgument}
          RESULT          OPTIONALLY-SIGNED     SET{
                                                ChainingResult,
                                                [0]  ReadResult}
          ERRORS {
               DsaReferral, Abandoned, AttributeError, NameError,
               ServiceError, SecurityError}

ChainedCompare        ::=
     ABSTRACT-OPERATION
          ARGUMENT        OPTIONALLY-SIGNED     SET{
                                                ChainingArgument,
                                                [0]  CompareArgume
          RESULT          OPTIONALLY-SIGNED     SET{
                                                ChainingResult,
                                                [0]  CompareResult}
          ERRORS {
               DsaReferral, Abandoned, AttributeError, NameError,
               ServiceError, SecurityError}
```

```
ChainedAbandon      ::=      Abandon

ChainedList     ::=
     ABSTRACT-OPERATION
          ARGUMENT          OPTIONALLY-SIGNED    SET{
                                                 ChainingArgument,
                                                 [0] ListArgument}
          RESULT            OPTIONALLY-SIGNED    SET{
                                                 ChainingResult,
                                                 [0] ListResult}
          ERRORS {
               DsaReferral, Abandoned, AttributeError, NameError,
               ServiceError, SecurityError }

ChainedSearch   ::=
     ABSTRACT-OPERATION
          ARGUMENT          OPTIONALLY-SIGNED    SET{
                                                 ChainingArgument,
                                                 [0] SearchArgument}
          RESULT            OPTIONALLY-SIGNED    SET{
                                                 ChainingResult,
                                                 [0] SearchResult}
          ERRORS {
               DsaReferral, Abandoned, AttributeError, NameError,
               ServiceError, SecurityError}



ChainedAddEntry ::=
     ABSTRACT-OPERATION
          ARGUMENT          OPTIONALLY-SIGNED    SET{
                                                 ChainingArgument,
                                                 [0] AddEntryArgument}
          RESULT            OPTIONALLY-SIGNED    SET{
                                                 ChainingResult,
                                                 [0] AddEntryResult}
          ERRORS {
               DsaReferral, Abandoned, AttributeError, NameError,
               ServiceError, SecurityError, UpdateError}

ChainedRemoveEntry ::=
     ABSTRACT-OPERATION
          ARGUMENT          OPTIONALLY-SIGNED    SET{
                                                 ChainingArgument,
                                                 [0] RemoveEntryArgument}
          RESULT            OPTIONALLY-SIGNED    SET{
                                                 ChainingResult,
                                                 [0] RemoveEntryResult}
          ERRORS {
               DsaReferral, Abandoned, NameError,
               ServiceError, SecurityError, UpdateError}

ChainedModifyEntry ::=
     ABSTRACT-OPERATION
          ARGUMENT          OPTIONALLY-SIGNED    SET{
                                                 ChainingArgument,
                                                 [0] ModifyEntryArgument}
          RESULT            OPTIONALLY-SIGNED    SET{
                                                 ChainingResult,
                                                 [0] ModifyEntryResult}
          ERRORS {
               DsaReferral, Abandoned, AttributeError, NameError,
               ServiceError, SecurityError, UpdateError}
```

```
ChainedModifyRDN  ::=
      ABSTRACT-OPERATION
            ARGUMENT        OPTIONALLY-SIGNED    SET{
                                                 ChainingArgument,
                                                 [0]  ModifyRDNArgument}
            RESULT          OPTIONALLY-SIGNED    SET{
                                                 ChainingResult,
                                                 [0]  ModifyRDNResult}
            ERRORS {
                  DsaReferral, Abandoned, NameError,
                  ServiceError, SecurityError, UpdateError}

-- errors and parameters --

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

-- common arguments/results --

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 { }}

ChainingResults      ::=       SET {
      info                       [0]    Domaininfo OPTIONAL,
      crossReferences            [1]    SEQUENCE OF CrossReference OPTIO
                                 [2]    SecurityParameters DEFAULT { }}

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

ReferenceType        ::=       ENUMERATED {
            superior                                (1),
            subordinate                             (2),
            cross                                   (3),
            nonSpecificSubordinate                  (4)}

TraceInformation     ::=       SEQUENCE OF
      SEQUENCE {
            targetObject         Name,
            dsa Name,
            OperationProgress}

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

```
DomainInfo          ::=     ANY

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 }

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

ANNEX B

(to Recommendation X.518)

Modelling of knowledge

This Annex is not part of the Recommendation.

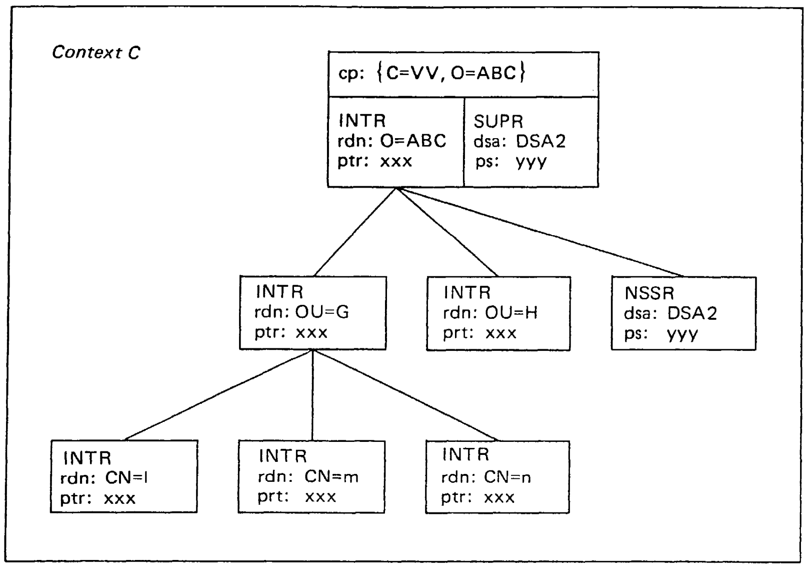
B.1 *Example of knowledge modelling*

The following example illustrates the knowledge information that would have to be maintained by the DSAs shown in Figure 5/X.518 (9). Figure 5/X.518 depicts a hypothetical DIT logically partitioned into five Naming Contexts (A, B, C, D and E) and physically distributed over three DSAs (DSA1, DSA2, DSA3). In the example, DSA1 holds context C, DSA2 holds contexts A, B, and E, and DSA3 holds context D.

The following abbreviations have been used in Figures B-1/X.518 to B-3/X.518.

SUPR:	superior reference
SUBR:	subordinate reference
INTR:	internal reference
NSSR:	non-specific subordinate reference
CROSSR:	cross reference
DSAn:	Distinguished Name of DSAn
PS:	Presentation Address
CP:	context prefix
RDN:	Relative Distinguished name
DSA:	Distinguished name of a DSA
PTR:	Pointer
AON:	Aliased Object Name**.**

Note – The following figures are intended only to provide a pictorial example of the concepts defined in this paragraph. How knowledge information is actually stored and managed in a particular DSA implementation is a local matter and is outside the scope of this Recommendation.



T0704610-88

FIGURE B-1/X.518

Knowledge information for DSA1

Figure B-1/X.518 illustrates the knowledge information that must be held by DSA1. This must include the following context prefixes and set of references:

Context Prefixes:	{C=WW, O=ABC}, context C.
Cross References:	{ }
Superior References:	{DSA2, presentation address of DSA2}
Internal References for Context C:	{C=WW, O=ABC}, {OU=G}, {OU=H} {OU=G, CN=1}, {OU=G, CN=m}, {OU=G, CN=n}.
Subordinate References:	{ }
Non-specific subordinate References:	{DSA2, presentation address of DSA2}.

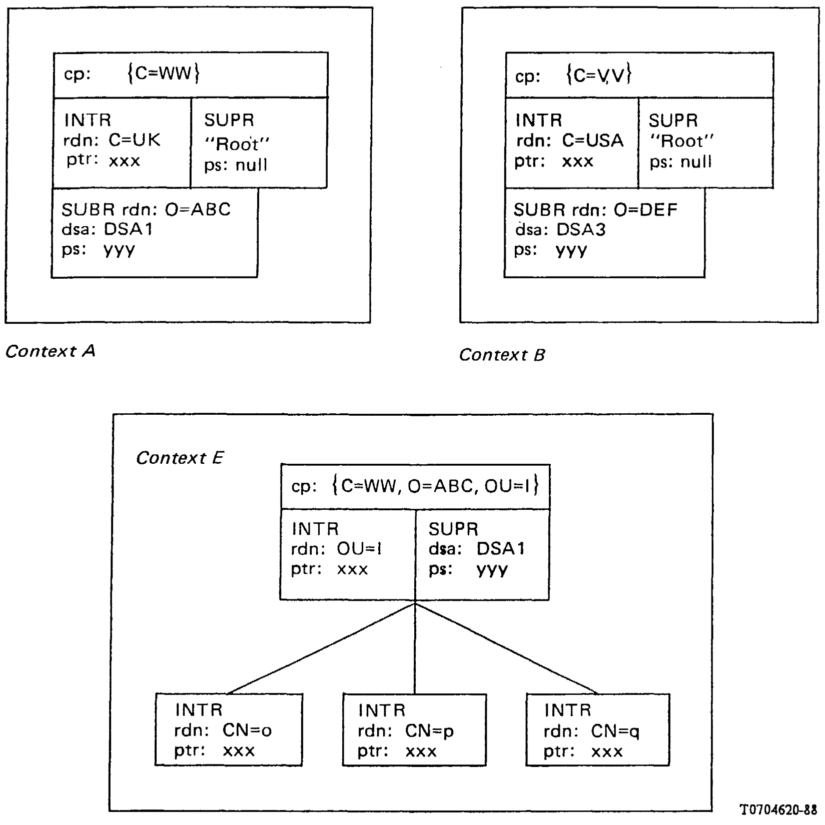


FIGURE B-2/X.518

Knowledge information for DSA1

Figure B-2/X.518 illustrates the knowledge information that must be held by DSA2. This must include the following context prefixes and set of references:

Context Prefixes:	{C=WW}, context A {C=VV}, context B {C=WW}, O=ABC, OU=I}, context E.
Cross References:	{ }
Superior References:	{ }
Internal References for Context A:	{C=WW}
Internal References for Context B:	{C=VV}
Internal References for Context E:	{C=WW, O=ABC, OU=I}, {CN=o}, {CN=p}, {CN=q}.
Subordinate References for Context A:	{C=WW, O=ABC}
Subordinate References for Context B:	{C=VV, O=DEF}
Non-specific subordinate References:	{ }

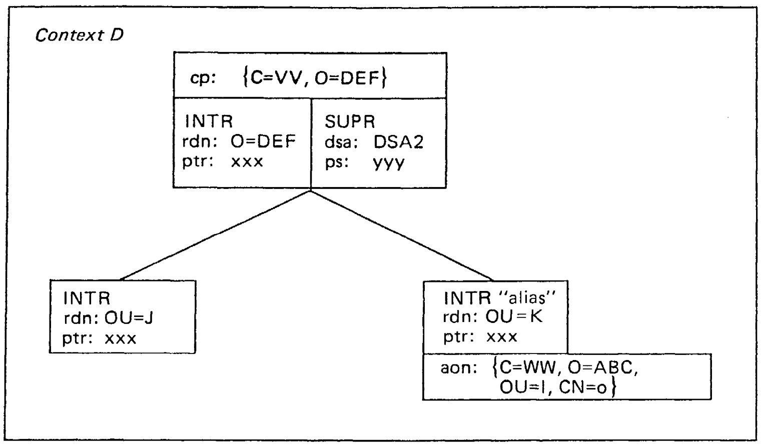


FIGURE B-3/X.518

Knowledge information for DSA3

Figure B-3/X.518 illustrates the knowledge information that must be held by DSA3. This must include the following context prefixes and set of references:

Context Prefixes:	{C=VV, O=DEF}, context D
Cross References:	{{C=WW, O=ABC, OU=H}, DSA1, presentation address of DSA1} (not shown in the figure above)
Superior References:	{DSA2, presentation address of DSA2}
Internal References for Context D:	{DSA1, presentation address of DSA1} {C=VV, O=DEF}, {OU=J}, {OU=K} alias for {C=WW, O=ABC, OU=I, CN=o} (alias information is not part of the knowledge)
Subordinate References:	{ }
Non-specific subordinate References:	{ }

B.2 *Example of distributed name resolution*

The following is an example of how Distributed Name Resolution is used to process different directory requests. The example is based on the hypothetical DIT shown in Figure 5/X.518 (9) and the corresponding DSA configuration(s) shown in Figures B-1/X.518 to B-3/X.518 (Annex B).

Assuming a chaining mode of propagating, the following requests addressed to DSA1 would be processed as follows:

1) A request with distinguished name {C=WW, O=ABC, OU=G, CN=1}
   - Will match context prefix {C=WW, O=ABC} of context C for which DSA1 has administrative authority. Therefore, name resolution will begin in DSA1 with context C.
   - Name resolution will proceed downwards in context C successfully matching each remaining RDN, until CN=1 is located.
2) A request with distinguished name {C=WW, O=JPR}
   - Will not match any context prefix held by DSA1, therefore DSA1 will use its superior reference to forward the request to its superior DSA, DSA2.
   - In DSA2, the request will match context prefix {C=WW} and name resolution will begin in DSA2 with context A.
   - Name resolution will not find a subordinate of C=WW to match RDN O=JPR, therefore the request will fail and the name will be determined to have been invalid (i.e. reference a non-existent object).

3) A request with distinguished name {C=VV, O=DEF, OU=K}

   – DSA1 will therefore forward the request to its superior DSA, DSA2.
   – The request will match context prefix {C=VV} of context B held by DSA2. Therefore, name resolution will begin in DSA2 with context B.
   – As name resolution attempts to match O=DEF, it will find a subordinate reference indicating that {C=VV, O=DEF} is the start of a new context held in DSA3.
   – Name resolution will continue in DSA3 until {C=VV, O=DEF, CN=K} is located.
   – Assuming that aliases are to be dereferenced, a new name will be constructed using the aliased name contained in the entry {C=VV, O=DEF, CN=K}. The resulting new name will be: {C=WW, O=ABC, OU=I, CN=o}.
   – DSA3 will resume processing of the request using the new name obtained by dereferencing.

ANNEX C

(to Recommendation X.518)

Distributed use of authentication

This Annex is not part of the Recommendation.

C.1 *Summary*

The security model is defined in § 10 of Recommendation X.501. The following is a summary of the main points of the model.

a) Simple Authentication of the operation initiator is not supported in the DSP.

b) Strong Authentication, by the signing of the request and of the result, is supported in the DSP.

c) Encryption of the request, or of the result, is not supported in the DSP.

d) Authentication of errors, including referrals, is not supported in the DSP.

This Annex describes how b) above is realized in the distributed Directory. It makes use of terminology and notation defined in Recommendation X.509.

C.2 *Simple authentication*

The DUA will be authenticated as part of the Bind Operation of the DAP. Thereafter, only the name of the DUA will be carried in the DSP, in the initiator field of the Chaining Argument.

C.3 *Distributed authentication model*

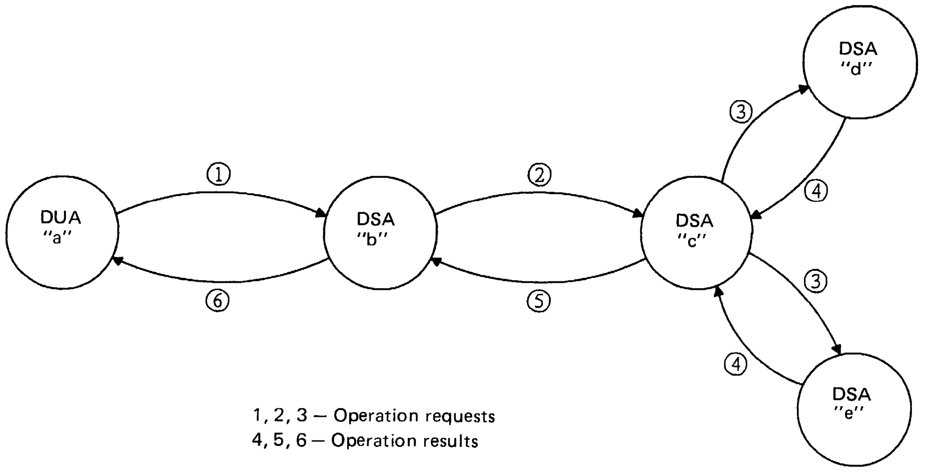


FIGURE C-1/X.518

Distributed authentication model

Figure C-1/X.518 illustrates the model to be used to specify the distributed authentication procedures. The model identifies the sequence of information flows for the general case of a list or search operation. The operation is considered as originating from DUA "a" citing a target object which resides in DSA "c"; in performing the operation, DSAs "b", "c", "d" and "e" are to be involved.

DUA "a" initially contacts any DSA (DSA "b") which does not hold the target object, but which is able to navigate, via chaining, to the DSA (DSA "c") holding the target object. If all the DSAs were operating in referral mode, then the model would be significantly simplified, and each DUA/DSA exchange would equate, in authentication terms, to the interaction between DUA "a" and DSA "b".

C.4 *DUA to DSA*

Originator authentication is realized as a consequence of exchange (1). In Figure C-1/X.518 the authentication procedure is as follows:

Let
OA = the Operation Argument i.e. Search, Read, Compare etc. Argument as defined in Part 3.

and
a(OA) = the Operation Argument signed by DUA "a".

Authentication will be determined by verification of the signature.

C.5 *Transference from the DAP to the DSP*

This procedure is effected by DSA "b" in Figure C-1/X.518 and represents the transference of the signed identity of the initiator from the DAP to the DSP.

DSA "b" formulates the appropriate Chaining Argument as described in § 12.3 of this Recommendation and combines it with the Operation Argument from the DAP thus forming a Chained Operation, i.e. Chained Read, Search, List etc. of the DSP. The Chained Operation so formed will be signed prior to passing it to other DSAs (DSA "c" in Figure C-1/X.518). The data structure can be represented as:

b{ChA,a{OA}} = the Chained Operation signed by DSA b

where

ChA = Chaining Argument.

Authentication information carried in the DSP between two DSAs [labelled exchange (2) in Figure C-1/X.518] therefore comprises two parts:

- the Operation Argument, signed by the initiator, which allows authentication of the initiator;
- the Chained Operation, signed by the sending DSA, which allows authentication of the sending DSA.

C.6 *Chaining through intermediate DSAs*

This procedure would be effected by DSA "c" in the model depicted in Figure C-1/X.518. DSA "c" will discard the signature provided by the sending DSA (DSA "b" in Figure C-1/X.518), and will modify the Chaining Argument, as described in § 12.3 of this Recommendation. DSA "c" shall then combine the modified Chaining Argument with the signed Operation Argument, and sign the result to create a modified signed Chained Operation. This can be represented by:

c{ChA ′, a{OA}} = the Chained Operation signed by DSA "c"

where

ChA ′ = modified Chaining Argument.

upon the nature of the operation, and upon the type of knowledge held, DSA "c" may perform request decomposition prior to chaining or multicasting any resultant operation(s). This has been represented in Figure C-1/X.518 by DSA "c" sending operations to DSA "d" and DSA "e"; in each case the authentication procedure is identical.

C.7 *Results authentication*

The results authentication service is requested by an initiator of a directory operation using the **signed** option within the **protectionRequest SecurityParameter**. In providing a response to such a request a DSA may optionally decide whether or not to sign any or all of the results; the results authentication service does not provide for the authentication of error responses.

Within the context of a particular DSA processing results from an arbitrary number of DSAs (each of which are associated with a particular service request) the following distinct cases are possible:

– the DSA provides a complete set of results for an operation without the need to perform any collating function (represented by DSA "d" and DSA "e" in Figure C-1/X.518);

– the DSA collates local results (sourced by this DSA) with the results from one or more other DSAs (represented by DSA "c" in Figure C-1/X.518);

– the DSA chains a result from a DSA to either another DSA or a DUA and does not contribute to the result set as it does so (represented by DSA "b" in Figure C-1/X.518).

C.7.1 *DSA results – no collation*

This paragraph addresses the role of a DSA in being the sole source of results to a particular operation request, i.e. the DSA has no collation function to perform. The paragraph considers the case for both the DSP and the DAP.

C.7.1.1 *DSP*

The DSA can choose to perform either of the following procedures:

– return the results unsigned, this can be represented by:

ChR,OR = Chained Operation Result (unsigned)

where

ChR = Chaining Results

OR = Operation Result;

– sign only the Operation Result, this can be represented by:

ChR, d(OR) = Operation Result signed by DSA "d";

– sign only the Chained Operation Result, which can be represented as:

d (ChR, OR) = Chained Operation Result signed by DSA "d"

– sign both the Operation Result and the Chained Operation Result, which can be represented by:

d{ChR, D{OR}} = Operation Result and Chained Operation Result signed by DSA "d".

Note – For the case where the Operation Result is signed, the signed result will be carried back to the initiator; for the case where the Chained Operation Result has been signed, the receiving DSA will have to discard the signature in order to modify the Chaining Results argument prior to forwarding the Chained Operation Result.

C.7.1.2 *DAP*

This is fully described in Recommendation X.511, a summary is reproduced here for completeness.

The DSA can choose to either return the results unsigned, which can be represented by:

OR = Operation Result

or, signed, which can be represented by:

d{OR} = Operation Result signed by DSA "d".

C.7.2 *DSA results – collation included*

This paragraph addresses the role of a DSA in returning the result of particular service requests where collation and integration of results from other DSAs is a necessary prerequisite. The paragraph considers the case for both the DSP and the DAP.

C.7.2.1 *DSP*

Recognizing that zero or more results received from other DSAs may be signed, this procedure enables a DSA to collate and integrate the results and sign zero or more constituent parts of the composite result and optionally, sign the composite result as a whole.

C.7.2.1.1 *Production of the chaining results argument*

This procedure requires that a DSA (represented by DSA "c" in Figure C-1/X.518) remove all of the Chained Operation Result signatures from the results received from external DSAs (DSA "d" and DSA "e" in Figure C-1/X.518). DSA "c" then possesses a set of unsigned Chaining results, a set of signed Operation Results, and a set of unsigned Operation Results.

All the Chaining Results are manipulated as described in § 12.4 of this Recommendation to create a single modified Chaining Result, denoted by:

i) ChR ′ = modified Chaining Results.

C.7.2.1.2 *Unsigned locally derived result*

If the DSA does not wish to sign the locally generated results, the set of unsigned Operation Results are merged with the local result to form a modified set of Operation Results, denoted by:

OR ′ = Merged Operation Result.

The complete set of Operation Results is then the union of the set of externally signed Operation Results denoted by:

d{OR}, e{OR} ...

and the Merged Operation Result, collectively denoted by:

(ii) OR ′, d{OR}, e{OR} ... = Operation Result.

C.7.2.1.3 *Signed locally derived result*

If the DSA does wish to sign the locally generated results, then the externally generated set of unsigned Operation Results are first merged together. The complete set of Operation Results is then the union of the locally signed set of Operation Results denoted by C{OR}, the merged set of externally unsigned Operation Results denoted by, OR", and the set of externally signed Operation Results denoted by:

d{OR}, e{OR}, ..., which are collectively denoted as:

(iii) c{OR}, OR", d{OR}, e{OR}, ... = Operation Result.

C.7.2.1.4 *Unsigned chained operation result*

If the DSA does not wish to sign the Chained Operation Result, then the latter will comprise the Chaining Results (identified in (i) above) added to the Operation Result identified in either (ii) or (iii) above, collectively, these are denoted by:

either:

ChR ′, OR ′, d{OR}, e{OR}, ... = Chained Operation Result (unsigned).

or,

ChR ′, c{OR}, OR″, d{OR}, e{OR}, ... = Chained Operation Result (unsigned) and Operation Result signed by DSA "c".

C.7.2.1.5 *Signed chained operation result*

If the DSA does wish to sign the Chained Operation Result, then the result will comprise the Chaining Results (identified in (i) above) added to the Operation Result (identified in either (ii) or (iii) above), collectively denoted as:

either:

c{ChR ′, OR ′, d{OR}, e{OR}, ...} = Chained Operation Result signed by DSA "c"

or,

c{ChR ′, c{OR}, OR ″, d{OR}, e{OR}, ...} = Chained Operation Result and Operation Result signed by DSA "c".

C.7.2.2 *DAP*

The procedure is very similar to that described in § C.7.2.1, with the exception that the Chaining Results argument is not passed in the DAP.

C.7.3 *DSA chained results*

This paragraph addresses the procedures to be effected by a DSA in chaining an operation result back to the requestor, DSA or DUA, within the DSP and DAP respectively.

C.7.3.1 *DSP*

The DSA initially removes the signature (if one exists) from the Chained Operation Result. It then manipulates the Chaining Results argument as described in this Recommendation, to produce a modified Chaining Results argument. The latter is then merged back with the Operation Result argument to produce a modified Chained Operation Result. Finally, the DSA may optionally sign the Chained Operation Result before passing it to the next DSA in the chain.

C.7.3.2 *DAP*

A DSA (represented by DSA "b" in Figure C-1/X.518) first removes the signature (if one exists) from the Chained Operation Result. It then analyses and discards the Chaining Results argument and, finally, it optionally signs the remaining Operation Result argument before passing the result to the DUA.

ANNEX D

(to Recommendation X.518)

Distributed directory object identifiers

This Annex is part of the Recommendation.

This Annex includes all of the ASN.1 object identifiers contained in this Recommendation in the form of the ASN.1 module **DistributedDirectoryObjectIdentifiers**.

```
DistributedDirectoryObjectIdentifiers     {joint-iso-ccitt ds(5) modules(1)
                                          distributedDirectoryObjectIdentifiers(13)}

DEFINITION    ::=
BEGIN

EXPORTS
        id-ot-dsa, id-pt-chainedRead, id-pt-chainedSearch, id-pt-chainedModify;

IMPORTS
        id-ot, id-pt
            FROM    UsefulDefinitions {joint-iso-ccitt ds(5) modules(1) usefulDefinitions(0)};

-- objects --

id-ot-dsa      OBJECT IDENTIFIER  ::=  {id-ot 3}

-- part types --

id-pt-chainedRead      OBJECT IDENTIFIER          ::=     {id-pt 4}
id-pt-chainedSearch    OBJECT IDENTIFIER          ::=     {id-pt 5}
id-pt-chainedModify    OBJECT IDENTIFIER          ::=     {id-pt 6}

END
```

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	**Data networks and open system communications**
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland

Geneva, 2008