ITU-Telecommunication Standardization Sector **Contribution _____ (WP5/4)**

STUDY GROUP **4**

Geneva, Switzerland, ? – ? January, 2001

Questions: 14/4, 15/4, 19/4

Title: Draft Rec. *"CORBA-Based TMN Services."*

Source: Editors

Contact:

Keith Allen
SBC Technology Resources
USA
Tel: +1 512 372 5741
Fax: +1 512 372 5791
E-mail: kallen@tri.sbc.com

Lakshmi Raman
Teraburst
USA
Tel: +1 408 541 1155 x322
Fax: +1 408 541 0439
E-mail: lraman@teraburst.com

ABSTRACT

This document defines a set of services that along with draft Recommendation X.780 composes a framework for CORBA-based TMN interfaces. It specifies protocol requirements, CORBA Common Object Service usage requirements, and TMN-specific support services. A CORBA IDL module defining the interfaces to the TMN-specific support services is provided.

INTERNATIONAL TELECOMMUNICATION UNION

TELECOMMUNICATION STANDARDIZATION SECTOR
STUDY PERIOD 1997 - 2000

COM 4-xxx-E
August 2000
Original: English

Question: 19/4

STUDY GROUP 4 – CONTRIBUTION ____

SOURCE*: EDITORs

TITLE: DRAFT NEW RECOMMENDATION Q.816: CORBA Based TMN Services

_____

Summary
This document defines a set of services that along with draft Recommendation X.780 composes a framework for CORBA-based TMN interfaces. It specifies protocol requirements, CORBA Common Object Service usage requirements, and TMN-specific support services. A CORBA IDL module defining the interfaces to the TMN-specific support services is provided

Source
ITU-T Recommendation Q.816 was developed by ITU-T Study Group 4 (1997-2000) and was approved under the WTSC Resolution 1 procedure on the **xx of xx xx**.

Keywords
Common Object Request Broker Architecture (CORBA), Interface Definition Language (IDL), CORBA services, Distributed Processing, TMN Interfaces, Managed Objects

Foreword

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.
The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.
The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.
In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

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

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.
As of the date of approval of this Recommendation, the ITU had/had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

Table Of Contents

Table Of Figures

Table Of Tables

Recommendation Q.816

CORBA-Based TMN Services (2001)

1 Scope

The TMN architecture defined in Recommendation M.3010 – 2000 introduces concepts from distributed processing and includes the use of multiple management protocols. The initial TMN interface specifications for intra- and inter-administration interfaces were developed using the Guidelines for the Definition of Managed Objects (GDMO) notation from OSI Systems Management with Common Management Information Protocol (CMIP) as the protocol. The inter-administration interface (X) included both CMIP and CORBA GIOP/IIOP as possible choices at the application layer.

CORBA, a distributed processing technology, is being considered for use in the TMN communication architecture primarily due to its acceptance by the Information Technology industry. This acceptance is expected to enhance the availability of CORBA-based interfaces due to better development tools and wide-spread expertise in developing CORBA-based interfaces. This technology, developed by the Object Management Group (OMG), is also being considered by multiple industries. Specifications using this technology provide support for standard application programming interfaces (APIs) and language bindings to programming languages, and they also facilitate software portability. The interoperability solutions offered by the object request broker combined with the inter-ORB protocols address interoperability between client and server. While CMIP and information models provide solutions for interoperability between manager and agent systems, CORBA defines inter-object interactions where the objects may be distributed.

1.1 Purpose

Several groups are developing network management specifications that use CORBA modeling techniques with IDL as the notation along with CORBA services. The scope of this standard is to define protocol requirements and common services suitable for use in the specification of interoperable CORBA-based network management interfaces. Previous standards for CORBA-based network management interfaces have mainly focused on TMN "X" interfaces, which are interfaces between administrations (carriers). The demands placed on these interfaces are different from those used "inside" an administration, "Q" interfaces. The scope of this Recommendation covers all interfaces in the TMN where CORBA may be used. It is expected that not all capabilities and services defined here are required in all TMN interfaces. This implies that the framework

can be used for interfaces between management systems at all levels of abstractions (inter and intra-administration) as well as between management systems and network elements.

This Recommendation is intended for use by various groups specifying network management interfaces. A number of factors are considered: the version of CORBA to use, the set of CORBA Common Object Services employed, and additional services. This Recommendation, along with the object modeling guidelines defined in ITU-T Rec. X.780 form a *framework* for CORBA-based TMN interfaces. Use of a common framework on telecommunications management interfaces has several advantages. Some examples are: facilitating reuse of models that are developed to meet the generic requirements of telecommunications; profiling CORBA services for use by the telecommunications industry; easing the definition of new services for TMN, reusing the semantics of the existing rich set of models; and harmonizing the modeling approach across groups using a single source similar to Recommendations X.720, 721 and 722 for CMIP. Re-using a common framework and generic information model for a variety of network technologies and network management applications will speed the introduction of new network services while keeping network management system development costs down.

The telecommunications industry has invested a great deal of time and energy in the development of information models for the CMIP network management protocol. A primary goal of this framework is the re-use of these information models by enabling their translation to CORBA Interface Definition Language (IDL) with little change in semantics (see Recommendation X.780). As a result, initial IDL information models are expected to be derived from CMIP models.

In addition to taking advantage of CMIP information models, another purpose of the framework is to take advantage of CORBA. The framework leverages the functions defined in the CORBA specifications, including a set of Common Object Services. Also, the framework tries to re-use CORBA approaches and design patterns wherever they fit. Finally, while re-using existing models is important, it is equally important that the framework support the development of new models. This framework does not require a GDMO model to be developed prior to the development of an IDL model. In fact, developing a new IDL information model for use within this framework is straightforward and guidelines for doing so are provided.

1.2 Application

As CORBA is introduced in TMN, different scenarios are possible that range from the use of gateways performing translations between systems using different network management protocols to cases where CORBA is natively supported by the communicating systems. The application of this framework is intended for scenarios where both the managed system and the managing system provide CORBA interfaces.

The framework does not address other inter-working scenarios requiring “gateway” systems where protocol and information model conversions are necessary for achieving interoperability. In particular, this framework is not specifically designed to support the

construction of gateways between CORBA and CMIP network management applications even though the semantics of the existing models are retained by this framework. A management system, however, might have to support multiple protocols, to inter-work in different environments.

A gateway approach has already been developed and standardized by the Joint Inter-Domain Management (JIDM) group. This gateway approach provides a one-to-one mapping of all constructs and capabilities available with CMIP and GDMO. However, many of the CORBA services and capabilities are not reused by this approach because the problem solved is to facilitate inter-working with systems that have been deployed using CMIP. In contrast, the problem domain for applying this framework is to support standards-based native CORBA network management interfaces. Such an approach takes advantage of the benefits offered by CORBA as a technology used by multiple industries.

ITU-T Recommendation X.780[1] accompanies this Recommendation and defines object modeling guidelines, superclasses for all managed objects and managed object factories for use with this framework, and a standard set of notifications. Together, X.780 and this Recommendation define a *framework* for CORBA-based TMN interfaces. Also, ITU-T Recommendation M.3120[30] provides a CORBA IDL version of the generic network information model originally defined in Recommendation M.3100. The IDL version follows the object modeling guidelines in X.780 and is designed to fit with the services defined here.

1.3 Document Roadmap

This document has the following structure:

Section 1.	Introduction, document roadmap, updates, and list of issues.
Section 2.	References.
Section 3.	Definitions of terms and abbreviations used throughout the rest of the document.
Section 4.	Requirements for the TMN CORBA-based services. These are the design goals the services must meet.
Section 5.	CORBA ORB and Service version requirements. Also provided is an overview of the services.
Section 6.	Requirements on the use of CORBA Common Object Services for network management interfaces.
Section 7.	Definition of TMN-specific support services. IDL interfaces for the support services are defined in Annex A.
Section 8.	Compliance and conformance guidelines.
Annex A.	TMN-specific support service IDL.
Appendix A.	Interworking Scenarios Between Models Using ITU Framework and ADSL/ATMF Compliant Models

1.4 Document Conventions

A few conventions are followed in this document to make the reader aware of the purpose of the text. While most of the document is normative, paragraphs succinctly stating mandatory requirements to be met by a management system (managing and/or managed) are preceded by a boldface "R" enclosed in parentheses, followed by a short name indicating the subject of the requirement, and a number. For example:

(R) EXAMPLE-1 An example mandatory requirement.

Requirements that may be optionally implemented by a management system are preceded by an "O" instead of an "R." For example:

(O) OPTION-1 An example optional requirement.

The requirement statements are used to create compliance and conformance profiles.

Many examples of CORBA IDL are included in this document, and IDL specifying the TMN specific services, and supporting data types, included in a normative annex. The IDL is written in a 9-point courier typeface:

```
// Example IDL
interface foo {
        void operation1 ();
};
```

Instructions for extracting the IDL from an electronic version of this document and compiling it are presented in the next section.

1.5 Compiling the IDL

An advantage of using IDL to specify network management interfaces is that IDL can be "compiled" into programming code by tools that accompany an ORB. This actually automates the development of some of the code necessary to enable network management applications to interoperate. This document has one annex that contains code that implementers will want to extract and compile. Annex A is normative and should be used by developers implementing systems that conform with this standard. The IDL in this document has been checked with two compilers to ensure its correctness. A compiler supporting the CORBA version specified in Section 5.2 must be used.

The annex has been formatted to make it simple to cut and paste it into a plain text file that may then be compiled. Below are tips on how to do this.

1. Cutting and pasting seems to work better from the Microsoft® Word® version of this document. Cutting and pasting from the Adobe® Acrobat® file format seems to include page headers and footers, which cannot be compiled.
2. All of Annex A, beginning with the line "/* This IDL code…" through the end should be stored in a file named "itut_q816.idl" in a directory where it will be found by the IDL compiler.

3. The headings embedded in the annex need not be removed. They have been encapsulated in IDL comments and will be ignored by the compiler.
4. Comments that begin with the special sequence "/**" are recognized by compilers that convert IDL to HTML. These comments often have special formatting instructions for these compilers. Those that will be working with the IDL may want to generate HTML as the resulting HTML files have links that make for quick navigation through the files.
5. The annex has been formatted with tab spaces at 8-space intervals and hard line feeds that should enable almost any text editor to work with the text.

2 References

2.1 Normative References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

[1] ITU-T Recommendation X.780, *Guidelines for Defining CORBA Managed Objects.*

[2] The Object Management Group (OMG), "The Common Object Request Broker: Architecture and Specification", OMG Document formal/99-10-07, Revision 2.3.1, October, 1999.

[3] The Object Management Group (OMG), "Naming Service Specification", OMG Document formal/2000-06-19, Version 1.0, April, 2000.

[4] The Object Management Group (OMG), "Notification Service Specification", OMG Document formal/2000-06-20, Version 1.0, June, 2000.

[5] The Object Management Group (OMG), "Telecom Log Service Specification", OMG Document formal/00-01-04, Version 1.0, January, 2000.

[6] The Object Management Group (OMG), "Security Services Specification", OMG Document formal/2000-06-25, Version 1.5, May, 2000.

[7] The Object Management Group (OMG), "Transaction Service Specification", OMG Document formal/2000-06-28, Version 1.1, May, 2000.

[8] The Object Management Group (OMG), "CORBA Messaging," OMG TC Document orbos/98-05-05, May, 1998.

[9] The Object Management Group (OMG), "JIDM Interaction Translation," Edition 4.31, OMG Document telecom/98-10-10, October 1998.

[10] Internet Engineering Task Force (IETF), "The TLS Protocol Version 1.0," RFC 2246, Version 1.0, January, 1999.

[11] The Institute of Electrical and Electronics Engineers (IEEE), "Information Technology – Portable Operating System Interface (POSIX) Part 2: Shell and Utilities," IEEE/ANSI Standard 1003.2-1992, 1992.

2.2 Additional References

The following standards contain information that was used in the development of this framework. As stated in the introduction, a primary design goal of this framework is to enable the re-use of existing network management information models, at least without significant semantic changes. These documents provide many of the details on the ITU-T's CMIP framework, and therefore define some of the functionality the CORBA framework must support.

[12] ITU-T Recommendation X.703 (1997), *Information Technology – Open Distributed Management Architecture*, October, 1997.

[13] ITU-T Recommendation X.710 (1997), *Common Management Service Definition for ITU-T Applications*, October, 1997.

[14] ITU-T Recommendation X.711 (1997), *Common Management Information Protocol Specification for ITU-T Applications*, October, 1997.

[15] CCITT Recommendation X.720 (1992) | ISO/IEC 10165-1 : 1992, *Information Technology – Open Systems Interconnections – Structure of Management Information: Management Information Model*.

[16] CCITT Recommendation X.721 (1992) | ISO/IEC 10165-2 : 1992, *Information Technology – Open Systems Interconnections – Structure of Management Information: Definition of Management Information*.

[17] CCITT Recommendation X.722 (1992) | ISO/IEC 10165-4 : 1992, *Information Technology – Open Systems Interconnections – Structure of Management Information: Guidelines for the Definitions of Managed Objects*.

[18] ITU-T Recommendation X.711 Cor. 2, *Corrigendum 2 to ITU-T Recommendation X.711*, January, 2000.

[19] ITU-T Recommendation X.720 Cor. 1, *Corrigendum 1 to CCITT Recommendation X.720*, February, 1994.

[20] ITU-T Recommendation X.721 Cor. 1, *Corrigendum 1 to CCITT Recommendation X.721*, February, 1994.

[21] ITU-T Recommendation X.721 Cor. 2, *Corrigendum 2 to CCITT Recommendation X.721*, October, 1996.

[22] ITU-T Recommendation X.721 Am. 1, *Amendment 1 to CCITT Recommendation X.721*, November, 1995.

[23] ITU-T Recommendation X.722 Cor. 1, *Corrigendum 1 to CCITT Recommendation X.722*, October, 1996.

[24] ITU-T Recommendation X.722 Cor. 2, *Corrigendum 2 to CCITT Recommendation X.722*, January, 2000.

[25] ITU-T Recommendation X.722 Am. 1, *Amendment 1 to CCITT Recommendation X.722*, November, 1995.

[26] ITU-T Recommendation X.722 Am. 2, *Amendment 2 to CCITT Recommendation X.722*, August, 1997.

[27] ITU-T Recommendation X.722 Am. 3, *Amendment 3 to CCITT Recommendation X.722*, August, 1997.

[28] CCITT Recommendation X.733 (1992) | ISO/IEC 10164-4 : 1992, *Information Technology – Open Systems Interconnection – Systems Management: Alarm Reporting Function.*

[29] ITU-T Recommendation M.3010 (2000), *Principles for a Telecommunications management network*, February, 2000.

[30] ITU-T Recommendation M.3120, *CORBA-Based Generic Network Information Model.*

[31] ITU-T Recommendation Q.821 (2000), *Stage 2 and Stage 3 description for the Q3 interface - Alarm Surveillance*,(to be published).

3 Definitions

This section provides definitions for acronyms used throughout the rest of the document.

AMI	Asynchronous Messaging Invocation.
API	Application Programming Interface.
ASN.1	Abstract Syntax Notation #1.
ATM	Asynchronous Transfer Mode.
AVA	Attribute Value Assertion.
CMIP	Common Management Information Protocol.
CORBA	Common Object Request Broker Architecture.
COS	Common Object Services.
DN	Distinguished Name.
EMS	Element Management System.
FIFO	First In, First Out.
GDMO	Guidelines for the Definition of Managed Objects.
GIOP	General Interoperability Protocol.
HTML	Hypertext Markup Language.
ID	Identifier.
IDL	Interface Definition Language.
IEEE	The Institute of Electrical and Electronics Engineers.
IETF	The Internet Engineering Task Force.
IIOP	Internet Interoperability Protocol.
IOR	Interoperable Object Reference.
ITU-T	International Telecommunication Union – Telecom.
JIDM	Joint Inter-Domain Management.
MO	Managed Object.
MOO	Multiple Object Operation.
NE	Network Element.

NMS	Network Management System.
OAM&P	Operations, Administration, Maintenance, and Provisioning.
ORB	Object Request Broker.
OID	Object Identifier.
OMG	Object Management Group.
OSI	Open Systems Interconnection.
PDU	Protocol Data Unit.
POA	Portable Object Adapter.
POSIX	Portable Operating System Interface.
POP	Point of Presence.
PM	Performance Management.
QoS	Quality of Service.
RDN	Relative Distinguished Name.
SDH.	Synchronous Digital Hierarchy.
SONET	Synchronous Optical Network.
SSL	Secure Socket Layer.
TII	Time-Independent Invocation.
TLS	Transport Layer Security.
TMN	Telecommunications Management Network.
TTP	Trail Termination Point.
UID	Universal Identifier.
UML	Unified Modeling Language.
UTC	Universal Time Code.

4 CORBA Based TMN Services Goals and Requirements

This section describes the key goals of the services framework and the requirements that help the CORBA Based TMN services support these goals. Section 4.1 introduces the goals of the CORBA framework. Section 4.2 then provides terminology and requirements. The requirements in Section 4 are requirements that the framework must satisfy. They are based on the telecommunications management needs. Sections 5, 6, 7, and 8 then describe a framework that meets these needs and define how to achieve the requirements of section 4 by using CORBA in a certain way. The rules in Section 5, 6, 7, and 8 on how to use CORBA also are referred to as requirements.

4.1 Goals

This document sets out to define a framework for defining how interfaces supported by management systems and network elements should be modeled. Some key goals of the framework are identified here:

- Application Interoperability
- Common Usage of CORBA Common Object Services
- Information Model Transparency

This section elaborates on these three goals.

4.1.1 Application Interoperability

A key goal of the TMN architecture, and in particular the information architecture, is to promote a standard framework for providing interoperability and information exchange between systems from a diverse set of network management system suppliers. Interoperability between systems involves many aspects of development. At its lowest layer, a common communication mechanism must be in place to support a common syntax, the establishment of connectivity and the exchange of operation requests/replies between systems. This aspect of interoperability is inherently supported by the CORBA specification.

For TMN, there is the need to provide application interoperability. That is, management systems from diverse suppliers will be utilized within a single administration's TMN to support different functions necessary to support management of its networks. To simplify integration of these various suppliers' systems, they must agree on the semantics of the information being exchanged. This is accomplished with the specification of an information model. Guidelines for the definition of CORBA-based information models are specified in X.780, but the services defined here must support those guidelines.

4.1.2 Common Usage of CORBA Common Object Services

A second aspect of this framework is the definition of common usage and profiling of the distributed processing environment of choice. This aspect of the framework should indicate reasonable expectations network management system suppliers may have for one another. Rather than re-defining the interface capabilities needed to support common network management functions such as object naming and notification filtering with each information model, the modeling guidelines in X.780 rely upon a set of support services. These support services enable the information models to be simpler, and also enhance interoperability.

In defining these services, special effort will be taken to make use of the CORBA Common Object Services. Specifically, this Recommendation will address the use of the CORBA ORB and CORBA Common Object Services (COS) that will impact system interoperability (i.e., issues involving the use of CORBA within a single system are outside the scope of this document). Where network management needs cannot be met by CORBA COS, additional services will be defined.

4.1.3 Information Model Transparency

If CORBA is used in places within the TMN architecture where existing information models (e.g., GDMO) are well established, then the framework must support the reuse of those models without any major changes.

The focus of this Recommendation is on the set of services required to allow the existing models to be used as they were originally intended with a reasonable amount of efficiency.

4.2 Information Modeling Dependencies

As described in the previous section, the explicit modeling of resources that are manageable across an interface is central to application interoperability. The guidelines for defining CORBA managed objects detailed in X.780 describe the rules for modeling manageable resources. They also embody several decisions that must be supported by the TMN CORBA-based services framework. This section summarizes those points.

4.2.1 Access Granularity

CORBA interface *granularity* refers to the relationship between the resources that are modeled on an interface and the means by which they are accessed using CORBA. X.780 uses an *instance-grain* modeling approach, which means each modeled resource is accessible using a unique CORBA object reference, known as an *interoperable reference* (IOR). The objects that represent manageable resources are called *managed objects*.

4.2.2 Representation of Containment and Naming

Containment is a logical representation of how modeled resources contain other modeled resources. Containment has traditionally been a very important relationship in network management applications because it is a convenient means of identifying the large number of resources that typically must be managed. X.780 guidelines require that a unique name be assigned to each managed object, based in part on the name of the object that contains it. The TMN CORBA-based services must provide a means to store these names (and hence the containment relationships they represent) as well as a means to find the IOR of an object based on its name.

4.2.3 Object Creation and Deletion

The CORBA ORB does not provide clients with a means to create objects on remote systems. Instead, typically *factory* objects are instantiated by remote systems, and these factory objects provide operations that may be invoked by clients to create objects on the remote system. For a number of information modeling purposes, X.780 specifies that a factory IDL interface shall be defined for each managed object IDL interface in an information model. So, object creation will be model-dependent and is not a good candidate for a TMN CORBA service. However, a service for managing system seeking a factory in order to request creation of an object in a managed system is defined in this Recommendation.

Object deletion is also an area in need of support. Often, CORBA objects are deleted by simply invoking some delete operation on the object, but this is not a good approach for network management applications because of their reliance on containment relationships. Deleting an object that contains other objects has impacts beyond the object being deleted. Also, as described in the previous section, support is required for storing the names of managed object instances, and this data must be updated when objects are deleted. The TMN CORBA services therefore need to provide support for deleting managed objects in an orderly fashion.

4.3 Scoping and Filtering

The ability to perform complex queries (i.e., GET operations), updates (i.e., SET operations), and delete operations on a group of Entities with a single operation request is a valuable component of TMN. Management systems may have to manage up to 10^7 instances of managed objects. Due to the size of the management information base, a managing system can not efficiently perform ad-hoc queries on individual instances of Managed Objects (i.e., Entities). Rather, the managing system expects a level of intelligence to be supported by the managed system.

The intelligence in the managed system allows the managing system to select a group of managed entities on which some operation will be performed. Managed entity selection involves two phases: scoping and filtering. This managed entity selection process is supported by a service defined later in this Recommendation. This service allows a managing system to select a scope of objects to act on (scope is defined through containment relationships, see Section 4.2.2). Once the scope of Entities is determined, the operation (specified by the scope and filtered request) is performed only on those Entities which meet criteria defined by a filter.

The use of scoping and filtering in this framework supports:

- Scoped and Filtered get: returns the values (for a list of attributes) from each of the Entities that meet the scope and filter criteria.
- Scoped and Filtered update: replaces an attribute value or adds/removes values to/from set-valued attributes, in the group of Entities meeting the scope and filter criteria, to the values specified in the scoped and filtered request. May be used to update one or multiple attributes in a single object or multiple objects.
- Scoped and Filtered deletion: deletes all Entities that meet the scope and filter criteria.

4.3.1 Scoping

Scoping entails the identification of the Entities to which a filter is to be applied. Scoping is applied based on the containment hierarchy as defined in Section 4.2.2. The scope is applied from some base managed entity down to some depth in the containment tree.

The base entity for the scope is defined as the root of the containment tree from which the search is to commence. A scoped request must specify the base managed entity of the scope. The depth of the scoping level can then be specified in one of four manners within the scoped request:

1. the base entity.
2. the nth level subordinates of the base entity.
3. the base entity and all of its subordinates down to and including the nth level.
4. the base entity and all of its subordinates (i.e., the whole subtree).

4.3.2 Filtering

Filters allow for the specification of criteria that Entities must meet in order to have a management operation performed. Together with scoping, filtering allows a single

operation to be performed across multiple managed objects with a single operation request.

A filter parameter is used to determine whether or not an operation should be performed on a managed object. A filter parameter applies a test that is either satisfied or not by a particular managed object. The filter is expressed in terms of assertions about the presence or value of certain attributes of the managed object, and it is satisfied if and only if it evaluates to TRUE.

4.3.2.1 Attribute Matching Rules

The following matching rules are defined that may be used in attribute value assertions (AVAs). These rules are:

- **Equality:** evaluates to TRUE if and only if the value supplied in the AVA is equal to the value of the attribute.

  For SET valued attributes, the AVA evaluates to TRUE if and only if the set of members supplied in the AVA is equal to the set of members in the attribute.

- **Greater or equal:** evaluates to TRUE if and only if the value supplied in the AVA is greater than or equal to the value of the attribute.

  For SET valued attributes, the value in the AVA shall contain exactly one member. The AVA evaluates to TRUE if and only if that member is greater than or equal to at least one of the members in the attribute value.

- **Less or Equal:** evaluates to TRUE if and only if the value supplied in the AVA is less than or equal to the value of the attribute.

  For SET valued attributes, the value in the AVA shall contain exactly one member. The AVA evaluates to TRUE if and only if that member is less than or equal to at least one of the members in the attribute value.

- **Present:** evaluates to TRUE if and only if such an attribute is present in the managed object.

- **Substrings:** evaluates to TRUE if and only if all of the substrings specified in the AVA appear in the attribute in the given order without overlapping and separated from the ends of the attribute value and from one another by zero or more string elements. In addition, for the AVA to evaluate to TRUE,

  - The first element in the initial substring, if present, shall match the first element in the attribute value;
  - The other substrings, if present, shall appear in the attribute value in the order that the substrings appear in the AVA; and

– The last element in the final substring, if present, shall match the last element in the attribute value.

For SET valued attributes, each value in the AVA shall contain exactly one member. The AVA evaluates to TRUE if and only if there is at least one of the members of the attribute value in which all of the substrings supplied in the AVA appear as described above.

(The remaining three matching tests apply to SET valued attributes only)

– **subset of:** evaluates to TRUE if and only if all asserted members are present in the attribute.
– **superset of:** evaluates to TRUE if and only if all members of attribute are present in the attribute value assertion.
– **non-null set intersection:** evaluates to TRUE if and only if at least one of the asserted members is present in the attribute.

4.4 Notifications

The framework needs to support the ability to:

- Deliver notifications
- Subscribe for notification types
- Forward notifications to multiple destinations
- Filter notifications
- Uniquely identify the resource that emits the notification

The framework must also support the requirements on notification content, clearing, and correlation algorithms found in ITU Rec. X.733[28] and ITU Rec. Q.821[31].

5 Framework Overview and Protocol Requirements

The previous section outlined the network management functions the framework must support. This section and the rest of the document provide the details on how the CORBA Based TMN Services will provide these functions. The aspects of the framework related to modeling objects are included in Recommendation X.780. First, a brief overview of the framework is presented, then some basic protocol requirements are defined.

5.1 Framework Overview

This framework for CORBA-based TMN interfaces is a collection of capabilities. A central piece of the framework is a set of CORBA Common Object Services. This framework defines their role in network management interfaces, and defines conventions for their use. The framework also defines support services that have not been standardized as CORBA Common Object Services, but are expected to be standard on network management interfaces conforming to this framework. IDL interfaces for these services are defined later in Annex A.

To support the software objects representing manageable resources, the framework requires that they implement some common basic capabilities. Therefore, two base classes are defined in Recommendation X.780 for use in modeling network management resources. Managed object classes (or object classes) in information models must inherit and implement a basic set of capabilities from these base classes in order to operate within this framework. Finally, some rules and conventions are defined for information modelers developing models for use with this framework. These consist of modeling guidelines, rules for converting GDMO models for CMIP to CORBA IDL definitions, and IDL style idioms. All of these are depicted graphically in Figure 1.

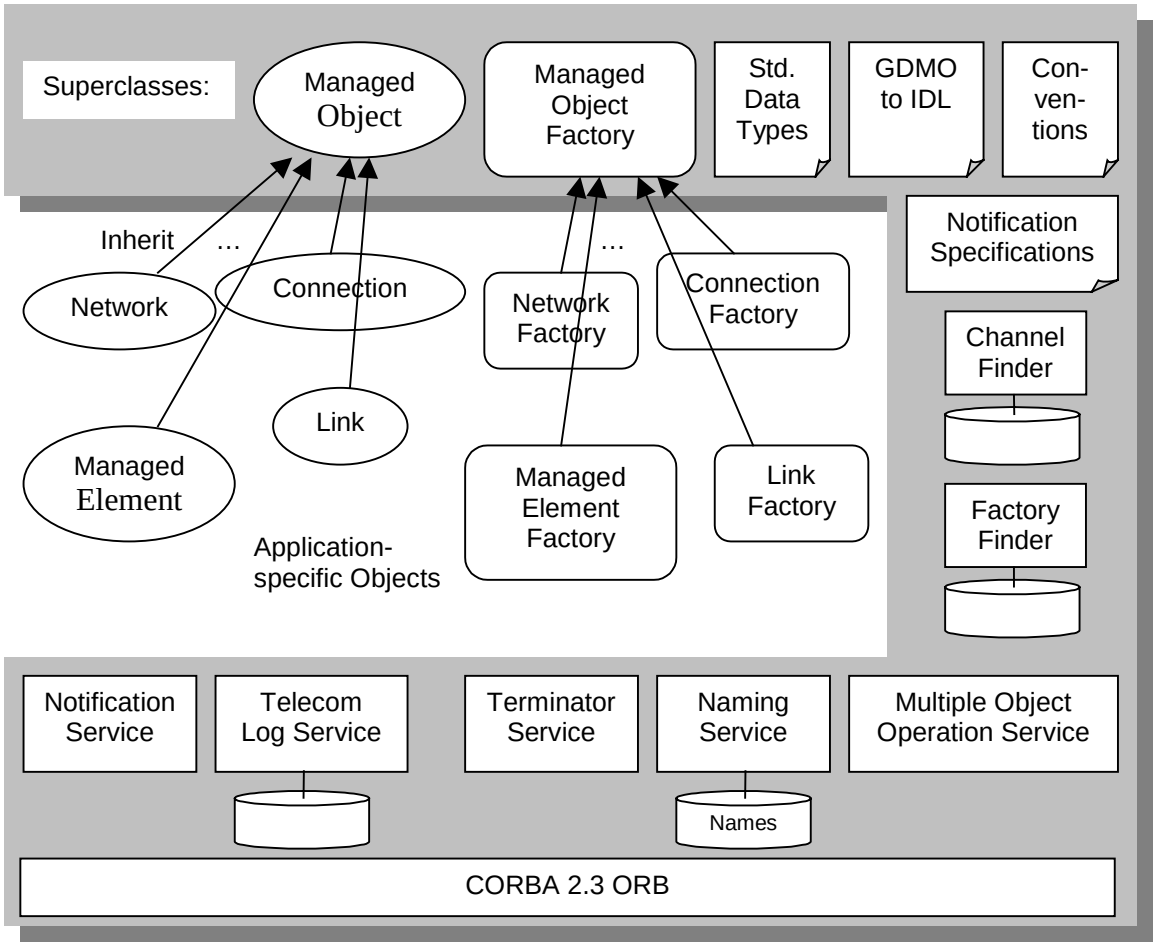

Figure 1. Overview of Framework

The figure shows the framework in gray. In the middle are the application-specific objects that are supported by the framework. Along the bottom is a box representing the CORBA ORB. Above that are a number of boxes with names in them representing the services that compose the framework. (Some also have icons depicting the databases they would have to maintain to perform their functions.) Along the top of the figure are icons representing two superclasses, one for managed objects and one for managed object factories. Each of the managed objects and managed object factories supported by this framework must ultimately inherit from these superclasses, respectively. Also shown on

the figure are icons of pages with up-turned corners representing standard object modeling conventions.

The framework services, represented as boxes with square corners, are defined in this document. The superclasses, notifications, and object modeling conventions are defined in ITU-T Recommendation X.780.

5.2 Framework Protocol Requirements

This section defines the versions of the services that are required to support this framework. CORBA services and protocol specifications are defined by the Object Management Group (OMG). The table below shows which version of the applicable OMG specification must be supported to comply with this framework and indicates the section where detailed requirements are defined for the service. A later version of a service that includes all the required capabilities of the stated version complies with this framework.

Service	Version	Section
ORB	2.3.1 [2]	5.2
Naming Service	1.0 [3]	6.1
Notification Service	1.0 [4]	6.2
Telecommunications Logging Service	1.0 [5]	6.3
Asynchronous Messaging	(determined by client system)	6.4
Security (if required)	Either the "Secure IOP protocol", or "CORBASecurity SSL Interoperability", as defined in [6]	6.5
Transaction Service	1.1 [7]	6.6

Table 1. CORBA Service Versions

The choice of version 2.3.1 for the basic ORB capabilities is important. CORBA 2.3 includes support for the Portable Object Adapter (POA) as well as for passing objects by value. POA is important to the framework because it enables implementations based on this framework to scale up to millions of instantiated objects, a magnitude required for network management applications. The framework also makes use of value type inheritance (which supports polymorphism) to retain flexibility but reduce the usage of CORBA “any” types, which can be inefficient and tedious for programmers.

The Naming, Notification, and Logging services are all the initial versions available from the OMG.

Asynchronous Messaging is really only a client-side consideration. An ORB with Asynchronous Messaging capabilities enables a client to use synchronous CORBA interfaces (those that would normally cause the client to block) in an asynchronous fashion. This capability is essential for clients that are single-threaded and cannot afford to block during network management operations. The availability of Asynchronous Messaging capabilities is important to this framework because it frees it from having to

define both synchronous and asynchronous interfaces. Clients need not use an ORB with Asynchronous Messaging if they are multi-threaded and therefore can afford to block during synchronous CORBA calls.

If security is required, this framework condones the use of ORBs that use SSL 3.0 for security until products supporting TLS become available, and the OMG migrates the CORBA Security. Until the OMG CORBA Security Service specification references TLS, the choice of which is supported in a product (if any) will have to be negotiated between individual suppliers and users. So, for now, the use of either one (or none) is compliant with this framework.

6 Framework Common Object Services Requirements

The CORBA ORB provides basic object-to-object interaction capabilities.[2] Additional capabilities are defined as separate, "Common Object Services." The CORBA Common Object Services are general purpose, domain-independent services that are fundamental for developing CORBA applications composed of distributed objects. They also provide the basic building blocks for application interoperability. The services are defined with object interfaces and can be combined in many different ways and put to many uses in different applications. In a specific domain, CORBA Common Object Services can be used to construct higher-level facilities and object frameworks that can inter-operate across multiple platform environments.

Many of these CORBA Common Object Services have already been implemented and are available as commercial, off-the-shelf software products. Also, programmers working in many industries will likely have experience with them in the near future. Re-using these Common Object Services instead of defining new ones strictly for the telecommunications industry or re-implementing the functionality in application-specific code will result in a quicker, more cost-efficient adoption of CORBA for network management.

The following sub-sections specify requirements on the use of CORBA Common Object Services to ensure interoperability between different network management systems and to preserve the telecommunications context..

6.1 The Naming Service

The OMG Naming Service is CORBA's directory service, or "white pages."[3] It allows a client to build a name-to-object association called a *name binding* that other clients can then use to find the object. (CORBA object references are binary and difficult for use by humans.) A name binding is always defined relative to a *naming context.* A naming context is an object that contains a set of name bindings in which each name is locally unique. A name binding is a data structure containing two strings and an object reference (address). The *ID* string is the identifier for the binding. A second string, called "kind," is also part of the data structure. Together, the ID and kind uniquely identify an object relative to a context. Different names can be bound to an object in the same or different contexts at the same time. The naming context can also be bound to a name in another

naming context. Binding contexts in other contexts creates a *naming graph* – a directed graph with nodes and labeled edges where nodes are contexts. Given a context in a naming graph, a sequence of name components (*ID-kind* pairs) can reference an object. This sequence of structures, called a *compound name*, defines a path in the naming graph that may be navigated to resolve the name and find the object.

There is no requirement that CORBA name bindings represent a containment relationship between objects, but the concept of containment is important in network management and needs to be communicated across network management interfaces. The CORBA Naming Service is the best way to accomplish this. The following paragraphs define a series of requirements on using the CORBA Naming Service to represent the containment relationships among managed object instances.

(R) NAME-1 Every managed object shall have one and only one name (DN). The components of the name may be obtained from multiple federated servers. Although the OMG Naming service supports multiple names per object, this framework restricts a managed object to using a single name. Support for multiple names is outside the scope of the framework.

(R) NAME-2 Since a simple name binding cannot identify an object and also contained objects, each managed object must actually have a corresponding Naming Context. A specially-named binding in each such context will bind the *ID* value "Object" with a reference to the actual managed object. (The *kind* field of this binding will be null.) Other naming contexts, representing contained managed objects, may also be bound to names in this context.

(R) NAME-3 The *ID* field of a name binding for a naming context representing a managed object will be application-dependent, and it may actually have semantic value beyond uniquely identifying a managed object, for a particular class of objects. For example, an *ID* value of "7" for an equipment holder object representing a slot in a shelf may indicate that this object represents the 7^{th} slot in the shelf. Special semantic value attached to IDs will be documented for each class of managed objects as part of the managed object interface specification. Note that the *ID* field is a string.

(R) NAME-4 The *kind* field of a name binding for a naming context representing a managed object shall be determined by *managed object name binding information*. This is information defined as constants in IDL modules specifically for the purpose of representing possible containment relationships. See ITU-T X.780 for details on the representation of managed object name binding information. In short, however, a name binding module will contain a constant string named "kind" that will be used as the value for the *kind* field in CORBA name bindings. The value of this string will usually be the unscoped class name of the managed object. This adds value by making it easier to identify the type of an object and by reducing the likelihood of name collisions. One factor complicating this is the release of new versions of an object, for example, an *equipmentR1* that extends an *equipment* object. When the new class merely extends the capabilities of an existing class without changing its purpose (that is, it still represents the

same managed resource), the *kind* field will usually be the original base class name. This, however is ultimately up to the object modeler who defines the name binding IDL module. Using the original value will enable existing applications to continue to use the new class as if it was the old version.

The following figure gives an example of name bindings according to the above requirements. In the figure, CORBA Naming Contexts are represented as folders. The contents of the folders are name bindings. The convention for representing a name component as a string with the format <ID>.<kind> is used. (Some example name bindings do not have a pointer shown in the diagram to reduce the complexity of the diagram.) The graph represents a Network object, named "CentralNet," that contains a Managed Element object named "Element9" and a Connection named "R5698."

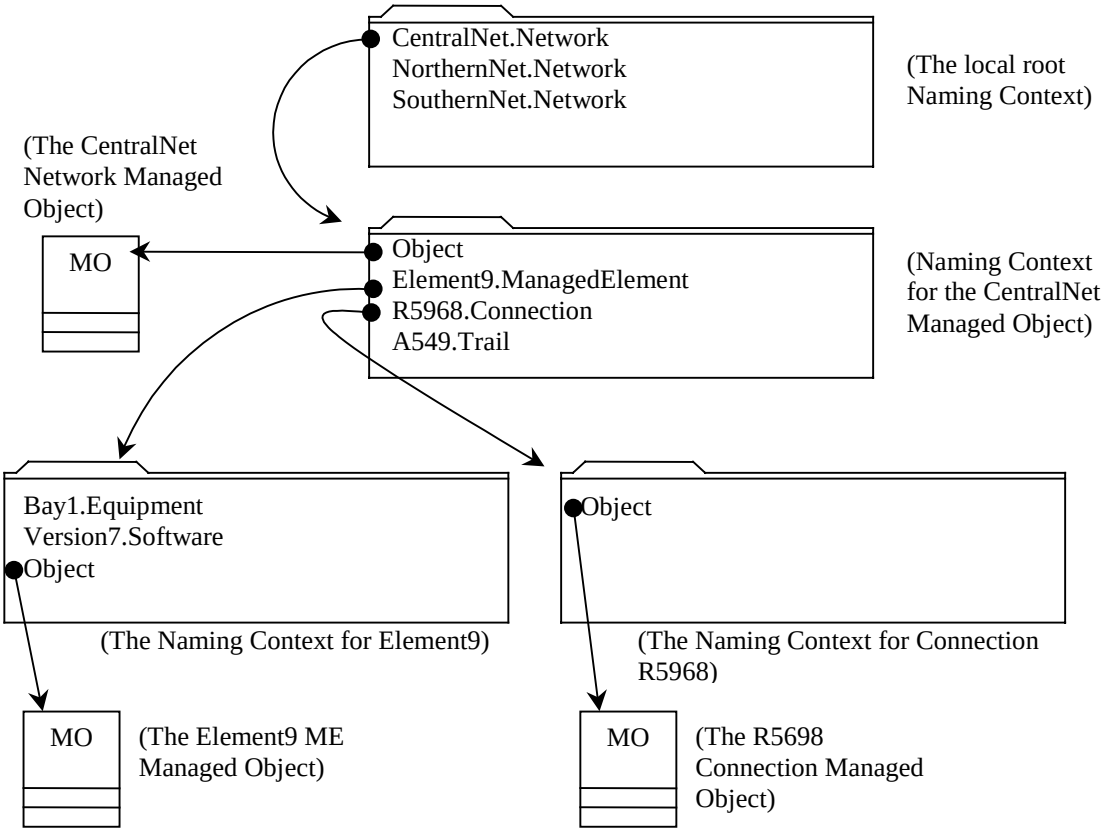

Figure 2. Naming Graph of Managed Objects

(R) NAME-5 Each managed system shall provide at least one local root naming context. Note on the figure above that the top-most naming context is referred to as a "local root" naming context. This is the naming context in which names for the top-most managed objects on the system will be bound, as well as names for certain support service objects.

A managed system may have multiple local root naming contexts. Since managed objects cannot have multiple names, they may be bound under only one local root. Support service objects, however, may have names bound under multiple root naming

contexts on the same system. One factor to consider when determining how many local root naming contexts a managed system will have is if the possibility exists that some of the managed objects might sometime have to be moved to another system. Moving an entire tree of managed objects, including the local root naming context, will be simpler than moving a subtree of objects.

(R) NAME-6 A managed system shall provide a local administrative procedure for assigning a CORBA name to each local root naming context on the system. All names exchanged across the managed interface will include the local root context name unless otherwise noted. This includes operation parameters and notifications.

This feature is to enable an administration to make names globally unique. Since the managed system must ensure that all names are unique relative to the local root naming context, by assigning a globally unique name to the local root naming context an administration can ensure that all names on a managed system are unique. The mechanism used to choose a globally unique name for the local root context is up to the administration. The format of the name will be the same as used by the CORBA Naming Service, *CosNaming::Name*. Multiple components are allowed, but administrations will likely want to keep local root context names short to reduce overhead.

In addition to making names unique, assigning a name to the local root naming context will make it easier for a managing system to resolve names. This is because the managing system can bind the local root naming contexts for all the systems it manages into its own local naming service. The name it uses for this binding will be the same name assigned to the root naming context on the managed system. See Figure 3 for an

example.

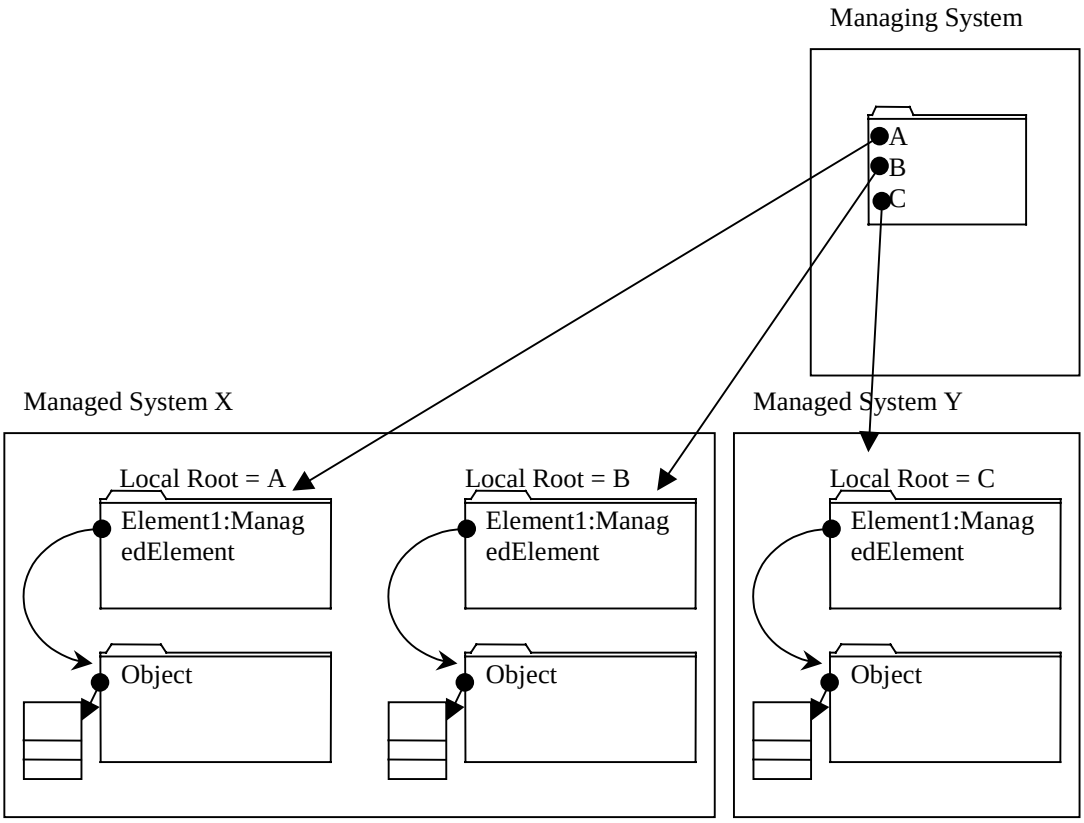

Figure 3. Assigning Names to Root Naming Contexts

The figure shows two element management systems on the bottom. System "X" has two objects of type *ManagedElement*, and System "Y" has 1. Each *ManagedElement* object belongs to its own local root naming context, which means System X has two local roots and System Y has one. There is also a network management system, and the local root contexts of both EMSs have been bound into the naming service on this system. This administration has chosen to assign the unique names "A" and "B" to the local root contexts on System X, and "C" to the local root context on System Y. References to the local root naming contexts have been bound with these names in the network management System.

Say System Y emits a notification concerning its *ManagedElement* object. The full name of that object (contained in the notification) will be "C/Element1.ManagedElement". Now let's say the NMS wants to retrieve more data from the object. In order to do so, it will have to resolve the name into a CORBA object reference. The NMS can accomplish this by simply performing a resolve operation using the full name on the local context where it bound the EMS local root contexts. Because the NMS' naming service is federated with the EMS naming services, the NMS' naming service can automatically forward the resolve operation to the naming service on the proper EMS, and return the object reference to the NMS application.

It is anticipated that the local root naming context name will be assigned during the initialization of a new system. Once in operation, it will be extremely difficult if not impossible to change.

Once assigned a name, the local root context's CORBA Interoperable Object Reference (IOR) will have to be bound to a naming context on the managing system, since up to now it has no idea the new system exists. This means the managed system will also have to provide a means for accessing the "stringified" IOR of the local root naming context. This value will then be transferred to the managing system by some means other than the management interface (e-mail, ftp, etc.). The managing system will require a way to accept this stringified IOR and bind it to a name on the managing system. As soon as the local root context's IOR is bound to a name on the managing system, the managing system can begin discovering the objects on the new system (using the Multiple Object Operation Service described later) and begin to manage it.

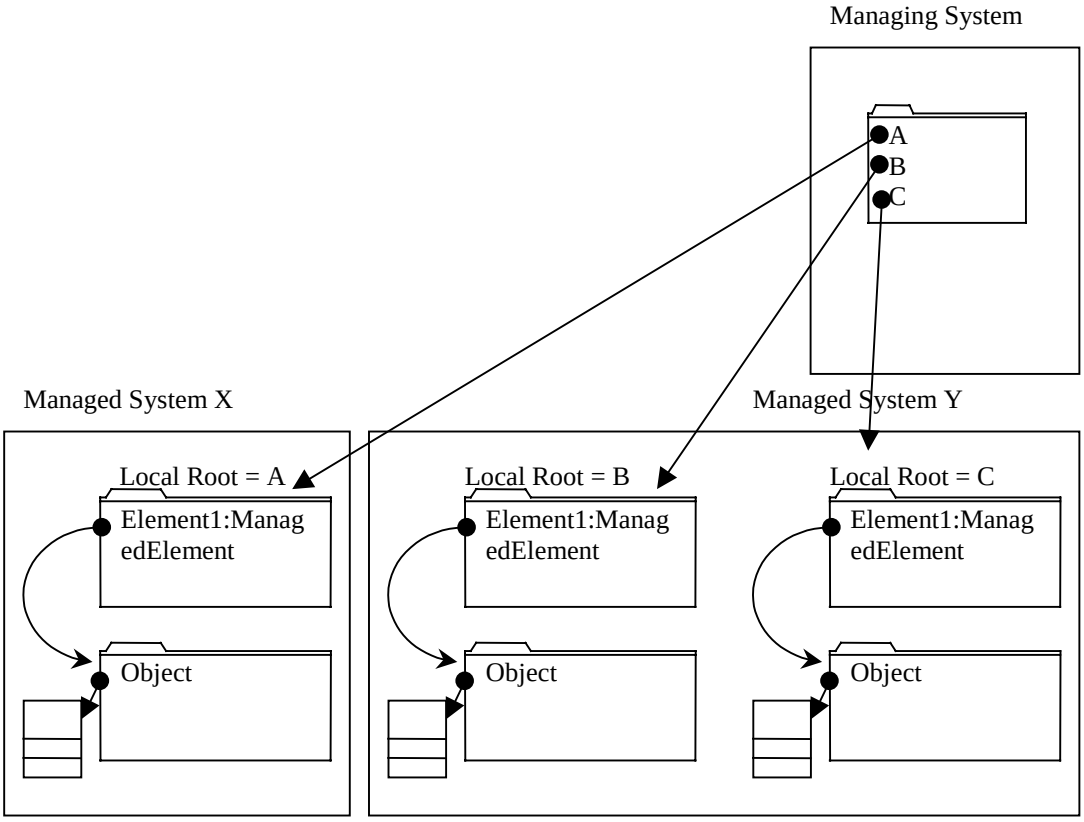

Figure 4. Moving a Local Root Naming Context and Contained Objects

Figure 4 shows how a local root naming context and all of the objects contained below it can be moved to another system without changing the names of the objects. The only change that might be required would be to change the object reference bound to the name in the network management system(s). Also, any outstanding references to moved objects would have to be refreshed. Moving only part of a tree contained below a local root naming context would require re-naming those objects.

6.2 Notification Service

The CORBA Notification Service supports the asynchronous exchange of event messages between clients using a subscribe-and-publish paradigm.[4] The Notification Service introduces event channels that broker event messages, notification suppliers that supply event messages, and notification consumers that consume event messages. The CORBA Notification Service preserves all of the semantics specified for the CORBA Event Service, allowing for backward compatibility with Event Service clients. The extended functionality that is important to the network management domain is the structured event, event filtering, and QoS (Quality of Service). The figure below depicts the general architecture of the Notification Service.

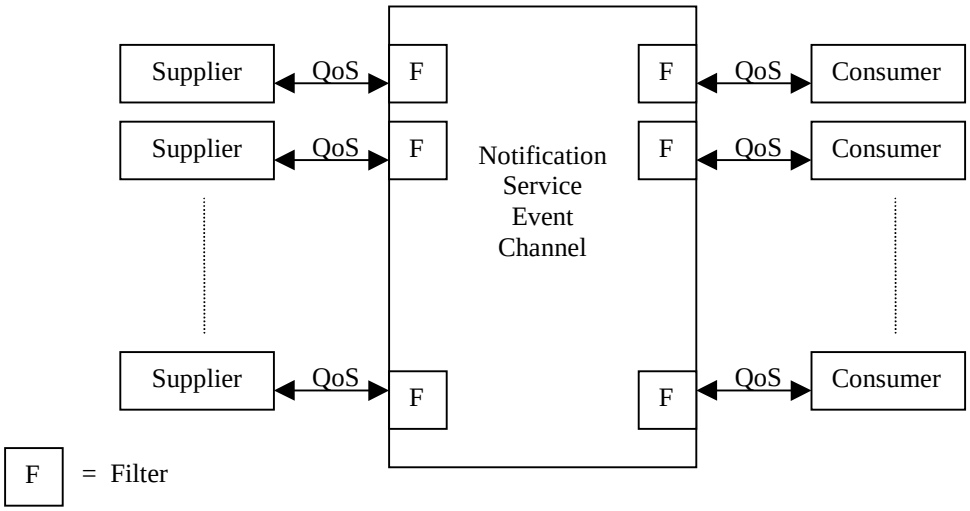

Figure 5. Architecture of the Notification Service

(R) NOTIF-1 The Notification Service shall support the push interface model. The managed object interface to the event channel shall be a push supplier.

(R) NOTIF-2 The managed system shall instantiate the Notification Service event channel object(s) that it will use. A managed system must instantiate at least one channel and may instantiate more than one. (These channels may either be Notification event channels or Telecom Log event channels. See Section 6.3) The framework does not support the creation or deletion of event channels across the management interface. Local administrative procedures may be provided for this purpose. (Event channels do, however, support the creation and deletion of filters across the management interface.)

(R) NOTIF-3 Each event channel shall be registered with the Channel Finder service. The Channel Finder service is a support service defined by this framework in Section 7.2. During registration the channel shall be associated with one or more managed objects that each forms the base of a tree of managed objects that send their events to the channel. Multiple channels may be associated with the same base managed object. A likely use of

this is to have different channels for different types of events. For example, one channel might handle performance management events while another handles alarms. When the channel is registered with the Channel Finder service it is also tied with a set of event types it handles and a set of managed object types that send their events to it. A managed object always sends its events to the channel associated with the nearest object above it that accepts that type of event from that class of object. Every notification from every managed object must go to at least one channel.

While this approach is quite flexible and enables complex arrangements of channels, because channels cannot be created across the management interface the complexity is under the control of the implementation of the managed system. It might be as simple as a single channel monitoring all managed objects on the system. (Please note again that while channels cannot be created across the interface, individual channels do support the creation and deletion of filters across the management interface. Thus, any number of clients may register for the events they wish to receive.)

(R) NOTIF-4 The Notification Service shall support structured events.

(O) NOTIF-5 The use of sequences of structured events is optional. Sequences of structured events are defined in [4] and are used to send multiple events in one message.

(O) NOTIF-6 The use of typed events is optional.

The message interface between suppliers and consumers shall be defined in IDL as if they were using typed events. This is done to enable capturing the notification in IDL (which cannot be done for structured events except with comments) as well as to support typed notifications for applications that wish to use them.

Rules for creating structured notifications based on these typed operations are provided below.

The OMG Notification Service definition does define rules for channels to automatically convert typed notifications to structured notifications. If the managed system natively creates typed notifications, but the client wishes to receive structured notifications, these rules shall be followed by the channel. Note, however, that this arrangement is likely less efficient than both systems using typed events. If the managed system natively creates structured notifications, it shall do so according to the rules below.

The structured notifications natively created by a managed system will differ slightly from the structured notifications created by automatic conversion from typed notifications. One reason for this is to make it possible for a managing system to tell the difference, and accept typed notifications if they are supported by the managed system. Another is to more efficiently use structured notifications. Managed systems that natively create structured notifications may exclude optional parameters from those notifications. Because a typed notification is created from a strongly-typed method invocation, a commercial notification channel that translates this to a structured

notification will include any null values as name-value pairs in the body of the structured event rather than exclude them. Note that allowing managed systems that natively create structured notifications to exclude optional parameters makes it unlikely that commercial notification channels will be able to support the automatic conversion of structured events to typed events.

To recap, a managed system shall send notifications either as structured events or typed events. If the managed system natively creates structured events, it shall do so according to the rules below. Because, for efficiency, these rules allow managed systems to exclude optional parameters from structured notifications, support for automatic conversion of these structured notifications to typed notifications by commercial notification channels is not expected. Thus, the managing system must accept structured events. If the managed system natively creates typed events, the managing system may rely on the notification channel to automatically convert them to structured events based on the OMG Notification Service's rules. Structured notifications rely upon the heavy use of CORBA "any" data types, however, which can be inefficient. Thus in this case managing system will likely prefer to accept typed notifications.

(R) NOTIF-7 The suppliers and consumers of structured events shall follow these rules for constructing and receiving the structured events. (See the figure below which depicts the Notification Structure and how elements from the IDL notification definition are to be mapped into it):

- The `domain_type` string in the fixed header of the structured event shall be set to "Telecommunications".
- The `type_name` string in the fixed header of the structured event shall be set to the scoped name of the operation defining the notification in IDL, for example, "`itut_x780::Notifications::attributeValueChange`".
- The `event_name` string in the fixed header of the structured event shall be null.
- Optional header fields may be included to support features like Quality of Service as appropriate.
- Each parameter in the operation shall be placed in a name-value pair in the filterable body portion of the structured event. The `fd_name` string of this pair shall be set to the name of the parameter and the type placed in the associated `fd_value` will be the type specified for the parameter. Using as an example the `equipmentAlarm` notification from the IDL presented later in this document, the first `fd_name` string would be set to "`eventTime`" and the first `fd_value` would contain an `ExternalTimeType` data type. Although all notification parameters go in the filterable body of the notification structure, depending on the data type of the parameter it may be difficult or even impossible to create a useful filter utilizing that parameter. Filter "matching rules" are based on the capabilities of the channel.
- Parameters that are denoted "optional" may optionally be excluded from the notification structure. If typed notifications are used, these parameters are included, but will usually have a special null value if not supported. For types for

which there is no special null value (such as integers) a special type consisting of a union between the base type (such as integer) and the null type is usually defined. These union types may be excluded from structured notifications when they have a null value, but if they are included, the union type must be used. This is to enable the same filters to be used for both structured and typed notifications.

- The remainder of the body of the structured event (the non-filterable part) shall be null.
- Parameters named "operation" shall be avoided in notification operations to potentially support the use of typed notifications. (When converting typed notifications to structured notifications, the parameters of an operation are automatically placed into a notification structure by the event channel. Unfortunately, the rules developed for doing this state that the name of the operation used to issue the notification goes not in the header of the event, but in the body of the of the structure as the first name-value pair. The `fd_name` string is set to "operation" and the `fd_value` is set to a string containing the name of the operation. Using a parameter named "operation" would then result in a second name-value pair with the name "operation," and the two could be confused.)

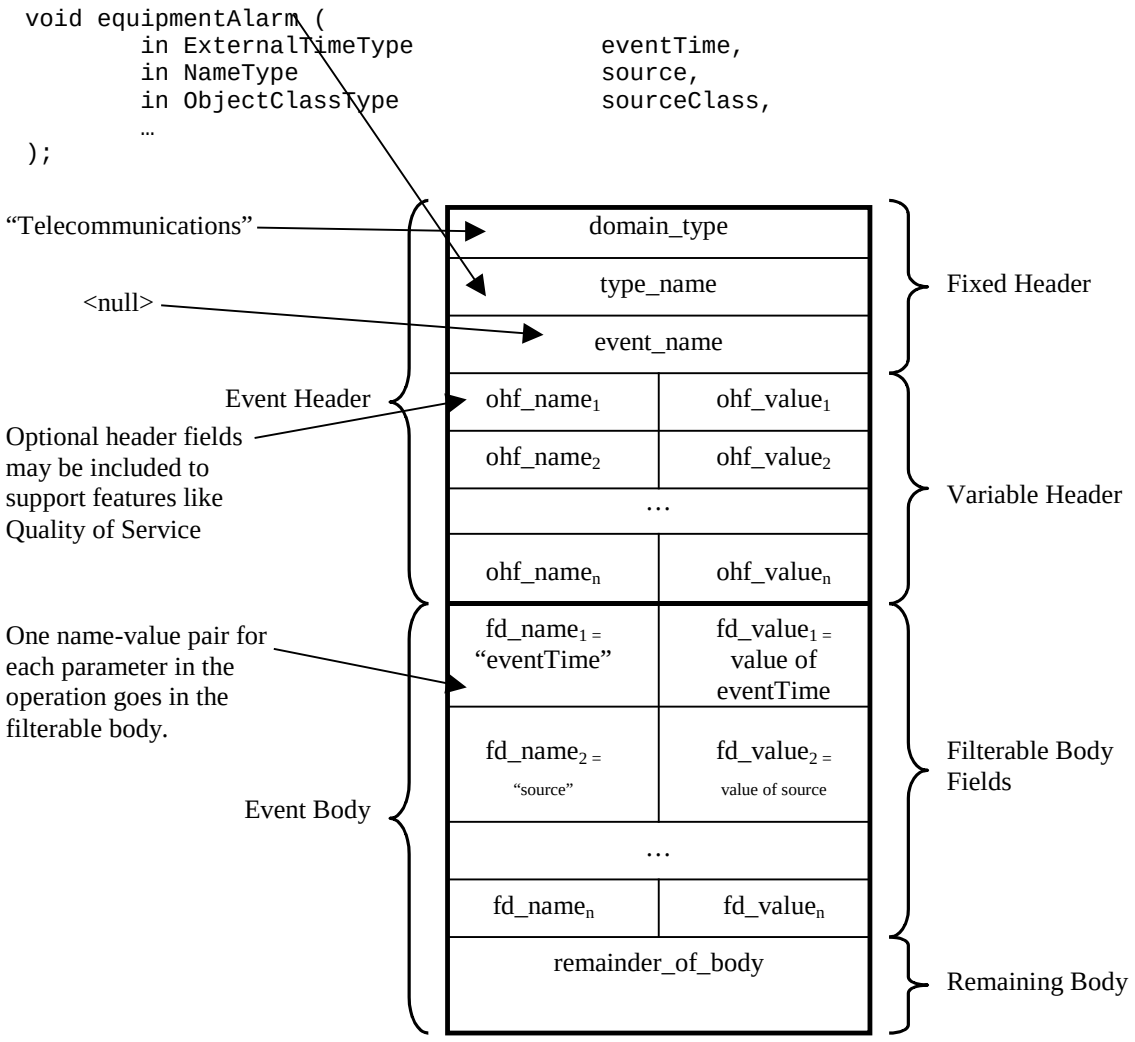

Figure 6. Mapping Notifications to Structured Events

(R) NOTIF-8 The Notification Service specification supports filter expressions that are used to determine if the event is to be forwarded. It also supports filter expressions that "map" values in the notification to parameters used to impact the operation of the event channel, such as the QoS used in delivering the event. For example, a mapping filter might be used to map a "severity=major" field from an event (which means nothing to an event channel) to a QoS parameter "priority=1" (which does mean something to the channel). The Notification Service shall support event filtering with filter objects that support constraints expressed in the default constraint grammar specified by the OMG. The Notification Service shall also support mapping filters.

(R) NOTIF-9 The Notification Service reliability QoS shall support EventReliability = Persistent & ConnectionReliability = Persistent.

Each event is guaranteed to be delivered to all consumers registered to receive it at the time the event was delivered to the channel, within expiry limits. If the connection between the channel and a consumer is lost for any reason, the

channel will persistently store any events destined for that consumer until each event time out due to expiry limits, or the consumer once again becomes available and the channel is subsequently able to deliver the events to all registered consumers. In addition, upon start from a failure the notification channel will automatically re-establish connections to all clients that were connected to it at the time the failure occurred.[4]

(R) NOTIF-10 The Notification Service order policy QoS shall allow the events to be delivered in the order of their arrival, i.e. FIFO. The Notification Service may also optionally support a priority-order QoS in which events could be buffered in priority order, such that higher priority events will be delivered before lower priority events.

(R) NOTIF-11 The Notification Service implementation deployed shall be compliant to the conformance statements of the OMG Notification Service specification with the exception of the pull interface model.

6.3 Telecom Log Service

The CORBA Telecom Log Service[5] is a CORBA-based log service that fully supports the ITU-T Recommendation X.735. The log is implemented as an Event Service or Notification Service event channel. The Log Service supports the following functionality:

- Writing to the log: Events supplied to the log are persistently stored as log records.
- Forwarding from the log: Events supplied to the log are also forwarded to other logs or to any application that wishes to receive them.
- Log generated events: The log itself will generate events.

Also the Log Service provides functions of log control and management, log record manipulation, log lifecycle management. The following figure gives a graphic representation of the Log Service.

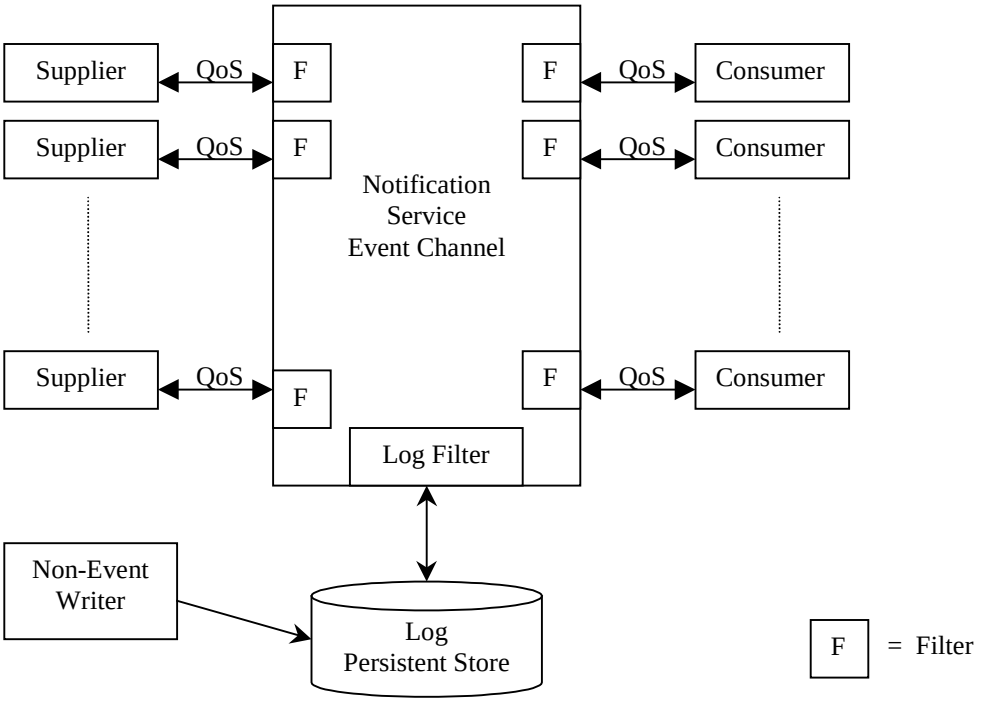

Figure 7. Telecom Log Service

By manipulating the Log Filter, a managing system is able to control which events are logged and which aren't, in exactly the same way it is able to control which events are forwarded and which aren't. The only exception is the "Non-event Writer," which is an application that writes data directly to the log.

Note that the definition of the OMG's Telecom Log pre-dates this framework. The notifications from the Telecom Log are structurally different from the other notifications in this framework even though some of the names of the notifications and parameters are semantically the same.

(R) LOG-1 The Log Service shall support all the Notification Service requirements. Log Event Channels must be registered with the Channel Finder service.

(R) LOG-2 The Log Record supported by the Log Service shall be the normal `struct LogRecord`. The support of `struct TypedLogRecord` is optional.

(R) LOG-3 The Log Service implementation shall be compliant with the conformance statement in the OMG Telecom Log Service specification with the exception of the pull interface model.

6.4 Messaging Service

The CORBA Messaging Service covers three areas: Asynchronous Method Invocation (AMI), Time Independent Invocation (TII), and Messaging Quality of Service (QoS).[8]

Of the three areas, the AMI has a significant role in the network management domain because it allows clients to make non-blocking requests on a CORBA object.

Note that CORBA is designed to enable an application to invoke a method on an object as if the object was local to the application, regardless of the object's actual location. Typically, when a method is invoked on an object in a non-distributed application, control passes to that object and the calling routine blocks until the method completes and control is passed back. These semantics are maintained in CORBA. In a distributed application, however, network latency can lead to poor performance. There are five possible solutions to consider:

1. Applications simply live with the delays.
2. TMN IDL interfaces are defined to be asynchronous. That is, management operations are always defined to return no results. Thus, invocations can be made without blocking. (This is supported by CORBA.) Results are returned later when the managed system performs a "callback" on the managing system.
3. TMN IDL interfaces always have two sets of operations, one that is asynchronous, and the other synchronous.
4. TMN IDL interfaces are defined to be synchronous, and manager applications improve performance by being multi-threaded and capable of blocking on multiple outstanding requests while continuing to process other work..
5. TMN IDL interfaces are defined to be synchronous, and manager applications improve performance by using the Asynchronous Method Invocation service, and an ORB that supports it.

This framework chooses a combination of 4 and 5. TMN IDL interfaces are defined to be synchronous. Manager applications that are multi-threaded can use these interfaces directly and experience good performance. Manager applications that cannot be multi-threaded shall use the AMI service to improve performance. Since multi-threaded managers do not need the AMI service, its use is optional.

The AMI is treated as a client side language mapping issue only. In most cases, server side implementations are not required to change. In certain situations, such as with a transactional server, the asynchrony of a client does matter and requires server side changes if it is expected to handle transactional asynchronous requests. Transactional asynchronous requests, however, will not be addressed in this document. The following figure depicts the basic concept of the OMG AMI model.

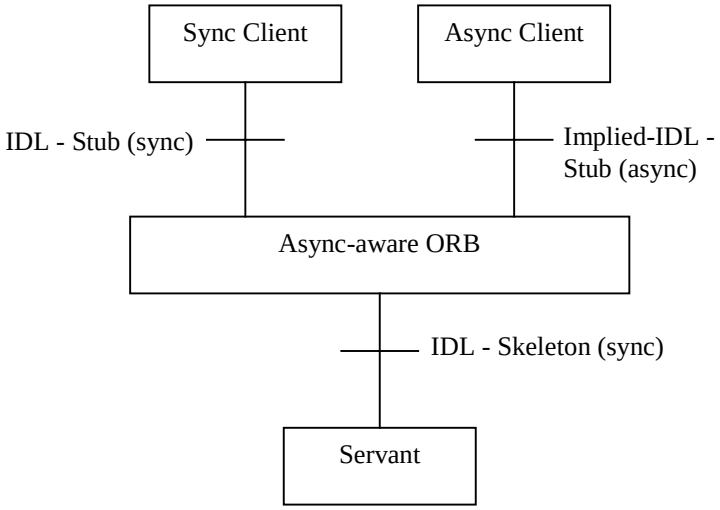

Figure 8. Asynchronous-aware ORB

The AMI specification provides two models of asynchronous requests: *callback* and *polling*. In the *callback* model, the client passes an object reference for a `ReplyHandler` object as a parameter when it invokes a two-way asynchronous operation on a server. When the server responds, the client ORB receives the response and dispatches it to the appropriate method on the `ReplyHandler` servant so the client can handle the reply. In other words, the ORB turns the response into a request on the client's `ReplyHandler`. The `ReplyHandler` is a normal CORBA object that is implemented by the programmer as with any object implementation. In the *polling* model, the client makes the request passing in all the parameters needed for the invocation, and is returned a `Poller` object which can be queried to obtain the results of the invocation. This `Poller` is an instance of a `valuetype,` which is a new IDL type introduced by the new *Objects-by-Value* specification. A `valuetype` has both data members and methods, which when invoked are just local function calls and not distributed CORBA operation invocations.

The value of the Asynchronous Method Invocation capability in network management applications is that it enables managing systems that wish to use asynchronous method calls to inter-operate with managed systems using the same interface definitions as those used by synchronous clients. No changes are required in the interface definition or the implementation of the managed system. The following requirements are proposed for implementations that optionally wish to support asynchronous method invocations (but not transactions with "ACID" capabilities).

(O) AMI-1 The AMI-aware CORBA implementation shall at least support the callback programming model.

(O) AMI-2 For each operation in an IDL interface, the AMI-aware CORBA implementation shall generate corresponding asynchronous callback method signatures.

These signatures are described in implied-IDL which is used to generate language-specific operation signatures.

(O) AMI-3 The AMI-aware CORBA ORB shall pass a type-specific `ExceptionHolder` value instance that contains the marshaled exceptions as its state to the `ReplyHandler` when exception replies are returned from the CORBA object. The AMI-aware IDL compiler would generate a type-specific `ExceptionHolder` for each IDL interface.

(O) AMI-4 The AMI-aware IDL compiler shall generate a type-specific reply handler for each IDL interface. The client will implement and register a reply handler with each asynchronous request and receive a callback when the reply is returned for that request. This reply handler is derived from the generic `Messaging::ReplyHandler`.

6.5 Security Service

The CORBA Security Service comprises the security functionality of authentication of principals (human users and objects), authorization of access to objects by principals, security auditing, communication security, non-repudiation, and administration.[6] All of this may be overkill for many applications, though. Instead, applications might require only the communication security and system-level authentication functionality based on Transport Layer Security (TLS) technology (and its precursor, SSL) for availability and simplicity reasons. Finally, some applications might require no security. The optional requirements below, therefore, reflect three possible choices:

1. No security.
2. ORBs use SSL to provide communications security and system-level authentication, which is essentially "session" security.
3. ORBs use the CORBA Security Service to provide communications security, authentication, non-repudiation, access control lists for groups or individuals accessing individual objects and operations, etc.

The actual level of service to be provided on an interface is left as a matter to be negotiated between the parties supplying the managed and managing systems.

(O) SEC-1 The CORBA interface may optionally support either the "Secure IOP protocol," or "CORBA Security SSL Interoperability," as defined in the CORBA Security Service Specification.[6]

(O) SEC-2 The CORBA Security Service may be used to support its wide range of capabilities.

(O) SEC-3 Support for the exchange of authentication certificates shall be an option left up to the administration.

6.6 Transaction Service

In a distributed computing environment such as CORBA, it is possible that updates from some clients could be overwritten by concurrent (or near concurrent) updates from other clients unless suitable safeguards are provided. Even though the Notification Service and Telecom Log Service provide a basis for making a client aware that its update has been overwritten, they do not provide a locking mechanism to prevent the occurrence of such overwrites. The OMG Transaction Service[7] provides a comprehensive locking mechanism for preventing the overwriting of one client's update by a concurrent update from a different client. This solution is designed for high reliability. However, the OMG Transaction Service may not be required in all applications and the additional overhead incurred may not be justified. If consistency of data after concurrent updates must be supported, the transaction service from the OMG should be considered.

(O) TRANS-1 The CORBA interface may optionally support the OMG Transaction Service to guarantee data consistency.

7 Framework Support Services

This section defines common support services included in the framework that are not standard OMG CORBA Common Object Services. Many network management applications perform functions that are not commonly required by general-purpose business applications, so it is not reasonable to expect the standard CORBA framework to supply all the necessary services for network management. This section defines services that will be broadly used by network management applications but are not as likely to be used by many other types of applications and are therefore unlikely to become CORBA Common Object Services. These services also provide functionality required to enable the re-use of existing information models without significant changes in semantics.

The advantages of placing this functionality in common support services is that it unburdens the managed object implementations from providing these services and allows the services to evolve to provide greater functionality without changing the managed object interfaces or implementations. The IDL describing the interfaces to these services can be found in Annex A.

7.1 The Factory Finder Service

The Factory Finder Service enables clients to find factories. A client finds a factory by querying this service with the class name of a factory. The service responds with a reference to a factory of that type. Note that the class name supplied by the client is the factory's class, not the class of the object to be created.

The Factory Finder Service is found by looking it up in the Naming Service.

Before factories can be found in the service, the service must know about them. Therefore, the Factory Finder Service also provides a means for factories to register and unregister themselves with the service. While it is not necessary to expose this capability across the management interface, these operations are defined to enable the

implementation of the Factory Finder Service as a component, separate from the objects comprising a specific information model. Thus, once implemented, the Factory Finder Service implementation will not have to be changed when new information models with new factories are defined. The Factory Finder Service could even be acquired from a third party.

The operations used to register and unregister channels are in a separate interface that is subclassed from the Factory Finder interface. Only the Factory Finder interface needs to be implemented to conform with this framework. The subclassed interface is provided for implementations that wish to use it to construct the Factory Finder service as a separate component.

This is the IDL that defines the interface to the Factory Finder Service (without comments):

```
interface FactoryFinder {

        ManagedObjectFactory find (in ObjectClassType factoryClass)
                raises (FactoryNotFound, ApplicationError);

        FactoryInfoListType list()
                raises (ApplicationError);

}; // end of FactoryFinder interface


interface FactoryFinderComponent : FactoryFinder {

        void register (in ObjectClassType factoryClass,
                in ManagedObjectFactory factoryRef)
                raises (ApplicationError);

        void unregister (in ObjectClassType factoryClass,
                in ManagedObjectFactory factoryRef)
                raises (FactoryNotFound, ApplicationError);

}; // end of FactoryFinderComponent interface
```

The *find* operation on the *FactoryFinder* interface is used by a client to find a factory of a particular type. The *list* operation returns a list of all the factories registered with the Factory Finder. The *register* operation on the *FactoryFinderComponent* interface is used by a factory to register itself with the service, and the *unregister* operation is used by a factory to delete its registration. These last two operations should not be used by managing systems.

(R) FACTORY_FINDER-1. A managed system shall instantiate at least one Factory Finder Service object. Also, each local root naming context on a system shall have at least one name binding for a Factory Finder Server Object. The value of the *ID* string in this binding shall simply identify the server, perhaps with a value similar to "FactoryFinder1". The *kind* string in the binding shall identify the class of the object ("itut_q816::FactoryFinder").

(R) FACTORY_FINDER-2. The Factory Finder server object(s) shall support the Factory Finder interface described above and defined in the CORBA IDL in Annex A. The Factory Finder server object(s) may support the Factory Finder Component interfaces defined above. The functionality described above shall be supported.

7.2 The Channel Finder Service

It is anticipated that a large network management system might have multiple event channels. These channels might handle different types of events, or they might handle events from different sets of objects. To ensure that a client does not miss any of the events in which it is interested, a means of identifying the channels present on a managed system, and the events they are handling, is needed. The Channel Finder Service performs this function. A client can invoke an operation on this service to list all of the event channels present on a managed system, along with the events they are handling. Once the client knows about the channels, it can interact with them to arrange to receive notifications.

A client finds a Channel Finder object by looking it up in the Naming Service.

Before channels can be listed by the service, the service must know about them. Therefore, the Channel Finder Service also provides a means for channels to be registered and unregistered with the service. While it is not necessary to expose this capability across the management interface, these operations are defined to enable the implementation of the Channel Finder Service as a component, separate from the objects comprising a specific information model. Thus, once implemented, the Channel Finder Service implementation will not have to be changed when new information models are defined. The Channel Finder Service could even be acquired from a third party.

The operations used to register and unregister channels are in a separate interface that is subclassed from the Channel Finder interface. Only the Channel Finder interface needs to be implemented to conform with this framework. The subclassed interface is provided for implementations that wish to use it to construct the channel finder service as a separate component.

7.2.1 Channel Finder Interface

This is the IDL that defines the interface to the Channel Finder Service (without comments):

```
interface ChannelFinder {

        ChannelInfoListType list()
                raises (ApplicationError);

}; // end of ChannelFinder interface


interface ChannelFinderComponent : ChannelFinder {
        void register (in string channelID,
                in ObjectClassType channelClass,
                in NameSetType baseObjects,
                in EventSetType eventTypes,
```

```
                    in EventSetType excludedEventTypes,
                    in ScopedNameSetType sourceClasses,
                    in ScopedNameSetType excludedSourceClasses,
                    in EventChannel channel)
                    raises (ChannelAlreadyRegistered, ApplicationError);

            void unregister (in string channelID)
                    raises (ChannelNotFound, ApplicationError);

    }; // end of ChannelFinderComponent interface
```

The *list* operation on the *ChannelFinder* interface can be used by a managing system to discover all the channels present on a managed system. The information returned is the same information included when a channel is registered, which is discussed below. The two operations on the *ChannelFinderComponent* interface are used by the managed system to register and unregister its channels with the service. These operations are not intended for use by a managing system.

When a channel is registered seven pieces of data are associated with the reference to the channel. First, a string identifier for the channel is specified. Second, the type of channel is identified (the framework has two types of channels, Notification channels and Log channels). After that, the set of objects covered by the channel is specified by the names of managed objects at the bases of trees of objects. These trees of objects are the sets of objects that send events to this channel, except that trees do not overlap. (A channel bound lower in the tree receives events from the objects "below" it. These events are not sent to the channels bound higher in the tree.) Multiple base objects may be associated with one channel. An empty set has the special meaning that all managed objects directly bound to the local root naming context that also contains this channel finder are covered by the channel.

Next, the types of events sent to this channel are identified. This is done with two sets of strings. The *eventTypes* set explicitly includes the name of each type of event sent to this channel. An example event type is "itut_x780::Notifications::equipmentAlarm". An empty set has the special meaning that all types of events are sent to this channel. The *excludedEventTypes* set lists the types of events that are not sent to this channel. If the *eventTypes* string set is null, then all events except those listed in the *excludedEventTypes* set are sent to the channel. If the *eventTypes* string set is not null, the *excludedEventTypes* parameter should be empty and is ignored.

Finally, the types of objects sending events to this channel are identified. This is also done with two sets of strings. The *sourceClasses* set explicitly includes the name of each managed object class that sends events to this channel. An example object class name is "itut_m3120::Equipment". An empty set has the special meaning that all types of managed objects send events to this channel. The *excludedSourceClasses* set lists the names of managed object classes that do not send events to this channel. If the *sourceClasses* string set is null, then all managed object classes except those listed in the *excludedSourceClasses* set send events to the channel. If the *sourceClasses* string set is not null, the *excludedSourceClasses* parameter should be empty and is ignored.

The combination of the base objects, list of event types, and source classes identifies the events handled by a channel. This can lead to some confusing combinations. While not all of these are expected to be implemented in systems, they are explained below just in case.

1. A channel may be registered with multiple base objects.
2. Multiple channels may be registered with the same base object. If the event lists and source object types associated with these channels overlap, the events must be made available at all channels listing the matching event and source object types.
3. An attempt to re-register a channel will result in an update to that channel registration. The new information will be associated with the channel and the old information will be discarded.
4. A managed object always sends an event to the channel with the associated base object that is the managed object's closest superior and that also handles that type of event from that class of object.

The figure below also graphically depicts some examples for further clarification.

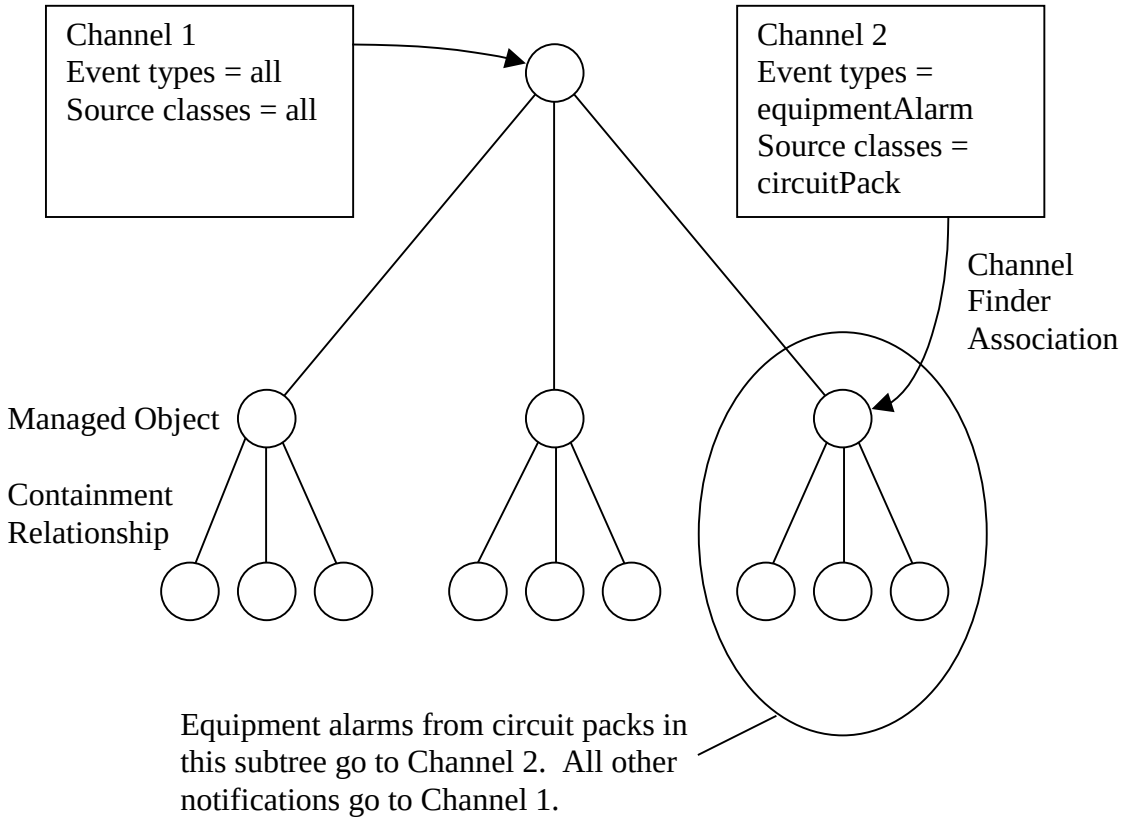

Figure 9. Event Channel Example

A special notification has been defined that must be sent whenever a channel registration is added, removed, or modified. This notification must be sent on all channels registered

immediately before the change occurred. Since this notification is emitted by the channel finder instead of a managed object, and since it is sent to all channels, the channel finder shall not list the channel change notification along with the other notifications handled by a channel. The IDL describing this notification is:

```
void channelChange (
        in ChannelModificationType      channelModification,
        in ChannelInfoType              channelInfo
);
```

The channel modification parameter indicates if a channel has been added, deleted, or modified (in terms of the event types it handles). The channel information parameter provides a string describing the type of channel, a reference to the channel, the base object(s) with which it is associated, and the source object classes and event types it handles.

It should be noted that while this approach to supporting multiple event channels is quite flexible and therefore complicated, the degree of complication is under the control of the managed system implementation. The creation and registration of event channels across the management interface is not supported. A complex managed system might support a local administrative procedure for adding, modifying, or removing channels to tune performance, or it might just update channels through software releases. A simple system will likely have one or perhaps two channels (one for high-priority events like equipment alarms and the other for everything else) associated with the root managed object. Also, please note that precluding a managing system from creating event channels does not preclude it from creating filters and "proxy suppliers" on existing channels. This gives the client capabilities equal to creating event forwarding discriminators in OSI network management systems.

7.2.2 Channel Finder Requirements

(R) CHANNEL_FINDER-1. A managed system shall instantiate at least one Channel Finder Service object. Also, each local root naming context on a system shall have at least one name binding for a Channel Finder Service object. The value of the *ID* string in this binding shall simply identify the server, perhaps with a value similar to "ChannelFinder1". The *kind* string in the binding shall identify the class of the object ("itut_q816::ChannelFinder").

(R) CHANNEL_FINDER-2. The Channel Finder server object(s) shall support the Channel Finder interface described above and defined in the CORBA IDL in Annex A. The Channel Finder server object(s) may support the Channel Finder Component interfaces defined above. The functionality described above shall be supported.

(R) CHANNEL_FINDER-3. Whenever a change to the channel registrations is made, the Channel Finder shall send a channel change notification on all channels registered immediately before the change.

(R) CHANNEL_FINDER-4. The network of event channels reported by a managed system shall handle all alarms from all managed objects on the system. A system that lists a set of channels that does not cover all events from all managed objects on the system does not comply with this framework.

7.3 The Terminator Service

The purpose of the Terminator Service is to provide a place in the framework to implement common functionality that would otherwise have to be implemented in the managed objects. The Terminator Service is used by managing systems to delete managed objects. It does so according to the delete policy of the managed object. (ITU-T Recommendation X.780 requires that every managed object have a *deletePolicy* attribute with one of three values: *notDeletable*, *deleteOnlyIfNoContainedObjects*, and *deleteContainedObjects*.) If the delete policy of the managed object is *notDeletable*, the Terminator Service does not delete the object, and instead raises an exception. If the delete policy is *deleteOnlyIfNoContainedObjects*, and the object does not contain any objects, then the Terminator Service deletes the object. Otherwise, it raises an exception. Finally, if the delete policy of the object is *deleteContainedObjects*, then the Terminator Service will delete the object as well as all of its contained objects, pursuant to some rules defined below.

ITU-T Recommendation X.780 also defines a *destroy* operation to be supported by all managed objects that is intended for use by the Terminator Service for actually deleting the managed object and releasing its resources. In addition to following the delete policies and actually deleting the managed objects, though, the Terminator Service is also a good place to implement code to maintain the integrity of the naming tree by removing name bindings for managed objects that are being deleted. Implementations may choose to implement this function elsewhere, but a goal of the framework is to enable implementations of managed objects that focus on representing network resources. It is believed that a service like this will help to make the implementation of managed objects simpler.

The IDL describing the Terminator Service provides two methods for deleting a managed object. One identifies the managed object by name, the other by reference. This is the IDL that defines the delete service interface:

```
interface TerminatorService {

 void deleteByName (in NameType name)
        raises (ApplicationError, DeleteError);

 void deleteByRef (in ManagedObject mo)
        raises (ApplicationError, DeleteError);

 }; // end of TerminatorService interface
```

(R) TERM-1. A managed system shall instantiate at least one Terminator Service object. Also, each local root naming context on a system shall have at least one name binding for a Terminator Service object. The value of the *ID* string in this binding shall

simply identify the server, perhaps with a value similar to "Terminator1". The *kind* string in the binding shall identify the class of the object ("itut_q816::TerminatorService").

(R) TERM-2. The interface supported by the Terminator Server object(s) shall be the Terminator interface described above and defined in the CORBA IDL in Annex A. The functionality described above must be supported.

(R) TERM-3. The Terminator Service shall delete objects according to the objects' delete policy attribute, which is set at creation and cannot be changed. Note that the Terminator Service is not a scoped service. The Terminator Service may actually delete multiple objects in response to a single request, but its focus is on the single object requested to be deleted. The Terminator Service shall implement the following rules when deleting an object:

1. No object shall ever be "orphaned." That is, no object may be deleted without deleting its subordinates.
2. If the object has a delete policy of *notDeletable*, the object shall not be deleted, nor are any of its subordinates if it has any. The *DeleteError* exception shall be raised with the error identifier set to the value *notDeletable*.
3. If the object has a delete policy of *deleteOnlyIfNoContainedObjects*, and it does not have any subordinates, the object shall be deleted. If the object has subordinates, regardless of their delete policies, it shall not be deleted nor shall any of its subordinates. The *DeleteError* exception shall be raised with the error identifier set to the value *containsObjects*.
4. If the object has a delete policy of *deleteContainedObjects*, and it does not have any subordinates, the object shall be deleted. If the object has subordinates, the Terminator Service shall check the delete policies of all the subordinates. If any are *notDeletable*, no objects are deleted. If any are *deleteOnlyIfNoContained* and they contain subordinates, no objects are deleted. Otherwise, the object and its subordinates are deleted.
5. The Terminator Service shall delete contained objects from the bottom up. If any contained object raises an exception during deletion, the Terminator Service shall not remove that object's name and shall not delete any of its superiors. The Terminator Service shall, however, continue trying to delete other contained objects. When all objects that can be deleted are deleted, the Terminator Service shall raise a *DeleteError* exception with the error identifier set to the value *undeletableContainedObject*. This best-effort approach to deleting contained objects is required to make the results deterministic. The client can identify the undeletable objects because they will be at the leaves of the tree remaining beneath the target object.
6. If the base object raises a *DeleteError* exception, the Terminator Service shall return the same exception (and included data). The object is not deleted and the object's name is not removed from the naming tree.

7.4 The Multiple-Object Operation Service

With potentially millions of entities to manage, there is a need for the framework to support operations on multiple objects with a single method invocation or perhaps a small number of invocations. The Multiple-Object Operation (MOO) Service provides this capability.

It is expected that each network management platform supporting a CORBA interface will provide at least one instance of the MOO Service. (For performance reasons, it is recommended that the MOO Service, Naming Service, and the managed objects be located on the same computing platform.) Managers will interact with the service using a limited number of interactions requiring relatively low bandwidth. The service will in turn interact with managed objects using either their published CORBA interfaces or some proprietary means. This high number of interactions is expected to require higher bandwidth, thus the need to co-locate the service with the managed objects.

Note that the MOO service is an example of application-specific access granularity discussed earlier in the introduction.

7.4.1 The MOO Service Interface

The MOO Service's interface, defined in Annex A, is weakly-typed. It provides a set of generic capabilities that may be invoked on sets of any kinds of managed objects, even objects of different types. The operations supported are:

- Scoped get: Returns the values from each of the objects for a list of attributes.
- Scoped update: Used to replace an attribute value or to add or remove values to/from set-valued attributes. May be used to update one or multiple attributes in a single object or multiple objects.
- Scoped delete: Deletes multiple objects.

The scoped get operation is defined on the *BasicMooService* interface. The scoped update and delete operations are defined on the *AdvancedMooService* interface, which inherits the scoped get operation from the basic service interface. This was done to allow for some flexibility in the implementation of multiple-object operation services. A basic service need only implement the scoped get operation.

Each of the scoped operations requires four parameters to define the set of objects on which the operation will be performed:

- Base object name: The name of the object at the root of a tree of objects on which the operation will potentially be performed.
- Scope: A discriminated union identifying the objects contained under the base object on which the operation potentially will be performed. The union has four cases. Two of the cases include an integer specifying a level of objects contained below the base object. The four choices are:
  - Base Object Only. If the scope is *baseObjectOnly* then only the named target (base) object is included in the scope.

- Whole Subtree. If the scope is *wholeSubtree* the scope is all of the objects contained below the base object, along with the base object.
- Individual Level. If the scope is *individualLevel*, the scope will also include an integer-valued *level*. All of the objects contained at a level below the base object equal to this value are in the scope. The objects directly contained by the base object are level one. If *level* equals zero, the scope is the base object.
- Base to Level. If the scope is *baseToLevel*, the scope will also include an integer-valued *level*. The scope will be all of the objects down to the given level, including the base object and the object at the given level. If *level* equals zero, the scope is the base object only.

Note that because this framework uses some unique naming conventions, the service has to do a little work to determine the actual depth for containment-based scopes. The goal is to make it as simple as possible for the client. First, the base object name will be the entire compound name including the final component with an ID value of "Object". The service will have to "back up" to the naming context that contains this binding and start counting from there. Also, the service shall automatically follow the "Object" bindings in the managed object naming contexts within the scope. This last hop will not count towards the depth.

- Filter: An expression written in a constraint language that is used to evaluate the attributes of an object. The operation is applied to those objects within the scope for which the filter expression evaluates to *true*.
- Language: a string indicating the language in which the filter expression is written.

The object names are the same as the *Name* type defined by the CORBA Naming Service. The scope is a signed short integer with values as described above. Finally, the filter and language parameters are strings.

Each of the operations may raise one of these exceptions:

- an *InvalidParameter* exception if one of the parameters has an invalid value. The name of the invalid parameter is returned. Here are some conditions under which this exception would be raised:
  - The base object name is not valid.
  - An unrecognized filter language value is supplied
- an *InvalidFilter* exception if the syntax of the filter is incorrect. This exception returns the text near the syntax error for trouble-shooting purposes.
- a *FilterComplexityLimit* exception if the syntax of the filter is correct, but the filter is too complex to be processed by the managed system.
- an *ApplicationError* exception to relate other problems on the server (such as a lack of resources) that make it impossible to carry out the requested operation.

Note that if an expression cannot be evaluated for a particular object because the types of its attributes do not match the expression, the filter is not invalid. That object may simply fail to pass the filter.

The other parameters for the operations as well as the return types are specific to the operation. For example, the scoped get operation takes a list of attribute names and returns a sequence of results, one from each object.

The object's name is associated with the results from that object. Because each of the operations could potentially return large amounts of data, the iterator design pattern is used for returning the results. An iterator is an object that is created to contain the results of an operation so that they may be returned to the client at a rate determined by the client. The client receives a reference to the iterator as part of the information returned by the method. The client may then invoke operations on the iterator to receive batches of results in sizes determined by the client. The iterator keeps track of how far through the results the client has progressed.

Note that the iterators are used to pace the return of information from the operations only, and should not control when the operations are actually invoked on individual objects. A scoped operation should be invoked on the objects and the results queued as soon as possible. Delaying the invocation of the operation on the individual managed objects until the results are requested through the iterator may be more efficient, but could lead to incorrect results or race conditions. Also, if the client requests more results than are currently available, the iterator must wait until it can return the requested number or until all results are ready. This is because in this frequently-used design pattern the client assumes it has retrieved all of the results when it gets fewer than requested. If a batch takes too long the client may reduce the batch size on subsequent requests.

The following sub-sections give additional details on each of the scoped operations.

7.4.1.1 Scoped Get

The IDL signature for the scoped get operation on the basic MOO service is:

```
GetResultsSetType scopedGet (
        in NameType baseName,
        in ScopeType scope,
        in FilterType filter,
        in LangaugeType language,
        in StringSetType attributes,
        in unsigned short howMany,
        out GetResultsIterator resultsIterator)
        raises (InvalidParameter,
                InvalidFilter,
                FilterComplexityLimit,
                ApplicationError);
```

As described above, the first four parameters are used to select a set of object on which to perform the get operation. For each of these the service will try to return a value for each of the attributes named in the "attributes" parameter, which is just a list of strings. A submitted null attribute list, however, has the special meaning that *all* attribute values for

the objects that pass the filter should be returned. The types involved in the return value are:

```
struct AttributeValueType {     // from itut_x780 framework
        string  attributeName;
        any     value;          // type will depend on the attribute
};

typedef sequence <AttributeValueType> AttributeSetType; // framework

struct GetResultsType {
        NameType                name;
        boolean                 notFilterable;
        AttributeSetType        attributes;
        StringSetType           failedAttributes;
};

typedef sequence <GetResultsType> GetResultsSetType;
```

The first two types form a *name-value pair* list. The return type is a sequence of structures, one for each object that passed the filter. In that structure is an object's name, a flag that will be true if the object could not be evaluated to see if it passed the filter, the list of attribute values from that object, and the names of any attributes that could not be retrieved from that object. Objects that could not be filtered are flagged as a special case because they may be objects in which the client was not even interested. If the object could not be filtered, the client will know the MOO server could not retrieve any attributes for that object, so the other two data members shall be empty. If an object passes the filter but an attribute value could not be retrieved either because the object did not have a matching attribute or some exception was raised on access, that attribute's name should be put on the failed attribute list for that object.

The *howMany* parameter indicates to the service how many objects' results should be included in the first batch of responses. (Zero is allowed, forcing all results to be returned through the iterator.) The *resultsIterator* output parameter is a reference to an iterator object that may be used to retrieve additional results in batches. If all the results were returned by the *scopedGet* operation, this reference will be null. The client must destroy this object when it is finished with it, and may do so before all the results are retrieved. The functionality of the CMIP Cancel Get operation is provided in this framework by destroying the results iterator.

7.4.1.2 Scoped Update

The IDL signature for the scoped update operation on the advanced MOO service is:

```
UpdateResultsSetType scopedUpdate (
        in NameType baseName,
        in ScopeType scope,
        in FilterType filter,
        in LanguageType language,
        in ModificationSeqType modifications,
        in boolean failuresOnly,
        in unsigned short howMany,
        out UpdateResultsIterator resultsIterator)
        raises (InvalidParameter,
```

```
                    InvalidFilter,
                    filterCoplexityLimit,
                    ApplicationError);
```

Again, the first four parameters are used to select the set of objects on which the update is performed. The modifications list is a list of structures, each with the name of an attribute, a value for that attribute, and an enumerated value indicating if the value should replace the attribute's current value, be added to the attribute's current value, or removed from it. The add and remove options are valid only if the attribute's type is a CORBA sequence and if the interface has add and remove operations for the attribute. The values in the modification list structures are passed across as CORBA *any* types. If the attribute's type is a CORBA sequence, a sequence of the proper type should be put in the *any* field, even if it contains only a single value. The IDL describing the modification list is:

```
enum ModificationOpType {set, add, remove};

struct ModificationType {
        string                  attribute;
        ModificationOpType      op;
        any                     value;
};

typedef sequence <ModificationType> ModificationSeqType;
```

The *failuresOnly* flag is used to indicate if the client wants the service to return results for all objects meeting the scope and filter, or just those objects for which at least one of the modifications could not be performed even though the scope and filter are satisfied.

The return value is a list of structures, each containing an object's name along with a Boolean value indicating if the object could not be evaluated to see if it passed the filter and a list of any attributes that could not be modified. If the service cannot interact with the object to determine if it passes the filter, the results for that object will have the *notFilterable* set to true and the *failedAttributes* data member will be empty. (This is flagged as a special case because the object may be one in which the client was not even interested.) The service will try to perform all the modifications in the list, in order, continuing to try the rest even if one modification fails. If any operation fails on an attribute, that attribute's name is added to the list of failures. If the *notFilterable* flag is false, and the *failedAttributes* data member is empty, the client will know all updates were performed on that object. The new types involved in the return value are:

```
struct UpdateResultsType {
        NameType        name;
        boolean         notFilterable;
        StringSetType   failedAttributes;
};

typedef sequence <UpdateResultsType> UpdateResultsSetType;
```

The *howMany* parameter indicates to the service how many objects' results should be included in the first batch of responses. (Zero is allowed, forcing all results to be

returned through the iterator.) The *resultsIterator* output parameter is a reference to an iterator object that may be used to retrieve additional results in batches. If all the results were returned by the *scopedUpdate* operation, this reference will be null. The client must destroy this object when it is finished with it, and may do so before all the results are retrieved.

7.4.1.3 Scoped Delete

The IDL signature for the scoped delete operation on the advanced MOO service is:

```
DeleteResultsSetType scopedDelete (
        in NameType baseName,
        in ScopeType scope,
        in FilterType filter,
        in LanguageType language,
        in boolean failuresOnly,
        in unsigned short howMany,
        out DeleteResultsIterator resultsIterator)
        raises (InvalidParameter,
                InvalidFilter,
                FilterComplexityLimit,
                ApplicationError);
```

Rather than accessing attribute values, this operation simply attempts to delete each object in the scope that passes the filter.

The *failuresOnly* flag is used to indicate if the client wants the service to return results for all objects meeting the scope and filter, or just those objects that could not be deleted. Because object deletion notification are typically sent, clients may often want to choose to receive results for only those objects that could not be deleted.

The return value lists the name of each object along with two flags that might indicate that the object could not be deleted. The *notFilterable* flag shall be true if the MOO service could interact with the object to even determine if it passed the filter. The *notDeletable* flag shall be true if the object passed the filter, but could not be deleted, either due to its delete policy, or because it raised an exception. An object that cannot be evaluated against the filter is flagged as a special case to let the client know it may be an object that it did not even intend for deletion.

```
struct DeleteResultsType {
        NameType                name;
        boolean                 notFilterable;
        boolean                 notDeletable;
};

typedef sequence <DeleteResultsType> DeleteResultsSetType;
```

The *howMany* parameter indicates to the service how many objects' results should be included in the first batch of responses. (Zero is allowed, forcing all results to be returned through the iterator.) The *resultsIterator* output parameter is a reference to an iterator object that may be used to retrieve additional results in batches. If all the results were returned by the *scopedDelete* operation, this reference will be null. The client must

destroy this object when it is finished with it, and may do so before all the results are retrieved.

Because many objects cannot be deleted if they contain other objects, for scopes based on containment relationships the service must begin deleting the "leaf" objects that are within scope and work toward the "root" object. When deleting objects, the MOO service must follow the rules for deleting an object based on the object's delete policy as described in Section 7.3. Because the rules are being applied to each of the objects in the scope, starting from the bottom up, however, the effect will be different than simply trying to delete the object at the root of a sub-tree. Also, the MOO service is best-effort. Therefore, it is possible for some of the objects in a scoped sub-tree to be deleted while others aren't. These are the rules that must be applied to scoped delete operations:

1. No objects may be "orphaned." That is, an object may not be deleted without deleting all of its contained (child) objects.
2. Performing a scoped delete on an entire sub-tree results in all of the objects in that sub-tree being deleted unless an object has a delete policy of *notDeletable*, the object raised an exception on the destroy operation, or an object has a subordinate that is not deletable.
3. Performing a scoped delete on part of a sub-tree requires evaluating each of the objects at the lowest scoped layer using the delete rules in Section 7.3. If an object at the lowest-layer of the scope may be deleted according to these rules, it and any subordinates are deleted. If a lowest-layer object cannot be deleted, it is not deleted nor are any of its superior objects. Other objects in the scope may be deleted, however, if the delete rules allow it. The service then moves up to the next layer, and so on.

7.4.2 The Default Filter Language

This section describes the default filtering constraint language that must be supported by all conformant implementations of the MOO Service. Conformant implementations may support other constraint grammars in addition to the grammar described here. An operation is provided on the Basic MOO Service interface to enable a client to retrieve the languages supported by a service. The grammar used in a request is indicated by a string-valued parameter named "language" on each scoped operation. A value of "MOO 1.0" (one space between "MOO" and "1.0") shall indicate the grammar described here. (A constant named *defaultLanguage* with this value is provided in the IDL module.)

The default grammar supported by each conformant implementation shall be the default constraint grammar defined for version 1.0 of the Notification Service[4] with changes as described in the following sub-sections. Note that by taking this approach, the framework fixes the support for comparison rules (or "matching rules") with the filter grammar. New rules (e.g., a case-insensitive string match) cannot be added with the addition of a new data type or attribute type. Instead, the grammar will have to be updated if new capabilities are required.

7.4.2.1 Applying the Constraint Language to Object Attributes

The default Notification Service constraint grammar introduced the special token '$' to denote the current event and run-time variables. For multiple-object operations, the '$' token shall denote the "current" object as well as run-time variables. That is, one can think of the MOO Service as selecting a set of objects based on the supplied base name and scope parameter, then applying the filter expression individually to each of the objects in that set. The "current" object is the object against which the expression is being evaluated. The following examples illustrate the use of the '$' token:

$.administrativeState	The administrative state attribute of the current object.
$curtime	A "built-in" variable named "curtime".

The identifiers that come after the "$." (dollar-sign period) are names of the attributes of the current object as found in the value object defined to return the attributes of an object. (See ITU-T X.780 for details on managed object attributes.) That is, "$.administrativeState" refers to the member named "administrativeState" in the value type returned by a call to the *getAttributes*() operation on the current object. (It is assumed that many implementations of the MOO Service will use the *getAttributes*() operation to retrieve the attributes from an object before evaluating the filter.)

The Notification Service constraint language, on which the MOO Service constraint language is built, has a "dot" operator (".") that can be used to access the individual members within a data structure, and data structures can be nested. Thus, an identifier like this one may be used to access a value within an attribute that has a data structure value:

$.systemTimingSource.primaryTimingSource

A comparison with an attribute name that is not present in an object always evaluates to false. To illustrate, in the expression "(A == 0) || (A != 0)" if there is no attribute named "A" present in the object both comparisons will evaluate to false and the expression will actually evaluate to false. The default Notification Service language does support an "exists" operation that can be used to test the existence of an attribute before including it in a comparison. Also, a comparison always evaluates to false if the types of the operands do not match. In the example above, if "A" is a string, the expression will be false.

Notice that the default Notification Service constraint grammar defines a set of runtime variables (which may be better thought of as "built-in" or "pre-defined" variables) but does not allow user-defined variables in filter expressions. In fact, there is no assignment operator that would support the use of user-defined variables. There are currently no built-in variables defined for the Scoping and Filtering Service and user-defined variables are not supported.

NOTE - Since the Notification Service evaluates objects based on the names of their attributes, care must be taken when defining attribute names (the names of the members

of the attribute value object defined for an interface). An attribute of type *AdministrativeStateType* named "adminState" with a value of "unlocked" will fail a filter of "administrativeState == unlocked" because the name does not match.

7.4.2.2 Support for Regular Expressions

The default Notification Service constraint language defines a substring operator to work like this: "A ~ B" is true if A is a substring of B. The default MOO Service constraint language extends this to allow A to be a regular expression. That is, "A ~ B" evaluates to true if A is a substring of B or if the regular expression defined in A is matched in B. For this framework, regular expressions are "modern" regular expressions as defined in Section 2.8 of POSIX 1003.2.[11]

A regular expression is a pattern that describes a set of strings. The inclusion of special characters known as "meta-characters" enables one string to describe a set of strings. The manual page for the "grep" command on most POSIX-compliant systems gives a complete description of regular expressions and their use.

Regular expression matching is added to the constraint language to satisfy the requirement to match sub-strings at the beginning, middle, or end of a string. POSIX regular expressions support this capability by using meta-characters that represent the beginning or end of a string("^" and "$"). Matching in the middle of a string is done by excluding these characters from the regular expression. Certainly, this requirement could have been met by only adding a couple of meta-characters to the string matching function. It was felt, however, that since regular expression matching is supported as a utility on POSIX-compliant systems, it made sense to go ahead and use this capability, which adds rich pattern matching to the language, rather than to require developers to implement a special capability offering far less functionality.

7.4.2.3 Support for Testing Set-valued and Sequence-Valued Attributes

Network management applications tend to rely heavily on the attributes of the managed objects, and often these attributes are actually sets or sequences of values. (Sets and sequences differ. Sets should not contain duplicates and the order of the elements is unimportant. In sequences, duplicate elements are allowed and order is important.) To support the use of set-valued and sequence-valued attributes in filter expressions, the default Notification Service constraint language needs to be extended. Two groups of extensions are required to support the use of sets and sequences. The first enables sets and sequences of literal values to be included in filter expression. The second defines operators for sets and sequences.

7.4.2.3.1 Sets and Sequences of Literal Values

Sets and sequences of literal values are included in filter expressions by enclosing a comma-separated list of literal values in curly braces. For example:

{1, 16, 21}	A set or sequence of integers
{5.2, 6.8, 7.01}	A set or sequence of floating-point numbers
{'apple', 'orange'}	A set or sequence of strings

{Critical, Major, Minor}	A set or sequence of enumerated values
{}	A null set or sequence.

The literal values must be of the "simple" types defined for the Notifications Service constraint language (Boolean, short, unsigned short, long, unsigned long, float, double, char, wchar, string, wstring), or enumerated values. All values in a set or sequence must be of the same type.

Obviously, in this constraint language, literal sets and sequences are defined in the same way. Actually, this matches the case with CORBA interface attribute types. Unlike some other interface syntax languages, OMG IDL has only a sequence structure, and no set type. To account for this, different operations for sets and sequences are defined. When a sequence operator is applied to a pair of sequences (either literal or attribute values), the sequences are treated as true sequences. That is, order is taken into account. When two sequences are involved in a set operation, however, the sequences are actually treated as sets. That is, the order of the values in the set does not matter. Also, while managed objects should never return duplicates in the value of a set-valued attribute, any duplicates should be ignored.

7.4.2.3.2 Set Operators

In order to include set-valued attributes in filter expressions, operators that work on sets are needed. This section extends the Notification Service constraint language by defining how the operators already defined for that service are to be applied to sets. One new operator, using the caret symbol ('^'), is defined for testing the intersection of two sets. Also, two built-in functions that take sets as arguments are defined.

Note that the default Notification Service constraint language already defines one operator that works on sets, the "in" operator. The expression "A in B" can only be applied if A is a simple type as defined above and B is a sequence of the same simple type. The expression evaluates to true if the value represented by A is equal to a value in B. Also, the default Notification Service constraint language supports the use of the "exist" operation on set-valued parameters. This behavior will also be supported for multiple object operations.

In general, to use any of the set operators in an expression such as "A <operator> B" one or both operands must be a sequence of one of the types listed above in the section on sets of literal values. If one operand is a sequence of type X, the other must either be a sequence of type X or a value of type X. Because one or both of the operands are actually sequences, not sets, the operations must ignore any duplicate values within a sequence and must not depend on any order of the values in a sequence. The operators extended to work on set-valued attributes are defined below:

A == B	True if all the values in each operand are present in the other.
A != B	False if all the values in each operand are present in the other.
A < B	True if all the values in A are in B and B contains at least one other value not in A.

A <= B	True if all the values in A are in B. (If A is a singly-valued attribute this is the same as "A in B".)
A > B	True if all the values in B are in A and A contains at least one other value not in B.
A >= B	True if all the values in B are in A.
A ^ B	True if any value in A is present in B (the intersection is not null).

In addition to these operations, two built-in functions that take a set as an argument and return a single value from that set are defined:

MAX(<set of values>)	Returns the highest value in the set.
MIN(<set of values>)	Returns the lowest value in the set.

If no maximum or minimum can be derived from the set (because the values are not numeric) the returned value should be indeterminate and any comparison to this indeterminate value should evaluate to false.

7.4.2.3.3 Sequence Operators

To support the inclusion of sequence-valued attributes in filter expressions, operators that work on sequences are needed. This section extends the Notification Service constraint language by defining operators that work on sequences.

Only a pair of operators are defined for sequences, since the only requirement was to do equality matching on sequences. The operators defined to work on sequence-valued operators are:

A % B	True if A and B have the same number of values and all the values in A match those in B, in order.
A !% B	False if A and B have the same number of values and all the values in A match those in B, in order.

7.4.3 MOO Service Requirements

This section summarizes the Multiple Object Operation Service requirements.

(R) MOO-1. A managed system shall instantiate at least one MOO Server object. Also, each local root naming context on a system shall have at least one name binding for a MOO Service object. The value of the *ID* string in this binding shall simply identify the server, perhaps with a value similar to "MOO1". The *kind* string in the binding shall identify the class of the object ("itut_q816::BasicMooService" or a sub-class).

(R) MOO-2. The interface supported by the MOO Server object(s) shall be the "Basic" MOO Service interface described above and defined in the CORBA IDL in Annex A.

(O) MOO-3. Optionally, the interface supported by the MOO Server object(s) may be the "Advanced" MOO Service interface described above and defined in the CORBA IDL in Annex A.

(R) MOO-4. The MOO Server object(s) shall at least support the default constraint language defined above for the specification of filters, and may support other grammars. The default constraint language, identified as "MOO 1.0", is the default constraint language defined by the CORBA Notification Service but extended as described above to support:

- Filtering on object attribute values rather than notification structure member values.
- Regular expression matching.
- Filtering on attributes containing sets or sequences of values.

7.5 The Heartbeat Service

The Heartbeat Service is used to verify the operation of the notification channels on a managed system, as well as the communications network between the managed system and managing system. It periodically sends a small notification to a managing system interested in receiving it that identifies the system that emitted the heartbeat, as well as the notification channel through which it was emitted. After configuring this service, a managing system can then set a filter for heartbeat notifications on any of the channels it is interested in assuring are functioning. Since these notifications flow through the same channels, software, and networks as notifications from other resources, they periodically verify the operation of these resources.

The Heartbeat Service is found by looking it up in the Naming Service.

The following IDL (without comments) describes the Heartbeat Service interface:

```
interface Heartbeat {

        attribute string systemLabel;

        unsigned short periodGet();

        void periodSet(in unsigned short period);

}; // end of Heartbeat interface

interface Notifications {
…
        void heartbeat (
                in string               systemLabel,
                in string               channelID,
                in unsigned short       period,
                in UtcT                 timeStamp
        );
…
}; // end of Notifications interface
```

As can be seen, the Heartbeat service has an attribute named *systemLabel*, and operations to set and get the period between heartbeats. *SystemLabel* is a user-supplied identifier.

The intended use is to allow a managing system to insert a label to identify the system providing the heartbeat.

The Heartbeat service periodically emits a notification on each event channel that it can find in the Channel Finder Service. The Channel Finder Service provides a listing of each channel on the system, with a channel ID for each, as well as additional information on the use of the channel. (The Channel Finder shall not list the Heartbeat notification as one of the notifications it handles. Heartbeat notifications are not sent by managed objects, and are sent to all channels.) At the end of each period, the Heartbeat Service sends a notification on each of the channels listed. The notification sent to each channel includes the channel ID of that channel.

The period between heartbeats is controlled using the *periodSet* operation. The value submitted to this operation is the period, in seconds, that the Heartbeat Service waits between emitting notifications. Updating the period causes the service to immediately emit a notification with the new period value, and begin a new period. Setting the period to zero causes the service to emit one final notification with a period value of zero, then no more (until the period is reset).

Each notification includes the value of the *systemLabel* attribute, the ID of the channel through which the notification was sent, the current value for the period, and a timestamp.

(R) HEARTBEAT-1. A managed system may instantiate at least one Heartbeat Service object. If the Heartbeat Service is supported, each local root naming context on a system shall have at least one name binding for a Heartbeat Service Object. The value of the *ID* string in this binding shall simply identify the server, with a value similar to "Heartbeat1". The *kind* string in the binding shall identify the class of the object ("itut_q816::Heartbeat").

(R) HEARTBEAT-2. The Heartbeat server object(s) shall support the Heartbeat interface described above and defined in the CORBA IDL in Annex A. The functionality described above shall be supported.

(R) HEARTBEAT-3. Updating of the period shall cause the service to deliver a notification to all channels with the new period value and then begin a new period. Setting the period to zero shall cause the service to emit one final notification with a period value of zero, then no more (until the period is reset).

(R) HEARTBEAT-4. Until the period is changed, the heartbeat notifications shall be sent to all the channels once within each period. The time between heartbeat notifications being sent to a channel shall never be greater than twice the period.

7.6 Other Support Services

This framework anticipates the need for other network management support services but recognizes it is impractical to make them all part of one framework document. Exactly

where the line gets drawn is a bit arbitrary, though. Because of its focus on TMN and the need to support existing information models, this framework includes services that equate to those provided by the CMIP protocol and the most basic TMN management information capabilities. Just as with CMIP, it is expected that additional support services will be defined, most likely in separate documents.

8 Compliance and Conformance

This section defines the criteria that must be met by other standards documents claiming compliance to this framework and the functions that must be implemented by systems claiming conformance to this specification.

8.1 System Conformance

8.1.1 Conformance Points

This section summarizes the individual functions described earlier in this document. These conformance points are then combined in profiles that must be supported by systems claiming conformance to this specification.

1. An implementation claiming conformance to the Naming Service requirements must:
   - Support the CORBA Naming Service version specified in Section 5.2.
   - Support all of the Naming Service requirements specified in Section 6.1.
2. An implementation claiming conformance to the Notification Service requirements must:
   - Support the CORBA Notification Service version specified in Section 5.2.
   - Support all of the Notification Service requirements specified in Section 6.2.
3. An implementation claiming conformance to the Telecom Logging Service requirements must:
   - Support the CORBA Telecom Logging Service version specified in Section 5.2.
   - Support all of the Logging Service requirements specified in Section 6.3.
4. An implementation claiming conformance to the Security Service requirements must:
   - Support the Security Service version specified in Section 5.2.
   - Support all of the Security Service requirements specified in Section 6.5.
5. An implementation claiming conformance to the Transaction Service requirements must:
   - Support the CORBA Transaction Service version specified in Section 5.2.
   - Support the Transaction Service requirements specified in Section 6.6.
6. An implementation claiming conformance to the Factory Finder Service must:
   - Support the Factory Finder service interface described in Section 7.1 and defined in the CORBA IDL in Annex A.
7. An implementation claiming conformance to the Channel Finder Service must:
   - Support the Channel Finder service interface described in Section 7.2 and defined in the CORBA IDL in Annex A.
8. An implementation claiming conformance to the Terminator Service must:

- Support the Terminator Service interface described in Section 7.3 and defined by the CORBA IDL in Annex A.

9. An implementation claiming conformance to the Basic MOO Service must:
   - Support the mandatory MOO service requirements described in Section 7.4.3.
10. An implementation claiming conformance to the Advanced MOO Service must:
    - Support the mandatory and optional MOO service requirements described in Section 7.4.3.
11. An implementation claiming conformance to the Heartbeat Service must:
    - Support the Heartbeat Service interface described in Section 7.5 and defined in the CORBA IDL in Annex A.

8.1.2 Basic Conformance Profile

A system claiming conformance to the ITU.Q.816 Basic Profile shall support:

1. The version of CORBA specified in Section 5.2.
2. The Naming Service requirements. (See conformance point 1.)
3. The Notification Service requirements. (See conformance point 2.)
4. The Factory Finder Service (See conformance point 6.)
5. The Channel Finder Service (See conformance point 7.)
6. The Terminator Service. (See conformance point 8.)
7. The Basic MOO Service. (See conformance point 9.)

8.2 Conformance Statement Guidelines

The users of this framework must be careful when writing conformance statements. Because IDL modules are being used as name spaces, they may, as allowed by OMG IDL rules, be split across files. Thus, when a module is extended its name won't change. Instead, a new IDL file will simply be added. Simply stating the name of a module in a conformance statement, therefore, will not suffice to identify a set of IDL interfaces. The conformance statement must identify a document and year of publication to make sure the right version of IDL is identified

Annex A Framework Support Services IDL

(Normative)

```
/* This IDL code is intended to be stored in a file named "itut_q816.idl"
located in the search path used by IDL compilers on your system. */

#ifndef ITUT_Q816_IDL
#define ITUT_Q816_IDL

#include <CosNotifyChannelAdmin.idl>
#include <CosTime.idl>
#include <itut_x780.idl>

#pragma prefix "itu.int"

module itut_q816 {

        // Types imported from CosNotifyChannelAdmin
        typedef CosNotifyChannelAdmin::EventChannel EventChannel;

        // Types imported from CosTime
        typedef TimeBase::UtcT UtcT;

        // Types imported from itut_x780
        typedef itut_x780::AttributeSetType AttributeSetType;
        typedef itut_x780::ManagedObject ManagedObject;
        typedef itut_x780::ManagedObjectFactory ManagedObjectFactory;
        typedef itut_x780::NameType NameType;
        typedef itut_x780::NameSetType NameSetType;
        typedef itut_x780::ObjectClassType ObjectClassType;
        typedef itut_x780::ObjectClassSetType ObjectClassSetType;
        typedef itut_x780::ScopedNameType ScopedNameType;
        typedef itut_x780::ScopedNameSetType ScopedNameSetType;
        typedef itut_x780::StringSetType StringSetType;

// Exceptions imported from itut_x780 (exceptions can't be typedeffed)
#define ApplicationError itut_x780::ApplicationError
#define DeleteError itut_x780::DeleteError
```

// Data Types and Structures

```
        /** EventSetType is a list of event types.  It is actually just a list
        of strings.  The values of the strings are the names of the event types
        (the strings that go in the "type_name" field of the structured event),
        which are the same as the scoped names of the operation defined on the
        Notifications interfaces to send the events.  For example:
        itut_x780::Notifications::objectCreation  */

        typedef sequence <ScopedNameType> EventSetType;

        /** A channel info structure contains information about an event
        channel.
        @member channelID       A string identifier for the channel.
        @member channelClass    the channel's scoped class name.
        @member baseObjects     The objects at the bases of the trees of
                                managed objects sending events to this channel.
                                A null list indicates that all base managed
                                objects on the system are covered by this
                                channel.
```

```
        @member eventTypes      The list of event types handled by this
                                channel.  A null list indicates all event types
                                are handled by this channel.
        @member excludedEventTypes If the eventTypes parameter is null, this
                                can be used to exclude event types.  If
                                eventTypes is not null, this should be null
                                and is ignored.
        @member sourceClasses   The list of types of objects that send events
                                to this channel.  A null list indicates all
                                types of managed objects send events to this
                                channel.
        @member excludedSourceClasses If the sourceClsses parameter is null,
                                this can be used to exclude source classes.
                                If sourceClasses is not null, this should be
                                null and is ignored.
        @member channel         a reference to the channel.
        */

        struct ChannelInfoType {
                string                  channelID;
                ObjectClassType         channelClass;
                NameSetType             baseObjects;
                EventSetType            eventTypes;
                EventSetType            excludedEventTypes;
                ObjectClassSetType      sourceClasses;
                ObjectClassSetType      excludedSourceClasses;
                EventChannel            channel;
        };

        /** A channel info set contains a list of channel references and the
        data associated with them.  */

        typedef sequence <ChannelInfoType> ChannelInfoSetType;

        /** Channel Modification indicates the type of event channel
        modification.  */

        enum ChannelModificationType {ChannelCreate, ChannelDelete,
                                      ChannelUpdate};

        /** The DeleteResultsType holds, for a single object, the results
        of a scoped delete operation.  If both boolean flags in the result
        are false, the object was deleted.
        @member name            The name of the object to which these results
                                apply.
        @member notFilterable   This flag will be true if the service could not
                                interact with the object to see if it even
                                passed the filter.
        @member notDeletable    This flag will be true if the object could not
                                be deleted due to its delete policy or because
                                it raised an exception.
        */

        struct DeleteResultsType {
                NameType                name;
                boolean                 notFilterable;
                boolean                 notDeletable;
        };

        /** The DeleteResultsSetType is a set of results returned by the
        scoped delete operation.  */

        typedef sequence <DeleteResultsType> DeleteResultsSetType;
```

```
/** A factory info structure contains information about a managed
object factory.
@member factoryClass    the factory's scoped class name
@member factoryRef      a reference to the factory
*/

struct FactoryInfoType {
        ObjectClassType         factoryClass;
        ManagedObjectFactory    factoryRef;
};

/** A factory info set contains a list of factory references and
their class names.  */

typedef sequence <FactoryInfoType> FactoryInfoSetType;

/** A Filter Type parameter conveys the filter expression used in a
scoped and filtered operation.
*/
typedef string FilterType;

/** Get Results structures hold a list of attribute values per object.
@member name            The CORBA name of the object
@member notFilterable   This flag will be true if the service could not
                        interact with the object to see if it even
                        passed the filter.  If true, the attributes and
                        failedAttributes members will be empty.
@mebmer attributes      The list of attributes retrieved from the
                        object.
@member failedAttributes The list of attributes whose values could
                        not be retrieved from the object.
*/

struct GetResultsType {
        NameType                name;
        boolean                 notFilterable;
        AttributeSetType        attributes;
        StringSetType           failedAttributes;
};

/** The Get Results Set is a set of results returned by a scoped get
operation.  */

typedef sequence <GetResultsType> GetResultsSetType;

/** A Language Type parameter conveys the filter expression language
used in a scoped and filtered operation.
*/

typedef string LanguageType;

/** A Language Set Type parameter contains a sequence of Languages. */

typedef sequence <LanguageType> LanguageSetType;

/** ModificationOp is used to indicate the type of update to be made to
an attribute. */

enum ModificationOpType {set, add, remove};

/** Modification structures identify an attribute and a modification to
be made to it.  Multiple updates may be made to a single attribute by
```

```
including multiple structures with the same attribute name in the
modification Set.
@member attrib          The name of the attribute to update.
@member op              The operation to be performed on the attribute.
@member val             The value to be used for the update operation.
                        It's type will depend on the attribute.
*/

struct ModificationType {
        string                  attrib; // the name of the attribute
        ModificationOpType      op;     // operation to be performed
        any                     value;  // value used to update attrib.
};

/** The Modification Sequence contains a sequence of modifications to
be made (in order) to each object in a scoped update operation. */

typedef sequence <ModificationType> ModificationSeqType;

/** Scope Choice enumerates four possible choices for a scope.  A
scope may include just the named base object, the entire subtree
of object below and including the base object, the objects at a certain
level below the base object (level 1 objects are directly contained by
the base object), or all of the objects down to a level, including the
base object and the level.
*/
enum ScopeChoiceType {baseObjectOnly, wholeSubtree, individualLevel,
                        baseToLevel};

/** Scope is used to convey which objects contained under the base
object, if any, are to be included in the scope of a scoped and
filtered operation. A level does not make sense for the baseObjectOnly
and wholeSubtree choices, but does for the other two.  To illustrate
the difference between the two options that include a level, a
scope choice of individualLevel with level = 1 would include all
of the objects directly contained by the base object.  A scope choice
of baseToLevel with level = 1 would include all of the objects
directly contained by the base object, and the base object.
*/

union ScopeType switch (ScopeChoiceType)
{
        /* The baseObjectOnly and wholeSubtree cases carry no value. */
        case individualLevel:   /* fall through */
        case baseToLevel:       short level;
};


/** Update Results structures hold the name of an object, a boolean
flag indicating if all modifications to that object were successful,
and a list of the attributes that could not be updated on that object.
The list will be null if the success flag is true.
@member name            the CORBA name of the object
@member notFilterable   This flag will be true if the service could not
                        interact with the object to see if it even
                        passed the filter.  If true, the client will
                        know no attributes could be set, so
                        the failedAttributes member will be empty.
@member failedAttributes the list of attributes that were not
                        correctly updated.
*/

struct UpdateResultsType {
```

```
        NameType        name;
        boolean         notFilterable;
        StringSetType   failedAttributes;
};

/** An Update Results Set is returned in response to a scoped update
operation (one that sets, adds to, or removes from the value of an
attribute). */

typedef sequence <UpdateResultsType> UpdateResultsSetType;
```

// Constants

```
/** Default filter is to allow everything through the filter*/

const FilterType defaultFilter = "TRUE";

/** Default language is the grammar described in this document */

const LanguageType defaultLanguage = "MOO 1.0";
```

// Exceptions

```
/**  A channel already registered exception is returned when an attempt
is made to register a channel with multiple channel IDs.  */

exception ChannelAlreadyRegistered {};

/**  A channel not found exception is returned when an event channel
cannot be found.  */

exception ChannelNotFound {};

/** A FactoryNotFound exception is raised when a requested factory
can't be found.  */

exception FactoryNotFound {};

/** A Filter Complexity Limit is raised when a filter expression in a
scoped operation is valid, but too complex to be processed.  */

exception FilterComplexityLimit {};

/** An invalid filter exception is raised when a client includes a
filter expression that cannot be parsed.  The text surrounding the
syntax error should be returned for trouble-shooting purposes. */

exception InvalidFilter {string badText;};

/** An Invalid Parameter exception is raised when the value of a
parameter is not valid for the operation.
@param parameter        the name of the bad parameter
*/

exception InvalidParameter {string parameter;};
```

// Interfaces

// Factory Finder Interface

```
/**
This interface defines a simple service for locating a managed object
factory.
*/

interface FactoryFinder {

        /** This method is used to find a managed object factory.
        @param factoryClass     The scoped class name of the factory,
                                NOT the managed object to be created.
        */

        ManagedObjectFactory find (in ObjectClassType factoryClass)
                raises (FactoryNotFound, ApplicationError);

        /** This method returns the list of factories registered
        with the factory finder. */

        FactoryInfoSetType list()
                raises (ApplicationError);

}; // end of FactoryFinder interface


/**
This interface extends the FactoryFinder interface to add methods
to support the registration and unregistration of factories.
*/

interface FactoryFinderComponent : FactoryFinder {

        /** This method is used by factories to register themselves.
        It should not be used by managing systems.
        @param factoryClass     The scoped class name of the factory,
                                NOT the managed object to be created.
        @param factoryRef       A reference to the factory.
        */

        void register (in ObjectClassType factoryClass,
                in ManagedObjectFactory factoryRef)
                raises (ApplicationError);

        /** This method is used by factories to unregister themselves,
        if necessary.  It should not be used by managing systems.
        @param factoryClass     The scoped class name of the factory,
                                NOT the managed object to be created.
        @param factoryRef       A reference to the factory.
        */

        void unregister (in ObjectClassType factoryClass,
                in ManagedObjectFactory factoryRef)
                raises (FactoryNotFound, ApplicationError);

}; // end of FactoryFinderComponent interface
```

// Channel Finder Interface

```
/**
This interface defines a simple service for locating event channels.
```

```
*/

interface ChannelFinder {

        /** This method returns the list of channels registered
        with the channel finder.  */

        ChannelInfoSetType list()
                raises (ApplicationError);

}; // end of ChannelFinder interface


/**
This interface extends the ChannelFinder interface to add methods
to support the registration and unregistration of channels.
*/

interface ChannelFinderComponent : ChannelFinder {

        /** This method is used by channels to register themselves.
        It should not be used by managing systems.  Re-registering
        a channel (re-using an existing channelID) results in
        updating that entry.  The other information previously
        associated with that entry is overwritten.  A
        ChannelAlreadyRegistered exception may be raised when an
        attempt is made to register a channel with multiple channelIDs.
        This should not be done.  (The service cannot guarantee that
        because two object references differ, they do not reference
        the same object.  It is therefore required that the managed
        system ensure that the same channel is not registered twice.)
        A channel change notification is sent whenever calling this
        method results in a change.
        @param channelID        A string identifier for the channel.
        @param channelClass     The scoped class name of the event
                                channel.
        @param baseObjects      The objects at the bases of the trees
                                of managed objects sending events to
                                this channel.  A null list indicates
                                that all base managed objects on the
                                system are covered by this channel.
        @param eventTypes       The list of event types handled by
                                this channel. A null list indicates all
                                event types are handled by this
                                channel.
        @param excludedEventTypes If the eventTypes parameter is null,
                                this can be used to exclude event
                                types.  If eventTypes is not null, this
                                should be null and is ignored.
        @param sourceClasses    The list of types of objects that send
                                events to this channel.  A null list
                                indicates all types of managed objects
                                send events to this channel.
        @param excludedSourceClasses If the sourceClsses parameter is
                                null, this can be used to exclude
                                source classes.  If sourceClasses is
                                not null, this should be null and is
                                ignored.
        @param channel          A reference to the channel.
        */

        void register (in string channelID,
                in ObjectClassType channelClass,
```

```
                        in NameSetType baseObjects,
                        in EventSetType eventTypes,
                        in EventSetType excludedEventTypes,
                        in ScopedNameSetType sourceClasses,
                        in ScopedNameSetType excludedSourceClasses,
                        in EventChannel channel)
                        raises (ChannelAlreadyRegistered, ApplicationError);

                /** This method is used by managed systems to unregister
                channels, if necessary.  It should not be used by managing
                systems.
                @param channel          A reference to the channel.
                */

                void unregister (in EventChannel channel)
                        raises (ChannelNotFound, ApplicationError);

        }; // end of ChannelFinderComponent interface
```

// Heartbeat Service Interface

```
        /**
        This interface defines a service used to periodically test the
        operation of the notification channels on a system.  The service
        supporting this interface periodically emits a short “heartbeat”
        notification on each channel on the system.
        */

        interface Heartbeat {

                /** The systemLabel attribute is sent in heartbeat
                notifications.  It is used to identify the heartbeat service
                instance from which the notification came.  Resetting this does
                not cause the service to immediately emit a notification, but
                the new value will be sent with the next notification.  */

                attribute string systemLabel;

                /**  The period is the interval, in seconds, at which the
                heartbeat service emits the heartbeat notification.  If it is
                zero, the service does not emit notifications.  */

                unsigned short periodGet();

                /** Updating of the period shall cause the service to deliver a
                notification to all channels with the new period value and then
                begin a new period.  Setting the period to zero shall cause the
                service to emit one final notification with a period value of
                zero, then no more (until the period is reset).  An attempt to
                set the period to a value outside the range supported will
                result in an ApplicationError with the error code set to
                invalidParameter.  */

                void periodSet(in unsigned short period)
                        raises(ApplicationError);

        }; // end of Heartbeat interface
```

// Terminator Service Interface

```
        /**
```

```
This interface defines a service that supports the deletion of managed
objects by clients.  A goal of the framework is to enable
implementations in which the managed objects do not have to maintain
the naming tree information.  The factories are one place to implement
the functions needed to create name bindings, and this service can be
used to clean up the naming tree after object deletion.  <p>

Also, this service can implement the rules for deleting objects based
on the delete policy of the managed objects.
*/

interface TerminatorService {

        /** This method is used to delete a managed object by
        specifying its name.  */

        void deleteByName (in NameType name)
                raises (ApplicationError, DeleteError);

        /** This method is used to delete a managed object by
        reference.  */

        void deleteByRef (in ManagedObject mo)
                raises (ApplicationError, DeleteError);

}; // end of TerminatorService interface
```

// DeleteResultsIterator Interface

```
/** The Delete Results Iterator interface is used to retrieve the
results from a scoped delete operation using the iterator design
pattern. */

interface DeleteResultsIterator {

        /** This method is used to retrieve the next “howMany” results
        in the result set.  */

        DeleteResultsSetType getNext(in unsigned short howMany)
                                raises (ApplicationError);

        /** This method is used to destroy the iterator and release its
        resources.  */

        void destroy();

}; // end of Delete Results Iterator interface
```

// GetResultsIterator Interface

```
/** The Get Results Iterator interface is used to retrieve the results
from a scoped get operation using the iterator design pattern. */

interface GetResultsIterator {

        /** This method is used to retrieve the next “howMany” results
        in the result set.  */

        GetResultsSetType getNext(in unsigned short howMany)
                                raises (ApplicationError);
```

```
        /** This method is used to destroy the iterator and release its
        resources.  */

        void destroy();

};  // end of Get Results Iterator interface
```

// UpdateResultsIterator Interface

```
/** The Update Results Iterator interface is used to retrieve the
results from a scoped update (set, add, remove) operation using the
iterator design pattern.
*/

interface UpdateResultsIterator {

        /** This method is used to retrieve the next “howMany” results
        in the result set.  */

        UpdateResultsSetType getNext(in unsigned short howMany)
                                raises (ApplicationError);

        /** This method is used to destroy the iterator and release its
        resources.  */

        void destroy();

};  // end of Update Results Iterator interface
```

// BasicMooService Interface

```
/** The basic scoping and filtering interface provides a common service
for performing attribute retrieval operations on multiple objects.
*/

interface BasicMooService {

        /** This operation is used to retrieve the list of filter
        languages supported by the service.  At the least, the
        list must include the value of the defaultLanguage constant
        defined above.  */

        LanguageSetType getFilterLanguages()
                                raises (ApplicationError);

        /** This operation is used to retrieve attributes from multiple
        objects using a small number of method invocations.  The method
        returns the first batch of results, one per object.  Each
        result has the name of the object and a list of name-value
        pairs indicating the attributes that could be retrieved with
        their values.

        @param baseName The name of the object at the base of the scope
                        tree.
        @param scope    A value indicating the contained objects to
                        include in the scope of the operations. See
                        ScopeType for details.
        @param filter   A string containing an expression to be
                        evaluated with attribute values from each
```

```
                        object in the scope.  Attribute values are
                        returned only for those objects for which the
                        expression evaluates to true.
                @param language A string identifying the language in which the
                        filter expression is written.
                @param attributes The names of the attributes for which values
                        should be returned.  If this list is null, all
                        attributes are to be returned.
                @param howMany  The maximum number of objects for which results
                        should be returned in the first batch.
                @param resultsIterator A reference to an iterator that can be
                        used to retrieve the rest of the results.  This
                        reference will be null if all results were
                        returned in the first batch.
                */

                GetResultsSetType scopedGet (
                        in NameType baseName,
                        in ScopeType scope,
                        in FilterType filter,
                        in LanguageType language,
                        in StringSetType attributes,
                        in unsigned short howMany,
                        out GetResultsIterator resultsIterator)
                        raises (InvalidParameter,
                                InvalidFilter,
                                FilterComplexityLimit,
                                ApplicationError);

        }; // end of BasicMooService interface
```

// AdvancedMooService Interface

```
        /** The advanced scoping and filtering interface provides a common
        service for performing multiple-attribute updates on multiple objects,
        and for deleting multiple objects.
        */

        interface AdvancedMooService : BasicMooService {

                /** This operation is used to modify multiple attributes in
                multiple objects using a small number of method invocations.
                The method returns the first batch of results, a list of
                objects for which one or more modifications failed.  Each
                result indicates the attribute(s) on that object that could not
                be updated.

                @param baseName The name of the object at the base of the scope
                        tree.
                @param scope    A value indicating the contained objects to
                        include in the scope of the operations. See
                        ScopeType for details.
                @param filter   A string containing an expression to be
                        evaluated with attribute values from each
                        object in the scope.  Updates are performed
                        only on those objects for which the expression
                        evaluates to true.
                @param language A string identifying the language in which the
                        filter expression is written.
                @param modifications The list of modifications to be made to
                        each object.
                @param failuresOnly If true only results for failed objects
```

```
                will be returned.
@param howMany  The maximum number of objects for which results
                should be returned in the first batch.
@param resultsIterator A reference to an iterator that can be
                used to retrieve the rest of the results.  This
                reference will be null if all results were
                returned in the first batch.
*/

UpdateResultsSetType scopedUpdate (
        in NameType baseName,
        in ScopeType scope,
        in FilterType filter,
        in LanguageType language,
        in ModificationSeqType modifications,
        in boolean failuresOnly,
        in unsigned short howMany,
        out UpdateResultsIterator resultsIterator)
        raises (InvalidParameter,
                InvalidFilter,
                FilterComplexityLimit,
                ApplicationError);

/** This operation is used to delete multiple objects using a
small number of method invocations.  The method returns the
first batch of results, a list of the objects that could not be
deleted.

@param baseName The name of the object at the base of the scope
                tree.
@param scope    A value indicating the contained objects to
                include in the scope of the operations. See
                ScopeType for details.
@param filter   A string containing an expression to be
                evaluated with attribute values from each
                object in the scope.  Only those objects for
                which the expression evaluates to true are
                deleted.
@param language A string identifying the language in which the
                filter expression is written.
@param failuresOnly If true only results for failed objects
                will be returned.
@param howMany  The maximum number of objects for which results
                should be returned in the first batch.
@param resultsIterator A reference to an iterator that can be
                used to retrieve the rest of the results.  This
                reference will be null if all results were
                returned in the first batch.
*/

DeleteResultsSetType scopedDelete (
        in NameType baseName,
        in ScopeType scope,
        in FilterType filter,
        in LanguageType language,
        in boolean failuresOnly,
        in unsigned short howMany,
        out DeleteResultsIterator resultsIterator)
        raises (InvalidParameter,
                InvalidFilter,
                FilterComplexityLimit,
                ApplicationError);
```

```
}; // end of AdvancedMooService interface
```

// Notifications Interface

```
/**  The notifications interface defines the notifications emitted by
the framework services, not the managed objects themselves.
*/

interface Notifications {

        /** The Channel Change notification is a special notification
        because it is emitted by the framework (the Channel Finder) and
        not a managed object.  It reports the addition, deletion, or
        change of a registered event channel.
        @param channelModification      indicates if a channel has been
                                        added, removed, or updated.
        @param channelInfo              provides the information about
                                        the affected channel.
        */

        void channelChange (
                in ChannelModificationType      channelModification,
                in ChannelInfoType              channelInfo
        );

        /** This operation signature defines the notification emitted
        by the heartbeat service.
        @param systemLabel      the current value of the Heartbeat
                                service systemLabel attribute.
        @param channelID        the ID of the channel through which the
                                notification was sent.
        @param period           the current value of the Heartbeat
                                service period attribute.
        @param timeStamp        the current time when the notification
                                is emitted.
        */

        void heartbeat (
                in string               systemLabel,
                in string               channelID,
                in unsigned short       period,
                in UtcT                 timeStamp
        );

        /** These constants defines the names of the notification
        declared above and are provided to help reduce errors.  */

        const string channelChangeTypeName =
                "itut_q816::Notifications::channelChange";

        const string heartbeatTypeName =
                "itut_q816::Notifications::heartbeat";

        /** These constants define the names of the parameters used in
        the notifications declared above and are provided to help
        reduce errors.  */

        const string channelIDName = "channelID";
        const string channelModificationName = "channelModification";
        const string channelInfoName = "channelInfo";
        const string periodName = "period";
        const string systemLabelName = "systemLabel";
```

```
                const string timeStampName = "timeStamp";

        }; // end of Notifications interface

}; // end of module itut_q816

#endif  // end of #ifndef ITUT_Q816_IDL
```

Appendix A Interworking Scenarios Between Models Using ITU Framework and ADSL/ATMF Compliant Models

A.1 Introduction

This appendix describes how systems designed using the approach in this framework may interwork with systems designed using the approach specified in the ATM Forum and the ASDL Forum.

The following approaches have been used to define interfaces for CORBA Based managed objects:

- A fine-grained model has a one-to-one relationship between CORBA interface instances (i.e., having their own Interoperable Object Reference, IOR) and managed object instances.
- A class-grained model has one CORBA interface for each managed object class. Some other mechanism (such as managed object name placed as an input parameter for every operation in that interface) has to be supported by the CORBA class grained interface to allow management of each managed object instance.

A, so called, Grain-neutral approach uses a structure holding both the managed object name and the IOR used to provide access to each managed objects. Note that the grain-neutral approach, while requiring the client to pass the managed object instance as a parameter to each operation (i.e., looks like class grained to the client), allows the implementation in the server to be either class grain or fine grain.

The framework in this Recommendation uses a fine-grained approach, along with a value-type based *attributesGet* operation, for defining CORBA based managed objects. The value type associated with a CORBA managed object subclass uses value type inheritance to extend the elements of the value type associated with its superclass.

The ATM Forum and ASDL Forum specifications includes the IOR for the class and the name of the managed object as a paired list of parameters for the operations (i.e., they are grain neutral). The client uses the name to reference the specific entity and does not require a separate IOR to be for each entity. These specifications also use structures (i.e., they do not employ inheritance) for their *attributesGet* operations.

A.2 Terminology

The following terms are introduced for the purpose of discussion:

- Grain Neutral Server – A managed system which implements objects defined using a grain neutral model with CORBA 2.1
- ITU Framework server - A managed system which implements objects defined using the Framework in this Recommendation, thus supporting CORBA 2.3 features
- Grain Neutral Managing System - A client capable of managing CORBA objects defined using grain neutral model with CORBA 2.1

– ITU Framework Managing System – A client capable of managing CORBA objects defined according to the Framework in this recommendation, thus supporting CORBA 2.3 features

A.3 Interworking Scenarios

A.3.1 Grain Neutral Server migrating to ITU Framework Server

Upon migration from a grain neutral server, an ITU Framework server may add new capabilities; however it must preserve the old capabilities as found in the grain neutral version

The new server should implement an adapter function. This function should present the grain neutral interfaces to existing Grain neutral managing systems. An implementation approach can employ delegation of operations invoked on grain neutral objects to the objects built according to the framework. Such a delegation approach will issue the same operations on the individual objects as would be invoked by an ITU Framework based managing system.

The class specific grain-neutral get all attribute operation parameters should be converted to the value type structured as per the framework in this specification.

There are functions to be performed by the interworking software beyond the differences resulting from the use of POA and value object. The delegation software needs to be customized to address the differences in the naming structure, including this Framework's use of the kind field. For example explain how the context object reference used in constructing the name needs to be replaced with kind field of COS naming structure.

A.3.2 Grain Neutral Client migrating to ITU Framework Client

Such an ITU Framework Client needs to manage both grain neutral servers as well as ITU Framework servers.

ITU framework client with CORBA 2.3 needs, in addition to using the naming tree for ITU Framework serves, to use the naming tree for grain neutral servers. CORBA 2.3 implementation may require an adaptation function where the application issuing the request to an object has to be converted to be appropriate for the grain neutral server. If the application uses the value type it has to be decomposed into the object specific get operation for the old server.

For an ITU Framework based client to manage both ITU Framework based servers and pre-ITU Framework servers, the ITU client will need to support stubs for both the ITU and pre-ITU servers. In addition, some interworking function may be needed, as discussed above and as shown in the figure below.

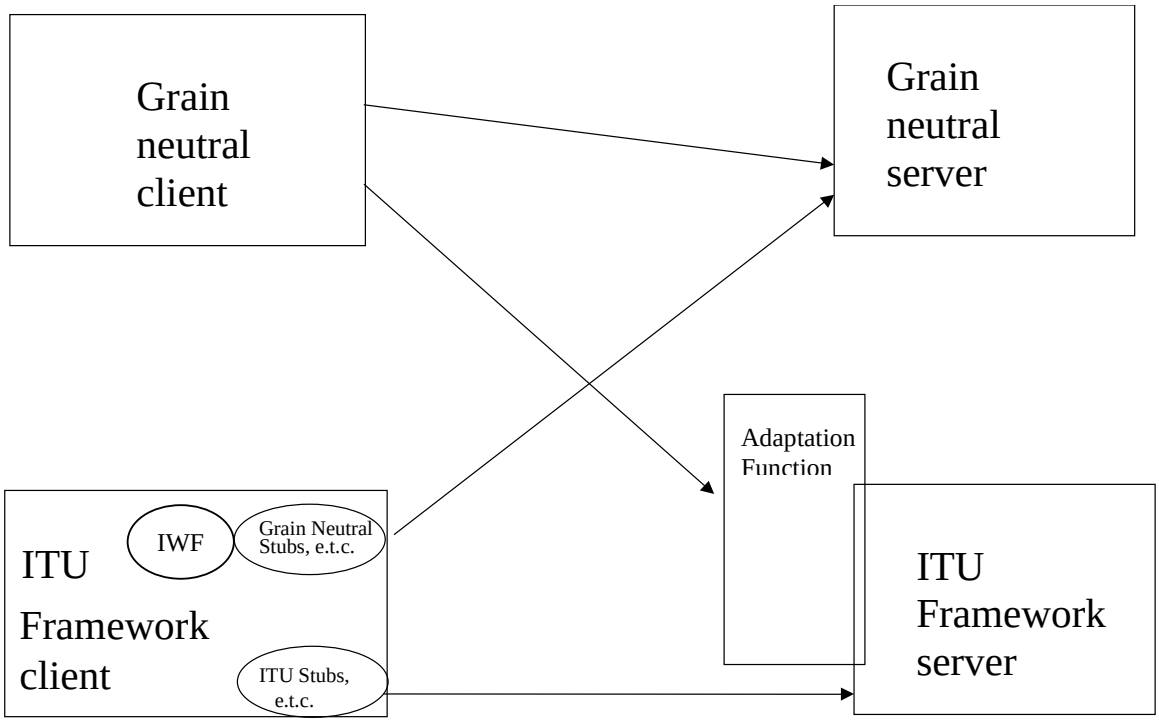

Figure 10. Interworking Scenarios