INTERNATIONAL STANDARD

ISO/IEC 19500-2

Second edition
2012-04-15

Information technology — Object Management Group — Common Object Request Broker Architecture (CORBA) —

Part 2: Interoperability

Technologies de l'information — OMG (Object Management Group) — CORBA (Common Object Request Broker Architecture) —

Partie 2: Interopérabilité

Reference number
ISO/IEC 19500-2:2012(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Table of Contents

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19500-2 was prepared by Technical Committee ISO/IEC JTC1, Information technology, in collaboration with the Object Management Group (OMG), following the submission and processing as a Publicly Available Specification (PAS) of the OMG Common Object Request Broker Architecture (CORBA) specification Part 2 Version 3.1 CORBA Interoperability.

ISO/IEC 19500-2 is related to:

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, Information Technology - Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, Information Technology - Open Distributed Processing - Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology - Open Distributed Processing - Interface Definition Language
- ISO/IEC 19500-2, Information Technology - Open Distributed Processing - CORBA Specification Part 1: CORBA Interfaces
- ISO/IEC 19500-3, Information Technology - Open Distributed Processing - CORBA Specification Part 3: CORBA Components

ISO/IEC 19500 consists of the following parts, under the general title *Information technology - Open distributed processing - CORBA specification*:

- Part 1: CORBA Interfaces
- Part 2: CORBA Interoperability
- Part 3: CORBA Components

It is the common core of the CORBA specification. Optional parts of CORBA, such as mappings to particular programming languages, Real-time CORBA extensions, and the minimum CORBA profile for embedded systems are documented in the other specifications that together comprise the complete CORBA specification. Please visit the CORBA download page at http://www.omg.org/technology/documents/corba_spec_catalog.htm to find the complete CORBA specification set.

Apart from this Foreword, the text of this International Standard is identical with that for the OMG specification for CORBA, v3.1.1, Part 2.

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability and portability can be integrated.

RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. The scopes and objectives of the RM-ODP Part 2 and the UML, while related, are not the same and, in a number of cases, the RM-ODP Part 2 and the UML specification use the same term for concepts which are related but not identical (e.g., interface). Nevertheless, a specification using the Part 2 modeling concepts can be expressed using UML with appropriate extensions (using stereotypes, tags, and constraints).

RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2. Given the relation between UML as a modeling language and Part 3 of the RM-ODP standard, it is easy to show that UML is suitable as a notation for the individual viewpoint specifications defined by the RM-ODP.

This International Standard for CORBA Interfaces is a standard for the technology specification of an ODP system. It defines a technology to provide the infrastructure required to support functional distribution of an ODP system, specifying functions required to manage physical distribution, communications, processing and storage, and the roles of different technology objects in supporting those functions.

Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices*: *Common Object Services Specification*.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities*: *Common Facilities Architecture*.
- **Application Objects**, which are products of a single vendor on in-house development group that controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

The architecture and specifications described in this standard are aimed at software designers and developers who want to produce applications that comply with OMG specifications for the Object Request Broker (ORB), or this standard (ISO/IEC 19500). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. The ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

This Part of this International Standard includes a non-normative annex.

Information technology - Object Management Group Common Object Request Broker Architecture (CORBA), Interoperability

1 Scope

This part of ISO/IEC 19500 specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant Object Request Brokers (ORBs). The approach to inter-ORB operation is universal, because elements can be combined in many ways to satisfy a very broad range of needs.

This part of ISO/IEC 19500 covers the specification of:

- ORB interoperability architecture
- Inter-ORB bridge support
- The General Inter-ORB Protocol (GIOP) for object request broker (ORB) interoperability. GIOP can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions defined by this standard.
- The Internet Inter-ORB Protocol (IIOP), a specific mapping of the GIOP which runs directly over connections that use the Internet Protocol and the Transmission Control Protocol (TCP/IP connections).
- The CORBA Security Attribute Service (SAS) protocol and its use within the CSIv2 architecture to address the requirements of CORBA security for interoperable authentication, delegation, and privileges.

This part of ISO/IEC 19500 provides a widely implemented and used particularization of ITU-T Rec. X.931 | ISO/IEC 14752. Open Distributed Processing - Protocol Support for Computational Interactions. It supports interoperability and location transparency in ODP systems.

2 Conformance and Compliance

An ORB is considered to be interoperability-compliant when it meets the following requirements:

- In the CORBA Core part, standard APIs are provided by an ORB to enable the construction of request-level inter-ORB bridges. APIs are defined by the Dynamic Invocation Interface, the Dynamic Skeleton Interface, and by the object identity operations described in the Interface Repository clause of this book.
- An Internet Inter-ORB Protocol (IIOP) (explained in the Building Inter-ORB Bridges clause) defines a transfer syntax and message formats (described independently as the General Inter-ORB Protocol), and defines how to transfer messages via TCP/IP connections. The IIOP can be supported natively or via a halfbridge.

Support for additional Environment Specific Inter-ORB Protocols (ESIOPs) and other proprietary protocols is optional in an interoperability-compliant system. However, any implementation that chooses to use the other protocols defined by the CORBA interoperability specifications must adhere to those specifications to be compliant with CORBA interoperability.

Figure 6.2 on page 12 shows examples of interoperable ORB domains that are CORBA-compliant. These compliance points support a range of interoperability solutions. For example, the standard APIs may be used to construct “half bridge” to the IIOP, relying on another “half bridge” to connect to another ORB. The standard APIs also support

construction of “full bridges,” without using the Internet IOP to mediate between separated bridge components. ORBs may also use the Internet IOP internally. In addition, ORBs may use GIOP messages to communicate over other network protocol families (such as Novell or OSI), and provide transport-level bridges to the IIOP.

The GIOP is described separately from the IIOP to allow future specifications to treat it as an independent compliance point.

2.1 Unreliable Multicast

Summary of Optional Verses Mandatory Interfaces

An interface to an MIOP gateway should be considered an optional interface within the MIOP specification.

Proposed Compliance Points

The MIOP specification is a single, optional compliance point within the CORBA Core specification.

Changes to Other OMG Specifications

This part of ISO/IEC 19500 contains an extension to the IOP module.

```
module IOP {
    const ProfileId         TAG_UIPMC = 3;
    const ComponentId       TAG_GROUP = 39;
    const ComponnetId       TAG_GROUP_IIOP = 40
};
```

3 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, Open Distributed Processing - Reference Model: Architecture
- ITU-T Recommendation X.920 (1999) | ISO/IEC 10750:1999, Open Distributed Processing - Interface Definition Language
- ITU-T Recommendation X.931(2000) | ISO/IEC 14752:2000, Open Distributed Processing - Protocol Support for Computational Interactions
- ISO/IEC 8859-1: 1998, Information Technology - 8-bit single byte coded graphic character sets - Part 1: Latin alphabet No. 1
- ISO/IEC 10646-1:1993 Information Technology - Universal Multiple-Octect coded character set (UCS) - Part 1: Architecture and Basic Multilingual Plane

- ISO/IEC 10646-1: 1993/Amd 1:1996 Transformation Format for 16 planes of group 00 (UTF - 16)
- ISO/IEC 10646-1: 1993/Amd 2:1996 UCS Transformation Format 8 (UTF - 8)
- ISO/IEC 19500-1: 2011 Open Distributed Processing - CORBA Specification Part 1: CORBA Interfaces, pas/2011-08-07

3.1 Other Specifications

- STD 007 (also, RFC 793), Transmission Control Protocol, J. Postel, Internet Engineering Task Force, Sept. 1981
- STD 005 (also, RFC 791), Internet Protocol, J. Postel, Internet Engineering Task Force, Sept. 1981
- OSF Character and Code Set Registry, OSF DCE FRC 40.1 (Public Version), S. (Martin) O'Donnell, June 1994.
- RPC Runtime Support For I18N Characters - Functional Specification, OSF DCE SIG RFC 41.2, M. Romagna, R. Mackey, November 1994.
- [JAV2I]Object Management Group, "Java to IDL," available from http://www.omg.org/spec/JAV2I/1.4
- [CORBASEC]Object Management Group, "Security Service," available from http://www.omg.org/spec/SEC/
- [ASMOTS]Object Management Group, "Additional Structuring Mechanisms for the OTS," available from http://www.omg.org/spec/OTS/
- [TRANS]Object Management Group, "Transaction Service," available from http://www.omg.org/spec/TRANS/
- [FIREWALL]Object Management Group, "CORBA Firewall Traversal Specification," available from http://www.omg.org/members/cgi-bin/doc?ptc/04-04-05.pdf
- [SCCP] Object Management Group, "CORBA / TC Interworking and SCCP-Inter ORB Protocol (SCCP)," available from http://www.omg.org/spec/SCCP
- [FTCORBA] Object Management Group, "Fault Tolerant Corba," clause 23 of CORBA 3.0.3, available from http://www.omg.org/cgi-bin/doc?formal/2004-03-01
- [RTCORBA] Object Management Group, "Real-Time CORBA, version 1.2," available from http://www.omg.org/spec/RT/
- [WATM] Object Management Group, "Wireless Access and Telecom Mobility in CORBA, Version 1.2," available from http://www.omg.org/spec/WATM/
- [DCOMI] Object Management Group, "Interoperability with non-CORBA Systems" clause 20 of CORBA 3.0.3, available from http://www.omg.org/cgi-bin/doc?formal/2004-03-01
- [TSAS] Object Management Group, "Telecommunications Service Access and Subscription Specification," available from http://www.omg.org/spec/TSAS/
- IETF RFC2119, "Key words for use in RFCs to Indicate Requirement Levels," S. Bradner, March 1997 (http://ietf.org/rfc/rfc2119)

4 Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

4.1 Recommendations | International Standards

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.902 | ISO/IEC 10746-2:

- behavior
- interface
- instance
- object
- service
- state
- transparency
- type

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.903 | ISO/IEC 10746-3:

- operation
- stub

4.2 Terms Defined in this Part of ISO/IEC 19500

adapter	Same as object adapter.
attribute	An identifiable association between an object and a value. An attribute A is made visable to clients as a pair of operations: get_A and set_A. Readonly attributes only generate a get operation.
client	The code or process that invokes an operation on an object.
data type	A categorization of values operation arguments, typically covering both behavior and representation (i.e., the traditional no-OO programming language notion of type.)
domain	A concept important to interoperability, it is a distinct scope, within which common characteristics are exhibited, common rules observed, and over which a distribution transparency is preserved.
dynamic invocation	Constructing and issuing a request whose signature is possibly not known until run-time.
dynamic skeleton	An interface-independent kind of skeleton, used by servers to handle requests whose signatures are possibly not known until run-time.

implementation	A definition that provides the information needed to create an object and allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of the data structure used to represent the core state associated with an object, as well as definitions of the methods that access that data structure. It will also typically include information about the intended interface of the object.
interface repository	A storage place for interface information.
interface type	A type satisfied by any object that satisfies a particular interface.
interoperability	The ability for two or more ORBs to cooperate to deliver requests to the proper object. Interoperating ORBs appear to a client to be a single ORB.
language binding or mapping	The means and conventions by which a programmer writing in a specific programming language accesses ORB capabilities.
method	An implementation of an operation. Code that may be executed to perform a requested service. Methods associated with an object may be structured into one or more programs.
object adapter	The ORB component which provides object reference, activation, and state related services to an object implementation. There may be different adapters provided for different kinds of implementations.
object implementation	Same as implementation.
object reference	A value that unambiguously identifies an object. Object references are never reused to identify another object.
objref	An abbreviation for object reference
ORB core	The ORB component which moves a request from a client to the appropriate adapter for the target object.
request	A message issued by a client to cause a service to be performed.
results	The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.
server	A process implementing one or more operations on one or more objects.
signature	Defines the parameters of a given operation including their number order, data types, and passing mode; the results if any; and the possible outcomes (normal vs. exceptional) that might occur.
skeleton	The object-interface-specific ORB component which assists an object adapter in passing requests to particular methods.
synchronous request	A request where the client pauses to wait for completion of the request. Contrast with deferred synchronous request and one-way request.
value	Any entity that may be a possible actual parameter in a request. Values that serve to identify objects are called object references.

4.3 Keywords for Requirment statements

The keywords “must,” “must not,” “shall,” “shall not,” “should,” “should not,” and “may” in this International Standard are to be interpreted as described in IETF RFC 2119.

5 Symbols (and abbreviated terms)

For the purposes of this International Standard, the following abbreviations apply:

ADT	Abstract Data Type
CSIv2	Common Secure Interoperability, version 2
CCCS	Client Conversion Code Sets
CCS	Conversion Code Sets
CDR	Common Data Representation
CMIR	Client Makes it Right
CNCS	Client Native Code Set
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
ESIOP	Environment Specific Inter-ORB Protocol
OMG	Object Management Group
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
ORB	Object Request Broker
SAS	Security Attribute Service
SCCS	Server Conversion Code Sets
SMIR	Server Makes It Right
SNCS	Server Native Code Set
TCS	Transmission Code Set
TCS-C	Char Transmission Code Set
TCS-W	Wchar Transmission Code Set
VSCID	Vender Service Context codeset ID

6 Interoperability Overview

6.1 General

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The approach to "interORBability" is universal because its elements can be combined in many ways to satisfy a very broad range of needs.

6.2 Elements of Interoperability

The elements of interoperability are as follows:

- ORB interoperability architecture
- Inter-ORB bridge support
- General and Internet Inter-ORB Protocols (GIOPs and IIOPs)

In addition, the architecture accommodates Environment Specific Inter-ORB Protocols (ESIOPs) that are optimized for particular environments such as DCE.

6.2.1 ORB Interoperability Architecture

The ORB Interoperability Architecture provides a conceptual framework for defining the elements of interoperability and for identifying its compliance points. It also characterizes new mechanisms and specifies conventions necessary to achieve interoperability between independently produced ORBs.

Specifically, the architecture introduces the concepts of *immediate* and *mediated bridging* of ORB domains. The Internet Inter-ORB Protocol (IIOP) forms the common basis for broad-scope mediated bridging. The inter-ORB bridge support can be used to implement both immediate bridges and to build "half-bridges" to mediated bridge domains.

By use of bridging techniques, ORBs can interoperate without knowing any details of that ORB's implementation, such as what particular IPC or protocols (such as ESIOPs) are used to implement the *CORBA* specification.

The IIOP may be used in bridging two or more ORBs by implementing "half bridges" that communicate using the IIOP. This approach works for both stand-alone ORBs, and networked ones that use an ESIOP.

The IIOP may also be used to implement an ORB's internal messaging, if desired. Since ORBs are not required to use the IIOP internally, the goal of not requiring prior knowledge of each others' implementation is fully satisfied.

6.2.2 Inter-ORB Bridge Support

The interoperability architecture clearly identifies the role of different kinds of domains for ORB-specific information. Such domains can include object reference domains, type domains, security domains (e.g., the scope of a *Principal* identifier), a transaction domain, and more.

Where two ORBs are in the same domain, they can communicate directly. In many cases, this is the preferable approach. This is not always true, however, since organizations often need to establish local control domains.

When information in an invocation must leave its domain, the invocation must traverse a bridge. The role of a bridge is to ensure that content and semantics are mapped from the form appropriate to one ORB to that of another, so that users of any given ORB only see their appropriate content and semantics.

The inter-ORB bridge support element specifies ORB APIs and conventions to enable the easy construction of interoperability bridges between ORB domains. Such bridge products could be developed by ORB vendors, Sieves, system integrators, or other third-parties.

Because the extensions required to support Inter-ORB Bridges are largely general in nature, do not impact other ORB operation, and can be used for many other purposes besides building bridges, they are appropriate for all ORBs to support. Other applications include debugging, interposing of objects, implementing objects with interpreters and scripting languages, and dynamically generating implementations.

The inter-ORB bridge support can also be used to provide interoperability with non-CORBA systems, such as Microsoft's Component Object Model (COM). The ease of doing this will depend on the extent to which those systems conform to the CORBA Object Model.

6.2.3 General Inter-ORB Protocol (GIOP)

The General Inter-ORB Protocol (GIOP) element specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. The GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. It does not require or rely on the use of higher level RPC mechanisms. The protocol is simple, scalable, and relatively easy to implement. It is designed to allow portable implementations with small memory footprints and reasonable performance, with minimal dependencies on supporting software other than the underlying transport layer.

While versions of the GIOP running on different transports would not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

6.2.4 Internet Inter-ORB Protocol (IIOP)®

The Internet Inter-ORB Protocol (IIOP)® element specifies how GIOP messages are exchanged using TCP/IP connections. The IIOP specifies a standardized interoperability protocol for the Internet, providing "out of the box" interoperation with other compatible ORBs based on the most popular product- and vendor-neutral transport layer. It can also be used as the protocol between half-bridges (see below).

The protocol is designed to be suitable and appropriate for use by any ORB to interoperate in Internet Protocol domains unless an alternative protocol is necessitated by the specific design center or intended operating environment of the ORB. In that sense it represents the basic inter-ORB protocol for TCP/IP environments, a most pervasive transport layer.

The IIOP's relationship to the GIOP is similar to that of a specific language mapping to OMG IDL; the GIOP may be mapped onto a number of different transports, and specifies the protocol elements that are common to all such mappings. The GIOP by itself, however, does not provide complete interoperability, just as IDL cannot be used to build complete programs. The IIOP and other similar mappings to different transports, are concrete realizations of the abstract GIOP definitions, as shown in Figure 6.1 on page 9.

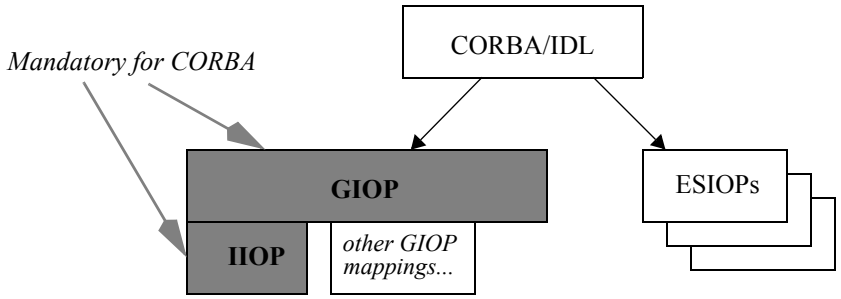


Figure 6.1 - Inter-ORB Protocol Relationships

6.2.5 Environment-Specific Inter-ORB Protocols (ESIOPs)

This part of ISO/IEC 19500 also makes provision for an open-ended set of Environment-Specific Inter-ORB Protocols (ESIOPs). Such protocols would be used for "out of the box" interoperation at user sites where a particular networking or distributing computing infrastructure is already in general use.

Because of the opportunity to leverage and build on facilities provided by the specific environment, ESIOPs might support specialized capabilities such as those relating to security and administration.

While ESIOPs may be optimized for particular environments, all ESIOP specifications will be expected to conform to the general ORB interoperability architecture conventions to enable easy bridging. The inter-ORB bridge support enables bridges to be built between ORB domains that use the IIOP and ORB domains that use a particular ESIOP.

6.3 Relationship to Previous Versions of CORBA

The ORB Interoperability Architecture builds on Common Object Request Broker Architecture by adding the notion of ORB Services and their domains. (ORB Services are described in "ORBs and ORB Services" on page 16). The architecture defines the problem of ORB interoperability in terms of bridging between those domains, and defines several ways in which those bridges can be constructed. The bridges can be internal (in-line) and external (request-level) to ORBs.

APIs included in the interoperability specifications include compatible extensions to previous versions of *CORBA* to support request-level bridging:

- A Dynamic Skeleton Interface (DSI) is the basic support needed for building request-level bridges. It is the server-side analogue of the Dynamic Invocation Interface and in the same way it has general applicability beyond bridging. For information about the Dynamic Skeleton Interface, refer to the *Dynamic Skeleton Interface* clause.
- APIs for managing object references have been defined, building on the support identified for the Relationship Service. The APIs are defined in Object Reference Operations in the *ORB Interface* clause of Part 1 of this International Standard (ISO/IEC 19500-1). The Relationship Service is described in the *Relationship Service* specification; refer to *CosObjectIdentity Module* in that specification.

6.4 Examples of Interoperability Solutions

The elements of interoperability (Inter-ORB Bridges, General and Internet Inter-ORB Protocols, Environment-Specific Inter-ORB Protocols) can be combined in a variety of ways to satisfy particular product and customer needs. This sub clause provides some examples.

6.4.1 Example 1

ORB product A is designed to support objects distributed across a network and provide "out of the box" interoperability with compatible ORBs from other vendors. In addition it allows bridges to be built between it and other ORBs that use environment-specific or proprietary protocols. To accomplish this, ORB A uses the IIOP and provides inter-ORB bridge support.

6.4.2 Example 2

ORB product B is designed to provide highly optimized, very high-speed support for objects located on a single machine. For example, to support thousands of Fresco GUI objects operated on at near function-call speeds. In addition, some of the objects will need to be accessible from other machines and objects on other machines will need to be infrequently accessed. To accomplish this, ORB A provides a half-bridge to support the Internet IOP for communication with other "distributed" ORBs.

6.4.3 Example 3

ORB product C is optimized to work in a particular operating environment. It uses a particular environment-specific protocol based on distributed computing services that are commonly available at the target customer sites. In addition, ORB C is expected to interoperate with other arbitrary ORBs from other vendors. To accomplish this, ORB C provides inter-ORB bridge support and a companion half-bridge product (supplied by the ORB vendor or some third-party) provides the connection to other ORBs. The half-bridge uses the IIOP to enable interoperability with other compatible ORBs.

6.4.4 Interoperability Compliance

An ORB is considered to be interoperability-compliant when it meets the following requirements:

- In the CORBA Core part, standard APIs are provided by an ORB to enable the construction of request-level inter-ORB bridges. APIs are defined by the Dynamic Invocation Interface, the Dynamic Skeleton Interface, and by the object identity operations described in the *Interface Repository* clause of Part 1 of this International Standard (ISO/IEC 19500-1).
- An Internet Inter-ORB Protocol (IIOP) (explained in the *Building Inter-ORB Bridges* clause) defines a transfer syntax and message formats (described independently as the General Inter-ORB Protocol), and defines how to transfer messages via TCP/IP connections. The IIOP can be supported natively or via a half-bridge.

Support for additional ESIOPs and other proprietary protocols is optional in an interoperability-compliant system. However, any implementation that chooses to use the other protocols defined by the CORBA interoperability specifications must adhere to those specifications to be compliant with CORBA interoperability.

Figure 6.2 on page 12 shows examples of interoperable ORB domains that are CORBA-compliant.

These compliance points support a range of interoperability solutions. For example, the standard APIs may be used to construct “half bridges” to the IIOP, relying on another “half bridge” to connect to another ORB. The standard APIs also support construction of “full bridges,” without using the Internet IOP to mediate between separated bridge components. ORBs may also use the Internet IOP internally. In addition, ORBs may use GIOP messages to communicate over other network protocol families (such as Novell or OSI), and provide transport-level bridges to the IIOP.

The GIOP is described separately from the IIOP to allow future specifications to treat it as an independent compliance point.

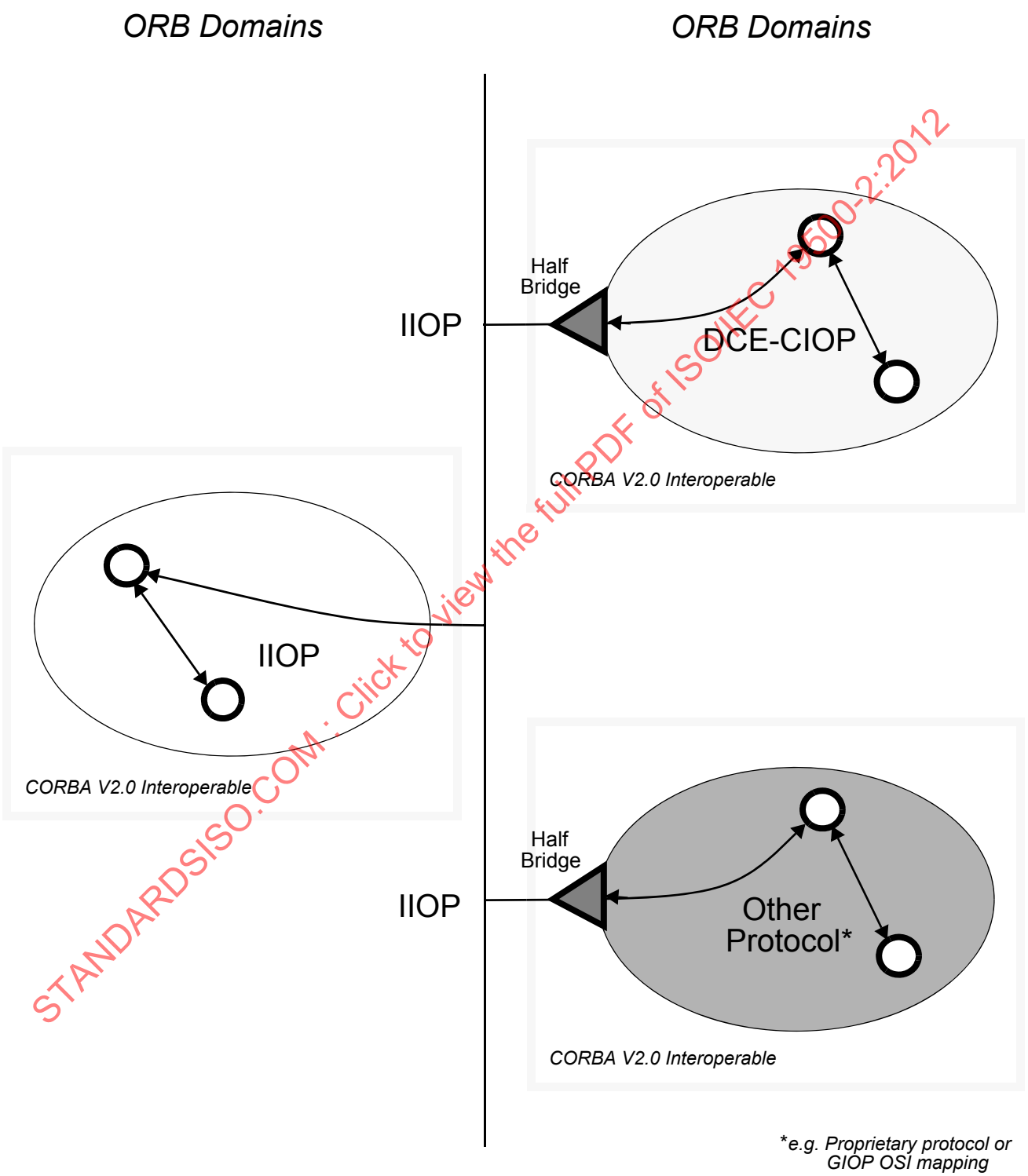


Figure 6.2 - Examples of CORBA Interoperability Compliance

6.5 Motivating Factors

This sub clause explains the factors that motivated the creation of interoperability specifications.

6.5.1 ORB Implementation Diversity

Today, there are many different ORB products that address a variety of user needs. A large diversity of implementation techniques is evident. For example, the time for a request ranges over at least 5 orders of magnitude, from a few microseconds to several seconds. The scope ranges from a single application to enterprise networks. Some ORBs have high levels of security, others are more open. Some ORBs are layered on a particular widely used protocol, others use highly optimized, proprietary protocols.

The market for object systems and applications that use them will grow as object systems are able to be applied to more kinds of computing. From application integration to process control, from loosely coupled operating systems to the information superhighway, CORBA-based object systems can be the common infrastructure.

6.5.2 ORB Boundaries

Even when it is not required by implementation differences, there are other reasons to partition an environment into different ORBs.

For security reasons, it may be important to know that it is not generally possible to access objects in one domain from another. For example, an “internet ORB” may make public information widely available, but a “company ORB” will want to restrict what information can get out. Even if they used the same ORB implementation, these two ORBs would be separate, so that the company could allow access to public objects from inside the company without allowing access to private objects from outside. Even though individual objects should protect themselves, prudent system administrators will want to avoid exposing sensitive objects to attacks from outside the company.

Supporting multiple ORBs also helps handle the difficult problem of testing and upgrading the object system. It would be unwise to test new infrastructure without limiting the set of objects that might be damaged by bugs, and it may be impractical to replace “the ORB” everywhere simultaneously. A new ORB might be tested and deployed in the same environment, interoperating with the existing ORB until either a complete switch is made or it incrementally displaces the existing one.

Management issues may also motivate partitioning an ORB. Just as networks are subdivided into domains to allow decentralized control of databases, configurations, resources, management of the state in an ORB (object reference location and translation information, interface repositories, per-object data) might also be done by creating sub-ORBs.

6.5.3 ORBs Vary in Scope, Distance, and Lifetime

Even in a single computing environment produced by a single vendor, there are reasons why some of the objects an application might use would be in one ORB, and others in another ORB. Some objects and services are accessed over long distances, with more global visibility, longer delays, and less reliable communication. Other objects are nearby, are not accessed from elsewhere, and provide higher quality service. By deciding which ORB to use, an implementor sets expectations for the clients of the objects.

One ORB might be used to retain links to information that is expected to accumulate over decades, such as library archives. Another ORB might be used to manage a distributed chess program in which the objects should all be destroyed when the game is over. Although while it is running, it makes sense for “chess ORB” objects to access the “archives ORB,” we would not expect the archives to try to keep a reference to the current board position.

6.6 Interoperability Design Goals

Because of the diversity in ORB implementations, multiple approaches to interoperability are required. Options identified in previous versions of *CORBA* include:

- *Protocol Translation*, where a gateway residing somewhere in the system maps requests from the format used by one ORB to that used by another.
- *Reference Embedding*, where invocation using a native object reference delegates to a special object whose job is to forward that invocation to another ORB.
- *Alternative ORBs*, where ORB implementations agree to coexist in the same address space so easily that a client or implementation can transparently use any of them, and pass object references created by one ORB to another ORB without losing functionality.

In general, there is no single protocol that can meet everyone's needs, and there is no single means to interoperate between two different protocols. There are many environments in which multiple protocols exist, and there are ways to bridge between environments that share no protocols.

This International Standard adopts a flexible architecture that allows a wide variety of ORB implementations to interoperate and that includes both bridging and common protocol elements.

The following goals guided the creation of interoperability specifications:

- The architecture and specifications should allow high-performance, small footprint, lightweight interoperability solutions.
- The design should scale, should not be unduly difficult to implement, and should not unnecessarily restrict implementation choices.
- Interoperability solutions should be able to work with any vendors' existing ORB implementations with respect to their CORBA-compliant core feature set, those implementations are diverse.
- All operations implied by the CORBA object model (i.e., the stringify and destringify operations defined on the **CORBA:ORB** pseudo-object and all the operations on **CORBA:Object**) as well as type management (e.g., narrowing, as needed by the C++ mapping) should be supported.

6.6.1 Non-Goals

The following were taken into account, but were not goals:

- Support for security
- Support for future ORB Services

7 ORB Interoperability Architecture

7.1 Overview

The original Interoperability RFP defines interoperability as the ability for a client on ORB A to invoke an OMG IDL-defined operation on an object on ORB B, where ORB A and ORB B are independently developed. It further identifies general requirements including in particular:

- Ability for two vendors' ORBs to interoperate without prior knowledge of each other's implementation.
- Support of all ORB functionality.
- Preservation of content and semantics of ORB-specific information across ORB boundaries (for example, security).

In effect, the requirement is for invocations between client and server objects to be independent of whether they are on the same or different ORBs, and not to mandate fundamental modifications to existing ORB products.

7.1.1 Domains

The CORBA Object Model identifies various distribution transparencies that must be supported within a single ORB environment, such as location transparency. Elements of ORB functionality often correspond directly to such transparencies. Interoperability can be viewed as extending transparencies to span multiple ORBs.

In this architecture a *domain* is a distinct scope, within which certain common characteristics are exhibited and common rules are observed over which a distribution transparency is preserved. Thus, interoperability is fundamentally involved with transparently crossing such domain boundaries.

Domains tend to be either administrative or technological in nature, and need not correspond to the boundaries of an ORB installation. Administrative domains include naming domains, trust groups, resource management domains, and other "run-time" characteristics of a system. Technology domains identify common protocols, syntaxes, and similar "build-time" characteristics. In many cases, the need for technology domains derives from basic requirements of administrative domains.

Within a single ORB, most domains are likely to have similar scope to that of the ORB itself: common object references, network addresses, security mechanisms, and more. However, it is possible for there to be multiple domains of the same type supported by a given ORB: internal representation on different machine types, or security domains. Conversely, a domain may span several ORBs: similar network addresses may be used by different ORBs, type identifiers may be shared.

7.1.2 Bridging Domains

The abstract architecture describes ORB interoperability in terms of the translation required when an object request traverses domain boundaries. Conceptually, a mapping or *bridging mechanism* resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain.

The concrete architecture identifies two approaches to inter-ORB bridging:

- At application level, allowing flexibility and portability.
- At ORB level, built into the ORB itself.

7.2 ORBs and ORB Services

The ORB Core is that part of the ORB which provides the basic representation of objects and the communication of requests. The ORB Core therefore supports the minimum functionality to enable a client to invoke an operation on a server object, with (some of) the distribution transparencies required by *CORBA*.

An object request may have implicit attributes which affect the way in which it is communicated - though not the way in which a client makes the request. These attributes include security, transactional capabilities, recovery, and replication. These features are provided by "ORB Services," which will in some ORBs be layered as internal services over the core, or in other cases be incorporated directly into an ORB's core. It is an aim of this part of ISO/IEC 19500 to allow for new ORB Services to be defined in the future, without the need to modify or enhance this architecture.

Within a single ORB, ORB services required to communicate a request will be implemented and (implicitly) invoked in a private manner. For interoperability between ORBs, the ORB services used in the ORBs, and the correspondence between them, must be identified.

7.2.1 The Nature of ORB Services

ORB Services are invoked implicitly in the course of application-level interactions. ORB Services range from fundamental mechanisms such as reference resolution and message encoding to advanced features such as support for security, transactions, or replication.

An ORB Service is often related to a particular transparency. For example, message encoding – the marshaling and unmarshaling of the components of a request into and out of message buffers – provides transparency of the representation of the request. Similarly, reference resolution supports location transparency. Some transparencies, such as security, are supported by a combination of ORB Services and Object Services while others, such as replication, may involve interactions between ORB Services themselves.

ORB Services differ from Object Services in that they are positioned below the application and are invoked transparently to the application code. However, many ORB Services include components that correspond to conventional Object Services in that they are invoked explicitly by the application.

Security is an example of service with both ORB Service and normal Object Service components, the ORB components being those associated with transparently authenticating messages and controlling access to objects while the necessary administration and management functions resemble conventional Object Services.

7.2.2 ORB Services and Object Requests

Interoperability between ORBs extends the scope of distribution transparencies and other request attributes to span multiple ORBs. This requires the establishment of relationships between supporting ORB Services in the different ORBs.

In order to discuss how the relationships between ORB Services are established, it is necessary to describe an abstract view of how an operation invocation is communicated from client to server object.

1. The client generates an operation request, using a reference to the server object, explicit parameters, and an implicit invocation context. This is processed by certain ORB Services on the client path.

2. On the server side, corresponding ORB Services process the incoming request, transforming it into a form directly suitable for invoking the operation on the server object.

3. The server object performs the requested operation.

4. Any result of the operation is returned to the client in a similar manner.

The correspondence between client-side and server-side ORB Services need not be one-to-one and in some circumstances may be far more complex. For example, if a client application requests an operation on a replicated server, there may be multiple server-side ORB service instances, possibly interacting with each other.

In other cases, such as security, client-side or server-side ORB Services may interact with Object Services such as authentication servers.

7.2.3 Selection of ORB Services

The ORB Services used are determined by:

- Static properties of both client and server objects; for example, whether a server is replicated.
- Dynamic attributes determined by a particular invocation context; for example, whether a request is transactional.
- Administrative policies (e.g., security).

Within a single ORB, private mechanisms (and optimizations) can be used to establish which ORB Services are required and how they are provided. Service selection might in general require negotiation to select protocols or protocol options. The same is true between different ORBs: it is necessary to agree which ORB Services are used, and how each transforms the request. Ultimately, these choices become manifest as one or more protocols between the ORBs or as transformations of requests.

In principle, agreement on the use of each ORB Service can be independent of the others and, in appropriately constructed ORBs, services could be layered in any order or in any grouping. This potentially allows applications to specify selective transparencies according to their requirements, although at this time CORBA provides no way to penetrate its transparencies.

A client ORB must be able to determine which ORB Services must be used in order to invoke operations on a server object. Correspondingly, where a client requires dynamic attributes to be associated with specific invocations, or administrative policies dictate, it must be possible to cause the appropriate ORB Services to be used on client and server sides of the invocation path. Where this is not possible - because, for example, one ORB does not support the full set of services required - either the interaction cannot proceed or it can only do so with reduced facilities or transparencies.

7.3 Domains

From a computational viewpoint, the OMG Object Model identifies various distribution transparencies which ensure that client and server objects are presented with a uniform view of a heterogeneous distributed system. From an engineering viewpoint, however, the system is not wholly uniform. There may be distinctions of location and possibly many others such as processor architecture, networking mechanisms and data representations. Even when a single ORB implementation is used throughout the system, local instances may represent distinct, possibly optimized scopes for some aspects of ORB functionality.

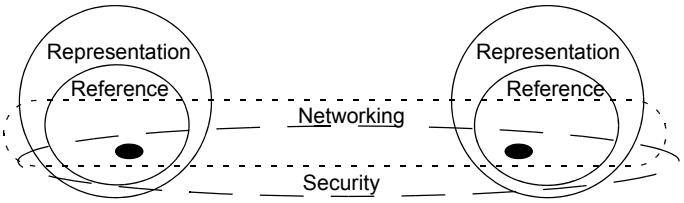


Figure 7.1 - Different Kinds of Domains can Coexist

Interoperability, by definition, introduces further distinctions, notably between the scopes associated with each ORB. To describe both the requirements for interoperability and some of the solutions, this architecture introduces the concept of *domains* to describe the scopes and their implications.

Informally, a domain is a set of objects sharing a common characteristic or abiding by common rules. It is a powerful modeling concept that can simplify the analysis and description of complex systems. There may be many types of domains (e.g., management domains, naming domains, language domains, and technology domains).

7.3.1 Definition of a Domain

Domains allow partitioning of systems into collections of components that have some characteristic in common. In this architecture a domain is a scope in which a collection of objects, said to be members of the domain, is associated with some common characteristic; any object for which the association does not exist, or is undefined, is not a member of the domain. A domain can be modeled as an object and may be itself a member of other domains.

It is the scopes themselves and the object associations or bindings defined within them which characterize a domain. This information is disjoint between domains. However, an object may be a member of several domains, of similar kinds as well as of different kinds, and so the sets of members of domains may overlap.

The concept of a domain boundary is defined as the limit of the scope in which a particular characteristic is valid or meaningful. When a characteristic in one domain is translated to an equivalent in another domain, it is convenient to consider it as traversing the boundary between the two domains.

Domains are generally either administrative or technological in nature. Examples of domains related to ORB interoperability issues are:

- Referencing domain – the scope of an object reference.
- Representation domain – the scope of a message transfer syntax and protocol.
- Network addressing domain – the scope of a network address.
- Network connectivity domain – the potential scope of a network message.
- Security domain – the extent of a particular security policy.
- Type domain – the scope of a particular type identifier.
- Transaction domain – the scope of a given transaction service.

Domains can be related in two ways: containment, where a domain is contained within another domain, and federation, where two domains are joined in a manner agreed to and set up by their administrators.

7.3.2 Mapping Between Domains: Bridging

Interoperability between domains is only possible if there is a well-defined mapping between the behaviors of the domains being joined. Conceptually, a mapping mechanism or bridge resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain. Note that the use of the term "bridge" in this context is conceptual and refers only to the functionality that performs the required mappings between distinct domains. There are several implementation options for such bridges and these are discussed elsewhere.

For full interoperability, it is essential that all the concepts used in one domain are transformable into concepts in other domains with which interoperability is required, or that if the bridge mechanism filters such a concept out, nothing is lost as far as the supported objects are concerned. In other words, one domain may support a superior service to others, but such a superior functionality will not be available to an application system spanning those domains.

A special case of this requirement is that the object models of the two domains need to be compatible. This part of ISO/IEC 19500 assumes that both domains are strictly compliant with the CORBA Object Model and the *CORBA* specifications. This includes the use of OMG IDL when defining interfaces, the use of the CORBA Core Interface Repository, and other modifications that were made to *CORBA*. Variances from this model could easily compromise some aspects of interoperability.

7.4 Interoperability Between ORBs

An ORB "provides the mechanisms by which objects transparently make and receive requests and responses. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments..." ORB interoperability extends this definition to cases in which client and server objects on different ORBs "transparently make and receive requests."

Note that a direct consequence of this transparency requirement is that bridging must be bidirectional: that is, it must work as effectively for object references passed as parameters as for the target of an object invocation. Were bridging unidirectional (e.g., if one ORB could only be a client to another), then transparency would not have been provided because object references passed as parameters would not work correctly: ones passed as "callback objects," for example, could not be used.

Without loss of generality, most of this text focuses on bridging in only one direction. This is purely to simplify discussions, and does not imply that unidirectional connectivity satisfies basic interoperability requirements.

7.4.1 ORB Services and Domains

In this architecture, different aspects of ORB functionality - ORB Services - can be considered independently and associated with different domain types. The architecture does not, however, prescribe any particular decomposition of ORB functionality and interoperability into ORB Services and corresponding domain types. There is a range of possibilities for such a decomposition:

1. The simplest model, for interoperability, is to treat all objects supported by one ORB (or, alternatively, all ORBs of a given type) as comprising one domain. Interoperability between any pair of different domains (or domain types) is then achieved by a specific all-encompassing bridge between the domains. (This is all *CORBA* implies.)

2. More detailed decompositions would identify particular domain types - such as referencing, representation, security, and networking. A core set of domain types would be pre-determined and allowance made for additional domain types to be defined as future requirements dictate (e.g., for new ORB Services).

7.4.2 ORBs and Domains

In many respects, issues of interoperability between ORBs are similar to those which can arise with a single type of ORB (e.g., a product). For example:

- Two installations of the ORB may be installed in different security domains, with different Principal identifiers. Requests crossing those security domain boundaries will need to establish locally meaningful Principals for the caller identity, and for any Principals passed as parameters.
- Different installations might assign different type identifiers for equivalent types, and so requests crossing type domain boundaries would need to establish locally meaningful type identifiers (and perhaps more).

Conversely, not all of these problems need to appear when connecting two ORBs of a different type (e.g., two different products). Examples include:

- They could be administered to share user visible naming domains, so that naming domains do not need bridging.
- They might reuse the same networking infrastructure, so that messages could be sent without needing to bridge different connectivity domains.

Additional problems can arise with ORBs of different types. In particular, they may support different concepts or models, between which there are no direct or natural mappings. CORBA only specifies the application level view of object interactions, and requires that distribution transparencies conceal a whole range of lower level issues. It follows that within any particular ORB, the mechanisms for supporting transparencies are not visible at the application-level and are entirely a matter of implementation choice. So there is no guarantee that any two ORBs support similar internal models or that there is necessarily a straightforward mapping between those models.

These observations suggest that the concept of an ORB (instance) is too coarse or superficial to allow detailed analysis of interoperability issues between ORBs. Indeed, it becomes clear that an ORB instance is an elusive notion: it can perhaps best be characterized as the intersection or coincidence of ORB Service domains.

7.4.3 Interoperability Approaches

When an interaction takes place across a domain boundary, a mapping mechanism, or bridge, is required to transform relevant elements of the interaction as they traverse the boundary. There are essentially two approaches to achieving this: mediated bridging and immediate bridging. These approaches are described in the following sub clauses.

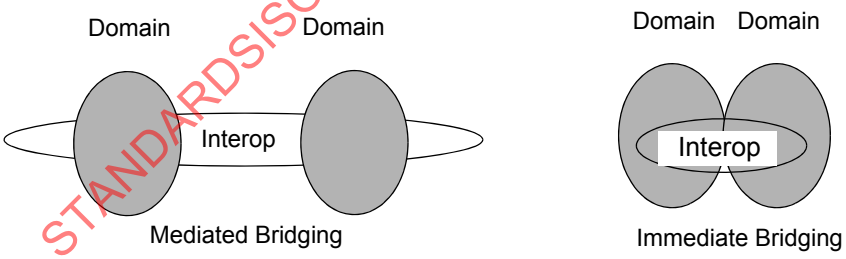


Figure 7.2 - Two bridging techniques, different uses of an intermediate form agreed on between the two domains.

7.4.3.1 Mediated Bridging

With mediated bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, between the internal form of that domain and an agreed, common form.

Observations on mediated bridging are as follows:

- The scope of agreement of a common form can range from a private agreement between two particular ORB/domain implementations to a universal standard.
- There can be more than one common form, each oriented or optimized for a different purpose.
- If there is more than one possible common form, then which is used can be static (e.g., administrative policy agreed between ORB vendors, or between system administrators) or dynamic (e.g., established separately for each object, or on each invocation).
- Engineering of this approach can range from in-line specifically compiled (compare to stubs) or generic library code (such as encryption routines), to intermediate bridges to the common form.

7.4.3.2 Immediate Bridging

With immediate bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, directly between the internal form of one domain and the internal form of the other.

Observations on immediate bridging are as follows:

- This approach has the potential to be optimal (in that the interaction is not mediated via a third party, and can be specifically engineered for each pair of domains) but sacrifices flexibility and generality of interoperability to achieve this.
- This approach is often applicable when crossing domain boundaries that are purely administrative (i.e., there is no change of technology). For example, when crossing security administration domains between similar ORBs, it is not necessary to use a common intermediate standard.

As a general observation, the two approaches can become almost indistinguishable when private mechanisms are used between ORB/domain implementations.

7.4.3.3 Location of Inter-Domain Functionality

Logically, an inter-domain bridge has components in both domains, whether the mediated or immediate bridging approach is used. However, domains can span ORB boundaries and ORBs can span machine and system boundaries; conversely, a machine may support, or a process may have access to more than one ORB (or domain of a given type). From an engineering viewpoint, this means that the components of an inter-domain bridge may be dispersed or co-located, with respect to ORBs or systems. It also means that the distinction between an ORB and a bridge can be a matter of perspective: there is a duality between viewing inter-system messaging as belonging to ORBs, or to bridges.

For example, if a single ORB encompasses two security domains, the inter-domain bridge could be implemented wholly within the ORB and thus be invisible as far as ORB interoperability is concerned. A similar situation arises when a bridge between two ORBs or domains is implemented wholly within a process or system that has access to both. In such cases, the engineering issues of inter-domain bridging are confined, possibly to a single system or process. If it were practical to implement all bridging in this way, then interactions between systems or processes would be solely within a single domain or ORB.

7.4.3.4 Bridging Level

As noted at the start of this sub clause, bridges may be implemented both internally to an ORB and as layers above it. These are called respectively “in-line” and “request-level” bridges.

Request-level bridges use the CORBA APIs, including the Dynamic Skeleton Interface, to receive and issue requests. However, there is an emerging class of “implicit context” which may be associated with some invocations, holding ORB Service information such as transaction and security context information, which is not at this time exposed through general purpose public APIs. (Those APIs expose only OMG IDL-defined operation parameters, not implicit ones.)

Rather, the precedent set with the Transaction Service is that special purpose APIs are defined to allow bridging of each kind of context. This means that request-level bridges must be built to specifically understand the implications of bridging such ORB Service domains, and to make the appropriate API calls.

7.4.4 Policy-Mediated Bridging

An assumption made through most of this part of ISO/IEC 19500 is that the existence of domain boundaries should be transparent to requests: that the goal of interoperability is to hide such boundaries. However, if this were always the goal, then there would be no real need for those boundaries in the first place.

Realistically, administrative domain boundaries exist because they reflect ongoing differences in organizational policies or goals. Bridging the domains will in such cases require *policy mediation*. That is, inter-domain traffic will need to be constrained, controlled, or monitored; fully transparent bridging may be highly undesirable. Resource management policies may even need to be applied, restricting some kinds of traffic during certain periods.

Security policies are a particularly rich source of examples: a domain may need to audit external access, or to provide domain-based access control. Only a very few objects, types of objects, or classifications of data might be externally accessible through a "firewall."

Such policy-mediated bridging requires a bridge that knows something about the traffic being bridged. It could in general be an application-specific policy, and many policy-mediated bridges could be parts of applications. Those might be organization-specific, off-the-shelf, or anywhere in between.

Request-level bridges, which use only public ORB APIs, easily support the addition of policy mediation components, without loss of access to any other system infrastructure that may be needed to identify or enforce the appropriate policies.

7.4.5 Configurations of Bridges in Networks

In the case of network-aware ORBs, we anticipate that some ORB protocols will be more frequently bridged to than others, and so will begin to serve the role of "backbone ORBs." (This is a role that the IIOP is specifically expected to serve.) This use of "backbone topology" is true both on a large scale and a small scale. While a large scale public data network provider could define its own backbone ORB, on a smaller scale, any given institution will probably designate one commercially available ORB as its backbone.

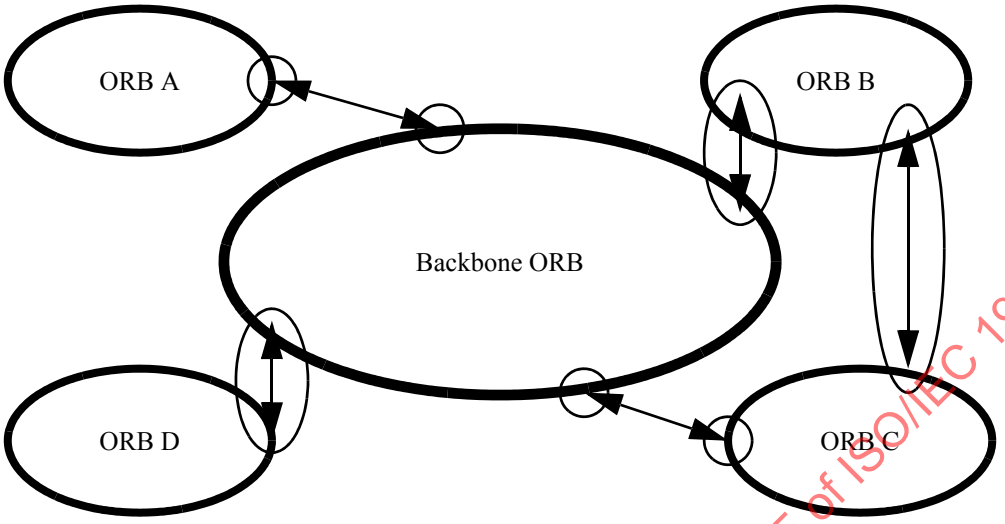


Figure 7.3 - An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

Adopting a backbone style architecture is a standard administrative technique for managing networks. It has the consequence of minimizing the number of bridges needed, while at the same time making the ORB topology match typical network organizations. (That is, it allows the number of bridges to be proportional to the number of protocols, rather than combinatorial.)

In large configurations, it will be common to notice that adding ORB bridges doesn't even add any new "hops" to network routes, because the bridges naturally fit in locations where connectivity was already indirect, and augment or supplant the existing network firewalls.

7.5 Object Addressing

The Object Model in "The Object Model" clause of Part 1 of this International Standard (ISO/IEC 19500-1) defines an object reference as an object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references.

The fundamental ORB interoperability requirement is to allow clients to use such object names to invoke operations on objects in other ORBs. Clients do not need to distinguish between references to objects in a local ORB or in a remote one. Providing this transparency can be quite involved, and naming models are fundamental to it.

This sub-clause discusses models for naming entities in multiple domains, and transformations of such names as they cross the domain boundaries. That is, it presents transformations of object reference information as it passes through networks of inter-ORB bridges. It uses the word "ORB" as synonymous with referencing domain; this is purely to simplify the discussion. In other contexts, "ORB" can usefully denote other kinds of domain.

7.5.1 Domain-relative Object Referencing

Since CORBA does not require ORBs to understand object references from other ORBs, when discussing object references from multiple ORBs one must always associate the object reference's domain (ORB) with the object reference. We use the notation *D0.R0* to denote an object reference *R0* from domain *D0*; this is itself an object reference. This is called "domain-relative" referencing (or addressing) and need not reflect the implementation of object references within any ORB.

At an implementation level, associating an object reference with an ORB is only important at an inter-ORB boundary; that is, inside a bridge. This is simple, since the bridge knows from which ORB each request (or response) came, including any object references embedded in it.

7.5.2 Handling of Referencing Between Domains

When a bridge hands an object reference to an ORB, it must do so in a form understood by that ORB: the object reference must be in the recipient ORB's native format. Also, in cases where that object originated from some other ORB, the bridge must associate each newly created "proxy" object reference with (what it sees as) the original object reference.

Several basic schemes to solve these two problems exist. These all have advantages in some circumstances; all can be used, and in arbitrary combination with each other, since CORBA object references are opaque to applications. The ramifications of each scheme merits attention, with respect to scaling and administration. The schemes include:

1. *Object Reference Translation Reference Embedding*: The bridge can store the original object reference itself, and pass an entirely different proxy reference into the new domain. The bridge must then manage state on behalf of each bridged object reference, map these references from one ORB's format to the other's, and vice versa.

2. *Reference Encapsulation*: The bridge can avoid holding any state at all by conceptually concatenating a domain identifier to the object name. Thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1... D4* it could be identified in *D4* as proxy reference *d3.d2.d1.d0.R*, where *dn* is the address of *Dn* relative to *Dn+1*.

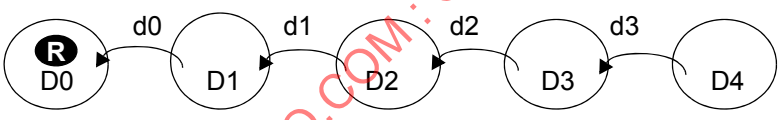


Figure 7.4 - Reference encapsulation adds domain information during bridging

3. *Domain Reference Translation*: Like object reference translation, this scheme holds some state in the bridge. However, it supports sharing that state between multiple object references by adding a domain-based route identifier to the proxy (which still holds the original reference, as in the reference encapsulation scheme). It achieves this by providing encoded domain route information each time a domain boundary is traversed; thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1...D4* it would be identified in *D4* as *(d3, x3).R*, and in *D2* as *(d1,x1).R*, and so on, where *dn* is the address of *Dn* relative to *Dn+1*, and *xn* identifies the pair *(dn-1, xn-1)*.

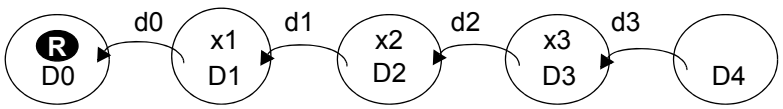


Figure 7.5 - Domain Reference Translation *substitutes domain references* during bridging

4. *Reference Canonicalization:* This scheme is like domain reference translation, except that the proxy uses a "well-known" (e.g., global) domain identifier rather than an encoded path. Thus a reference *R*, originating in domain *D0* would be identified in other domains as *D0.R*.

Observations about these approaches to inter-domain reference handling are as follows:

- Naive application of reference encapsulation could lead to arbitrarily large references. A "topology service" could optimize cycles within any given encapsulated reference and eliminate the appearance of references to local objects as alien references.
- A topology service could also optimize the chains of routes used in the domain reference translation scheme. Since the links in such chains are re-used by any path traversing the same sequence of domains, such optimization has particularly high leverage.
- With the general purpose APIs defined in *CORBA*, object reference translation can be supported even by ORBs not specifically intended to support efficient bridging, but this approach involves the most state in intermediate bridges. As with reference encapsulation, a topology service could optimize individual object references. (APIs are defined by the Dynamic Skeleton Interface and Dynamic Invocation Interface.)
- The chain of addressing links established with both object and domain reference translation schemes must be represented as state within the network of bridges. There are issues associated with managing this state.
- Reference canonicalization can also be performed with managed hierarchical name spaces such as those now in use on the Internet and X.500 naming.

7.6 An Information Model for Object References

This sub clause provides a simple, powerful information model for the information found in an object reference. That model is intended to be used directly by developers of bridging technology, and is used in that role by the IIOP, described in the *General Inter-ORB Protocol* clause of this standard, *Object References*.

7.6.1 What Information Do Bridges Need?

The following potential information about object references has been identified as critical for use in bridging technologies:

- *Is it null?* Nulls only need to be transmitted and never support operation invocation.
- *What type is it?* Many ORBs require knowledge of an object's type in order to efficiently preserve the integrity of their type systems.
- *What protocols are supported?* Some ORBs support objrefs that in effect live in multiple referencing domains, to allow clients the choice of the most efficient communications facilities available.
- *What ORB Services are available?* As noted in Selection of ORB Services on page 17, several different ORB Services might be involved in an invocation. Providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

7.6.2 Interoperable Object References: IORs

To provide the information above, an "Interoperable Object Reference," (IOR) data structure has been provided. This data structure need not be used internally to any given ORB, and is not intended to be visible to application-level ORB programmers. It should be used only when crossing object reference domain boundaries, within bridges.

This data structure is designed to be efficient in typical single-protocol configurations, while not penalizing multiprotocol ones.

```
module IOP {                                        // IDL

    // Standard Protocol Profile tag values

    typedef unsigned long           ProfileId;

    typedef CORBA::OctetSeq ProfileData;
    struct TaggedProfile {
        ProfileId                   tag;
        ProfileData                 profile_data;
    };
    typedef sequence <TaggedProfile> TaggedProfileSeq ;

    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.

    struct IOR {
        string                      type_id;
        TaggedProfileSeq            profiles;
    };

    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedProfile.

    typedef unsigned long ComponentId;
    typedef CORBA::OctetSeq ComponentData;

    struct TaggedComponent {
        ComponentId                 tag;
        ComponentData               component_data;
    };

    typedef sequence<TaggedComponent> TaggedComponentSeq;
    typedef sequence <TaggedComponent> MultipleComponentProfile;
    typedef CORBA::OctetSeq ObjectKey;
};
```

7.6.3 IOR Profiles

Object references have at least one *tagged profile*. Each profile supports one or more protocols and encapsulates all the basic information the protocols it supports need to identify an object. Any single profile holds enough information to drive a complete invocation using any of the protocols it supports; the content and structure of those profile entries are wholly specified by these protocols.

When a specific protocol is used to convey an object reference passed as a parameter in an IDL operation invocation (or reply), an IOR which reflects, in its contained profiles, the full protocol understanding of the operation client (or server in case of reply) may be sent. A receiving ORB which operates (based on topology and policy information available to it) on

profiles rather than the received IOR as a whole, to create a derived reference for use in its own domain of reference, is placing itself as a bridge between reference domains. Interoperability inhibiting situations can arise when an orb sends an IOR with multiple profiles (using one of its supported protocols) to a receiving orb, and that receiving orb later returns a derived reference to that object, which has had profiles or profile component data removed or transformed from the original IOR contents.

To assist in classifying behavior of ORBS in such bridging roles, two classes of IOR conformance may be associated with the conformance requirements for a given ORB interoperability protocol:

- Full IOR conformance requires that an orb which receives an IOR for an object passed to it through that ORB interoperability protocol, shall recover the original IOR, in its entirety, for passing as a reference to that object from that orb through that same protocol.
- Limited-Profile IOR conformance requires that an orb which receives an IOR passed to it through a given ORB interoperability protocol, shall recover all of the standard information contained in the IOR profile for that protocol, whenever passing a reference to that object, using that same protocol, to another ORB.

NOTE: Conformance to IIOP versions 1.0, 1.1, and 1.2 only requires support of limited-Profile IOR conformance, specifically for the IIOP IOR profile. However, due to interoperability problems induced by Limited-Profile IOR conformance, it is now deprecated by the CORBA 2.4 specification for an orb to not support Full IOR conformance. Some future IIOP versions could require Full IOR conformance.

An ORB may be unable to use any of the profiles provided in an IOR for various reasons which may be broadly categorized as transient ones like temporary network outage, and non-transient ones like unavailability of appropriate protocol software in the ORB. The decision about the category of outage that causes an ORB to be unable to use any profile from an IOR is left up to the ORB. At an appropriate point, when an ORB discovers that it is unable to use any profile in an IOR, depending on whether it considers the reason transient or non-transient, it should raise the standard system exception TRANSIENT with standard minor code 2, or IMP_LIMIT with the standard minor code 1.

Each profile has a unique numeric tag, assigned by the OMG. The ones defined here are for the IIOP (see clause 9.8.3, IIOP IOR Profile Components, on page 114) and for use in “multiple component profiles.” Profile tags in the range **0x80000000** through **0xffffffff** are reserved for future use, and are not currently available for assignment.

Null object references are indicated by an empty set of profiles, and by a “Null” type ID (a string that contains only a single terminating character). Type IDs may only be “Null” in any message, requiring the client to use existing knowledge or to consult the object, to determine interface types supported. The type ID is a Repository ID identifying the interface type, and is provided to allow ORBs to preserve strong typing. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability specification, needs to have a much broader scope to address various problems in system evolution and maintenance. Type IDs support detection of type equivalence, and in conjunction with an Interface Repository, allow processes to reason about the relationship of the type of the object referred to and any other type.

The type ID, if provided by the server, indicates the most derived type that the server wishes to publish, at the time the reference is generated. The object’s actual most derived type may later change to a more derived type. Therefore, the type ID in the IOR can only be interpreted by the client as a hint that the object supports at least the indicated interface. The client can succeed in narrowing the reference to the indicated interface, or to one of its base interfaces, based solely on the type ID in the IOR, but must not fail to narrow the reference without consulting the object via the “_is_a” or “_get_interface” pseudo-operations.

ORBs claiming to support the Full-IOR conformance are required to preserve all the semantic content of any IOR (including the ordering of each profile and its components), and may only apply transformations which preserve semantics (e.g., changing Byte order for encapsulation).

For example, consider an echo operation for object references:

```
interface Echoer {Object echo(in Object o);};
```

Assume that the method body implementing this “echo” operation simply returns its argument. When a client application invokes the echo operation and passes an arbitrary object reference, if both the client and server ORBs claim support to Full IOR conformance, the reference returned by the operation is guaranteed to have not been semantically altered by either client or server ORB. That is, all its profiles will remain intact and in the same order as they were present when the reference was sent. This requirement for ORBs which claim support for Full-IOR conformance, ensures that, for example, a client can safely store an object reference in a naming service and get that reference back again later without losing information inside the reference.

7.6.4 Standard IOR Profiles

```
module IOP {
    const ProfileId          TAG_INTERNET_IOP = 0;
    const ProfileId          TAG_MULTIPLE_COMPONENTS = 1;
    const ProfileId          TAG_SCCP_IOP = 2;
    const ProfileId          TAG_UIPMC = 3;
};
```

7.6.4.1 The TAG_INTERNET_IOP Profile

The **TAG_INTERNET_IOP** tag identifies profiles that support the Internet Inter-ORB Protocol. The **ProfileBody** of this profile, described in detail in IIOP IOR Profiles on page 112, contains a CDR encapsulation of a structure containing addressing and object identification information used by IIOP. Version 1.1 of the **TAG_INTERNET_IOP** profile also includes a **sequence<TaggedComponent>** that can contain additional information supporting optional IIOP features, ORB services such as security, and future protocol extensions.

Protocols other than IIOP (such as ESIOPs and other GIOPs) can share profile information (such as object identity or security information) with IIOP by encoding their additional profile information as components in the **TAG_INTERNET_IOP** profile. All **TAG_INTERNET_IOP** profiles support IIOP, regardless of whether they also support additional protocols. Interoperable ORBs are not required to create or understand any other profile, nor are they required to create or understand any of the components defined for other protocols that might share the **TAG_INTERNET_IOP** profile with IIOP.

The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in IIOP IOR Profiles on page 112.

7.6.4.2 The TAG_MULTIPLE_COMPONENTS Profile

The **TAG_MULTIPLE_COMPONENTS** tag indicates that the value encapsulated is of type **MultipleComponentProfile**. In this case, the profile consists of a list of protocol components, the use of which must be specified by the protocol using this profile. This profile may be used to carry IOR components, as specified in IOR Components on page 29.

The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type shown above.

7.6.4.3 The TAG_SCCP_IOP Profile

See the OMG specification [SCCP] and Annex A of Part 2 of this International Standard for additional information.

7.6.4.4 The TAG_UIPMC Profile

The **TAG_UIPMC** tag is used by MIOP. See clause 11 of this Part of this International standard and Annex A of Part 2 of this International Standard for additional information.

7.6.5 IOR Components

TaggedComponents contained in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles are identified by unique numeric tags using a namespace distinct form that is used for profile tags. Component tags are assigned by the OMG.

Specifications of components must include the following information:

- *Component ID*: The compound tag that is obtained from OMG.
- *Structure and encoding*: The syntax of the component data and the encoding rules. If the component value is encoded as a CDR encapsulation, the IDL type that is encapsulated and the GIOP version which is used for encoding the value, if different than GIOP 1.0, must be specified as part of the component definition.
- *Semantics*: How the component data is intended to be used.
- *Protocols*: The protocol for which the component is defined, and whether it is intended that the component be usable by other protocols.
- *At most once*: whether more than one instance of this component can be included in a profile.

Specifications of protocols must describe how the components affect the protocol. In addition, a protocol definition must specify, for each TaggedComponent, whether inclusion of the component in profiles supporting the protocol is required (MANDATORY PRESENCE) or not required (OPTIONAL PRESENCE). An ORB claiming to support Full-IOR conformance shall not drop optional components, once they have been added to a profile.

7.6.6 Standard IOR Components

The following are standard IOR components that can be included in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles, and may apply to IIOP, other GIOPs, ESIOPs, or other protocols. An ORB must not drop these components from an existing IOR.

```
module IOP {
    const ComponentId   TAG_ORB_TYPE = 0;
    const ComponentId   TAG_CODE_SETS = 1;
    const ComponentId   TAG_POLICIES = 2;
    const ComponentId   TAG_ALTERNATE_IIOP_ADDRESS = 3;

    const ComponentId   TAG_ASSOCIATION_OPTIONS = 13;
    const ComponentId   TAG_SEC_NAME = 14;
    const ComponentId   TAG_SPKM_1_SEC_MECH = 15;
    const ComponentId   TAG_SPKM_2_SEC_MECH = 16;
    const ComponentId   TAG_KerberosV5_SEC_MECH = 17;
    const ComponentId   TAG_CSI_ECMA_Secret_SEC_MECH = 18;
    const ComponentId   TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
    const ComponentId   TAG_SSL_SEC_TRANS = 20;
    const ComponentId   TAG_CSI_ECMA_Public_SEC_MECH = 21;
    const ComponentId   TAG_ GENERIC_SEC_MECH = 22;
```

```
    const ComponentId    TAG_FIREWALL_TRANS = 23;
    const ComponentId    TAG_SCCP_CONTACT_INFO = 24;
    const ComponentId    TAG_JAVA_CODEBASE = 25;
    const ComponentId    TAG_TRANSACTION_POLICY = 26;
    const ComponentId    TAG_MESSAGE_ROUTERS = 30;
    const ComponentId    TAG_OTS_POLICY = 31;
    const ComponentId    TAG_INV_POLICY = 32;
    const ComponentId    TAG_CSI_SEC_MECH_LIST = 33;
    const ComponentId    TAG_NULL_TAG = 34;
    const ComponentId    TAG_SECIOP_SEC_TRANS = 35;
    const ComponentId    TAG_TLS_SEC_TRANS = 36;
    const ComponentId    TAG_ACTIVITY_POLICY = 37;
    const ComponentId    TAG_RMI_CUSTOM_MAX_STREAM_FORMAT = 38;
    const ComponentId    TAG_GROUP = 39;
    const ComponentId    TAG_GROUP_IIOP = 40;
    const ComponentId    TAG_PASSTHRU_TRANS = 41;
    const ComponentId    TAG_FIREWALL_PATH = 42;
    const ComponentId    TAG_IIOP_SEC_TRANS = 43;

    const ComponentId    TAG_INET_SEC_TRANS = 123;
};
```

7.6.6.1 TAG_ORB_TYPE Component

It is often useful in the real world to be able to identify the particular kind of ORB an object reference is coming from, to work around problems with that particular ORB, or exploit shared efficiencies.

The **TAG_ORB_TYPE** component has an associated value of type **unsigned long**, encoded as a CDR encapsulation, designating an ORB type ID allocated by the OMG for the ORB type of the originating ORB. Anyone may register any ORB types by submitting a short (one-paragraph) description of the ORB type to the OMG, and will receive a new ORB type ID in return. A list of ORB type descriptions and values will be made available on the OMG web server.

The **TAG_ORB_TYPE** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.1 or greater, it is optionally present.

7.6.6.2 TAG_ALTERNATE_IIOP_ADDRESS Component

In cases where the same object key is used for more than one internet location, the following standard IOR Component is defined for support in IIOP version 1.2.

The **TAG_ALTERNATE_IIOP_ADDRESS** component has an associated value of type:

```
struct {
    string HostID,
    unsigned short Port
};
```

encoded as a CDR encapsulation.

Zero or more instances of the **TAG_ALTERNATE_IIOP_ADDRESS** component type may be included in a version 1.2 **TAG_INTERNET_IOP** Profile. Each of these alternative addresses may be used by the client orb, in addition to the host and port address expressed in the body of the Profile. In cases where one or more **TAG_ALTERNATE_IIOP_ADDRESS** components are present in a **TAG_INTERNET_IOP** Profile, no order of use is prescribed by Version 1.2 of IIOP.

7.6.6.3 Other Components

The following standard components are specified in various OMG specifications:

- **TAG_CODE_SETS** - See CodeSet Component of IOR Multi-Component Profile in clause 7.10.2.4 in this part of this International Standard.
- **TAG_POLICIES** - See the "CORBA Messaging" clause of ISO/IEC 19500-1.
- **TAG_SEC_NAME** - See the Mechanism Tags sub clause of [CORBASEC].
- **TAG_ASSOCIATION_OPTIONS** - See the Tag Association Options sub clause of [CORBASEC].
- **TAG_SSL_SEC_TRANS** - See the Mechanism Tags sub clause of [CORBASEC].
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form TAG_*_SEC_MECH - See the "Mechanism Tags" sub clause of [CORBASEC].
- **TAG_FIREWALL_SEC** - See [FIREWALL].
- **TAG_SCCP_CONTACT_INFO** - See [SCCP].
- **TAG_JAVA_CODEBASE** - See [JAV2I].
- **TAG_TRANSACTION_POLICY** - See [TRANS].
- **TAG_MESSAGE_ROUTERS** - See the "CORBA Messaging" clause of ISO/IEC 19500-1.
- **TAG_OTS_POLICY** - See [TRANS].
- **TAG_INV_POLICY** - See [TRANS].
- **TAG_INET_SEC_TRANS** - See [CORBASEC].
- **TAG_CSI_SEC_MECH_LIST, TAG_NULL_TAG, TAG_SECIOP_SEC_TRANS, TAG_TLS_SEC_TRANS** - See the "Secure Interoperability" clause 10 in this part of this International Standard.
- **TAG_ACTIVITY_POLICY** - See [ASMOTS].
- **TAG_RMI_CUSTOM_MAX_STREAM_FORMAT** - See [JAV2I].
- **TAG_GROUP** and **TAG_GROUP_IIOP** - See the "Unreliable Multicast Inter-ORB Protocol" clause 11 in this part of this International Standard.
- **TAG_IIOP_SEC_TRANS** - See the "Secure Interoperability" clause 10 in this part of this International Standard.

7.6.7 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in another profile.
2. Any invocation uses information from exactly one profile.
3. Information used to drive multiple inter-ORB protocols may coexist within a single profile, possibly with some information (e.g., components) shared between the protocols, as specified by the specific protocols.

4. Unless otherwise specified in the definition of a particular profile, multiple profiles with the same profile tag may be included in an IOR.
5. Unless otherwise specified in the definition of a particular component, multiple components with the same component tag may be part of a given profile within an IOR.
6. A **TAG_MULTIPLE_COMPONENTS** profile may hold components shared between multiple protocols. Multiple such profiles may exist in an IOR.
7. The definition of each protocol using a **TAG_MULTIPLE_COMPONENTS** profile must specify which components it uses, and how it uses them.
8. Profile and component definitions can be either public or private. Public definitions are those whose tag and data format is specified in OMG documents. For private definitions, only the tag is registered with OMG.
9. Public component definitions shall state whether or not they are intended for use by protocols other than the one(s) for which they were originally defined, and dependencies on other components.

The OMG is responsible for allocating and registering protocol and component tags. Neither allocation nor registration indicates any "standard" status, only that the tag will not be confused with other tags. Requests to allocate tags should be sent to *tag_request@omg.org*.

7.6.8 IOR Creation and Scope

IORs are created from object references when required to cross some kind of referencing domain boundary. ORBs will implement object references in whatever form they find appropriate, including possibly using the IOR structure. Bridges will normally use IORs to mediate transfers where that standard is appropriate.

7.6.9 Stringified Object References

Object references can be "stringified" (turned into an external string form) by the **ORB::object_to_string** operation, and then "destringified" (turned back into a programming environment's object reference representation) using the **ORB::string_to_object** operation.

There can be a variety of reasons why being able to parse this string form might *not* help make an invocation on the original object reference:

- Identifiers embedded in the string form can belong to a different domain than the ORB attempting to destringify the object reference.
- The ORBs in question might not share a network protocol, or be connected.
- Security constraints may be placed on object reference destringification.

Nonetheless, there is utility in having a defined way for ORBs to generate and parse stringified IORs, so that in some cases an object reference stringified by one ORB could be destringified by another.

To allow a stringified object reference to be internalized by what may be a different ORB, a stringified IOR representation is specified. This representation instead establishes that ORBs could parse stringified object references using that format. This helps address the problem of bootstrapping, allowing programs to obtain and use object references, even from different ORBs.

The following is the representation of the stringified (externalized) IOR:

```
(1)  <oref> ::= <prefix> <hex_Octets>
(2)  <prefix> ::= <i><o><r>":"
(3)  <hex_Octets> ::= <hex_Octet> {<hex_Octet>}*
(4)  <hex_Octet>::= <hexDigit> <hexDigit>
(5)  <hexDigit> ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
(6)  <digit> ::= "0" | "1" | "2" | "3" | "4" | "5" |
(7)                          "6" | "7" | "8" | "9"
(8)  <a>::= "a" | "A"
(9)  <b>::= "b" | "B"
(10) <c>::= "c" | "C"
(11) <d>::= "d" | "D"
(12) <e>::= "e" | "E"
(13) <f>::= "f" | "F"
(14) <i>:: ="i" | "I"
(15) <o>:: ="o" | "O"
(16) <r>:: ="r" | "R"
```

NOTE: The case for characters in a stringified IOR is not significant.

The hexadecimal strings are generated by first turning an object reference into an IOR, and then encapsulating the IOR using the encoding rules of CDR, as specified in GIOP 1.0. (See clause 9.4, CDR Transfer Syntax, on page 71 for more information.) The content of the encapsulated IOR is then turned into hexadecimal digit pairs, starting with the first octet in the encapsulation and going until the end. The high four bits of each octet are encoded as a hexadecimal digit, then the low four bits.

7.6.10 Object URLs

To address the problem of bootstrapping and allow for more convenient exchange of human-readable object references, **ORB::string_to_object** allows URLs in the **corbaloc** and **corbaname** formats to be converted into object references.

If conversion fails, **string_to_object** raises a BAD_PARAM exception with one of following standard minor codes, as appropriate:

Minor Code	Description
7	string_to_object conversion failed due to bad scheme name
8	string_to_object conversion failed due to bad address
9	string_to_object conversion failed due to bad schema specific part
10	string_to_object conversion failed due to non-specific reason

7.6.10.1 corbaloc URL

The **corbaloc** URL scheme provides stringified object references that are more easily manipulated by users than **IOR** URLs. Currently, **corbaloc** URLs denote objects that can be contacted by IIOP or **resolve_initial_references**. Other transport protocols can be explicitly specified when they become available. Examples of IIOP and **resolve_initial_references** (**rir:**) based **corbaloc** URLs are:

corbaloc::555xyz.com/Prod/TradingService
corbaloc:iiop:1.1@555xyz.com/Prod/TradingService
corbaloc::555xyz.com,:556xyz.com:80/Dev/NameService
corbaloc:rir:/TradingService
corbaloc:rir:/NameService
corbaloc:iiop:192.168.14.25:555/NameService
corbaloc::[1080::8:800:200C:417A]:88/DefaultEventChannel

A **corbaloc** URL contains one or more:

- protocol identifiers
- protocol specific components such as address and protocol version information

When the **rir** protocol is used, no other protocols are allowed. After the addressing information, a **corbaloc** URL ends with a single object key. The full syntax is:

```
<corbaloc>          = "corbaloc:"<obj_addr_list>["/"<key_string>]
<obj_addr_list>     = [<obj_addr> ","]* <obj_addr>
<obj_addr>          = <prot_addr> | <future_prot_addr>
<prot_addr>         = <rir_prot_addr> | <iiop_prot_addr>

<rir_prot_addr>     = <rir_prot_token>":"
<rir_prot_token>    = "rir"

<iiop_prot_addr>    = <iiop_id><iiop_addr>
<iiop_id>           = ":" | <iiop_prot_token>":"
<iiop_prot_token>    = "iiop"
<iiop_addr>         = [<version> <host> [":" <port>]]
<host>              = DNS_style_host_name | ip_address
<host>              = DNS_style_host_name | IPv4_address
                        | "[" IPv6_address "]"
<version>           = <major> "." <minor> "@" | empty_string
<port>              = number
<major>             = number
<minor>             = number

<future_prot_addr>  = <future_prot_id><future_prot_addr>
<future_prot_id>    = <future_prot_token>":"
<future_prot_token> = possible examples: "atm" | "dce"
<future_prot_addr>  = protocol specific address

<key_string>        = <string> | empty_string
```

Where:

obj_addr_list: comma-separated list of protocol id, version, and address information. This list is used in an implementation-defined manner to address the object An object may be contacted by any of the addresses and protocols.

NOTE: If the **rir** protocol is used, no other protocols are allowed.

obj_addr: A protocol identifier, version tag, and a protocol specific address. The comma ',' and '/' characters are specifically prohibited in this component of the URL.

rir_prot_addr: resolve_initial_references protocol identifier. This protocol does not have a version tag or address. See 7.6.10.2, 'corbaloc:rir URL'.

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address. See corbaloc:iiop URL on page 35 for the iiop specific definitions.

future_prot_addr: a placeholder for future **corbaloc** protocols.

future_prot_id: token representing a protocol terminated with a ":".

future_prot_token: token representing a protocol. Currently only "**iiop**" and **"rir"** are defined.

future_prot_addr: a protocol specific address and possibly protocol version information. An example of this for **iiop** is "**1.1@555xyz.com**."

key_string: a stringified object key.

The **key_string** corresponds to the octet sequence in the **object_key** member of a GIOP **Request** or **LocateRequest** header as defined in 15.4 of CORBA 2.3. The **key_string** uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

";" | "/" | ":" | "?"| ":" | "@" | "&" | "=" | "+" | "$" |
"," | "-" | "_" | "!" | "~" | "*" | "'" | "(" | ")"

The **key_string** is not NUL-terminated.

7.6.10.2 corbaloc:rir URL

The **corbaloc:rir** URL is defined to allow access to the ORB's configured initial references through a URL. The protocol address syntax is:

<rir_prot_addr> = <rir_prot_token>":"
<rir_prot_token> = "rir"

Where:

rir_prot_addr: resolve_initial_references protocol identifier. There is no version or address information when **rir** is used.

rir_prot_token: The token "**rir**" identifies this protocol.

For a **corbaloc:rir** URL, the **<key_string>** is used as the argument to **resolve_initial_references**. An empty **<key_string>** is interpreted as the default "**NameService**."

The **rir** protocol cannot be used with any other protocol in a URL.

7.6.10.3 corbaloc:iiop URL

The **corbaloc:iiop** URL is defined for use in TCP/IP- and DNS-centric environments The full protocol address syntax is:

<iiop_prot_addr> = <iiop_id><iiop_addr>
<iiop_id> = <iiop_default> | <iiop_prot_token>":"
<iiop_default> = ":"
<iiop_prot_token> = "iiop"

```
<iiop_addr>          = [<version> <host> [":" <port>]]
<host>               = DNS_style_host_name | IPv4_address
                         | "[" IPv6_address "]"
<version>            = <major> "." <minor> "@" | empty_string
<port>               = number
<major>              = number
<minor>              = number
```

Where:

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address.

iiop_id: tokens recognized to indicate an iiop protocol corbaloc.

iiop_default: default token indicating iiop protocol, "**:**".

iiop_prot_token: iiop protocol token, "**iiop**."

iiop_address: a single address.

host: DNS-style host name or IP address. If not present, the local host is assumed.

version: a major and minor version number, separated by '.' and followed by '@'. If the version is absent, 1.0 is assumed.

IPv4_address: numeric IPv4 address (dotted decimal notation).

IPv6_address: numeric IPv6 address (colon separated hexadecimal or mixed hexadecimal/decimal notation as described in RFC 2373).

port: port number the agent is listening on (see below). Default is *2809*.

7.6.10.4 corbaloc Server Implementation

The only requirements on an object advertised by a **corbaloc** URL are that there must be a software agent listening on the host and port specified by the URL. This agent must be capable of handling GIOP **Request** and **LocateRequest** messages targeted at the object key specified in the URL.

A normal CORBA server meets these criteria. It is also possible to implement lightweight object location forwarding agents that respond to GIOP `Request` messages with `Reply` messages with a **LOCATION_FORWARD** status, and respond to GIOP **LocateRequest** messages with **LocateReply** messages.

7.6.10.5 corbaname URL

The **corbaname** URL scheme is described in the Naming Service specification. It extends the capabilities of the **corbaloc** scheme to allow URLs to denote entries in a Naming Service. Resolving **corbaname** URLs does not require a Naming Service implementation in the ORB core. Some examples are:

corbaname::555objs.com#a/string/path/to/obj

This URL specifies that at host **555objs.com**, an object of type **NamingContext** (with an object key of **NameService**) can be found, or alternatively, that an agent is running at that location which will return a reference to a **NamingContext**. The (stringified) name **a/string/path/to/obj** is then used as the argument to a **resolve** operation on that `NamingContext`. The URL denotes the object reference that results from that lookup.

corbaname:rir:#a/local/obj

This URL specifies that the stringified name **a/local/obj** is to be resolved relative to the naming context returned by **resolve_initial_references("NameService")**.

7.6.10.6 Future corbaloc URL Protocols

This part of ISO/IEC 19500 only defines use of iiop with corbaloc. New protocols can be added to **corbaloc** as required. Each new protocol must implement the **<future_prot_addr>** component of the URL and define described in corbaloc URL on page 33.

A possible example of a future corbaloc URL that incorporates an ATM address is:

corbaloc:iiop:xyz.com,atm:E.164:358.400.1234567/dev/test/objectX

7.6.10.7 Future URL Schemes

Several currently defined non-CORBA URL scheme names are reserved. Implementations may choose to provide these or other URL schemes to support additional ways of denoting objects with URLs.

Table 7.1 lists the required and some optional formats.

Table 7.1

Scheme	Description	Status
IOR:	Standard stringified IOR format	Required
corbaloc:	Simple object reference. rir: must be supported.	Required
corbaname:	CosName URL	Required
file://	Specifies a file containing a URL/IOR	Optional
ftp://	Specifies a file containing a URL/IOR that is accessible via ftp protocol.	Optional
http://	Specifies an HTTP URL that returns an object URL/IOR.	Optional

7.7 Service Context

Emerging specifications for Object Services occasionally require service-specific context information to be passed implicitly with requests and replies. The Interoperability specifications define a mechanism for identifying and passing this service-specific context information as "hidden" parameters. The specification makes the following assumptions:

- Object Service specifications that need additional context passed will completely specify that context as an OMG IDL data type.
- ORB APIs will be provided that will allow services to supply and consume context information at appropriate points in the process of sending and receiving requests and replies.
- It is an ORB's responsibility to determine when to send service-specific context information, and what to do with such information in incoming messages. It may be possible, for example, for a server receiving a request to be unable to de-encapsulate and use a certain element of service-specific context, but nevertheless still be able to successfully reply to the message.

As shown in the following OMG IDL specification, the IOP module provides the mechanism for passing Object Service–specific information. It does not describe any service-specific information. It only describes a mechanism for transmitting it in the most general way possible. The mechanism is currently used by the DCE ESIOP and could also be used by the Internet Inter-ORB protocol (IIOP) General Inter-ORB Protocol (GIOP).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by OMG. Service context ID values are of type **unsigned long**. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The marshaling of Object Service data is described by the following OMG IDL:

```
module IOP {                                        // IDL

    typedef unsigned long           ServiceId;
    typedef CORBA::OctetSeq ContextData;

    struct ServiceContext {
        ServiceId           context_id;
        ContextData         context_data;
    };
    typedef sequence <ServiceContext>ServiceContextList;
};
```

The context data for a particular service will be encoded as specified for its service-specific OMG IDL definition, and that encoded representation will be encapsulated in the **context_data** member of **IOP::ServiceContext** (see Encapsulation on page 79). The **context_id** member contains the service ID value identifying the service and data format. Context data is encapsulated in octet sequences to permit ORBs to handle context data without unmarshaling, and to handle unknown context data types.

During request and reply marshaling, ORBs will collect all service context data associated with the *Request* or *Reply* in a **ServiceContextList**, and include it in the generated messages. No ordering is specified for service context data within the list. The list is placed at the beginning of those messages to support security policies that may need to apply to the majority of the data in a request (including the message headers).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by the OMG. Service context ID values are of type unsigned long. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The high-order 24 bits of a service context ID contain a 24-bit vendor service context codeset ID (VSCID); the low-order 8 bits contain the rest of the service context ID. A vendor (or group of vendors) who wishes to define a specific set of service context IDs should obtain a unique VSCID from the OMG, and then define a specific set of service context IDs using the VSCID for the high-order bits.

The VSCIDs of zero to 15 inclusive (0x000000 to 0x00000f) are reserved for use for OMG-defined standard service context IDs (i.e., service context IDs in the range 0-4095 are reserved as OMG standard service contexts).

7.7.1 Standard Service Contexts

```
module IOP {                                        // IDL
    const ServiceId         TransactionService = 0;
```

```
        const ServiceId         CodeSets = 1;
        const ServiceId         ChainBypassCheck = 2;
        const ServiceId         ChainBypassInfo = 3;
        const ServiceId         LogicalThreadId = 4;
        const ServiceId         BI_DIR_IIOP = 5;
        const ServiceId         SendingContextRunTime = 6;
        const ServiceId         INVOCATION_POLICIES = 7;
        const ServiceId         FORWARDED_IDENTITY = 8;
        const ServiceId         UnknownExceptionInfo = 9;
        const ServiceId         RTCorbaPriority = 10;
        const ServiceId         RTCorbaPriorityRange = 11;
        const ServiceId         FT_GROUP_VERSION = 12;
        const ServiceId         FT_REQUEST = 13;
        const ServiceId         ExceptionDetailMessage = 14;
        const ServiceId         SecurityAttributeService = 15;
        const ServiceId         ActivityService = 16;
        const ServiceId         RMICustomMaxStreamFormat = 17;
        const ServiceId         ACCESS_SESSION_ID = 18;
        const ServiceId         SERVICE_SESSION_ID = 19;
        const ServiceId         FIREWALL_PATH = 20;
        const ServiceId         FIREWALL_PATH_RESP = 21;
};
```

The standard **ServiceId**s currently defined are:

- **TransactionService** identifies a CDR encapsulation of the **CosTransactions::PropogationContext** defined in the Object Transaction Service specification [TRANS].
- **CodeSets** identifies a CDR encapsulation of the **CONV_FRAME::CodeSetContext** defined in GIOP Code Set Service Context on page 51.
- DCOM-CORBA Interworking uses three service contexts as defined in “DCOM-CORBA Interworking” in [DCOMI]. They are:
  - **ChainBypassCheck**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassCheck**. This is carried only in a **Request** message as described in [DCOMI].
  - **ChainBypassInfo**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassInfo**. This is carried only in a **Reply** message as described in [DCOMI].
  - **LogicalThreadId**, which carries a CDR encapsulation of the **struct CosBridging::LogicalThreadId** as described in [DCOMI].
- **BI_DIR_IIOP** identifies a CDR encapsulation of the **IIOP::BiDirIIOPServiceContext** defined in the General Inter-ORB Protocol clause of this part in this International Standard.
- **SendingContextRunTime** identifies a CDR encapsulation of the IOR of the **SendingContext::RunTime** object. See the Value Type Semantics clause (Part 1 of this International Standard).
- For information on **INVOCATION_POLICIES** refer to the CORBA Messaging clause of Part 1 of this International Standard.
- For information on **FORWARDED_IDENTITY** refer to the Firewall Traversal specification ([FIREWALL]).
- **UnknownExceptionInfo** identifies a CDR encapsulation of a marshaled instance of a **java.lang.throwable** or one of its subclasses as described in the Java to IDL Mapping specification [JAVA2I].
- For information on **RTCorbaPriority** refer to the Real-time CORBA specification [RTCORBA].

- For information on **RTCorbaPriorityRange** refer to the Real-time CORBA specification [RTCORBA].
- **FT_GROUP_VERSION, FT_REQUEST** - refer to the Fault Tolerant CORBA [FTCORBA].
- **ExceptionDetailMessage** identifies a CDR encapsulation of a wstring, encoded using GIOP 1.2 with a TCS-W of UTF-16. This service context may be sent on Reply messages with a reply_status of SYSTEM_EXCEPTION or USER_EXCEPTION. The usage of this service context is defined by language mappings.
- **SecurityAttributeService** - refer to the Secure Interoperability clause of this Part of this International Standard.
- **ActivityService** - refer to the Additional Structuring Mechanisms for OTS specification [ASMOTS].
- **RMICustomMaxStreamFormat** - refer to the Java to IDL Language Mapping specification [JAVA2I].
- **ACCESS_SESSION_ID** and **SERVICE_SESSION_ID** - refer to the Telecommunication Service Access and Subscription Specification [TSAS}.
- **FIREWALL_PATH** and **FIREWALL_PATH_RESP** - refer to the Firewall Traversal specification ([FIREWALL]).

7.7.2 Service Context Processing Rules

Service context IDs are associated with a specific version of GIOP, but will always be allocated in the OMG service context range. This allows any ORB to recognize when it is receiving a standard service context, even if it has been defined in a version of GIOP that it does not support.

The following are the rules for processing a received service context:

- The service context is in the OMG defined range:
  - If it is valid for the supported GIOP version, then it must be processed correctly according to the rules associated with it for that GIOP version level.
  - If it is not valid for the GIOP version, then it may be ignored by the receiving ORB; however, it must be passed on through a bridge and must be made available to interceptors. No exception shall be raised.
- The service context is not in the OMG-defined range:
  - The receiving ORB may choose to ignore it, or process it if it "understands" it; however, the service context must be passed on through a bridge and must made available to interceptors.

7.8 Coder/Decoder Interfaces

The formats of IOR components and service context data used by ORB services are often defined as CDR encapsulations encoding instances of IDL defined data types. The **Codec** provides a mechanism to transfer these components between their IDL data types and their CDR encapsulation representations.

A **Codec** is obtained from the **CodecFactory**. The **CodecFactory** is obtained through a call to **ORB::resolve_initial_references ("CodecFactory")**.

7.8.1 Codec Interface

```
module IOP {
    local interface Codec {
        exception InvalidTypeForEncoding {};
        exception FormatMismatch {};
        exception TypeMismatch {};
```

```
        CORBA::OctetSeq encode (in any data)
            raises (InvalidTypeForEncoding);
        any decode (in CORBA::OctetSeq data)
            raises (FormatMismatch);
        CORBA::OctetSeq encode_value (in any data)
            raises (InvalidTypeForEncoding);
        any decode_value (
            in CORBA::OctetSeq data,
            in CORBA::TypeCode tc)
            raises (FormatMismatch, TypeMismatch);
    };
};
```

7.8.1.1 Exceptions

InvalidTypeForEncoding

This exception is raised by **encode** or **encode_value** when the type is invalid for the encoding. For example, this exception is raised if the encoding is **ENCODING_CDR_ENCAPS** version 1.0 and a type that does not exist in that version, such as wstring, is passed to the operation.

FormatMismatch

This exception is raised by **decode** or **decode_value** when the data in the octet sequence cannot be decoded into an **any**.

TypeMismatch

This exception is raised by **decode_value** when the given **TypeCode** does not match the given octet sequence.

7.8.1.2 Operations

encode

Convert the given **any** into an octet sequence based on the encoding format effective for this **Codec**. This operation may raise **InvalidTypeForEncoding**.

Parameter: **data**	The data, in the form of an **any**, to be encoded into an octet sequence.
Return Value:	An octet sequence containing the encoded **any**. This octet sequence contains both the **TypeCode** and the data of the type.

decode

Decode the given octet sequence into an **any** based on the encoding format effective for this **Codec**. This operation raises **FormatMismatch** if the octet sequence cannot be decoded into an **any**.

Parameter: **data**	The data, in the form of an octet sequence, to be decoded into an **any**.
Return Value:	An **any** containing the data from the decoded octet sequence.

encode_value

Convert the given **any** into an octet sequence based on the encoding format effective for this **Codec**. Only the data from the **any** is encoded, not the **TypeCode**. This operation may raise InvalidTypeForEncoding.

Parameter: **data**	The data, in the form of an **any**, to be encoded into an octet sequence.
Return Value:	An octet sequence containing the data from the encoded **any**.

decode_value

Decode the given octet sequence into an **any** based on the given **TypeCode** and the encoding format effective for this **Codec**. This operation raises FormatMismatch if the octet sequence cannot be decoded into an **any**.

Parameters:	
data	The data, in the form of an octet sequence, to be decoded into an **any**.
tc	The **TypeCode** to be used to decode the data.
Return Value:	An **any** containing the data from the decoded octet sequence.

7.8.2 Codec Factory

```
module IOP {
    typedef short EncodingFormat;
    const EncodingFormat ENCODING_CDR_ENCAPS = 0;

    struct Encoding {
        EncodingFormat format;
        octet major_version;
        octet minor_version;
    };

    local interface CodecFactory {
        exception UnknownEncoding {};
        Codec create_codec (in Encoding enc)
            raises (UnknownEncoding);
    };
};
```

7.8.2.1 Encoding Structure

The **Encoding** structure defines the encoding format of a **Codec**. It details the encoding format, such as CDR Encapsulation encoding, and the major and minor versions of that format. The encodings which shall be supported are:

- **ENCODING_CDR_ENCAPS**, version 1.0;
- **ENCODING_CDR_ENCAPS**, version 1.1;
- **ENCODING_CDR_ENCAPS**, version 1.2;
- **ENCODING_CDR_ENCAPS** for all future versions of GIOP as they arise.

Vendors are free to support additional encodings.

7.8.2.2 CodecFactory Interface

create_codec

Create a **Codec** of the given encoding.

This operation raises **UnknownEncoding** if this factory cannot create a **Codec** of the given encoding.

Parameter: **enc**	The **Encoding** for which to create a **Codec**.
Return Value:	A **Codec** obtained with the given encoding.

7.9 Feature Support and GIOP Versions

The association of service contexts with GIOP versions, (along with some other supported features tied to GIOP minor version), is shown in Table 7.2.

Table 7.2 Feature Support Tied to Minor GIOP Version Number

Feature	V1.0	V1.1	V1.2	V1.3	V1.4
TransactionService Service Context	yes	yes	yes	yes	yes
CodeSets Service Context		yes	yes	yes	yes
DCOM Bridging Service Contexts: ChainBypassCheck ChainBypassInfo LogicalThreadId			yes	yes	yes
Object by Value Service Context: SendingContextRunTime			yes	yes	yes
Bi-Directional IIOP Service Context: BI_DIR_IIOP			yes	yes	yes
Asynch Messaging Service Context: INVOCATION_POLICIES			optional[$]	yes	yes
Firewall Service Context: FORWARDED_IDENTITY			optional[$]	yes	yes
Java Language Throwable Service Context: UnknownExceptionInfo			yes	yes	yes
Realtime CORBA Service Contexts RTCorbaPriority RTCorbaPriorityRange			optional (Realtime CORBA only)	optional (Realtime CORBA only)	optional (Realtime CORBA only)
ExceptionDetailMessage Service Context			optional	yes	yes
FT_GROUP_VERSION			optional[$$]	yes	yes

Table 7.2 Feature Support Tied to Minor GIOP Version Number *(Continued)*

Feature	V1.0	V1.1	V1.2	V1.3	V1.4
FT_REQUEST			optional[$$]	yes	yes
SecurityAttributeService			optional[$$]	yes	yes
ActivityService			optional[$$]	yes	yes
IOR components in IIOP profile		yes	yes	yes	yes
TAG_ORB_TYPE		yes	yes	yes	yes
TAG_CODE_SETS		yes	yes	yes	yes
TAG_ALTERNATE_IIOP_ADDRESS			yes	yes	yes
TAG_ASSOCIATION_OPTION		yes	yes	yes	yes
TAG_SEC_NAME		yes	yes	yes	yes
TAG_SSL_SEC_TRANS		yes	yes	yes	yes
TAG_GENERIC_SEC_MECH		yes	yes	yes	yes
TAG_*_SEC_MECH		yes	yes	yes	yes
TAG_JAVA_CODEBASE			yes	yes	yes
TAG_FIREWALL_TRANS			optional[$]	yes	yes
TAG_SCCP_CONTACT_INFO			optional[$]	yes	yes
TAG_TRANSACTION_POLICY			optional[$]	yes	yes
TAG_MESSAGE_ROUTERS			optional[$]	yes	yes
TAG_OTS_POLICY			optional[$]	yes	yes
TAG_INV_POLICY			optional[$]	yes	yes
TAG_INET_SEC_TRANS			optional[$]	yes	yes
Extended IDL data types		yes	yes	yes	yes
Bi-Directional GIOP Features			yes	yes	yes
Value types and Abstract Interfaces			yes	yes	yes
TAG_CSI_SEC_MECH_LIST			optional[$$]	yes	yes
TAG_NULL_TAG			optional[$$]	yes	yes
TAG_SECIOP_SEC_TRANS, TAG_IIOP_SEC_TRANS			optional[$$]	yes	yes
TAG_TLS_SEC_TRANS			optional[$$]	yes	yes
TAG_ACTIVITY_POLICY			optional[$$]	yes	yes
_component				yes	yes
tk_abstract_interface tk_local_interfacel			optional[$$]	yes	yes

Table 7.2 Feature Support Tied to Minor GIOP Version Number *(Continued)*

Feature	V1.0	V1.1	V1.2	V1.3	V1.4
tk_component tk_home tk_event				yes	yes
_repository_id					yes
IPV6 addresses in IOR					yes
TAG_GROUP, TAG_GROUP_IIOP and TAG_UIPMC + Group features					optional
RMICustomMaxStreamFormat TAG_RMI_CUSTOM_MAX_STREAM_FORMAT					optional

NOTE: $, $$All features that have been added after CORBA 2.3 have been marked as optional in GIOP 1.2. These features cannot be compulsory in GIOP 1.2 since there is no way to incorporate them in deployed implementations of 1.2. However, in order to have the additional features of CORBA 2.4 work properly these optional features must be supported by the GIOP 1.2 implementation connecting CORBA 2.4$ or later ORBs.

7.10 Code Set Conversion

7.10.1 Character Processing Terminology

This sub clause introduces a few terms and explains a few concepts to help understand the character processing portions of this clause.

7.10.1.1 Character Set

A finite set of different characters used for the representation, organization, or control of data. In this part of ISO/IEC 19500, the term “character set” is used without any relationship to code representation or associated encoding. Examples of character sets are the English alphabet, Kanji or sets of ideographic characters, corporate character sets (commonly used in Japan), and the characters needed to write certain European languages.

7.10.1.2 Coded Character Set, or Code Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. In this part of ISO/IEC 19500, the term “code set” is used as an abbreviation for the term “coded character set.” Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.) and Unicode.

7.10.1.3 Code Set Classifications

Some language environments distinguish between byte-oriented and “wide characters.” The byte-oriented characters are encoded in one or more 8-bit bytes. A typical single-byte encoding is ASCII as used for western European languages like English. A typical multi-byte encoding that uses from one to three 8-bit bytes for each character is eucJP (Extended UNIX Code - Japan, packed format) as used for Japanese workstations.

Wide characters are a fixed 16 or 32 bits long, and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits is insufficient and a fixed-width encoding is needed. A typical example is Unicode (a "universal" character set defined by the Unicode Consortium, which uses an encoding scheme identical to ISO 10646 UCS-2, or 2-byte Universal Character Set encoding). An extended encoding scheme for Unicode characters is UTF-16 (UCS Transformation Format, 16-bit representations).

The C language has data types **`char`** for byte-oriented characters and **`wchar_t`** for wide characters. The language definition for C states that the sizes for these characters are implementation-dependent. Some environments do not distinguish between byte-oriented and wide characters (e.g., Ada and Smalltalk). Here again, the size of a character is implementation-dependent. The following table illustrates code set classifications as used in this document.

Table 7.3 - Code Set Classification

Orientation	Code Element Encoding	Code Set Examples	C Data Type
byte-oriented	single-byte	ASCII, ISO 8859-1 (Latin-1), EBCDIC, ...	**char**
	multi-byte	UTF-8, eucJP, Shift-JIS, JIS, Big5, ...	**char[]**
non-byte-oriented	fixed-length	ISO 10646 UCS-2 (Unicode), ISO 10646 UCS-4, UTF-16, ...	**wchar_t**

7.10.1.4 Narrow and Wide Characters

Some language environments distinguish between "narrow" and "wide" characters. Typically the narrow characters are considered to be 8-bit long and are used for western European languages like English, while the wide characters are 16-bit or 32-bit long and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits are insufficient. However, as noted above there are common encoding schemes in which Asian characters are encoded using multi-byte code sets and it is incorrect to assume that Asian characters are always encoded as "wide" characters.

Within this text, the general terms "narrow character" and "wide character" are only used in discussing OMG IDL.

7.10.1.5 Char Data and Wchar Data

The phrase "**char** data" in this text refers to data whose IDL types have been specified as **char** or **string**. Likewise "**wchar** data" refers to data whose IDL types have been specified as **wchar** or **wstring**.

7.10.1.6 Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy one or more bytes. A byte as used in this part of ISO/IEC 19500 is synonymous with octet, which occupies 8 bits.

7.10.1.7 Multi-Byte Character Strings

A character string represented in a byte-oriented encoding where each character can occupy one or more bytes is called a multi-byte character string. Typically, wide characters are converted to this form from a (fixed-width) process code set before transmitting the characters outside the process (see below about process code sets). Care must be taken to correctly process the component bytes of a character's multi-byte representation.

7.10.1.8 Non-Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy fixed 16 or 32 bits.

7.10.1.9 Char and Wchar Transmission Code Set (TCS-C and TCS-W)

These two terms refer to code sets that are used for transmission between ORBs after negotiation is completed. As the names imply, the first one is used for **char** data and the second one for **wchar** data. Each TCS can be byte-oriented or non-byte oriented.

7.10.1.10 Process Code Set and File Code Set

Processes generally represent international characters in an internal fixed-width format which allows for efficient representation and manipulation. This internal format is called a "process code set." The process code set is irrelevant outside the process, and hence to the interoperation between CORBA clients and servers through their respective ORBs.

When a process needs to write international character information out to a file, or communicate with another process (possibly over a network), it typically uses a different encoding called a "file code set." In this part of ISO/IEC 19500, unless otherwise indicated, all references to a program's code set refer to the file code set, not the process code set. Even when a client and server are located physically on the same machine, it is possible for them to use different file code sets.

7.10.1.11 Native Code Set

A native code set is the code set that a client or a server uses to communicate with its ORB. There might be separate native code sets for **char** and **wchar** data.

7.10.1.12 Transmission Code Set

A transmission code set is the commonly agreed upon encoding used for character data transfer between a client's ORB and a server's ORB. There are two transmission code sets established per session between a client and its server, one for **char** data (TCS-C) and the other for **wchar** data (TCS-W). Figure 7.6 illustrates these relationships:

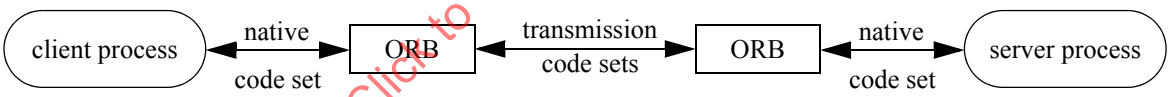


Figure 7.6 - Transmission Code Sets

The intent is for TCS-C to be byte-oriented and TCS-W to be non-byte-oriented. However, this part of ISO/IEC 19500 does allow both types of characters to be transmitted using the same transmission code set. That is, the selection of a transmission code set is orthogonal to the wideness or narrowness of the characters, although a given code set may be better suited for either narrow or wide characters.

7.10.1.13 Conversion Code Set (CCS)

With respect to a particular ORB's native code set, the set of other or target code sets for which an ORB can convert all code points or character encodings between the native code set and that target code set. For each code set in this CCS, the ORB maintains appropriate translation or conversion procedures and advertises the ability to use that code set for transmitted data in addition to the native code set.

7.10.2 Code Set Conversion Framework

7.10.2.1 Requirements

The file code set that an application uses is often determined by the platform on which it runs. In Japan, for example, Japanese EUC is used on Unix systems, while Shift-JIS is used on PCs. Code set conversion is therefore required to enable interoperability across these platforms. This part of ISO/IEC 19500 defines a framework for the automatic conversion of code sets in such situations. The requirements of this framework are:

1. Backward compatibility. In previous CORBA specifications, IDL type **char** was limited to ISO 8859-1. The conversion framework should be compatible with existing clients and servers that use ISO 8859-1 as the code set for **char**.

2. Automatic code set conversion. To facilitate development of CORBA clients and servers, the ORB should perform any necessary code set conversions automatically and efficiently. The IDL type **octet** can be used if necessary to prevent conversions.

3. Locale support. An internationalized application determines the code set in use by examining the LOCALE string (usually found in the LANG environment variable), which may be changed dynamically at run time by the user. Example LOCALE strings are fr_FR.ISO8859-1 (French, used in France with the ISO 8859-1 code set) and ja_JP.ujis (Japanese, used in Japan with the EUC code set and X11R5 conventions for LOCALE). The conversion framework should allow applications to use the LOCALE mechanism to indicate supported code sets, and thus select the correct code set from the registry.

4. CMIR and SMIR support. The conversion framework should be flexible enough to allow conversion to be performed either on the client or server side. For example, if a client is running in a memory-constrained environment, then it is desirable for code set converters to reside in the server and for a Server Makes It Right (SMIR) conversion method to be used. On the other hand, if many servers are executed on one server machine, then converters should be placed in each client to reduce the load on the server machine. In this case, the conversion method used is Client Makes It Right (CMIR).

7.10.2.2 Overview of the Conversion Framework

Both the client and server indicate a native code set indirectly by specifying a locale. The exact method for doing this is language-specific, such as the XPG4 C/C++ function **`setlocale`**. The client and server use their native code set to communicate with their ORB. (Note that these native code sets are in general different from process code sets and hence conversions may be required at the client and server ends.)

The conversion framework is illustrated in Figure 7.7. The server-side ORB stores a server's code set information in a component of the IOR multiple-component profile structure (see Interoperable Object References: IORs on page 25)[1]. The code sets actually used for transmission are carried in the service context field of an IOP (Inter-ORB Protocol) request header (see Service Context on page 37 and GIOP Code Set Service Context on page 51). Recall that there are two code sets (TCS-C and TCS-W) negotiated for each session.

1. Version 1.1 of the IIOP profile body can also be used to specify the server's code set information, as this version introduces an extra field that is a sequence of tagged components.

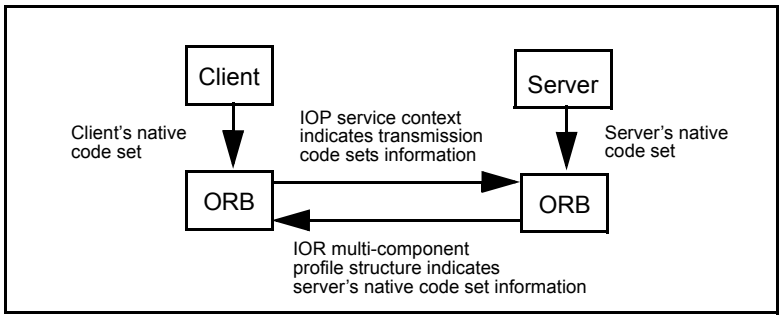


Figure 7.7 - Code Set Conversion Framework Overview

If the native code sets used by a client and server are the same, then no conversion is performed. If the native code sets are different and the client-side ORB has an appropriate converter, then the CMIR conversion method is used. In this case, the server's native code set is used as the transmission code set. If the native code sets are different and the client-side ORB does not have an appropriate converter but the server-side ORB does have one, then the SMIR conversion method is used. In this case, the client's native code set is used as the transmission code set.

The conversion framework allows clients and servers to specify a native **char** code set and a native **wchar** code set, which determine the local encodings of IDL types **char** and **wchar**, respectively. The conversion process outlined above is executed independently for the **char** code set and the **wchar** code set. In other words, the algorithm that is used to select a transmission code set is run twice, once for **char** data and once for **wchar** data.

The rationale for selecting two transmission code sets rather than one (which is typically inferred from the locale of a process) is to allow efficient data transmission without any conversions when the client and server have identical representations for **char** and/or **wchar** data. For example, when a Windows NT client talks to a Windows NT server and they both use Unicode for wide character data, it becomes possible to transmit wide character data from one to the other without any conversions. Of course, this becomes possible only for those wide character representations that are well-defined, not for any proprietary ones. If a single transmission code set was mandated, it might require unnecessary conversions. (For example, choosing Unicode as the transmission code set would force conversion of all byte-oriented character data to Unicode.)

7.10.2.3 ORB Databases and Code Set Converters

The conversion framework requires an ORB to be able to determine the native code set for a locale and to convert between code sets as necessary. While the details of exactly how these tasks are accomplished are implementation-dependent, the following databases and code set converters might be used:

- Locale database. This database defines a native code set for a process. This code set could be byte-oriented or non-byte-oriented and could be changed programmatically while the process is running. However, for a given session between a client and a server, it is fixed once the code set information is negotiated at the session's setup time.
- Environment variables or configuration files. Since the locale database can only indicate one code set while the ORB needs to know two code sets, one for **char** data and one for **wchar** data, an implementation can use environment variables or configuration files to contain this information on native code sets.
- Converter database. This database defines, for each code set, the code sets to which it can be converted. From this database, a set of "conversion code sets" (CCS) can be determined for a client and server. For example, if a server's native code set is eucJP, and if the server-side ORB has eucJP-to-JIS and eucJP-to-SJIS bilateral converters, then the server's conversion code sets are JIS and SJIS.

- Code set converters. The ORB has converters which are registered in the converter database.

7.10.2.4 CodeSet Component of IOR Multi-Component Profile

The code set component of the IOR multi-component profile structure contains:

- server's native **char** code set and conversion code sets, and
- server's native **wchar** code set and conversion code sets.

Both **char** and **wchar** conversion code sets are listed in order of preference. The code set component is identified by the following tag:

```
const IOP::ComponentID TAG_CODE_SETS = 1;
```

This tag has been assigned by OMG (See Standard IOR Components on page 29). The following IDL structure defines the representation of code set information within the component:

```
module CONV_FRAME {                         // IDL
    typedef unsigned long CodeSetId;
    typedef sequence<CodeSetId> CodeSetIdSeq;
    struct CodeSetComponent {
        CodeSetId               native_code_set;
        CodeSetIdSeq            conversion_code_sets;
    };
    struct CodeSetComponentInfo {

        CodeSetComponent        ForCharData;
        CodeSetComponent        ForWcharData;
    };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See Character and Code Set Registry on page 56 for further information). Data within the code set component is represented as a structure of type **CodeSetComponentInfo**, and is encoded as a CDR encapsulation. In other words, the **char** code set information comes first, then the **wchar** information, represented as structures of type **CodeSetComponent**.

A null value should be used in the **native_code_set** field if the server desires to indicate no native code set (possibly with the identification of suitable conversion code sets).

If the code set component is not present in a multi-component profile structure, then the default **char** code set is ISO 8859-1 for backward compatibility. However, there is no default **wchar** code set. If a server supports interfaces that use wide character data but does not specify the **wchar** code sets that it supports, client-side ORBs will raise exception INV_OBJREF, with standard minor code 1.

If a client application invokes an operation that results in an attempt by the client ORB to marshal wchar or wstring data for an in parameter (or to unmarshal wchar or wstring data for an in/out parameter, out parameter or the return value), and the associated Object Reference does not include a codeset component, then the client ORB shall raise the INV_OBJREF standard system exception with standard minor code 2 as a response to the operation invocation.

Non-presence of a codeset component in an IOR means that:

- The server and/or server-side ORB support only ISO 8859-1 for char/string, and
- the server and/or server-side ORB don't support wchar/wstring.

Thus if client tries to send wchar or wstring data on an **any** type, and there is no codeset component in target server's IOR, the client-side ORB can raise an exception BAD_PARAM, with standard minor code set to 42.

7.10.2.5 GIOP Code Set Service Context

The code set GIOP service context contains:

- **char** transmission code set, and
- **wchar** transmission code set

in the form of a code set service. This service is identified by:

```
const IOP::ServiceID  CodeSets = 1;
```

The following IDL structure defines the representation of code set service information:

```
module CONV_FRAME {                          // IDL
    typedef unsigned long CodeSetId;
    struct CodeSetContext {
        CodeSetId           char_data;
        CodeSetId           wchar_data;
    };
};
```

For GIOP versions 1.1, 1.2 and 1.3, Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See Character and Code Set Registry on page 56 for further information).

For GIOP versions greater than 1.3, Code sets are identified by a 32 bit integer id, from either the OSF Character and Code set registry (See Character and Code Set Registry on page 56 for further information) or the IANA Character Set registry (current version at http://www.iana.org/assignments/character-sets).

The OSF Registry and the IANA Registry have non-overlapping ranges, so there is no need for mapping values from one codeset registry to the other.

NOTE: A server's **char** and **wchar** Code set components are usually different, but under some special circumstances they can be the same. That is, one could use the same code set for both **char** data and **wchar** data. Likewise the **CodesetId**s in the service context don't have to be different.

7.10.2.6 Code Set Negotiation

The client-side ORB determines a server's native and conversion code sets from the code set component in an IOR multi-component profile structure, and it determines a client's native and conversion code sets from the locale setting (and/or environment variables/configuration files) and the converters that are available on the client. From this information, the client-side ORB chooses **char** and **wchar** transmission code sets (TCS-C and TCS-W). For both requests and replies, the **char** TCS-C determines the encoding of **char** and **string** data, and the **wchar** TCS-W determines the encoding of **wchar** and **wstring** data.

Code set negotiation is not performed on a per-request basis, but only when a client initially connects to a server. All text data communicated on a connection are encoded as defined by the TCSs selected when the connection is established.

A codeset service context must be sent by the client (i.e., codeset negotiation must be completed) over a specific transport connection, before the client or the server may send international character values (i.e., char or string values with non Latin-1 encodings, or Wchar or Wstring values) in messages on that transport connection.

If used, the codeset service context shall be sent before, or included with, the first request message sent on that transport connection.

A request sent by the client before sending a codeset service context or not containing a service context itself, implies the client is using the default codesets on that connection (i.e., Latin-1 for string, and no ability to send Wstring on any message over that connection).

Some existing Standard Service contexts have defined their encapsulated data content including International Character information, and have also specified that the codeset used is that which is negotiated using codeset negotiation. Such service contexts may not be sent until after the Codeset Service context is sent (i.e., in the GIOP message, the codeset service context must precede any service context which depends on it being present.). Such Service Contexts that exist today are grandfathered in. Barring that exception, since all encapsulation definitions need to specify the Codeset used for their encodings, it is an error for a Service Context to depend on information that is not contained within the encapsulation to determine the codeset used within it.

Figure 7.8 illustrates there are two channels for character data flowing between the client and the server. The first, TCS-C, is used for **char** data and the second, TCS-W, is used for **wchar** data. Also note that two native code sets, one for each type of data, could be used by the client and server to talk to their respective ORBs (as noted earlier, the selection of the particular native code set used at any particular point is done via **`setlocale`** or some other implementation-dependent method).

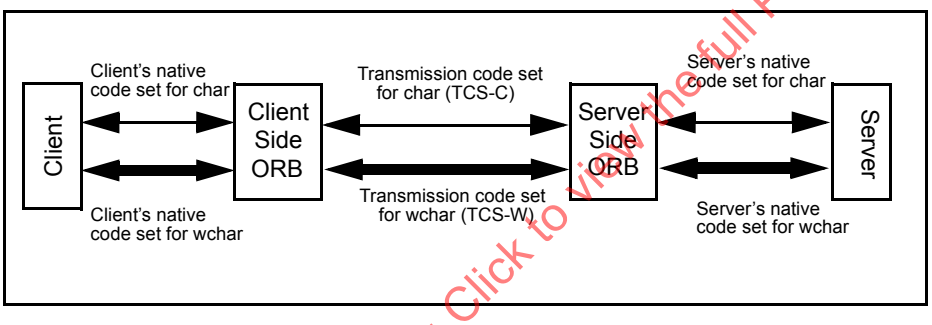


Figure 7.8 - Transmission Code Set Use

Let us look at an example. Assume that the code set information for a client and server is as shown in the table below. (Note that this example concerns only **char** code sets and is applicable only for data described as **char**s in the IDL.)

	Client	**Server**
Native code set:	SJIS	eucJP
Conversion code sets:	eucJP, JIS	SJIS, JIS

The client-side ORB first compares the native code sets of the client and server. If they are identical, then the transmission and native code sets are the same and no conversion is required. In this example, they are different, so code set conversion is necessary. Next, the client-side ORB checks to see if the server's native code set, eucJP, is one of the conversion code sets supported by the client. It is, so eucJP is selected as the transmission code set, with the client (i.e., its ORB) performing conversion to and from its native code set, SJIS, to eucJP. Note that the client may first have to convert all its data described as **char**s (and possibly **wchar_t**s) from process codes to SJIS first.

Now let us look at the general algorithm for determining a transmission code set and where conversions are performed. First, we introduce the following abbreviations:

- CNCS - Client Native Code Set;
- CCCS - Client Conversion Code Sets;
- SNCS - Server Native Code Set;
- SCCS - Server Conversion Code Sets; and
- TCS - Transmission Code Set.

The algorithm is as follows:

```
if (CNCS==SNCS)
        TCS = CNCS;                 // no conversion required
else {
        if (elementOf(SNCS,CCCS))
            TCS = SNCS;   // client converts to server's native code set
        else if (elementOf(CNCS,SCCS))
            TCS = CNCS;   // server converts from client's native code set
        else if (intersection(CCCS,SCCS) != emptySet) {
            TCS = oneOf(intersection(CCCS,SCCS));
            // client chooses TCS, from intersection(CCCS,SCCS), that is
            // most preferable to server;
            // client converts from CNCS to TCS and server
            // from TCS to SNCS
        else if (compatible(CNCS,SNCS))
            TCS = fallbackCS;       // fallbacks are UTF-8 (for char data) and
                                    // UTF-16 (for wchar data)
        else
            raise CODESET_INCOMPATIBLE exception;
}
```

The algorithm first checks to see if the client and server native code sets are the same. If they are, then the native code set is used for transmission and no conversion is required. If the native code sets are not the same, then the conversion code sets are examined to see if

1. the client can convert from its native code set to the server's native code set,
2. the server can convert from the client's native code set to its native code set, or
3. transmission through an intermediate conversion code set is possible.

If the third option is selected and there is more than one possible intermediate conversion code set (i.e., the intersection of CCCS and SCCS contains more than one code set), then the one most preferable to the server is selected.[1]

If none of these conversions is possible, then the fallback code set (UTF-8 for **char** data and UTF-16 for **wchar** data—see below) is used. However, before selecting the fallback code set, a compatibility test is performed. This test looks at the character sets encoded by the client and server native code sets. If they are different (e.g., Korean and French), then meaningful communication between the client and server is not possible and a CODESET_INCOMPATIBLE exception with standard minor code 1 is raised. This test is similar to the DCE compatibility test and is intended to catch those cases where conversion from the client native code set to the fallback, and the fallback to the server native code set would result

1. Recall that server conversion code sets are listed in order of preference.

in massive data loss. (See clause 7.10.5, Relevant OSFM Registry Interfaces, on page 56 for the relevant OSF registry interfaces that could be used for determining compatibility.) If either the CNCS or SNCS is from the IANA Character Set registry, then the codesets are automatically assumed to be compatible and the fallback codeset is used.

A DATA_CONVERSION exception is raised when a client or server attempts to transmit a character that does not map into the negotiated transmission code set. For example, not all characters in Taiwan Chinese map into Unicode. When an attempt is made to transmit one of these characters via Unicode, an ORB is required to raise a DATA_CONVERSION exception, with standard minor code 1.

In summary, the fallback code set is UTF-8 for **char** data (identified in the Registry as 0x05010001, "X/Open UTF-8; UCS Transformation Format 8 (UTF-8)"), and UTF-16 for **wchar** data (identified in the Registry as 0x00010109, "ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form"). As mentioned above the fallback code set is meaningful only when the client and server character sets are compatible, and the fallback code set is distinguished from a default code set used for backward compatibility.

If a server's native **char** code set is not specified in the IOR multi-component profile, then it is considered to be ISO 8859-1 for backward compatibility. However, a server that supports interfaces that use wide character data is required to specify its native **wchar** code set; if one is not specified, then the client-side ORB raises exception INV_OBJREF, with standard minor code set to 1.

Similarly, if no **char** transmission code set is specified in the code set service context, then the **char** transmission code set is considered to be ISO 8859-1 for backward compatibility. If a client transmits wide character data and does not specify its **wchar** transmission code set in the service context, then the server-side ORB raises exception BAD_PARAM, with standard minor code set to 23.

If the client delivers a codeset via a CodeSetContext that the server does not support as a transmission codeset then the server returns a CODESET_INCOMPATIBLE exception with the standard minor code 2.

If the client (or the server if Bi-Directional GIOP is in use) sends multiple codeset service contexts on the same connection, with different parameter values, then the behavior is undefined. The receiver of a codeset service context with different values from those received on the same connection and processed previously may return a MARSHAL system exception with the standard minor code 9.

To guarantee "out-of-the-box" interoperability, clients and servers must be able to convert between their native **char** code set and UTF-8 and their native **wchar** code set (if specified) and Unicode. Note that this does not require that all server native code sets be mappable to Unicode, but only those that are exported as native in the IOR. The server may have other native code sets that aren't mappable to Unicode and those can be exported as SCCSs (but not SNCSs). This is done to guarantee out-of-the-box interoperability and to reduce the number of code set converters that a CORBA-compliant ORB must provide.

ORB implementations are strongly encouraged to use widely-used code sets for each regional market. For example, in the Japanese marketplace, all ORB implementations should support Japanese EUC, JIS, and Shift JIS to be compatible with existing business practices.

7.10.3 Mapping to Generic Character Environments

Certain language environments do not distinguish between byte-oriented and wide characters. In such environments both **char** and **wchar** are mapped to the same "generic" character representation of the language. **String** and **wstring** are likewise mapped to generic strings in such environments. Examples of language environments that provide generic character support are Smalltalk and Ada.

Even while using languages that do distinguish between wide and byte-oriented characters (e.g., C and C++), it is possible to mimic some generic behavior by the use of suitable macros and support libraries. For example, developers of Windows NT and Windows 95 applications can write portable code between NT (which uses Unicode strings) and Windows 95 (which uses byte-oriented character strings) by using a set of macros for declaring and manipulating characters and character strings.

Another way to achieve generic manipulation of characters and strings is by treating them as abstract data types (ADTs). For example, if strings were treated as abstract data types and the programmers are required to create, destroy, and manipulate strings only through the operations in the ADT interface, then it becomes possible to write code that is representation-independent. This approach has an advantage over the macro-based approach in that it provides portability between byte-oriented and wide character environments even without recompilation (at runtime the string function calls are bound to the appropriate byte-oriented/wide library). Another way of looking at it is that the macro-based genericity gives compile-time flexibility, while ADT-based genericity gives runtime flexibility.

Yet another way to achieve generic manipulation of character data is through the ANSI C++ Strings library defined as a template that can be parameterized by `char`, `wchar_t`, or other integer types.

Given that there can be several ways of treating characters and character strings in a generic way, this standard cannot, and therefore does not, specify the mapping of **char**, **wchar, string**, and **wstring** to all of them. It only specifies the following normative requirements that are applicable to generic character environments:

- **wchar** must be mapped to the generic character type in a generic character environment.
- **wstring** must be mapped to a string of such generic characters in a generic character environment.
- The language binding files (i.e., stubs) generated for these generic environments must ensure that the generic type representation is converted to the appropriate code sets (i.e., CNCS on the client side and SNCS on the server side) before character data is given to the ORB runtime for transmission.

7.10.3.1 Describing Generic Interfaces

To describe generic interfaces in IDL we recommend using **wchar** and **wstring**. These can be mapped to generic character types in environments where they do exist and to wide characters where they do not. Either way interoperation between generic and non-generic character type environments is achieved because of the code set conversion framework.

7.10.3.2 Interoperation

Let us consider an example to see how a generic environment can interoperate with a non-generic environment. Let us say there is an IDL interface with both **char** and **wchar** parameters on the operations, and let us say the client of the interface is in a generic environment while the server is in a non-generic environment (for example the client is written in Smalltalk and the server is written in C++).

Assume that the server's (byte-oriented) native **char** code set (SNCS) is eucJP and the client's native **char** code set (CNCS) is SJIS. Further assume that the code set negotiation led to the decision to use eucJP as the **char** TCS-C and Unicode as the **wchar** TCS-W.

As per the above normative requirements for mapping to a generic environment, the client's Smalltalk stubs are responsible for converting all **char** data (however they are represented inside Smalltalk) to SJIS and all **wchar** data to the client's **wchar** code set before passing the data to the client-side ORB. Note that this conversion could be an identity mapping if the internal representation of narrow and wide characters is the same as that of the native code set(s). The client-side ORB now converts all **char** data from SJIS to eucJP and all **wchar** data from the client's **wchar** code set to Unicode, and then transmits to the server side.

The server side ORB and stubs convert the eucJP data and Unicode data into C++'s internal representation for **char**s and **wchar**s as dictated by the IDL operation signatures. Notice that when the data arrives at the server side it does not look any different from data arriving from a non-generic environment (e.g., that is just like the server itself). In other words, the mappings to generic character environments do not affect the code set conversion framework.

7.10.4 Example of Generic Environment Mapping

This sub clause shows how **char**, **wchar**, **string**, and **wstring** can be mapped to the generic C/C++ macros of the Windows environment. This is merely to illustrate one possibility. This sub clause is not normative and is applicable only in generic environments. See Mapping to Generic Character Environments on page 54.

7.10.4.1 Generic Mappings

Char and **string** are mapped to C/C++ **`char`** and **`char*`** as per the standard C/C++ mappings. **wchar** is mapped to the **`TCHAR`** macro which expands to either **`char`** or **`wchar_t`** depending on whether **`_UNICODE`** is defined. **wstring** is mapped to pointers to **`TCHAR`** as well as to the string class **`CORBA::Wstring_var`**. Literal strings in IDL are mapped to the **`_TEXT`** macro as in **`_TEXT(<literal>)`**.

7.10.4.2 Interoperation and Generic Mappings

We now illustrate how the interoperation works with the above generic mapping. Consider an IDL interface operation with a **wstring** parameter, a client for the operation which is compiled and run on a Windows 95 machine, and a server for the operation which is compiled and run on a Windows NT machine. Assume that the locale (and/or the environment variables for CNCS for **wchar** representation) on the Windows 95 client indicates the client's native code set to be SJIS, and that the corresponding server's native code set is Unicode. The code set negotiation in this case will probably choose Unicode as the TCS-W.

Both the client and server sides will be compiled with **`_UNICODE`** defined. The IDL type **wstring** will be represented as a string of **`wchar_t`** on the client. However, since the client's locale or environment indicates that the CNCS for wide characters is SJIS, the client side ORB will get the **wstring** parameter encoded as a SJIS multi-byte string (since that is the client's native code set), which it will then convert to Unicode before transmitting to the server. On the server side the ORB has no conversions to do since the TCS-W matches the server's native code set for wide characters.

We therefore notice that the code set conversion framework handles the necessary translations between byte-oriented and wide forms.

7.10.5 Relevant OSFM Registry Interfaces

7.10.5.1 Character and Code Set Registry

The OSF character and code set registry is defined in *OSF Character and Code Set Registry* (see References in the Preface) and current registry contents may be obtained directly from the Open Software Foundation (obtain via anonymous ftp to ftp.opengroup.org:/pub/code_set_registry). This registry contains two parts: character sets and code sets. For each listed code set, the set of character sets encoded by this code set is shown.

Each 32-bit code set value consists of a high-order 16-bit organization number and a 16-bit identification of the code set within that organization. As the numbering of organizations starts with 0x0001, a code set null value (0x00000000) may be used to indicate an unknown code set.

When associating character sets and code sets, OSF uses the concept of "fuzzy equality," meaning that a code set is shown as encoding a particular character set if the code set can encode "most" of the characters.

"Compatibility" is determined with respect to two code sets by examining their entries in the registry, paying special attention to the character sets encoded by each code set. For each of the two code sets, an attempt is made to see if there is at least one (fuzzy-defined) character set in common, and if such a character set is found, then the assumption is made that these code sets are "compatible." Obviously, applications that exploit parts of a character set not properly encoded in this scheme will suffer information loss when communicating with another application in this "fuzzy" scheme.

The ORB is responsible for accessing the OSF registry and determining "compatibility" based on the information returned.

OSF members and other organizations can request additions to both the character set and code set registries by email to *cs-registry@opengroup.org*; in particular, one range of the code set registry (**`0xf5000000`** through **`0xffffffff`**) is reserved for organizations to use in identifying sets that are not registered with the OSF (although such use would not facilitate interoperability without registration).

7.10.5.2 Access Routines

The following routines are for accessing the OSF character and code set registry. These routines map a code set string name to code set id and vice versa. They also help in determining character set compatibility. These routine interfaces, their semantics and their actual implementation are not normative (i.e., ORB vendors do not have to bundle the OSF registry implementation with their products for compliance).

The following routines are adopted from *RPC Runtime Support For I18N Characters - Functional Specification* (see References in the Preface).

7.10.5.2.1 dce_cs_loc_to_rgy

Maps a local system-specific string name for a code set to a numeric code set value specified in the code set registry.

Synopsis

```
void dce_cs_loc_to_rgy(
idl_char *local_code_set_name,
unsigned32 *rgy_code_set_value,
unsigned16 *rgy_char_sets_number,
unsigned16 **rgy_char_sets_value,
error_status_t *status);
```

Parameters - Input	
local_code_set_name	A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes plus a terminating NULL character.
Parameters - Output	
rgy_code_set_value 0	The registered integer value that uniquely identifies the code set specified by local_code_set_name.
rgy_char_sets_number	The number of character sets that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter.
rgy_char_sets_value	A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter. The routine dynamically allocates this value.

status	Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. The possible status codes and their meanings are as follows: • dce_cs_c_ok – Code set registry access operation succeeded. • dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info. • dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set name specified. • dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the name specified.

Description

The dce_cs_loc_to_rgy() routine maps operating system-specific names for character/code set encodings to their unique identifiers in the code set registry.

The dce_cs_loc_to_rgy() routine takes as input a string that holds the host-specific “local name” of a code set and returns the corresponding integer value that uniquely identifies that code set, as registered in the host’s code set registry. If the integer value does not exist in the registry, the routine returns the status dce_cs_c_unknown.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a code set value from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the array after it is used, since the array is dynamically allocated.

7.10.5.2.2 dce_cs_rgy_to_loc

Maps a numeric code set value contained in the code set registry to the local system-specific name for a code set.

Synopsis

```
void dce_cs_rgy_to_loc(
     unsigned32 *rgy_code_set_value,
     idl_char **local_code_set_name,
     unsigned16 *rgy_char_sets_number,
     unsigned16 **rgy_char_sets_value,
     error_status_t *status);
```

Parameters - Input	
rgy_code_set_value	The registered hexadecimal value that uniquely identifies the code set.
Parameters - Output	

local_code_set_name	A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes and a terminating NULL character.
rgy_char_sets_number	The number of character sets that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value.
rgy_char_sets_value	A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value. The routine dynamically allocates this value.
status	Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. The possible status codes and their meanings are as follows: • dce_cs_c_ok – Code set registry access operation succeeded. • dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info. • dce_cs_c_unknown – The requested code set value was not found in the code set registry. • dce_cs_c_notfound – No local code set name was found in the registry that corresponds to the specific code set registry ID value. This implies that the code set is not supported in the local system environment.

Description

The dce_cs_rgy_to_loc() routine maps a unique identifier for a code set in the code set registry to the operating system-specific string name for the code set, if it exists in the code set registry.

The dce_cs_rgy_to_loc() routine takes as input a registered integer value of a code set and returns a string that holds the operating system-specific, or local name, of the code set.

If the code set identifier does not exist in the registry, the routine returns the status dce_cs_c_unknown and returns an undefined string.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a local code set name from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the rgy_char_sets_value array after it is used.

7.10.5.2.3 rpc_cs_char_set_compat_check

Evaluates character set compatibility between a client and a server.

Synopsis

```
void rpc_cs_char_set_compat_check(
     unsigned32 client_rgy_code_set_value,
```

```
unsigned32 server_rgy_code_set_value,
error_status_t *status);
```

Parameters - Input	
client_rgy_code_set_value	The registered hexadecimal value that uniquely identifies the code set that the client is using as its local code set.
server_rgy_code_set_value	The registered hexadecimal value that uniquely identifies the code set that the server is using as its local code set.
Parameters - Output	
status	Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. The possible status codes and their meanings are as follows: • rpc_s_ok – Successful status. • rpc_s_ss_no_compat_charsets – No compatible code set found. The client and server do not have a common encoding that both could recognize and convert. • The routine can also return status codes from the dce_cs_rgy_to_loc() routine.

Description

The rpc_cs_char_set_compat_check() routine provides a method for determining character set compatibility between a client and a server; if the server's character set is incompatible with that of the client, then connecting to that server is most likely not acceptable, since massive data loss would result from such a connection.

The routine takes the registered integer values that represent the code sets that the client and server are currently using and calls the code set registry to obtain the registered values that represent the character set(s) that the specified code sets support. If both client and server support just one character set, the routine compares client and server registered character set values to determine whether or not the sets are compatible. If they are not, the routine returns the status message rpc_s_ss_no_compat_charsets.

If the client and server support multiple character sets, the routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible, and returns a success status code to the caller.

7.10.5.2.4 rpc_rgy_get_max_bytes

Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host

Synopsis

```
void rpc_rgy_get_max_bytes(
unsigned32 rgy_code_set_value,
unsigned16 *rgy_max_bytes,
error_status_t *status);
```

Parameters - Input	
rgy_code_set_value	The registered hexadecimal value that uniquely identifies the code set.
Parameters - Output	
rgy_max_bytes	The registered decimal value that indicates the number of bytes this code set uses to encode one character.
status	Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not. The possible status codes and their meanings are as follows: • rpc_s_ok – Operation succeeded. • dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info. • dce_cs_c_unknown – No code set value was not found in the registry that corresponds to the code set value specified. • dce_cs_c_notfound – No local code set name was found in the registry that corresponds to the value specified.

Description

The rpc_rgy_get_max_bytes() routine reads the code set registry on the local host. It takes the specified registered code set value, uses it as an index into the registry, and returns the decimal value that indicates the number of bytes that the code set uses to encode one character.

This information can be used for buffer sizing as part of the procedure to determine whether additional storage needs to be allocated for conversion between local and network code sets.

8 Building Inter-ORB Bridges

8.1 Introduction

This clause provides an implementation-oriented conceptual framework for the construction of bridges to provide interoperability between ORBs. It focuses on the layered *request level bridges* that the CORBA Core specifications facilitate, although ORBs may always be internally modified to support bridges.

Key features of the specifications for inter-ORB bridges are as follows:

- Enables requests from one ORB to be translated to requests on another.
- Provides support for managing tables keyed by object references.

The OMG IDL specification for interoperable object references, which are important to inter-ORB bridging, is shown in Interoperable Object References: IORs on page 25.

8.2 In-Line and Request-Level Bridging

Bridging of an invocation between a client in one domain and a server object in another domain can be mediated through a standardized mechanism, or done immediately using non-standard ones.

The question of how this bridging is constructed is broadly independent of whether the bridging uses a standardized mechanism. There are two possible options for where the bridge components are located:

1. Code inside the ORB may perform the necessary translation or mappings; this is termed *in-line bridging*.
2. Application style code outside the ORB can perform the translation or mappings; this is termed *request-level bridging*.

Request-level bridges that mediate through a common protocol (using networking, shared memory, or some other IPC provided by the host operating system) between distinct execution environments will involve components, one in each ORB, known as “half bridges.”

When that mediation is purely internal to one execution environment, using a shared programming environment’s binary interfaces to CORBA- and OMG-IDL-defined data types, this is known as a “full bridge.”[1] From outside the execution environment this will appear identical to some kinds of in-line bridging, since only that environment knows the construction techniques used. However, full bridges more easily support portable policy mediation components, because of their use of only standard CORBA programming interfaces.

Network protocols may be used immediately “in-line,” or to mediate between request-level half bridges. The General Inter-ORB Protocol can be used in either manner. In addition, this text provides for Environment Specific Inter-ORB Protocols (ESIOP), allowing for alternative mediation mechanisms.

1. Special initialization supporting object referencing domains (e.g., two protocols) to be exposed to application programmers to support construction of this style bridge.

Note that mediated, request-level half-bridges can be built by anyone who has access to an ORB, without needing information about the internal construction of that ORB. Immediate-mode request-level half-bridges (i.e., ones using non-standard mediation mechanisms) can be built similarly without needing information about ORB internals. Only in-line bridges (using either standard or non-standard mediation mechanisms) need potentially proprietary information about ORB internals.

8.2.1 In-line Bridging

In-line bridging is in general the most direct method of bridging between ORBs. It is structurally similar to the engineering commonly used to bridge between systems within a single ORB (e.g., mediating using some common inter-process communications scheme, such as a network protocol). This means that implementing in-line bridges involves as fundamental a set of changes to an ORB as adding a new inter-process communications scheme. (Some ORBs may be designed to facilitate such modifications, though.)

In this approach, the required bridging functionality can be provided by a combination of software components at various levels:

- As additional or alternative services provided by the underlying ORBs
- As additional or alternative stub and skeleton code.

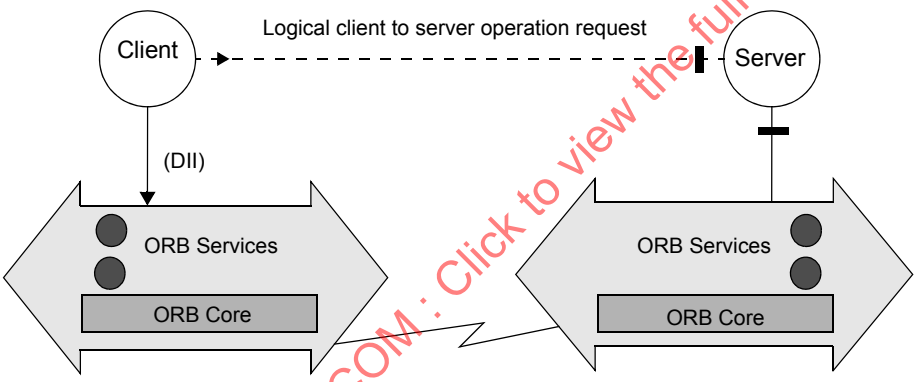


Figure 8.1 - In-Line bridges are built using *ORB internal APIs*

8.2.2 Request-level Bridging

The general principle of request-level bridging is as follows:

1. The original request is passed to a proxy object in the client ORB.
2. The proxy object translates the request contents (including the target object reference) to a form that will be understood by the server ORB.
3. The proxy invokes the required operation on the apparent server object.
4. Any operation result is passed back to the client via a complementary route.

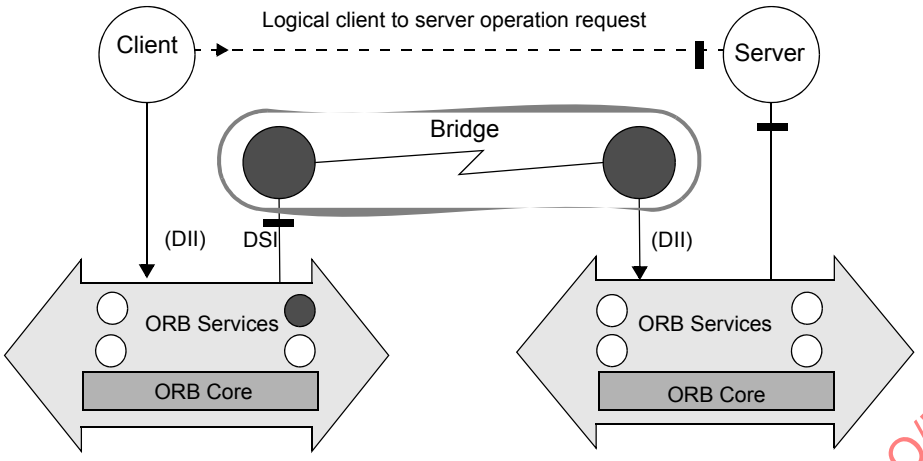


Figure 8.2 - Request-Level bridges are built using *public ORB APIs*

The request translation involves performing object reference mapping for all object references involved in the request (the target, explicit parameters, and perhaps implicit ones such as transaction context). As elaborated later, this translation may also involve mappings for other domains: the security domain of **CORBA::Principal** parameters, type identifiers, and so on.

It is a language mapping requirement of the CORBA Core specification that all dynamic typing APIs (e.g., **Any, NamedValue**) support such manipulation of parameters even when the bridge was not created with compile-time knowledge of the data types involved.

8.2.3 Collocated ORBs

In the case of immediate bridging (i.e., not via a standardized, external protocol) the means of communication between the client-side bridge component and that on the server-side is an entirely private matter. One possible engineering technique optimizes this communication by coalescing the two components into the same system or even the same address space. In the latter case, accommodations must be made by both ORBs to allow them to share the same execution environment.

Similar observations apply to request-level bridges, which in the case of collocated ORBs use a common binary interface to all OMG IDL-defined data as their mediating data format.

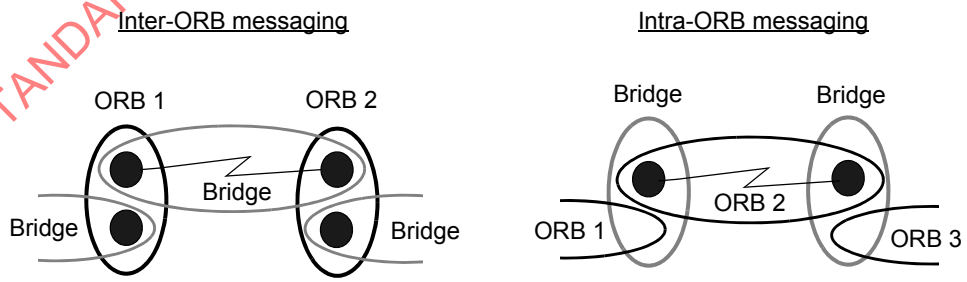


Figure 8.3 - When the two ORBs are collocated in a bridge execution environment, network communications will be purely intra-ORB. If the ORBs are not collocated, such communications must go between ORBs.

An advantage of using bridges spanning collocated ORBs is that all external messaging can be arranged to be intra-ORB, using whatever message-passing mechanisms each ORB uses to achieve distribution within a single ORB, multiple machine system. That is, for bridges between networked ORBs such a bridge would add only a single “hop,” a cost analogous to normal routing.

8.3 Proxy Creation and Management

Bridges need to support arbitrary numbers of proxy objects, because of the (bidirectional) object reference mappings required. The key schemes for creating and managing proxies are *reference translation* and *reference encapsulation*, as discussed in Handling of Referencing Between Domains on page 24.

- Reference translation approaches are possible with CORBA V2.0 Core APIs. Proxies themselves can be created as normal objects using the Basic Object Adapter (BOA) and the Dynamic Skeleton Interface (DSI).
- Reference Encapsulation is not supported by the BOA, since it would call for knowledge of more than one ORB. Some ORBs could provide other object adapters that support such encapsulation.

Note that from the perspective of clients, they only deal with local objects; clients do not need to distinguish between proxies and other objects. Accordingly, all CORBA operations supported by the local ORB are also supported through a bridge. The ORB used by the client might, however, be able to recognize that encapsulation is in use, depending on how the ORB is implemented.

Also, note that the **CORBA::InterfaceDef** used when creating proxies (e.g., the one passed to **CORBA::BOA::create**) could be either a proxy to one in the target ORB, or could be an equivalent local one. When the domains being bridged include a type domain, then the **InterfaceDef** objects cannot be proxies since type descriptions will not have the same information. When bridging CORBA-compliant ORBs, type domains by definition do not need to be bridged.

8.4 Interface-specific Bridges and Generic Bridges

Request-level bridges may be:

- *Interface-specific*: they support predetermined IDL interfaces only, and are built using IDL-compiler generated stub and skeleton interfaces.
- *Generic*: capable of bridging requests to server objects of arbitrary IDL interfaces, using the interface repository and other dynamic invocation support (DII and DSI).

Interface-specific bridges may be more efficient in some cases (a generic bridge could conceivably create the same stubs and skeletons using the interface repository), but the requirement for prior compilation means that this approach offers less flexibility than using generic bridges.

8.5 Building Generic Request-Level Bridges

The CORBA Core specifications define the following interfaces. These interfaces are of particular significance when building a generic request-level bridge:

- ***Dynamic Invocation Interface (DII)*** lets the bridge make arbitrary invocations on object references whose types may not have been known when the bridge was developed or deployed.
- ***Dynamic Skeleton Interface (DSI)*** lets the bridge handle invocations on proxy object references that it implements, even when their types may not have been known when the bridge was developed or deployed.
- ***Interface Repositories*** are consulted by the bridge to acquire the information used to drive DII and DSI, such as the type codes for operation parameters, return values, and exceptions.

- ***Object Adapters*** (such as the Basic Object Adapter) are used to create proxy object references both when bootstrapping the bridge and when mapping object references, which are dynamically passed from one ORB to the other.
- ***CORBA Object References*** support operations to fully describe their interfaces and to create tables mapping object references to their proxies (and vice versa).

Interface repositories accessed on either side of a half bridge need not have the same information, though of course the information associated with any given repository ID (e.g., an interface type ID, exception ID) or operation ID must be the same.

Using these interfaces and an interface to some common transport mechanism such as TCP, portable request-level half bridges connected to an ORB can:

- Use DSI to translate all CORBA invocations on proxy objects to the form used by some mediating protocol such as IIOP (see the *General Inter-ORB Protocol* clause in this standard).
- Translate requests made using such a mediating protocol into DII requests on objects in the ORB.

As noted in In-Line and Request-Level Bridging on page 63, translating requests and responses (including exceptional responses) involves mapping object references (and other explicit and implicit parameter data) from the form used by the ORB to the form used by the mediating protocol, and vice versa. Explicit parameters, which are defined by an operation's OMG-IDL definition, are presented through DII or DSI and are listed in the Interface Repository entry for any particular operation.

Operations on object references such as **hash()** and **is_equivalent()** may be used to maintain tables that support such mappings. When such a mapping does not exist, an object adapter is used to create ORB-specific proxy object references, and bridge-internal interfaces are used to create the analogous data structure for the mediating protocol.

8.6 Bridging Non-Referencing Domains

In the simplest form of request-level bridging, the bridge operates only on IDL-defined data, and bridges only object reference domains. In this case, a proxy object in the client ORB acts as a representative of the target object and is, in almost any practical sense, indistinguishable from the target server object - indeed, even the client ORB will not be aware of the distinction.

However, as alluded to above, there may be multiple domains that need simultaneous bridging. The transformation and encapsulation schemes described above may not apply in the same way to Principal or type identifiers. Request-level bridges may need to translate such identifiers, in addition to object references, as they are passed as explicit operation parameters.

Moreover, there is an emerging class of "implicit context" information that ORBs may need to convey with any particular request, such as transaction and security context information. Such parameters are not defined as part of an operation's OMG-IDL signature, hence are "implicit" in the invocation context. Bridging the domains of such implicit parameters could involve additional kinds of work, needing to mediate more policies than bridging the object reference, Principal, and type domains directly addressed by CORBA.

CORBA does not yet have a generic way (including support for both static and dynamic invocations) to expose such implicit context information.

8.7 Bootstrapping Bridges

A particularly useful policy for setting up bridges is to create a pair of proxies for two Naming Service naming contexts (one in each ORB) and then install those proxies as naming contexts in the other ORB's naming service. (The Naming Service is described in the Naming Service specification.) This will allow clients in either ORB to transparently perform naming context lookup operations on the other ORB, retrieving (proxy) object references for other objects in that ORB. In this way, users can access facilities that have been selectively exported from another ORB, through a naming context, with no administrative action beyond exporting those initial contexts. (See the *ORB Interface* clause in CORBA, Part 1 for additional information.)

This same approach may be taken with other discovery services, such as a trading service or any kind of object that could provide object references as operation results (and in "out" parameters). While bridges can be established that only pass a predefined set of object references, this kind of minimal connectivity policy is not always desirable.

9 General Inter-ORB Protocol

9.1 Overview

This clause specifies a General Inter-ORB Protocol (GIOP) for ORB interoperability, which can be mapped onto any connection-oriented transport protocol that meets a minimal set of assumptions. This clause also defines a specific mapping of the GIOP, which runs directly over TCP/IP connections, called the Internet Inter-ORB Protocol (IIOP). The IIOP must be supported by conforming networked ORB products regardless of other aspects of their implementation. Such support does not require using it internally; conforming ORBs may also provide bridges to this protocol.

9.2 Goals of the General Inter-ORB Protocol

The GIOP and IIOP support protocol-level ORB interoperability in a general, low-cost manner. The following objectives were pursued vigorously in the GIOP design:

- ***Widest possible availability*** - The GIOP and IIOP are based on the most widely-used and flexible communications transport mechanism available (TCP/IP), and defines the minimum additional protocol layers necessary to transfer CORBA requests between ORBs.
- ***Simplicity*** - The GIOP is intended to be as simple as possible, while meeting other design goals. Simplicity is deemed the best approach to ensure a variety of independent, compatible implementations.
- ***Scalability*** - The GIOP/IIOP protocol should support ORBs, and networks of bridged ORBs, to the size of today's Internet, and beyond.
- ***Low cost*** - Adding support for GIOP/IIOP to an existing or new ORB design should require small engineering investment. Moreover, the run-time costs required to support IIOP in deployed ORBs should be minimal.
- ***Generality*** - While the IIOP is initially defined for TCP/IP, GIOP message formats are designed to be used with any transport layer that meets a minimal set of assumptions; specifically, the GIOP is designed to be implemented on other connection-oriented transport protocols.
- ***Architectural neutrality*** - The GIOP specification makes minimal assumptions about the architecture of agents that will support it. The GIOP specification treats ORBs as opaque entities with unknown architectures.

The approach a particular ORB takes to providing support for the GIOP/IIOP is undefined. For example, an ORB could choose to use the IIOP as its internal protocol, it could choose to externalize IIOP as much as possible by implementing it in a half-bridge, or it could choose a strategy between these two extremes. All that is required of a conforming ORB is that some entity or entities in, or associated with, the ORB be able to send and receive IIOP messages.

9.3 GIOP Overview

The GIOP specification consists of the following elements:

- *The Common Data Representation (CDR) definition*. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for "on-the-wire" transfer between ORBs and Inter-ORB bridges (agents).
- *GIOP Message Formats*. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.

- *GIOP Transport Assumptions*. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

The IIOP specification adds the following element to the GIOP specification:

- *Internet IOP Message Transport*. The IIOP specification describes how agents open TCP/IP connections and use them to transfer GIOP messages.

The IIOP is not a separate specification; it is a specialization, or mapping, of the GIOP to a specific transport (TCP/IP). The GIOP specification (without the transport-specific IIOP element) may be considered as a separate conformance point for future mappings to other transport layers.

The complete OMG IDL specifications for the GIOP and IIOP are shown in OMG IDL on page 119. Fragments of the specification are used throughout this document as necessary.

9.3.1 Common Data Representation (CDR)

CDR is a transfer syntax, mapping from data types defined in OMG IDL to a bicanonical, low-level representation for transfer between agents. CDR has the following features:

- ***Variable byte ordering*** - Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.

- ***Aligned primitive types*** - Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.

- ***Complete OMG IDL Mapping*** - CDR describes representations for all OMG IDL data types, including transferable pseudo-objects such as TypeCodes. Where necessary, CDR defines representations for data types whose representations are undefined or implementation-dependent in the CORBA Core specifications.

9.3.2 GIOP Message Overview

The GIOP specifies formats for messages that are exchanged between inter-operating ORBs. GIOP message formats have the following features:

- ***Few, simple messages*** - With only seven message formats, the GIOP supports full CORBA functionality between ORBs, with extended capabilities supporting object location services, dynamic migration, and efficient management of communication resources. GIOP semantics require no format or binding negotiations. In most cases, clients can send requests to objects immediately upon opening a connection.

- ***Dynamic object location*** - Many ORBs' architectures allow an object implementation to be activated at different locations during its lifetime, and may allow objects to migrate dynamically. GIOP messages provide support for object location and migration, without requiring ORBs to implement such mechanisms when unnecessary or inappropriate to an ORB's architecture.

- ***Full CORBA support*** - GIOP messages directly support all functions and behaviors required by CORBA, including exception reporting, passing operation context, and remote object reference operations (such as **CORBA::Object::get_interface**).

GIOP also supports passing service-specific context, such as the transaction context defined by the Transaction Service (the Transaction Service is described in *CORBAservices: Common Object Service Specifications*). This mechanism is designed to support any service that requires service related context to be implicitly passed with requests.

9.3.3 GIOP Message Transfer

The GIOP specification is designed to operate over any connection-oriented transport protocol that meets a minimal set of assumptions (described in GIOP Message Transport on page 107). GIOP uses underlying transport connections in the following ways:

- ***Asymmetrical connection usage*** - The GIOP defines two distinct roles with respect to connections, client, and server. The client side of a connection originates the connection, and sends object requests over the connection. In GIOP versions 1.0 and 1.1, the server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction, which was included to make GIOP 1.0 and 1.1 much simpler and avoid certain race conditions, has been relaxed for GIOP version 1.2 and later, as specified in the BiDirectional GIOP specification, see Bi-Directional GIOP on page 115.

- ***Request multiplexing*** - If desirable, multiple clients within an ORB may share a connection to send requests to a particular ORB or server. Each request uniquely identifies its target object. Multiple independent requests for different objects, or a single object, may be sent on the same connection.

- ***Overlapping requests*** - In general, GIOP message ordering constraints are minimal. GIOP is designed to allow overlapping asynchronous requests; it does not dictate the relative ordering of requests or replies. Unique request/reply identifiers provide proper correlation of related messages. Implementations are free to impose any internal message ordering constraints required by their ORB architectures.

- ***Connection management*** - GIOP defines messages for request cancellation and orderly connection shutdown. These features allow ORBs to reclaim and reuse idle connection resources.

- ***GIOP versions for requests and replies*** - The GIOP version of the message carrying a response to a request shall be the same as the GIOP version of the message carrying the request. This rule does not apply when the server is responding with a MessageError because it does not support the GIOP minor version in the request.

9.4 CDR Transfer Syntax

The Common Data Representation (CDR) transfer syntax is the format in which the GIOP represents OMG IDL data types in an octet stream.

An octet stream is an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport. For the purposes of this discussion, an octet stream is an arbitrarily long (but finite) sequence of eight-bit values (octets) with a well-defined beginning. The octets in the stream are numbered from *0* to *n-1*, where *n* is the size of the stream. The numeric position of an octet in the stream is called its *index*. Octet indices are used to calculate alignment boundaries, as described in Alignment on page 72.

GIOP defines two distinct kinds of octet streams:

- Message - an octet stream constituting the basic unit of information exchange in GIOP, described in detail in GIOP Message Formats on page 93.

- Encapsulation - an octet stream into which OMG IDL data structures may be marshaled independently, apart from any particular message context, described in detail in Encapsulation on page 79.

9.4.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see GIOP Message Formats on page 93) include tags in message headers that indicate the byte ordering in the message. Encapsulations include an initial flag that indicates the byte ordering within the encapsulation, described in Encapsulation on page 79. The byte ordering of any encapsulation may be different from the message or encapsulation within which it is nested. It is the responsibility of the message recipient to translate byte ordering if necessary. Primitive data types are encoded in multiples of octets. An **octet** is an 8-bit value.

9.4.1.1 Alignment

In order to allow primitive data to be moved into and out of octet streams with instructions specifically designed for those primitive data types, in CDR all primitive data types must be aligned on their natural boundaries (i.e., the alignment boundary of a primitive datum is equal to the size of the datum in **octets**). Any primitive of size *n* octets must start at an octet stream index that is a multiple of *n*. In CDR, *n* is one of 1, 2, 4, or 8.

Where necessary, an alignment gap precedes the representation of a primitive datum. The value of **octets** in alignment gaps is undefined. A gap must be the minimum size necessary to align the following primitive. Table 9.1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 9.1

TYPE	OCTET ALIGNMENT
char	1
wchar	1, 2 or 4 for GIOP 1.1 1 for GIOP 1.2 and later
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enum	4

Alignment is defined above as being relative to the beginning of an octet stream. The first octet of the stream is octet index zero (0); any data type may be stored starting at this index. Such octet streams begin at the start of a GIOP message header (see GIOP Message Header on page 94) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See Encapsulation on page 79).

9.4.1.2 Integer Data Types

Figure 9.1 on page 73 illustrates the representations for OMG IDL integer data types, including the following data types:

- **short**
- **unsigned short**
- **long**
- **unsigned long**
- **long long**
- **unsigned long long**

The figure illustrates bit ordering and size. Signed types (**short**, **long,** and **long long**) are represented as two's complement numbers; unsigned versions of these types are represented as unsigned binary numbers.

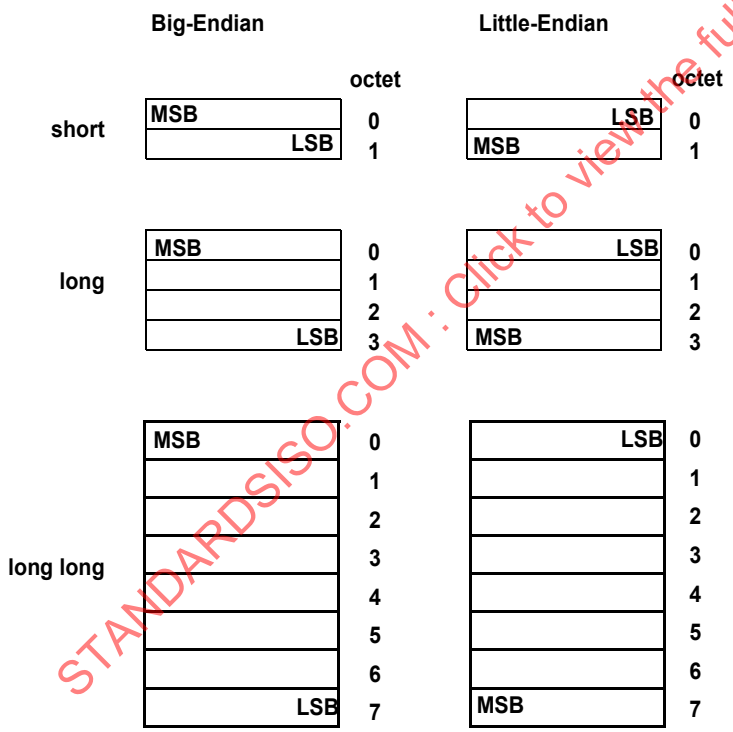


Figure 9.1 - Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

9.4.1.3 Floating Point Data Types

Figure 9.3 on page 78 illustrates the representation of floating point numbers. These exactly follow the IEEE standard formats for floating point numbers[1], selected parts of which are abstracted here for explanatory purposes. The diagram shows three different components for floating points numbers, the sign bit (s), the exponent (e), and the fractional part (f) of the mantissa. The sign bit has values of 0 or 1, representing positive and negative numbers, respectively.

For single-precision float values the exponent is 8 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 127. The fractional mantissa (f1 - f3) is a 23-bit value f where 1.0 <= f < 2.0, f1 being most significant and f3 being least significant. The value of a normalized number is described by:

$$-1^{sign} \times 2^{(exponent - 127)} \times (1 + fraction)$$

For double-precision values the exponent is 11 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are most significant. The exponent is represented as excess 1023. The fractional mantissa (f1 - f7) is a 52-bit value m where 1.0 <= m < 2.0, f1 being most significant and f7 being least significant. The value of a normalized number is described by:

$$-1^{sign} \times 2^{(exponent - 1023)} \times (1 + fraction)$$

For double-extended floating-point values the exponent is 15 bits long, comprising e1 and e2 in the figure, where the 7 bits in e1 are the most significant. The fractional mantissa (f1 through f14) is 112 bits long, with f1 being the most significant. The value of a **long double** is determined by:

$$-1^{sign} \times 2^{(exponent - 16383)} \times (1 + fraction)$$

1. "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

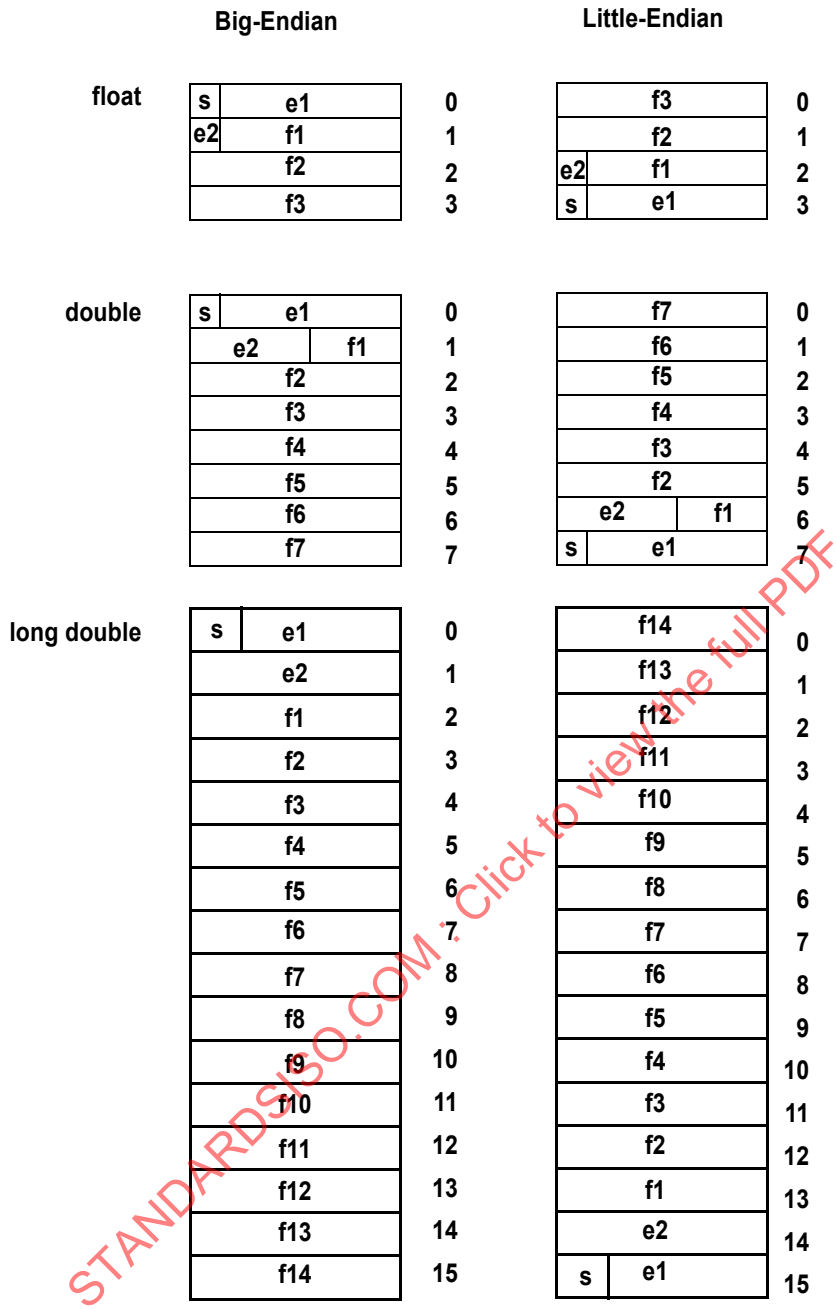


Figure 9.2 - Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single, double precision, and double extended floating point numbers.

9.4.1.4 Octet

Octets are uninterpreted 8-bit values whose contents are guaranteed not to undergo any conversion during transmission. For the purposes of describing possible **octet** values in this part of ISO/IEC 19500, octets may be considered as unsigned 8-bit integer values.

9.4.1.5 Boolean

Boolean values are encoded as single octets, where **TRUE** is the value 1, and **FALSE** as 0.

9.4.1.6 Character Types

An IDL character is represented as a single octet; the code set used for transmission of character data (e.g., TCS-C) between a particular client and server ORBs is determined via the process described in Code Set Conversion on page 45. In the case of multi-byte encodings of characters, a single instance of the **char** type may hold only one octet of any multi-byte character encoding.

NOTE: Full representation of multi-byte characters will require the use of an array of IDL **char** variables.

For GIOP version 1.1, the transfer syntax for an IDL wide character depends on whether the transmission code set (TCS-W, which is determined via the process described in Code Set Conversion on page 45) is byte-oriented or non-byte-oriented:

- Byte-oriented (e.g., SJIS). Each wide character is represented as one or more octets, as defined by the selected TCS-W.
- Non-byte-oriented (e.g., Unicode UTF-16). Each wide character is represented as one or more codepoints. A codepoint is the same as "Coded-Character data element," or "CC data element" in ISO terminology. Each codepoint is encoded using a fixed number of bits as determined by the selected TCS-W. The OSF Character and Code Set Registry may be examined using the interfaces in Relevant OSFM Registry Interfaces on page 56 to determine the maximum length (max_bytes) of any character codepoint.

For GIOP version 1.2, and later **wchar** is encoded as an unsigned binary octet value, followed by the elements of the octet sequence representing the encoded value of the **wchar**. The initial octet contains a count of the number of elements in the sequence, and the elements of the sequence of octets represent the **wchar**, using the negotiated wide character encoding.

NOTE: The GIOP 1.2 and later encoding of **wchar** is similar to the encoding of an octet sequence, except for its use of a single octet to encode the value of the length.

For GIOP versions prior to 1.2 and later, interoperability for wchar is limited to the use of two- octet fixed-length encoding.

Wchar values in encapsulations are assumed to be encoded using GIOP version 1.2 and later CDR.

If UTF-16 is selected as the TCS-W, the CDR encoding purposes can be big endian or little endian, but defaults to big endian. By placing a BOM (byte order marker) at the front of the wstring or wchar encoding, it can be sent either big-endian or little-endian. In particular, the CDR rules for endian-ness of UTF-16 encoded wstring or wchar values are as follows:

- If the first two bytes (after the length indication) are FE FF, it's big-endian.
- If the first two bytes (after the length indication) are FF FE, it's little-endian.
- **If the first two b**ytes (after the length indication) are neither, it's big-endian.

If an ORB decides to use BOM to indicate endianness, it shall add the BOM to the beginning of wchar or wstring values when encoding the value, since it is not present in wchar or wstring values passed by the user.

If a BOM is present at the beginning of a wchar or wstring received in a GIOP message, the ORB shall remove the BOM before passing the value to the user.

If a client orb erroneously sends **wchar** or **wstring** data in a GIOP 1.0 message, the server shall generate a MARSHAL standard system exception, with standard minor code 5.

If a server erroneously sends **wchar** data in a GIOP 1.0 response, the client ORB shall raise a MARSHAL exception to the client application with standard minor code 6.

For GIOP 1.1, 1.2, and 1.3, UCS-2 and UCS-4 should be encoded using the endianess of the GIOP message, for backward compatibility.

For GIOP 1.4, the byte order rules for UCS-2 and UCS-4 are the same as for UTF-16.

UTF-16LE and UTF-16BE, from IANA codeset registry, have their own endianess definition. Thus these should be encoded using the endianess specified by their endianness definition.

9.4.2 OMG IDL Constructed Types

Constructed types are built from OMG IDL's data types using facilities defined by the OMG IDL language.

9.4.2.1 Alignment

Constructed types have no alignment restrictions beyond those of their primitive components. The alignment of those primitive types is not intended to support use of marshaling buffers as equivalent to the implementation of constructed data types within any particular language environment. GIOP assumes that agents will usually construct structured data types by copying primitive data between the marshaled buffer and the appropriate in-memory data structure layout for the language mapping implementation involved.

9.4.2.2 Struct

The components of a structure are encoded in the order of their declaration in the structure. Each component is encoded as defined for its data type.

9.4.2.3 Union

Unions are encoded as the discriminant tag of the type specified in the union declaration, followed by the representation of any selected member, encoded as its type indicates.

9.4.2.4 Array

Arrays are encoded as the array elements in sequence. As the array length is fixed, no length values are encoded. Each element is encoded as defined for the type of the array. In multidimensional arrays, the elements are ordered so the index of the first dimension varies most slowly, and the index of the last dimension varies most quickly.

9.4.2.5 Sequence

Sequences are encoded as an unsigned long value, followed by the elements of the sequence. The initial unsigned long contains the number of elements in the sequence. The elements of the sequence are encoded as specified for their type.

9.4.2.6 Enum

Enum values are encoded as unsigned longs. The numeric values associated with enum identifiers are determined by the order in which the identifiers appear in the enum declaration. The first enum identifier has the numeric value zero (0). Successive enum identifiers take ascending numeric values, in order of declaration from left to right.

9.4.2.7 Strings and Wide Strings

A string is encoded as an **unsigned long** indicating the length of the string in octets, followed by the string value in single- or multi-byte form represented as a sequence of octets. The string contents include a single terminating null character. The string length includes the null character, so an empty string has the length of the encoding of the null character in the transmission character set.

For GIOP version 1.1, 1.2, and 1.3, when encoding a string, always encode the length as the total number of bytes used by the encoding string, regardless of whether the encoding is byte-oriented or not.

For GIOP version 1.1, a wide string is encoded as an **unsigned long** indicating the length of the string in octets or unsigned integers (determined by the transfer syntax for wchar) followed by the individual wide characters. The string contents include a single terminating null character. The string length includes the null character. The terminating null character for a wstring is also a wide character.

For GIOP version 1.2 and 1.3, when encoding a **wstring**, always encode the length as the total number of octets used by the encoded value, regardless of whether the encoding is byte-oriented or not. For GIOP version 1.2 and 1.3 a **wstring** is not terminated by a null character. In particular, in GIOP version 1.2 and 1.3 a length of 0 is legal for **wstring**.

NOTE: For GIOP versions prior to 1.2 and 1.3, interoperability for **wstring** is limited to the use of two-octet fixed-length encoding.

Wstring values in encapsulations are assumed to be encoded using GIOP version 1.2 and 1.3 CDR.

9.4.2.8 Fixed-Point Decimal Type

The IDL **fixed** type has no alignment restrictions, and is represented as shown in Figure 9.4. Each **octet** contains (up to) two decimal digits. If the **fixed** type has an odd number of decimal digits, then the representation begins with the first (most significant) digit — d0 in the figure. Otherwise, this first half-octet is all zero, and the first digit is in the second half-octet — d1 in the figure. The sign configuration, in the last half-octet of the representation, is 0xD for negative numbers and 0xC for positive and zero values.

The number of digits present must equal the number of significant digits specified in the IDL definition for the fixed type being marshaled, with the exception of the inclusion of a leading 0x0 half octet when there are an even number of significant digits.

Decimal digits are encoded as hexadecimal values in each half-octet as follows:

Decimal Digit	Half-Octet Value
0	0x0
1	0x1
2	0x2
...	...
9	0x9

Figure 9.3 - Decimal Digit Encoding for Fixed Type

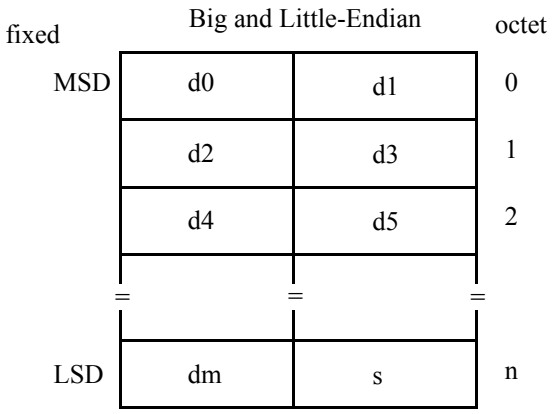


Figure 9.4 - IDL Fixed Type Representation

9.4.3 Encapsulation

Encapsulations are octet streams into which OMG IDL data structures may be marshaled independently, apart from any particular message context. Once a data structure has been encapsulated, the **octet** stream can be represented as the OMG IDL opaque data type **sequence<octet>**, which can be marshaled subsequently into a message or another encapsulation. Encapsulations allow complex constants (such as TypeCodes) to be pre-marshaled; they also allow certain message components to be handled without requiring full unmarshaling. Whenever encapsulations are used in CDR or the GIOP, they are clearly noted.

The GIOP and IIOP explicitly use encapsulations in three places: *TypeCodes* (see TypeCode on page 87), the IIOP protocol profile inside an IOR (see Object References on page 93), and in service-specific context (see Service Context on page 37). In addition, some ORBs may choose to use an encapsulation to hold the **object_key** (see IIOP IOR Profiles on page 112), or in other places that a **sequence<octet>** data type is in use.

When encapsulating OMG IDL data types, the first octet in the stream (index 0) contains a boolean value indicating the byte ordering of the encapsulated data. If the value is **FALSE** (0), the encapsulated data is encoded in big-endian order; if **TRUE** (1), the data is encoded in little-endian order, exactly like the byte order flag in GIOP message headers (see GIOP Message Header on page 94). This value is not part of the data being encapsulated, but is part of the octet stream holding the encapsulation. Following the byte order flag, the data to be encapsulated is marshaled into the buffer as defined by CDR encoding rules. Marshaled data are aligned relative to the beginning of the octet stream (the first octet of which is occupied by the byte order flag).

When the encapsulation is encoded as type **sequence<octet>** for subsequent marshaling, an unsigned long value containing the sequence length is prefixed to the octet stream, as prescribed for sequences (see Sequence on page 77). The length value is not part of the encapsulation's octet stream, and does not affect alignment of data within the encapsulation.

Note that this guarantees a four-octet alignment of the start of all encapsulated data within GIOP messages and nested encapsulations.[2]

Whenever the use of an encapsulation is specified, the GIOP version to use for encoding the encapsulation, if different than GIOP version 1.0, shall be explicitly defined (i.e., the default is GIOP 1.0).

If a parameter with IDL char or string type is defined to be carried in an encapsulation using GIOP version greater than 1.0, the transmission Code Set for characters (TCS-C), to be used when encoding the encapsulation, shall also be explicitly defined.

If a parameter with IDL wchar or wstring type is defined to be carried in an encapsulation using GIOP version greater than 1.0, the transmission Code Set for wide characters (TCS-W), to be used when encoding the encapsulation shall also be explicitly defined.

9.4.4 Value Types

Value types are built from OMG IDL's value type definitions. Their representation and encoding is defined in this sub clause.

Value types may be used to transmit and encode complex state. The general approach is to support the transmission of the data (state) and type information encoded as **RepositoryID**s.

The loading (and possible transmission) of code is outside of the scope of the GIOP definition, but enough information is carried to support it, via the CodeBase object.

The format makes a provision for the support of custom marshaling (i.e. the encoding and transmission of state using application-defined code). Consistency between custom encoders and decoders is not ensured by the protocol.

The encoding supports all of the features of value types as well as supporting the "chunking" of value types. It does so in a compact way.

At a high level the format can be described as the linearization of a graph. The graph is the depth-first exploration of the transitive closure that starts at the top-level value object and follows its "reference to value objects" fields (an ordinary remote reference is just written as an IOR). It is a recursive encoding similar to the one used for **TypeCodes**. An indirection is used to point to a value that has already been encoded.

The data members are written beginning with the highest possible base type to the most derived type in the order of their declaration.

9.4.4.1 Partial Type Information and Versioning

The format provides support for partial type information and versioning issues in the receiving context. However the encoding has been designed so that this information is only required when "advanced features" such as truncation are used.

The presence (or absence) of type information and codebase URL information is indicated by flags within the <value_tag>, which is a **long** in the range between **0x7fffff00** and **0x7fffffff** inclusive. The last octet of this tag is interpreted as follows:

- The least significant bit (<value_tag> & **0x00000001**) is the value **1** if a <codebase_URL> is present. If this bit is **0**, no <codebase_URL> follows in the encoding. The <codebase_URL> is a blank-separated list of one or more URLs.

2. Accordingly, in cases where encapsulated data holds data with natural alignment of greater than four octets, some processors may need to copy the octet data before removing it from the encapsulation. For example, an appropriate way to deal with long long discriminator type in an encapsulation for a union TypeCode is to encode the body of the encapsulation as if it was aligned at the 8 byte boundary, and then copy the encoded value into the encapsulation. This may result in long long data values inside the encapsulation being aligned on only a 4 byte boundary when viewed from outside the encapsulation.

- The second and third least significant bits (<value_tag> & **0x00000006**) are:
  - the value **0** if no type information is present in the encoding. This indicates the actual parameter is the same type as the formal argument.
  - the value **2** if only a single repository id is present in the encoding, which indicates the most derived type of the actual parameter (which may be either the same type as the formal argument or one of its derived types).
  - the value **6** if the partial type information list of repository ids is present in the encoding as a list of repository ids.

When a list of **RepositoryIDs** is present, the encoding is a **long** specifying the number of **RepositoryIDs**, followed by the **RepositoryIDs**. The first **RepositoryID** is the id for the most derived type of the value. If this type has any base types, the sending context is responsible for listing the **RepositoryIDs** for all the base types to which it is safe to truncate the value passed. These truncatable base types are listed in order, going up the derivation hierarchy. The sending context may choose to (but need not) terminate the list at any point after it has sent a **RepositoryID** for a type well known to the receiving context. A well known type is any of the following:

- A type that is a formal parameter, result of the method call, or exception, for which this GIOP message is being marshaled.
- A base type of a well known type.
- A member type of a well known type.
- An element type of a well known type.

For value types that have an RMI: **RepositoryId**, ORBs must include at least the most derived **RepositoryId**, in the value type encoding.

For value types marshaled as abstract interfaces (see Abstract Interfaces on page 93), **RepositoryId** information must be included in the value type encoding.

If the receiving context needs more typing information than is contained in a GIOP message that contains a codebase URL information, it can go back to the sending context and perform a lookup based on that **RepositoryID** to retrieve more typing information (e.g., the type graph).

CORBA **RepositoryIDs** may contain standard version identification (major and minor version numbers or a hash code information). The ORB run time may use this information to check whether the version of the value being transmitted is compatible with the version expected. In the event of a version mismatch, the ORB may apply product-specific truncation/conversion rules (with the help of a local interface repository or the **SendingContext::RunTime** service). For example, the Java serialization model of truncation/conversion across versions can be supported. See the JDK 1.1 documentation for a detailed specification of this model.

9.4.4.2 Example

The following examples demonstrate legal combinations of truncatability, actual parameter types and GIOP encodings. This is not intended to be an exhaustive list of legal possibilities.

The following example uses valuetypes **animal** and **horse**, where **horse** is derived from **animal**. The actual parameters passed to the specified operations are **an_animal** of runtime type **animal** and **a_horse** of runtime type **horse**.

The following combinations of truncatability, actual parameter types and GIOP encodings are legal.

1. If there is a single operation:

op1(in animal a);

a). If the type **horse** cannot be truncated to **animal** (i.e., **horse** is declared):

valuetype horse: animal ...

then the encoding is as shown below:

Actual Invocation	Legal Encoding
op1(a_horse)	2 horse
	6 1 horse

Note that if the type horse is not available to the receiver, then the receiver throws a demarshaling exception.

b). If the type **horse** can be truncated to **animal** (i.e., **horse** is declared):

valuetype horse: truncatable animal ...

then the encoding is as shown below

Actual Invocation	Legal Encoding
op1(a_horse)	6 2 horse animal

Note that if the type horse is not available to the receiver, then the receiver tries to truncate to animal.

c). Regardless of the truncation relationships, when the exact type of the formal argument is sent, then the encoding is as shown below:

Actual Invocation	Legal Encoding
op1(an_animal)	0
	2 animal
	6 1 animal

2. Given the additional operation:

op2(in horse h);

(i.e., the sender knows that both types **horse** and **animal** and their derivation relationship are known to the receiver)

a). If the type horse cannot be truncated to animal (i.e., horse is declared):

valuetype horse: animal ...

then the encoding is as shown below:

Actual Invocation	Legal Encoding
op2(a_horse)	2 horse
	6 1 horse

Note that the demarshaling exception of case 1 will not occur, since horse is available to the receiver.

b). If the type horse can be truncated to animal (i.e., horse is declared):

valuetype horse: truncatable animal ...

then the encoding is as shown below:

Actual Invocation	Legal Encoding
op2 (a_horse)	2 horse
	6 1 horse
	6 2 horse animal

Note that truncation will not occur, since horse is available to the receiver.

9.4.4.3 Scope of the Indirections

The special value **0xffffffff** introduces an indirection (i.e., it directs the decoder to go somewhere else in the marshaling buffer to find what it is looking for). This can be codebase URL information that has already been encoded, a **RepositoryID** that has already been encoded, a list of repository IDs that has already been encoded, or another value object that is shared in a graph. **0xffffffff** is always followed by a **long** indicating where to go in the buffer. A repositoryID or URL, which is the target of an indirection used for encoding a valuetype must have been introduced as the type or codebase information for a valuetype.

It is not permissible for a repositoryID marshaled for some purpose other than as the type information of a valuetype to use indirection to reference a previously marshaled value. The encoding used to indicate an indirection is the same as that used for recursive **TypeCodes** (i.e., a **0xffffffff** indirection marker followed by a **long** offset (in units of **octets**) from the beginning of the long offset). As an example, this means that an offset of negative four (-4) is illegal, because it is self-indirecting to its indirection marker. Indirections may refer to any preceding location in the GIOP message, including previous fragments if fragmentation is used. This includes any previously marshaled parameters. Non-negative offsets are reserved for future use. Indirections may not cross encapsulation boundaries.

Fragmentation support in GIOP versions 1.1, 1.2, and 1.3 introduces the possibility of a header for a **FragmentMessage** being marshaled between the target of an indirection and the start of the encapsulation containing the indirection. The octets occupied by any such headers are not included in the calculation of the offset value.

9.4.4.4 Null Values

All value types have a distinguished “null.” All null values are encoded by the <null_tag> (0x0). The CDR encoding of null values includes no type information.

9.4.4.5 Other Encoding Information

A “new” value is coded as a value header followed by the value’s state. The header contains a tag and codebase URL information if appropriate, followed by the **RepositoryID** and an octet flag of bits. Because the same **RepositoryID** (and codebase URL information) could be repeated many times in a single request when sending a complex graph, they are encoded as a regular string the first time they appear, and use an indirection for later occurrences.

9.4.4.6 Fragmentation

It is anticipated that value types may be rather large, particularly when a graph is being transmitted. Hence the encoding supports the breaking up of the serialization into an arbitrary number of chunks in order to facilitate incremental processing.

Values with **truncatable** base types need a length indication in case the receiver needs to truncate them to a base type. Value types that are custom marshaled also need a length indication so that the ORB run time can know exactly where they end in the stream without relying on user-defined code. This allows the ORB to maintain consistency and ensure the integrity of the GIOP stream when the user-written custom marshaling and demarshaling does not marshal the entire value state. For simplicity of encoding, we use a length indication for all values whether or not they have a truncatable base type or use custom marshaling.

If limited space is available for marshaling, it may be necessary for the ORB to send the contents of a marshaling buffer containing a partially marshaled value as a GIOP fragment. At that point in the marshaling, the length of the entire value being marshaled may not be known. Calculating this length may require processing as costly as marshaling the entire value. It is therefore desirable to allow the value to be encoded as multiple chunks, each with its own length. This allows the portion of a value that occupies a marshaling buffer to be sent as a chunk of known length with no need for additional length calculation processing.

The data may be split into multiple chunks at arbitrary points except within primitive CDR types, arrays of primitive types, strings, and wstrings, or between the tag and offset of indirections. It is never necessary to end a chunk within one of these types as the length of these types is known before starting to marshal them so they can be added to the length of the currently open chunk. It is the responsibility of the CDR stream to hide the chunking from the marshaling code.

The presence (or absence) of chunking is indicated by flags within the <value_tag>. The fourth least significant bit (<value_tag> & **0x00000008**) is the value 1 if a chunked encoding is used for the value's state. The chunked encoding is required for custom marshaling and truncation. If this bit is 0, the state is encoded as <**octets**>.

Each chunk is preceded by a positive long, which specifies the number of octets in the chunk.

A chunked value is terminated by an end tag that is a non-positive long so the start of the next value can be differentiated from the start of another chunk. In the case of values that contain other values (e.g., a linked list) the "nested" value is started without there being an end tag. The absolute value of an end tag (when it finally appears) indicates the nesting level of the value being terminated. A single end tag can be used to terminate multiple nested values. The detailed rules are as follows:

- End tags, chunk size tags, and value tags are encoded using non-overlapping ranges so that the unmarshaling code can tell after reading each chunk whether:
  - another chunk follows (positive tag).
  - one or multiple value types are ending at a given point in the stream (negative tag).
  - a nested value follows (special large positive tag).
- The end tag is a negative long whose value is the negation of the absolute nesting depth of the value type ending at this point in the CDR stream. Any value types that have not already been ended and whose nesting depth is greater than the depth indicated by the end tag are also implicitly ended. The end tag value **0** is reserved for future use (e.g., supporting a nesting depth of more than **2^31**). The outermost value type will always be terminated by an end tag with a value of **-1**. Enclosing non-chunked valuetypes are not considered when determining the nesting depth.

The following example describes how end tags may be used. Consider a valuetype declaration that contains two member values:

```
// IDL
    valuetype simpleNode{ ... };
    valuetype node truncatable simpleNode {
    public node node1;
    public node node2;
};
```

When an instance of type '**node**' is passed as a parameter of type '**simpleNode**' a chunked encoding is used. In all cases, the outermost value is terminated with an end tag with a value of **-1**. The nested value '**node1**' is terminated with an end tag with a value of **-2** since only the second-level value '**node1**' ends at that point. Since the nested value '**node2**' coterminates with the outermost value, either of the following end tag layouts is legal:

- A single end tag with a value of **-1** marks the termination of the outermost value, implying the termination of the nested value, '**node2**'as well. This is the most compact marshaling.
- An end tag with a value of **-2** marks the termination of the nested value, '**node2**. This is then followed by an end tag with a value of **-1** to mark the termination of the outermost value.

  Because data members are encoded in their declaration order, declaring a value type data member of a value type last is likely to result in more compact encoding on the wire because it maximizes the number of values ending at the same place and so allows a single end tag to be used for multiple values. The canonical example for that is a linked list.

- For the purposes of chunking, values encoded as indirections or null are treated as non-value data.
- Chunks are never nested. When a value is nested within another value, the outer value's chunk ends at the place in the stream where the inner value starts. If the outer value has non-value data to be marshaled following the inner value, the end tag for the inner value is followed by a continuation chunk for the remainder of the outer value.
- Regardless of the above rules, any value nested within a chunked value is always chunked. Furthermore, any such nested value that is truncatable must encode its type information as a list of **RepositoryIDs** (see Section 9.4.4.1, Partial Type Information and Versioning, on page 80).
- The scope of an encoded valuetype is a complete GIOP message or an encapsulation. Starting a new encapsulation starts a new scope. Ending an encapsulation ends the current scope and restores the previous scope. Starting a new scope starts a new count of end tag nesting (initially 0), chunking status (initially false) and chunk position (initially 0).
- Chunks in the same scope are never nested. When a value is nested within another value, the outer value's chunk ends at the place in the stream where the inner value starts. If the outer value has non-value data to be marshaled following the inner value, the end tag for the inner value is followed by a continuation chunk for the remainder of the outer value.
- Regardless of the above rules, any value nested within a chunked value in the same scope is always chunked. Furthermore, any such nested value that is truncatable must encode its type information as a list of RepositoryIDs (see Partial Type Information and Versioning on page 80).

Truncating a value type in the receiving context may require keeping track of unused nested values (only during unmarshaling) in case further indirection tags point back to them. These values can be held in their "raw" GIOP form, as fully unmarshaled value objects, or in any other product-specific form.

Value types that are custom marshaled are encoded as chunks in order to let the ORB run-time know exactly where they end in the stream without relying on user-defined code.

9.4.4.7 Notation

The on-the-wire format is described by a BNF grammar with conventions similar to the ones used to define IDL syntax. ***The terminals of the grammar are to be interpreted differently.*** We are describing a protocol format. Although the terminals have the same names as IDL tokens they represent either:

- constant tags, or
- the GIOP CDR encoding of the corresponding IDL construct.

For example, **long** is a shorthand for the GIOP encoding of the IDL **long** data type -with all the GIOP alignment rules. Similarly **struct** is a shorthand for the GIOP CDR encoding of a **struct**.

A **(type) constant** means that an instance of the given type having the given value is encoded according to the rules for that type. So that **(long) 0** means that a CDR encoding for a long having the value **0** appears at that location.

9.4.4.8 The Format

```
(1)  <value> ::= <value_tag> [  <codebase_URL> ]
                 [ <type_info> ] <state>
                                 |   <value_ref>
(2)  <value_ref>::=<indirection_tag>  <indirection> | <null_tag>
(3)  <value_tag>::= long// 0x7fffff00 <= value_tag <= 0x7fffffff
(4)  <type_info> ::= <rep_ids> | <repository_id>
(5)  <state>::= <octets> |<value_data>* [ <end_tag> ]
(6)  <value_data>::= <value_chunk> | <value>
(7)  <rep_ids> ::= long <repository_id>+
                                 |   <indirection_tag> <indirection>
(8)  <repository_id> ::= string | <indirection_tag> <indirection>
(9)  <value_chunk> ::= <chunk_size_tag> <octets>
(10)  <null_tag> ::= (long) 0
(11)  <indirection_tag> ::= (long) 0xffffffff
(12)  <codebase_URL> ::= string | <indirection_tag> <indirection>
(13)  <chunk_size_tag> ::= long
                 //  0 < chunk_size_tag < 2^31-256 (0x7fffff00)
(14)  <end_tag>::= long // -2^31 < end_tag < 0
(15)  <indirection> ::=  long // -2^31 < indirection < 0
(16)  <octets> ::= octet | octet <octets>
```

The concatenated octets of consecutive value chunks within a value encode state members for the value according to the following grammar:

```
(1)<state members> ::= <state_member>
                                 |   <state_member> <state members>
(2)  <state_member> ::=<value_ref>
                 // All legal IDL types should be here
                                 |   octet
                                 |   boolean
                                 |   char
                                 |   short
```

| **unsigned short**
| **long**
| **unsigned long**
| **float**
| **wchar**
| **wstring**
| **string**
| **struct**
| **union**
| **sequence**
| **array**
| **Object**
| **any**
| **long long**
| **unsigned long long**
| **double**
| **long double**
| **fixed**

9.4.5 Pseudo-Object Types

CORBA defines some kinds of entities that are neither primitive types (integral or floating point) nor constructed ones.

9.4.5.1 TypeCode

In general, **TypeCodes** are encoded as the **TCKind** enum value, potentially followed by values that represent the **TypeCode** parameters. Unfortunately, **TypeCodes** cannot be expressed simply in OMG IDL, since their definitions are recursive. The basic **TypeCode** representations are given in Table 9.2 on page 88. The *integer value* column of this table gives the **TCKind** enum value corresponding to the given **TypeCode**, and lists the parameters associated with such a **TypeCode**. The rest of this sub clause presents the details of the encoding.

Basic TypeCode Encoding Framework

The encoding of a **TypeCode** is the **TCKind** enum value (encoded, like all **enum** values, using four octets), followed by zero or more parameter values. The encodings of the parameter lists fall into three general categories, and differ in order to conserve space and to support efficient traversal of the binary representation:

- Typecodes with an *empty parameter list* are encoded simply as the corresponding **TCKind** enum value.
- Typecodes with *simple parameter lists* are encoded as the **TCKind** enum value followed by the parameter value(s), encoded as indicated in Table 9.2. A “simple” parameter list has a fixed number of fixed length entries, or a single parameter that has its length encoded first.
- All other typecodes have *complex parameter lists*, which are encoded as the **TCKind** enum value followed by a CDR encapsulation octet sequence (see Encapsulation on page 79) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 9.2.

The third column of Table 9.2 shows whether each parameter list is *empty*, *simple*, or *complex*. Also, note that an internal indirection facility is needed to represent some kinds of typecodes; this is explained in Indirection: Recursive and Repeated TypeCodes on page 91. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 9.2 on page 88. The ordering and meaning of parameters is a superset of those given in the *ORB Interface* clause, CORBA, Part 1 specification. More information is needed by CDR's representation in order to provide the full semantics of **TypeCodes** as shown by the API.

- Each parameter is written in the form *type (name),* where *type* describes the parameter's type, and *name* describes the parameter's meaning.
- The encoding of some parameter lists (specifically, **tk_struct, tk_union, tk_enum, and tk_except**) contain a counted sequence of tuples.

  Such counted tuple sequences are written in the form *count {parameters}*, where *count* is the number of tuples in the encoded form, and the *parameters* enclosed in braces are available in each tuple instance. First the *count*, which is an `unsigned long`, and then each *parameter* in each tuple (using the noted type), is encoded in the CDR representation of the typecode. Each tuple is encoded, first parameter followed by second, before the next tuple is encoded (first, then second, etc.).

Note that the tuples identifying **struct**, union, **exception**, and **enum** members must be in the order defined in the OMG IDL definition text. Also, that the types of discriminant values in encoded **tk_union TypeCodes** are established by the second encoded parameter (*discriminant type*), and cannot be specified except with reference to a specific OMG IDL definition.[3]

Table 9.2

TCKind	Integer Value	Type	Parameters
tk_null	0	empty	– none –
tk_void	1	empty	– none –
tk_short	2	empty	– none –
tk_long	3	empty	– none –
tk_ushort	4	empty	– none –
tk_ulong	5	empty	– none –
tk_float	6	empty	– none –
tk_double	7	empty	– none –
tk_boolean	8	empty	– none –
tk_char	9	empty	– none –
tk_octet	10	empty	– none –
tk_any	11	empty	– none –
tk_TypeCode	12	empty	– none –
tk_Principal	13	empty	– none –
tk_objref	14	complex	string (repository ID), string(name)

Table 9.2

TCKind	Integer Value	Type	Parameters
tk_struct	15	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
tk_union	16	complex	string (repository ID), string(name), TypeCode (discriminant type), long (default used), ulong (count) {*discriminant type*[a] (label value), string (member name), TypeCode (member type)}
tk_enum	17	complex	string (repository ID), string (name), ulong (count) {string (member name)}
tk_string	18	simple	ulong (max length[b])
tk_sequence	19	complex	TypeCode (element type), ulong (max length[c])
tk_array	20	complex	TypeCode (element type), ulong (length)
tk_alias	21	complex	string (repository ID), string (name), TypeCode
tk_except	22	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
tk_longlong	23	empty	– none –
tk_ulonglong	24	empty	– none –
tk_longdouble	25	empty	– none –
tk_wchar	26	empty	– none –

3. This means that, for example, two OMG IDL unions that are textually equivalent, except that one uses a “char” discriminant, and the other uses a “long” one, would have different size encoded TypeCodes.

Table 9.2

TCKind	Integer Value	Type	Parameters
tk_wstring	27	simple	ulong(max length or zero if unbounded)
tk_fixed	28	simple	ushort(digits), short(scale)
tk_value	29	complex	string (repository ID), string (name, may be empty), short(ValueModifier), TypeCode(of concrete base)[d], ulong (count), {string (member name), TypeCode (member type), short(Visibility)}
tk_value_box	30	complex	string (repository ID), string(name), TypeCode
tk_native	31	complex	string (repository ID), string(name)
tk_abstract_interface	32	complex	string(RepositoryId), string(name)
tk_local_interface	33	complex	string(RepositoryId), string(name)
tk_component	34	complex	string (repository ID), string(name)
tk_home	35	complex	string (repository ID), string(name)
tk_event	36	complex	string (repository ID), string (name, may be empty), short(ValueModifier), TypeCode(of concrete base)[e], ulong (count), {string (member name), TypeCode (member type), short(Visibility)}
– none –	0xffffffff	simple	long (indirection[f])

a. The type of union label values is determined by the second parameter, discriminant type.
b. For unbounded strings, this value is zero.
c. For unbounded sequences, this value is zero.
d. Should be **tk_null** if there is no concrete base.
e. Should be tk_null if there is no concrete base.
f. See Indirection: Recursive and Repeated TypeCodes on page 91.

9.4.5.1.1 Encoded Identifiers and Names

The Repository ID parameters in **tk_objref**, **tk_struct**, **tk_union**, **tk_enum**, **tk_alias**, **tk_except**, **tk_native**, **tk_value**, **tk_value_box**, and **tk_abstract_interface** TypeCodes are Interface Repository **RepositoryId** values, whose format is described in the specification of the Interface Repository.

For GIOP 1.2 onwards, repositoryID values are required to be sent, if known by the ORB[4]. For GIOP 1.2 and 1.3 an empty repositoryID string is only allowed if a repositoryID value is not available to the ORB sending the type code.

For GIOP 1.0 and 1.1, **RepositoryId** values are required for **tk_objref** and **tk_except** TypeCodes; for **tk_struct**, **tk_union**, **tk_enum**, and **tk_alias** TypeCodes **RepositoryIds** are optional and encoded as empty strings if omitted.

The **name** parameters in **tk_objref, tk_struct, tk_union, tk_enum, tk_alias, tk_value, tk_value_box, tk_abstract_interface, tk_native** and **tk_except** TypeCodes and the **member name** parameters in **tk_struct, tk_union, tk_enum, tk_value**, and **tk_except** TypeCodes are not specified by (or significant in) GIOP. Agents should not make assumptions about type equivalence based on these name values; only the structural information (including **RepositoryId** values, if provided) is significant. If provided, the strings should be the simple, unscoped names supplied in the OMG IDL definition text. If omitted, they are encoded as empty strings.

When a reference to a base **Object** is encoded, there are two allowed encodings for the Repository ID: either "**IDL:omg.org/CORBA/Object:1.0**" or "" may be used.

Encoding the tk_union Default Case

In **tk_union TypeCodes**, the **long default used** value is used to indicate which tuple in the sequence describes the union's **default** case. If this value is less than zero, then the union contains no default case. Otherwise, the value contains the zero-based index of the default case in the sequence of tuples describing union members.

The discriminant value used in the actual typecode parameter associated with the default member position in the list, may be any valid value of the discriminant type, and has no semantic significance (i.e., it should be ignored and is only included for syntactic completeness of union type code marshaling).

TypeCodes for Multi-Dimensional Arrays

The **tk_array TypeCode** only describes a single dimension of any array. **TypeCodes** for multi-dimensional arrays are constructed by nesting **tk_array TypeCodes** within other **tk_array TypeCodes**, one per array dimension. The outermost (or top-level) **tk_array TypeCode** describes the leftmost array index of the array as defined in IDL; the innermost nested **tk_array TypeCode** describes the rightmost index.

Indirection: Recursive and Repeated TypeCodes

The typecode representation of OMG IDL data types that can indirectly contain instances of themselves (e.g., **struct foo {sequence <foo> bar;}**) must also contain an indirection. Such an indirection is also useful to reduce the size of encodings; for example, unions with many cases sharing the same value.

CDR provides a constrained indirection to resolve this problem:

- The indirection applies only to TypeCodes nested within some "top-level" TypeCode. Indirected TypeCodes are not "freestanding," but only exist inside some other encoded TypeCode.
- For GIOP 1.2 and below, the indirection applies only to TypeCodes nested within some "top-level" TypeCode. Indirected TypeCodes are not "freestanding," but only exist inside some other encoded TypeCode.

4. A type code passed via a GIOP 1.2 connection shall contain non-empty repositoryID strings, unless a repositoryID value is not available to the sending ORB for a specific type code. This situation can arise, for example, if an ORB receives a type code containing empty repository IDs via a GIOP 1.0 or 1.1 connection and passes that type code on via a GIOP 1.2 connection).

- For GIOP 1.3 and above, the indirection applies only to TypeCodes nested within some "top-level" TypeCode, or from one top-level TypeCode to another. Indirected TypeCodes nested within a top-level TypeCode can only reference TypeCodes that are part of the same top-level TypeCode, including the top-level TypeCode itself. Indirected top-level TypeCodes can reference other top-level TypeCodes but cannot reference TypeCodes nested within some other top-level TypeCode.

- Only the second (and subsequent) references to a **TypeCode** in that scope may use the indirection facility. The first reference to that **TypeCode** must be encoded using the normal rules. In the case of a recursive **TypeCode**, this means that the first instance will not have been fully encoded before a second one must be completely encoded.

The indirection is a numeric octet offset within the scope of the "top-level" **TypeCode** and points to the **TCKind** value for the typecode. (Note that the byte order of the **TCKind** value can be determined by its encoded value.) This indirection may well cross encapsulation boundaries, but this is not problematic because of the first constraint identified above. Because of the second constraint, the value of the offset will always be negative.

Fragmentation support in GIOP versions 1.1, 1.2, and 1.3 introduces the possibility of a header for a **FragmentMessage** being marshaled between the target of an indirection and the start of the encapsulation containing the indirection. The octets occupied by any such headers are not included in the calculation of the offset value.

The encoding of such an indirection is as a **TypeCode** with a "**TCKind** value" that has the special value 2^{32}-1 (**0xffffffff**, all ones). Such typecodes have a single (simple) parameter, which is the **long** offset (in units of octets) from the simple parameter. (This means that an offset of negative four (-4) is illegal because it will be self-indirecting.)

9.4.5.2 Any

Any values are encoded as a **TypeCode** (encoded as described above) followed by the encoded value. For **Any** values containing a **tk_null** or **tk_void TypeCode**, the encoded value shall have zero length (i.e., shall be absent).

9.4.5.3 Principal

Principal pseudo objects are encoded as **sequence<octet>**. In the absence of a Security service specification, **Principal** values have no standard format or interpretation, beyond serving to identify callers (and potential callers). This text does not prescribe any usage of **Principal** values.

By representing **Principal** values as **sequence<octet>**, GIOP guarantees that ORBs may use domain-specific principal identification schemes; such values undergo no translation or interpretation during transmission. This allows bridges to translate or interpret these identifiers as needed when forwarding requests between different security domains.

9.4.5.4 Context

Context pseudo objects are encoded as **sequence<string>**. The strings occur in pairs. The first string in each pair is the context property name, and the second string in each pair is the associated value. If an operation has an IDL context sub clause but the client does not supply any properties matching the context sub clause at run time, an empty sequence is marshaled.

9.4.5.5 Exception

Exceptions are encoded as a string followed by exception members, if any. The string contains the RepositoryId for the exception, as defined in the *Interface Repository* clause of CORBA (Part 1). Exception members (if any) are encoded in the same manner as a struct.

If an ORB receives a non-standard system exception that it does not support, or a user exception that is not defined as part of the operation's definition, the exception shall be mapped to UNKNOWN, with standard minor code set to 2 for a system exception, or set to 1 for a user exception.

9.4.6 Object References

Object references are encoded in OMG IDL (as described in Object Addressing on page 23). IOR profiles contain transport-specific addressing information, so there is no general-purpose IOR profile format defined for GIOP. Instead, this part of ISO/IEC 19500 describes the general information model for GIOP profiles and provides a specific format for the IIOP (see IIOP IOR Profiles on page 112).

In general, GIOP profiles include at least these three elements:

1. The version number of the transport-specific protocol specification that the server supports.
2. The address of an endpoint for the transport protocol being used.
3. An opaque datum (an **object_key**, in the form of an octet sequence) used exclusively by the agent at the specified endpoint address to identify the object.

9.4.7 Abstract Interfaces

Abstract interfaces are encoded as a union with a **boolean** discriminator. The **union** has an *object reference* (see Object References on page 93) if the discriminator is **TRUE**, and a *value type* (see Value Types on page 80) if the discriminator is **FALSE**. The encoding of value types marshaled as abstract interfaces always includes **RepositoryId** information. If there is no indication whether a nil abstract interface represents a nil object reference or a null valuetype, it shall be encoded as a null valuetype.

9.5 GIOP Message Formats

GIOP has restriction on client and server roles with respect to initiating and receiving messages. For the purpose of GIOP versions 1.0 and 1.1, a client is the agent that opens a connection (see more details in Connection Management on page 108) and originates requests. Likewise, for GIOP versions 1.0 and 1.1, a server is an agent that accepts connections and receives requests.When Bidirectional GIOP is in use for GIOP protocol version 1.2 and 1.3, either side may originate messages, as specified in Bi-Directional GIOP on page 115.

GIOP message types are summarized in Table 9.3, which lists the message type names, whether the message is originated by client, server, or both, and the value used to identify the message type in GIOP message headers.

Table 9.3

Message Type	Originator	Value	GIOP Versions
Request	Client	0	1.0, 1.1, 1.2, 1.3
Request	Both	0	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
Reply	Server	1	1.0, 1.1, 1.2, 1.3

Table 9.3

Message Type	Originator	Value	GIOP Versions
Reply	Both	1	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
CancelRequest	Client	2	1.0, 1.1, 1.2, 1.3
CancelRequest	Both	2	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
LocateRequest	Client	3	1.0, 1.1, 1.2, 1.3
LocateRequest	Both	3	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
LocateReply	Server	4	1.0, 1.1, 1.2, 1.3
LocateReply	Both	4	1.2 with BiDir GIOP in use, 1.3 with BiDir GIOP in use
CloseConnection	Server	5	1.0, 1.1, 1.2, 1.3
CloseConnection	Both	5	1.2, 1.3
MessageError	Both	6	1.0, 1.1, 1.2, 1.3
Fragment	Both	7	1.1, 1.2, 1.3

9.5.1 GIOP Message Header

All GIOP messages begin with the following header, defined in OMG IDL:

```
module GIOP { // IDL extended for version 1.1, 1.2, and 1.3
    struct Version {
        octet       major;
        octet       minor;
    };

#if MAX_GIOP_VERSION_NUMBER == 0
    // GIOP 1.0
    enum MsgType_1_0 { // Renamed from MsgType
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError
    };

#else
    // GIOP 1.1
    enum MsgType_1_1 {
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError,
```

```
        Fragment            // GIOP 1.1 addition
    };
#endif  // MAX_GIOP_VERSION_NUMBER

    // GIOP 1.0
    typedef char Magicn[4]
    struct MessageHeader_1_0 {// Renamed from MessageHeader
        Magicn              magic;
        Version             GIOP_version;
        boolean             byte_order;
        octet               message_type;
        unsigned long       message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        Magicn              magic;
        Version             GIOP_version;
        octet               flags;          // GIOP 1.1 change
        octet               message_type;
        unsigned long       message_size;
    };

    // GIOP 1.2, 1.3
    typedef MessageHeader_1_1 MessageHeader_1_2;
    typedef MessageHeader_1_1 MessageHeader_1_3;
};
```

The message header clearly identifies GIOP messages and their byte-ordering. The header is independent of byte ordering except for the field encoding message size.

- **magic** identifies GIOP messages. The value of this member is always the four (upper case) characters “GIOP,” encoded in ISO Latin-1 (8859.1).

- **GIOP_version** contains the version number of the GIOP protocol being used in the message. The version number applies to the transport-independent elements of this part of ISO/IEC 19500 (i.e., the CDR and message formats) that constitute the GIOP. This is not equivalent to the IIOP version number (as described in Object References on page 93) though it has the same structure. The major GIOP version number of this text is one (1); the minor versions are zero (0), one (1), and two (2).

  A server implementation supporting a minor GIOP protocol version 1.n (with n > 0 and n < 3), must also be able to process GIOP messages having minor protocol version 1.m, with m less than n. A GIOP server, which receives a request having a greater minor version number than it supports, should respond with an error message having the highest minor version number that that server supports, and then close the connection.

  A client should not send a GIOP message having a higher minor version number than that published by the server in the tag Internet IIOP Profile body of an IOR.

- **byte_order** (in GIOP 1.0 only) indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of **FALSE** (0) indicates big-endian byte ordering, and **TRUE** (1) indicates little-endian byte ordering.

- flags (in GIOP 1.1, 1.2, and 1.3) is an 8-bit octet. The least significant bit indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of **FALSE** (0) indicates big-endian byte ordering, and **TRUE** (1) indicates little-endian byte ordering. The byte order for fragment messages must match the byte order of the initial message that the fragment extends.

  The second least significant bit indicates whether or not more framents follow. A value of **FALSE** (0) indicates this message is the last fragment, and **TRUE** (1) indicates more fragments follow this message.

  The most significant 6 bits are reserved. These 6 bits must have value 0 for GIOP version 1.1, 1.2, and 1.3.

- **message_type** indicates the type of the message, according to Table 9.3; these correspond to enum values of type **MsgType**.

- **message_size** contains the number of octets in the message following the message header, encoded using the byte order specified in the byte order bit (the least significant bit) in the **flags** field (or using the byte_order field in GIOP 1.0). It refers to the size of the message body, not including the 12-byte message header. This count includes any alignment gaps and must match the size of the actual request parameters (plus any final padding bytes that may follow the parameters to have a fragment message terminate on an 8-byte boundary).

  A MARSHAL exception with minor code 9 indicates that fewer bytes were present in a message than indicated by the count. (This condition can arise if the sender sends a message in fragments, and the receiver detects that the final fragment was received but contained insufficient data for all parameters to be unmarshaled.).

  A MARSHAL exception with minor code 8 indicates that more bytes were present in a message than indicated by the count. Depending on the ORB implementation, this condition may be reported for the current message or the next message that is processed (when the receiver detects that the previous message is not immediately followed by the GIOP magic number).

  The use of a message size of 0 with a **Request**, **LocateRequest**, **Reply**, or **LocateReply** message is reserved for future use.

  For GIOP version 1.2, and 1.3, if the second least significant bit of **Flags** is 1, the sum of the **message_size** value and 12 must be evenly divisible by 8.

Messages with different GIOP minor versions may be mixed on the same underlying transport connection.

9.5.2 Request Message

Request messages encode CORBA object invocations, including attribute accessor operations, and **CORBA::Object** operations **get_interface, repository_id, and get_implementation**. Requests flow from client to server.

Request messages have three elements, encoded in this order:

- A GIOP message header
- A Request Header
- The Request Body

9.5.2.1 Request Header

The request header is specified as follows:

```
module GIOP { // IDL extended for version 1.1, 1.2, and 1.3

    // GIOP 1.0
    struct RequestHeader_1_0 { // Renamed from RequestHeader
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        IOP::ObjectKey              object_key;
        string                      operation;
        CORBA::OctetSeq             requesting_principal;
    };

    typedef octet RequestReserved[3];
    struct RequestHeader_1_1 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        RequestReserved             reserved; // Added in GIOP 1.1
        IOP::ObjectKey              object_key;
        string                      operation;
        CORBA::OctetSeq             requesting_principal;
    };

    // GIOP 1.2, 1.3
    typedef short               AddressingDisposition;
    const short                 KeyAddr = 0;
    const short                 ProfileAddr = 1;
    const short                 ReferenceAddr = 2;

    struct IORAddressingInfo {
        unsigned long               selected_profile_index;
        IOP::IOR                    ior;
    };

    union TargetAddress switch (AddressingDisposition) {
        case KeyAddr:           IOP::ObjectKey object_key;
        case ProfileAddr:       IOP::TaggedProfile profile;
        case ReferenceAddr:     IORAddressingInfo ior;
    };

    struct RequestHeader_1_2 {
        unsigned long               request_id;
        octet                       response_flags;
        RequestReserved             reserved; // Added in GIOP 1.1
        TargetAddress               target;
        string                      operation;
        IOP::ServiceContextList     service_context;
        // requesting_principal not in GIOP 1.2 and 1.3
    };
    typedef RequestHeader_1_2 RequestHeader_1_3;
};
```

The members have the following definitions:

- **request_id** is used to associate reply messages with request messages (including **LocateRequest** messages). The client (requester) is responsible for generating values so that ambiguity is eliminated; specifically, a client must not re-use **request_id** values during a connection if:

  (a) the previous request containing that ID is still pending, or

  (b) if the previous request containing that ID was canceled and no reply was received. (See the semantics of the CancelRequest Message on page 102).

- **response_flags** is set to 0x0 for a **SyncScope** of **NONE** and **WITH_TRANSPORT**. The flag is set to 0x1 for a **SyncScope** of **WITH_SERVER**. A non exception reply to a request message containing a **response_flags** value of 0x1 should contain an empty body, i.e., the equivalent of a void operation with no out/inout parameters. The flag is set to 0x3 for a **SyncScope** of **WITH_TARGET**. These values ensure interworking compatibility between this and previous versions of **GIOP**.

  For GIOP 1.0 and 1.1 a **response_expected** value of **TRUE** is treated like a **response_flags** value of \x03, and a **response_expected** value of **FALSE** is treated like a **response_flags** value of \x00.

- **reserved** is always set to **0** in GIOP 1.1. These three octets are reserved for future use.

- For GIOP 1.0 and 1.1, **object_key** identifies the object that is the target of the invocation. It is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.

- For GIOP 1.2**, 1.3**, **target** identifies the object that is the target of the invocation. The possible values of the union are:

  - **KeyAddr** is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
  - **ProfileAddr** is the transport-specific GIOP profile selected for the target's IOR by the client ORB.
  - **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB. The first profile has an index of zero.

- **operation** is the IDL identifier naming, within the context of the interface (not a fully qualified scoped name), the operation being invoked. In the case of attribute accessors, the names are **_get_<attribute>** and **_set_<attribute>**. The case of the operation or attribute name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  In the case of **CORBA::Object** operations that are defined in the *ORB Interface* clause, CORBA, Part 1 and that correspond to GIOP request messages, the operation names are **_interface**, **_is_a**, **_non_existent, _domain_managers, _component,** and **_repository_id**.

NOTE: The name **_get_domain_managers** is not used, to avoid conflict with a get operation invoked on a user defined attribute with name **domain_managers**.

For GIOP 1.2 and later versions, only the operation name **_non_existent** shall be used.
The correct operation name to use for GIOP 1.0 and 1.1 is **_non_existent**. Due to a typographical error in CORBA 2.0, 2.1, and 2.2, some legacy implementations of GIOP 1.0 and 1.1 respond to the operation name **_not_existent**. For maximum interoperability with such legacy implementations, new implementations of GIOP 1.0 and 1.1 may wish to respond to both operation names, **_non_existent** and **_not_existent**.

- **service_context** contains ORB service data being passed from the client to the server, encoded as described in Service Context on page 37.
- **requesting_principal** contains a value identifying the requesting principal. It is provided to support the **BOA::get_principal** operation. The usage of the **requesting_principal** field is deprecated for GIOP versions 1.0 and 1.1. The field is not present in the request header for GIOP version 1.2 and 1.3.

There is no padding after the request header when an unfragmented request message body is empty.

9.5.2.2 Request Body

In GIOP versions 1.0 and 1.1, request bodies are marshaled into the CDR encapsulation of the containing Message immediately following the Request Header. In GIOP version 1.2 and 1.3, the Request Body is always aligned on an 8-octet boundary. The fact that GIOP specifies the maximum alignment for any primitive type is 8 guarantees that the Request Body will not require remarshaling if the Message or Request header are modified. The data for the request body includes the following items encoded in this order:

- All **in** and **inout** parameters, in the order in which they are specified in the operation's OMG IDL definition, from left to right.
- An optional **Context** pseudo object, encoded as described in Context on page 92. This item is included only if the operation's OMG IDL definition includes a context expression, and only includes context members as defined in that expression.

For example, the request body for the following OMG IDL operation:

```
double example (in short m, out string str, inout long p);
```

would be equivalent to this structure:

```
struct example_body {
    short           m;      // leftmost in or inout parameter
    long            p;      // ... to the rightmost
};
```

9.5.3 Reply Message

Reply messages are sent in response to **Request** messages if and only if the response expected flag in the request is set to **TRUE**. Replies include inout and out parameters, operation results, and may include exception values. In addition, Reply messages may provide object location information. In GIOP versions 1.0 and 1.1, replies flow only from server to client.

Reply messages have three elements, encoded in this order:

- A GIOP message header
- A ReplyHeader structure
- The reply body

9.5.3.1 Reply Header

The reply header is defined as follows:

```
module GIOP {                           // IDL extended for 1.2 and 1.3

#if MAX_GIOP_MINOR_VERSION < 2

    // GIOP 1.0 and 1.1
    enum ReplyStatusType_1_0 { // Renamed from ReplyStatusType
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    // GIOP 1.0
    struct ReplyHeader_1_0 { // Renamed from ReplyHeader
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        ReplyStatusType_1_0         reply_status;
    };

    // GIOP 1.1
    typedef ReplyHeader_1_0 ReplyHeader_1_1;
    // Same Header contents for 1.0 and 1.1

#endif // MAX_GIOP_VERSION_NUMBER

#if MAX_GIOP_MINOR_VERSION >= 2

    // GIOP 1.2, 1.3
    enum ReplyStatusType_1_2 {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD,
        LOCATION_FORWARD_PERM,// new value for 1.2
        NEEDS_ADDRESSING_MODE // new value for 1.2
    };

    struct ReplyHeader_1_2 {
        unsigned long               request_id;
        ReplyStatusType_1_2         reply_status;
        IOP::ServiceContextList     service_context;
    };
    typedef ReplyHeader_1_2 ReplyHeader_1_3;
#endif // MAX_GIOP_VERSION_NUMBER

};
```

The members have the following definitions:

- **request_id** is used to associate replies with requests. It contains the same **request_id** value as the corresponding request.

- **reply_status** indicates the completion status of the associated request, and also determines part of the reply body contents. If no exception occurred and the operation completed successfully, the value is NO_EXCEPTION and the body contains return values. Otherwise the body

  - contains an exception, or
  - directs the client to reissue the request to an object at some other location, or
  - directs the client to supply more addressing information.

- **service_context** contains ORB service data being passed from the server to the client, encoded as described in GIOP Message Transfer on page 71.

There is no padding after the reply header when an unfragmented reply message body is empty.

9.5.3.2 Reply Body

In GIOP version 1.0 and 1.1, reply bodies are marshaled into the CDR encapsulation of the containing Message immediately following the Reply Header. In GIOP version 1.2 and 1.3, the Reply Body is always aligned on an 8-octet boundary. The fact that GIOP specifies the maximum alignment for any primitive type is 8 guarantees that the ReplyBody will not require remarshaling if the Message or the Reply Header are modified. The data for the reply body is determined by the value of **reply_status**. There are the following types of reply body:

- If the **reply_status** value is NO_EXCEPTION, the body is encoded as if it were a structure holding first any operation return value, then any **inout** and **out** parameters in the order in which they appear in the operation's OMG IDL definition, from left to right. (That structure could be empty.)

- If the **reply_status** value is USER_EXCEPTION or SYSTEM_EXCEPTION, then the body contains the exception that was raised by the operation, encoded as described in Exception on page 92. (Only the user-defined exceptions listed in the operation's OMG IDL definition may be raised.)

  When a GIOP Reply message contains a '**reply_status**' value of SYSTEM_EXCEPTION, the body of the Reply message conforms to the following structure:

```
module GIOP {                                   // IDL
    struct SystemExceptionReplyBody {
        string              exception_id;
         unsigned long      minor_code_value;
        unsigned long       completion_status;
        };
};
```

  The high-order 20 bits of **minor_code_value** contain a 20-bit "Vendor Minor Codeset ID" (**VMCID**); the low-order 12 bits contain a minor code. A vendor (or group of vendors) wishing to define a specific set of system exception minor codes should obtain a unique **VMCID** from the OMG, and then use those 4096 minor codes as they see fit; for example, defining up to 4096 minor codes for each system exception. Any vendor may use the special **VMCID** of zero (0) without previous reservation, but minor code assignments in this codeset may conflict with other vendor's assignments, and use of the zero **VMCID** is officially deprecated.

NOTE: OMG standard minor codes are identified with the 20 bit **VMCID \x4f4d0**. This appears as the characters 'O' followed by the character 'M' on the wire, which is defined as a 32-bit constant called **OMGVMCID \x4f4d0000** (see the *ORB Interface* clause, CORBA, Part 1) so that allocated minor code numbers can be or-ed with it to obtain the **minor_code_value**.

- If the **reply_status** value is **LOCATION_FORWARD**, then the body contains an object reference (IOR) encoded as described in Object References on page 93. The client ORB is responsible for re-sending the original request to that (different) object. This resending is transparent to the client program making the request.
- The usage of the **reply_status** value **LOCATION_FORWARD_PERM** behaves like the usage of **LOCATION_FORWARD**, but when used by a server it also provides an indication to the client that it may replace the old IOR with the new IOR. Both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.
- If the **reply_status** value is **NEEDS_ADDRESSING_MODE**, then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the original request using the requested addressing mode. The resending is transparent to the client program making the request.

NOTE: Usage of **LOCATATION_FORWARD_PERM** is now deprecated, due to problems it causes with the semantics of the **Object::hash()** operation. **LOCATATION_FORWARD_PERM** features could be removed from some future GIOP versions if solutions to these problems are not provided.

For example, the reply body for a successful response (the value of **reply_status** is **NO_EXCEPTION**) to the *Request* example shown on page 99 would be equivalent to the following structure:

```
struct example_reply {
    double          return_value;       // return value
    string          str;
    long            p;                  // ... to the rightmost
};
```

Note that the **object_key** field in any specific GIOP profile is server-relative, not absolute. Specifically, when a new object reference is received in a **LOCATION_FORWARD Reply** or in a **LocateReply** message, the **object_key** field embedded in the new object reference's GIOP profile may not have the same value as the **object_key** in the GIOP profile of the original object reference. For details on location forwarding, see Object Location on page 110.

9.5.4 CancelRequest Message

CancelRequest messages may be sent, in GIOP versions 1.0 and 1.1, only from clients to servers. **CancelRequest** messages notify a server that the client is no longer expecting a reply for a specified pending **Request** or **LocateRequest** message.

CancelRequest messages have two elements, encoded in this order:

- A GIOP message header
- A **CancelRequestHeader**

9.5.4.1 Cancel Request Header

The cancel request header is defined as follows:

```
module GIOP {                           // IDL
    struct CancelRequestHeader {
        unsigned long           request_id;
    };
};
```

The **request_id** member identifies the **Request** or **LocateRequest** message to which the cancel applies. This value is the same as the **request_id** value specified in the original **Request** or **LocateRequest** message.

When a client issues a cancel request message, it serves in an advisory capacity only. The server is not required to acknowledge the cancellation, and may subsequently send the corresponding reply. The client should have no expectation about whether a reply (including an exceptional one) arrives.

9.5.5 LocateRequest Message

LocateRequest messages may be sent from a client to a server to determine the following regarding a specified object reference:

- whether the current server is capable of directly receiving requests for the object reference, and if not,
- to what address requests for the object reference should be sent.

Note that this information is also provided through the **Request** message, but that some clients might prefer not to support retransmission of potentially large messages that might be implied by a **LOCATION_FORWARD** status in a **Reply** message. That is, client use of this represents a potential optimization.

LocateRequest messages have two elements, encoded in this order:

- A GIOP message header
- A **LocateRequestHeader**

9.5.5.1 LocateRequest Header

The **LocateRequest** header is defined as follows:

```
module GIOP {                       // IDL extended for version 1.2 and 1.3

// GIOP 1.0
    struct LocateRequestHeader_1_0 {
                                    // Renamed LocationRequestHeader
        unsigned long               request_id;
        IOP::ObjectKey              object_key;
    };

// GIOP 1.1
    typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
    // Same Header contents for 1.0 and 1.1

// GIOP 1.2, 1.3
    struct LocateRequestHeader_1_2 {
        unsigned long           request_id;
        TargetAddress           target;
    };
    typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;
};
```

The members are defined as follows:

- **request_id** is used to associate **LocateReply** messages with **LocateRequest** ones. The client (requester) is responsible for generating values; see Request Message on page 96 for the applicable rules.
- For GIOP 1.0 and 1.1, **object_key** identifies the object being located. In an IIOP context, this value is obtained from the **object_key** field from the encapsulated **IIOP::ProfileBody** in the IIOP profile of the IOR for the target object. When GIOP is mapped to other transports, their IOR profiles must also contain an appropriate corresponding value. This value is only meaningful to the server and is not interpreted or modified by the client.
- For GIOP 1.2, 1.3, target identifies the object being located. The possible values of this union are:
  - **KeyAddr** is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
  - **ProfileAddr** is the transport-specific GIOP profile selected for the target's IOR by the client ORB.
  - **IORAddressingInfo** is the full IOR of the target object. The **selected_profile_index** indicates the transport-specific GIOP profile that was selected by the client ORB.

See Object Location on page 110 for details on the use of **LocateRequest**.

9.5.6 LocateReply Message

LocateReply messages are sent from servers to clients in response to **LocateRequest** messages. In GIOP versions 1.0 and 1.1 the **LocateReply** message is only sent from the server to the client.

A **LocateReply** message has three elements, encoded in this order:

1. A GIOP message header
2. A **LocateReplyHeader**
3. The locate reply body

9.5.6.1 Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                       // IDL extended for GIOP 1.2 and 1.3
#if MAX_GIOP_MINOR_VERSION < 2
    // GIOP 1.0 and 1.1
    enum LocateStatusType_1_0 {// Renamed from LocateStatusType
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    // GIOP 1.0
    struct LocateReplyHeader_1_0 {// Renamed from LocateReplyHeader
        unsigned long           request_id;
        LocateStatusType_1_0    locate_status;
    };
```

```
    // GIOP 1.1
    typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
    // same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2, 1.3
    enum LocateStatusType_1_2 {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD,
        OBJECT_FORWARD_PERM,        // new value for GIOP 1.2
        LOC_SYSTEM_EXCEPTION,       // new value for GIOP 1.2
        LOC_NEEDS_ADDRESSING_MODE   // new value for GIOP 1.2
    };

    struct LocateReplyHeader_1_2 {
        unsigned long           request_id;
        LocateStatusType_1_2    locate_status;
    };
    typedef LocateReplyHeader_1_2 LocateReplyHeader_1_3;
#endif // MAX_GIOP_VERSION_NUMBER
};
```

The members have the following definitions:

- **request_id** - is used to associate replies with requests. This member contains the same **request_id** value as the corresponding **LocateRequest** message.
- **locate_status -** the value of this member is used to determine whether a **LocateReply** body exists. Values are:
  - **UNKNOWN_OBJECT** - the object specified in the corresponding **LocateRequest** message is unknown to the server; no body exists.
  - **OBJECT_HERE** - this server (the originator of the **LocateReply** message) can directly receive requests for the specified object; no body exists.
  - **OBJECT_FORWARD** and **OBJECT_FORWARD_PERM** - a **LocateReply** body exists.
  - **LOC_SYSTEM_EXCEPTION** - a **LocateReply** body exists.
  - **LOC_NEEDS_ADDRESSING_MODE** - a **LocateReply** body exists.

9.5.6.2 LocateReply Body

The body is empty, except for the following cases:

- If the **LocateStatus** value is **OBJECT_FORWARD** or **OBJECT_FORWARD_PERM**, the body contains an object reference (IOR) that may be used as the target for requests to the object specified in the **LocateRequest** message. The usage of **OBJECT_FORWARD_PERM** behaves like the usage of **OBJECT_FORWARD**, but when used by the server it also provides an indication to the client that it may replace the old IOR with the new IOR. When using **OBJECT_FORWARD_PERM**, both the old IOR and the new IOR are valid, but the new IOR is preferred for future use.
- If the **LocateStatus** value is **LOC_SYSTEM_EXCEPTION**, the body contains a marshaled **GIOP::SystemExceptionReplyBody**.

- If the **LocateStatus** value is **LOC_NEEDS_ADDRESSING_MODE,** then the body contains a **GIOP::AddressingDisposition**. The client ORB is responsible for re-sending the **LocateRequest** using the requested addressing mode.

NOTE: Usage of **OBJECT_FORWARD_PERM** is now deprecated, due to problems it causes with the semantics of the **Object::hash** operation. **OBJECT_FORWARD_PERM** features could be removed from some future GIOP versions if solutions to these problems are not provided.

LocateReply bodies are marshaled immediately following the **LocateReply** header.

9.5.6.3 Handling ForwardRequest Exception from ServantLocator

If the **ServantLocator** in a POA raises a ForwardRequest exception the ORB shall send a **LocateReply** message to the client with **locate_status** set to **OBJECT_FORWARD**, and with the body containing the object reference from the ForwardRequest exception's **forward_reference** field.

9.5.7 CloseConnection Message

CloseConnection messages are sent only by servers in GIOP protocol versions 1.0 and 1.1. They inform clients that the server intends to close the connection and must not be expected to provide further responses. Moreover, clients know that any requests for which they are awaiting replies will never be processed, and may safely be reissued (on another connection). In GIOP version 1.2 or later both sides of the connection may send the **CloseConnection** message.

The **CloseConnection** message consists only of the GIOP message header, identifying the message type.

For details on the usage of **CloseConnection** messages, see Connection Management on page 108.

9.5.8 MessageError Message

The **MessageError** message is sent in response to any GIOP message whose version number or message type is unknown to the recipient or any message received whose header is not properly formed (e.g., has the wrong magic value). Error handling is context-specific.

The **MessageError** message consists only of the GIOP message header, identifying the message type.

9.5.9 Fragment Message

This message is added in GIOP 1.1.

The **Fragment** message is sent following a previous request or response message that has the more fragments bit set to **TRUE** in the **flags** field.

All of the GIOP messages begin with a GIOP header. One of the fields of this header is the **message_size** field, a 32-bit unsigned number giving the number of bytes in the message following the header. Unfortunately, when actually constructing a GIOP **Request** or **Reply** message, it is sometimes impractical or undesirable to ascertain the total size of the message at the stage of message construction where the message header has to be written. GIOP 1.1 provides an alternative indication of the size of the message, for use in those cases.

In GIOP 1.1, a **Request** or **Reply** message can be broken into multiple fragments. In GIOP 1.2 and later, a **Request**, **Reply**, **LocateRequest**, or **LocateReply** message can be broken into multiple fragments. The first fragment is a regular message (e.g., **Request** or **Reply**) with the **more** fragments bit in the **flags** field set to **TRUE**. This initial fragment can be followed by one or more messages using the fragment messages. The last fragment shall have the more fragment bit in the flag field set to **FALSE**.

A **CancelRequest** message may be sent by the client before the final fragment of the message being sent. In this case, the server should assume no more fragments will follow.

NOTE: A GIOP client that fragments the header of a **Request** message before sending the request ID may not send a **CancelRequest** message pertaining to that request ID and may not send another **Request** message until after the request ID is sent.

A primitive data type of 8 bytes or smaller should never be broken across two fragments.

In GIOP 1.1, the data in a fragment is marshaled with alignment relative to its position in the fragment, not relative to its position in the whole unfragmented message.

For GIOP version 1.2 and later, the total length (including the message header) of a fragment other than the final fragment of a fragmented message are required to be a multiple of 8 bytes in length, allowing bridges to defragment and/or refragment messages without having to remarshal the encoded data to insert or remove padding.

For GIOP version 1.2 and later, a fragment header is included in the message, immediately after the GIOP message header and before the fragment data. The request ID, in the fragment header, has the same value as that used in the original message associated with the fragment.

The byte order and GIOP protocol version of a fragment shall be the same as that of the message it continues.

```
module GIOP {//IDL extension for GIOP 1.2 and later

    struct FragmentHeader_1_2 {
        unsigned long request_id;
    };
    typedef FragmentHeader_1_2 FragmentHeader_1_3;
};
```

9.6 GIOP Message Transport

The GIOP is designed to be implementable on a wide range of transport protocols. The GIOP definition makes the following assumptions regarding transport behavior:

- The transport is connection-oriented. GIOP uses connections to define the scope and extent of request IDs.
- The transport is reliable. Specifically, the transport guarantees that bytes are delivered in the order they are sent, at most once, and that some positive acknowledgment of delivery is available.
- The transport can be viewed as a byte stream. No arbitrary message size limitations, fragmentation, or alignments are enforced.
- The transport provides some reasonable notification of disorderly connection loss. If the peer process aborts, the peer host crashes, or network connectivity is lost, a connection owner should receive some notification of this condition.

- The transport's model for initiating connections can be mapped onto the general connection model of TCP/IP. Specifically, an agent (described herein as a server) publishes a known network address in an IOR, which is used by the client when initiating a connection.

The server does not actively initiate connections, but is prepared to accept requests to connect (i.e., it *listens* for connections in TCP/IP terms). Another agent that knows the address (called a client) can attempt to initiate connections by sending *connect* requests to the address. The listening server may *accept* the request, forming a new, unique connection with the client, or it may *reject* the request (e.g., due to lack of resources). Once a connection is open, either side may *close* the connection. (See Connection Management on page 108 for semantic issues related to connection closure.) A candidate transport might not directly support this specific connection model; it is only necessary that the transport's model can be mapped onto this view.

9.6.1 Connection Management

For the purposes of this discussion, the roles client and server are defined as follows:

- A client initiates the connection, presumably using addressing information found in an object reference (IOR) for an object to which it intends to send requests.
- A server accepts connections, but does not initiate them.

These terms only denote roles with respect to a connection. They do not have any implications for ORB or application architectures.

In GIOP protocol versions 1.0 and 1.1, connections are not symmetrical. Only clients can send **Request**, **LocateRequest**, and **CancelRequest** messages over a connection, in GIOP 1.0 and 1.1. In all GIOP versions, a server can send **Reply**, **LocateReply,** and **CloseConnection** messages over a connection; however, in GIOP 1.2 and later the client can send them as well. Either client or server can send **MessageError** messages, in GIOP 1.0 and 1.1.

If multiple GIOP versions are used on an underlying transport connection, the highest GIOP version used on the connection can be used for handling the close. A **CloseConnection** message sent using any GIOP version applies to all GIOP versions used on the connection (i.e., the underlying transport connection is closed for all GIOP versions). In particular, if GIOP version 1.2 or higher has been used on the connection, the client can send the **CloseConnection** message by using the highest GIOP version in use.

Only GIOP messages are sent over GIOP connections.

Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection. Request IDs may be re-used if there is no possibility that the previous request using the ID may still have a pending reply. Note that cancellation does not guarantee no reply will be sent. It is the responsibility of the client to generate and assign request IDs. Request IDs must be unique among both **Request** and **LocateRequest** messages.

9.6.1.1 Connection Closure

Connections can be closed in two ways: orderly shutdown, or abortive disconnect.

For GIOP versions 1.0, and 1.1:

- Orderly shutdown is initiated by servers sending a **CloseConnection** message, or by clients just closing down a connection.
- Orderly shutdown may be initiated by the client at any time.

- A server may not initiate shutdown if it has begun processing any requests for which it has not either received a **CancelRequest** or sent a corresponding reply.
- If a client detects connection closure without receiving a **CloseConnection** message, it must assume an abortive disconnect has occurred, and treat the condition as an error.

For GIOP Version 1.2 and later:

- Orderly shutdown is initiated by either the originating client ORB (connection initiator) or by the server ORB (connection responder) sending a **CloseConnection** message
- If the ORB sending the **CloseConnection** is a server, or bidirectional GIOP is in use, the sending ORB must not currently be processing any Requests from the other side.
- The ORB that sends the **CloseConnection** must not send any messages after the **CloseConnection**.
- If either ORB detects connection closure without receiving a **CloseConnection** message, it must assume an abortive disconnect has occurred, and treat the condition as an error.
- If bidirectional GIOP is in use, the conditions of Bi-Directional GIOP on page 115 apply.

For all uses of **CloseConnection** (for GIOP versions 1.0, 1.1, 1.2, and later):

- If there are any pending non-oneway requests, which were initiated on a connection by the ORB shutting down that connection, the connection-peer ORB should consider them as canceled.
- If an ORB receives a **CloseConnection** message from its connection-peer ORB, it should assume that any outstanding messages (i.e., without replies) were received after the connection-peer ORB sent the CloseConnection message, were not processed, and may be safely re-sent on a new connection.
- After issuing a **CloseConnection** message, the issuing ORB may close the connection. Some transport protocols (not including TCP) do not provide an "orderly disconnect" capability, guaranteeing reliable delivery of the last message sent. When GIOP is used with such protocols, an additional handshake needs to be provided as part of the mapping to that protocol's connection mechanisms, to guarantee that both ends of the connection understand the disposition of any outstanding GIOP requests.

9.6.1.2 Multiplexing Connections

A client, if it chooses, may send requests to multiple target objects over the same connection, provided that the connection's server side is capable of responding to requests for the objects. It is the responsibility of the client to optimize resource usage by reusing connections, if it wishes. If not, the client may open a new connection for each active object supported by the server, although this behavior should be avoided.

9.6.2 Message Ordering

Only the client (connection originator) may send **Request, LocateRequest**, and **CancelRequest** messages, if Bi-Directional GIOP is not in use.

Clients may have multiple pending requests. A client need not wait for a reply from a previous request before sending another request.

Servers may reply to pending requests in any order. **Reply** messages are not required to be in the same order as the corresponding **Requests**.

The ordering restrictions regarding connection closure mentioned in Connection Management, above, are also noted here. Servers may only issue **CloseConnection** messages when **Reply** messages have been sent in response to all received **Request** messages that require replies.

9.7 Object Location

The GIOP is defined to support object migration and location services without dictating the existence of specific ORB architectures or features. The protocol features are based on the following observations.

A given transport address does not necessarily correspond to any specific ORB architectural component (such as an *object adapter*, *object server process*, *Inter-ORB bridge*, and so forth). It merely implies the existence of some agent with which a connection may be opened, and to which requests may be sent.

The "agent" (owner of the server side of a connection) may have one of the following roles with respect to a particular object reference:

- The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be an Inter-ORB bridge that transforms the request and passes it on to another process or ORB. From GIOP's perspective, it is only important that requests can be sent directly to the agent.
- The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any **Request** messages sent to the agent would result in either exceptions or replies with **LOCATION_FORWARD** status, providing new addresses to which requests may be sent. Such agents would also respond to **LocateRequest** messages with appropriate **LocateReply** messages.
- The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.
- The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time (perhaps during the same connection).

Agents are not required to implement location forwarding mechanisms. An agent can be implemented with the policy that a connection either supports direct access to an object, or returns exceptions. Such an ORB (or inter-ORB bridge) always returns **LocateReply** messages with either **OBJECT_HERE** or **UNKNOWN_OBJECT** status, and never **OBJECT_FORWARD** status.

Clients must, however, be able to accept and process **Reply** messages with **LOCATION_FORWARD** status, since any ORB may choose to implement a location service. Whether a client chooses to send **LocateRequest** messages is at the discretion of the client. For example, if the client routinely expected to see **LOCATION_FORWARD** replies when using the address in an object reference, it might always send **LocateRequest** messages to objects for which it has no recorded forwarding address. If a client sends **LocateRequest** messages, it should be prepared to accept **LocateReply** messages.

A client shall not make any assumptions about the longevity of object addresses returned by **LOCATION_FORWARD** (**OBJECT_FORWARD**) mechanisms. Once a connection based on location-forwarding information is closed, a client can attempt to reuse the forwarding information it has, but, if that fails, it shall restart the location process using the original address specified in the initial object reference.

For GIOP version 1.2 and later, the usage of **LOCATION_FORWARD_PERM** (**OBJECT_FORWARD_PERM**) behaves like the usage of **LOCATION_FORWARD** (**OBJECT_FORWARD**), but when used by the server it also provides an indication to the client that it may replace the old IOR with the new IOR. When using **LOCATION_FORWARD_PERM** (**OBJECT_FORWARD_PERM**), both the old IOR and the new **IOR** are valid, but the new IOR is preferred for future use.

NOTE: Usage of **LOCATION_FORWARD_PERM** and **OBJECT_FORWARD_PERM** is now deprecated, due to problems it causes with the semantics of the **Object::hash** operation. **LOCATION_FORWARD_PERM** and **OBJECT_FORWARD_PERM** features could be removed from some future GIOP versions if solutions to these problems are not provided.

Even after performing successful invocations using an address, a client should be prepared to be forwarded. The only object address that a client should expect to continue working reliably is the one in the initial object reference. If an invocation using that address returns **UNKNOWN_OBJECT**, the object should be deemed non-existent.

In general, the implementation of location forwarding mechanisms is at the discretion of ORBs, available to be used for optimization and to support flexible object location and migration behaviors.

9.8 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[5]. Specific APIs for libraries supporting TCP/IP may vary, so this discussion is limited to an abstract view of TCP/IP and management of its connections. The mapping of GIOP message transfer to TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

IIOP 1.0 is based on GIOP 1.0.

IIOP 1.1 can be based on either GIOP 1.0 or 1.1. An IIOP 1.1 client must support GIOP 1.1, and may also support GIOP 1.0. An IIOP 1.1 server must support processing both GIOP 1.0 and GIOP 1.1 messages.

IIOP 1.2 can be based on any of the GIOP minor versions 1.0, 1.1, or 1.2. An IIOP 1.2 client must support GIOP 1.2, and may also support lesser GIOP minor versions. An IIOP 1.2 server must also support processing messages with all lesser GIOP versions.

IIOP 1.3 can be based on any of the GIOP minor versions 1.0, 1.1, 1.2, or 1.3. An IIOP 1.3 client must support GIOP 1.3, and may also support lesser GIOP minor versions. An IIOP 1.3 server must also support processing messages with all lesser GIOP versions.

IIOP 1.4 can be based on any of the GIOP minor versions 1.0, 1.1, 1.2, 1.3, or 1.4. An IIOP 1.4 client must support GIOP 1.4, and may also support lesser GIOP minor versions. An IIOP 1.4 server must also support processing messages with all lesser GIOP versions.

Conformance to IIOP versions 1.1, 1.2, 1.3, and 1.4 requires support of Limited-Profile IOR conformance (see Interoperable Object References: IORs on page 25), specifically for the IIOP IOR Profile. As of CORBA 2.4, this limited IOR conformance is deprecated, and ORBs implementing IIOP are strongly recommended to support Full IOR conformance. Some future IIOP versions could require support of Full IOR conformance.

9.8.1 TCP/IP Connection Usage

Agents that are capable of accepting object requests or providing locations for objects (i.e., servers) publish TCP/IP addresses in IORs, as described in IIOP IOR Profiles on page 112. A TCP/IP address consists of an IP host address, typically represented by a host name, and a TCP port number. Servers must listen for connection requests.

A client needing an object's services must initiate a connection with the address specified in the IOR, with a connect request.

---

5. Postel, J., "Transmission Control Protocol – DARPA Internet Program Protocol Specification," RFC-793, Information Sciences Institute, September 1981

The listening server may accept or reject the connection. In general, servers should accept connection requests if possible, but ORBs are free to establish any desired policy for connection acceptance (e.g., to enforce fairness or optimize resource usage).

Once a connection is accepted, the client may send **Request, LocateRequest**, or **CancelRequest** messages by writing to the TCP/IP socket it owns for the connection. The server may send **Reply, LocateReply**, and **CloseConnection** messages by writing to its TCP/IP connection. In GIOP 1.2, and later, the client may send the **CloseConnection** message, and if BiDirectional GIOP is in use, the client may also send **Reply and LocateReply messages.**

After receiving a **CloseConnection** message, an ORB must close the TCP/IP connection. After sending a **CloseConnection**, an ORB may close the TCP/IP connection immediately, or may delay closing the connection until it receives an indication that the other side has closed the connection. For maximum interoperability with ORBs using TCP implementations that do not properly implement orderly shutdown, an ORB may wish to only shutdown the sending side of the connection, and then read any incoming data until it receives an indication that the other side has also shutdown, at which point the TCP connection can be closed completely.

Given TCP/IP's flow control mechanism, it is possible to create deadlock situations between clients and servers if both sides of a connection send large amounts of data on a connection (or two different connections between the same processes) and do not read incoming data. Both processes may block on write operations, and never resume. It is the responsibility of both clients and servers to avoid creating deadlock by reading incoming messages and avoiding blocking when writing messages, by providing separate threads for reading and writing, or any other workable approach. ORBs are free to adopt any desired implementation strategy, but should provide robust behavior.

9.8.2 IIOP IOR Profiles

IIOP profiles, identifying individual objects accessible through the Internet Inter-ORB Protocol, have the following form:

```
module IIOP {  // IDL extended for version 1.1, 1.2, and later
    struct Version {
        octet           major;
        octet           minor;
    };

    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version                     iiop_version;
        string                      host;
        unsigned short              port;
        IOP::ObjectKey              object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2 and later
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        IOP::ObjectKey              object_key;
```

```
    // Added in 1.1 unchanged for 1.2 and later
        IOP::TaggedComponentSeq components;
    };
};
```

IIOP Profile version number:

- Indicates the IIOP protocol version.
- Major number can stay the same if the new changes are backward compatible.
- Clients with lower minor version can attempt to invoke objects with higher minor version number by using only the information defined in the lower minor version protocol (ignore the extra information).

Profiles supporting only IIOP version 1.0 use the **ProfileBody_1_0** structure, while those supporting IIOP version 1.1 or 1.2 or later use the **ProfileBody_1_1** structure. An instance of one of these structure types is marshaled into an encapsulation octet stream. This encapsulation (a **sequence <octet>**) becomes the **profile_data** member of the **IOP::TaggedProfile** structure representing the IIOP profile in an IOR, and the tag has the value **TAG_INTERNET_IOP** (as defined earlier).

The version number published in the Tag Internet IIOP Profile body signals the highest GIOP minor version number that the server supports at the time of publication of the IOR.

If the major revision number is 1, and the minor revision number is greater than 0, then the length of the encapsulated profile may exceed the total size of components defined in this part of ISO/IEC 19500 for profiles with minor revision number 0. ORBs that support only revision 1.0 IIOP profiles must ignore any data in the profile that occurs after the **object_key**. If the revision of the profile is 1.0, there shall be no extra data in the profile (i.e., the length of the encapsulated profile must agree with the total size of components defined for version 1.0).

For Version 1.2 and later of IIOP, no order of use is prescribed in the case where more than one TAG Internet IOP Profile is present in an IOR.

The members of **IIOP::ProfileBody_1_0 and IOP::ProfileBody_1_1** are defined as follows:

- **iiop_version** describes the version of IIOP that the agent at the specified address is prepared to receive. When an agent generates IIOP profiles specifying a particular version, it must be able to accept messages complying with the specified version or any previous minor version (i.e., any smaller version number). The major version number of this text is 1; the minor versions defined to date are 0, 1, and 2. Compliant ORBs must generate version 1.1 profiles, and must accept any profile with a major version of 1, regardless of the minor version number. If the minor version number is 0, the encapsulation is fully described by the **ProfileBody_1_0** structure. If the minor version number is 1 or 2, the encapsulation is fully described by the **ProfileBody_1_1** structure. If the minor version number is greater than 2, then the length of the encapsulated profile may exceed the total size of components defined in this text for profiles with minor version number 1 or 2. ORBs that support only version 1.1 or 1.2 IIOP profiles must ignore, but preserve, any data in the profile that occurs after the **components** member, for IIOP profiles with minor version greater than 1.2.

NOTE: As of version 1.2 of GIOP and IIOP and minor versions beyond, the minor version in the **TAG_INTERNET_IOP** profile signals the highest minor revision of GIOP supported by the server at the time of publication of the IOR.

- **host** identifies the Internet host to which GIOP messages for the specified object may be sent. In order to promote a very large (Internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host, rather than an unqualified (or partially qualified) name. However, per Internet standards, the host string may also contain a host address expressed in standard “dotted decimal” form (e.g., “192.231.79.52”).

- **port** contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.
- **object_key** is an opaque value supplied by the agent producing the IOR. This value will be used in request messages to identify the object to which the request is directed. An agent that generates an object key value must be able to map the value unambiguously onto the corresponding object when routing requests internally.
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. **TaggedComponent**s that apply to IIOP 1.2 are described below in IIOP IOR Profile Components on page 114. Other components may be included to support enhanced versions of IIOP, to support ORB services such as security, and to support other GIOPs, ESIOPs, and proprietary protocols. If an implementation puts a non-standard component in an IOR, it cannot be assured that any or all non-standard components will remain in the IOR.

  The relationship between the IIOP protocol version and component support conformance requirements is as follows:

  - Each IIOP version specifies a set of standard components and the conformance rules for that version. These rules specify which components are mandatory and which are optional. A conformant implementation has to conform to these rules, and is not required to conform to more than these rules.
  - New components can be added, but they do not become part of the versions conformance rules.
  - When there is a need to specify conformance rules that include the new components, there will be a need to create a new IIOP version.

Note that host addresses are restricted in this version of IIOP to be Class A, B, or C Internet addresses. That is, Class D (multi-cast) addresses are not allowed. Such addresses are reserved for use in future versions of IIOP.

Agents may freely choose TCP port numbers for communication; IIOP supports multiple agents per host.

9.8.3 IIOP IOR Profile Components

The following components are part of IIOP 1.1, 1.2, and later conformance. All these components are optional.

- **TAG_ORB_TYPE**
- **TAG_CODE_SETS**
- **TAG_SEC_NAME**
- **TAG_ASSOCIATION_OPTIONS**
- **TAG_GENERIC_SEC_MECH**
- **TAG_SSL_SEC_TRANS**
- **TAG_SPKM_1_SEC_MECH**
- **TAG_SPKM_2_SEC_MECH**
- **TAG_KerberosV5_SEC_MECH**
- **TAG_CSI_ECMA_Secret_SEC_MECH**
- **TAG_CSI_ECMA_Hybrid_SEC_MECH**
- **TAG_SSL_SEC_TRANS**
- **TAG_CSI_ECMA_Public_SEC_MECH**

- **TAG_FIREWALL_TRANS**
- **TAG_JAVA_CODEBASE**
- **TAG_TRANSACTION_POLICY**
- **TAG_MESSAGE_ROUTERS**
- **TAG_INET_SEC_TRANS**

The following components are part of IIOP 1.2, and later conformance. All these components are optional.

- **TAG_ALTERNATE_IIOP_ADDRESS**
- **TAG_POLICIES**
- **TAG_DCE_STRING_BINDING**
- **TAG_DCE_BINDING_NAME**
- **TAG_DCE_NO_PIPES**
- **TAG_DCE_MECH**
- **TAG_COMPLETE_OBJECT_KEY**
- **TAG_ENDPOINT_ID_POSITION**
- **TAG_LOCATION_POLICY**
- **TAG_OTS_POLICY**
- **TAG_INV_POLICY**
- **TAG_CSI_SEC_MECH_LIST**
- **TAG_NULL_TAG**
- **TAG_SECIOP_SEC_TRANS**
- **TAG_TLS_SEC_TRANS**
- **TAG_ACTIVITY_POLICY**

9.9 Bi-Directional GIOP

The specification of GIOP connection management, in GIOP minor versions 1.0 and 1.1, states that connections are not symmetrical. For example, only clients that initialize connections can send requests, and only servers that accept connections can receive them.

This GIOP 1.0 and 1.1 restriction gives rise to significant difficulties when operating across firewalls. It is common for firewalls not to allow incoming connections, except to certain well-known and carefully configured hosts, such as dedicated HTTP or FTP servers. For most CORBA-over-the-internet applications it is not practicable to require that all potential client firewalls install GIOP proxies to allow incoming connections, or that any entities receiving callbacks will require prior configuration of the firewall proxy.

An applet, for example, downloaded to a host inside such a firewall will be restricted in that it cannot receive requests from outside the firewall on any object it creates, as no host outside the firewall will be able to connect to the applet through the client's firewall, even though the applet in question would typically only expect callbacks from the server it initially registered with.

In order to circumvent this unnecessary restriction, GIOP minor protocol version 1.2 or later specifies that the asymmetry stipulation above be relaxed in cases where the client and the server agree on it. In these cases, the client (the applet in the above case) would still initiate the connection to the server, but any requests from the server on any objects.

The client creates an object for exporting to a server, and arranges that the server receive an IOR for the object. The most common use case would be for the client to pass the IOR as a parameter in a GIOP request, but other mechanisms are possible, such as the use of a Name Service. If the client ORB policy permits bi-directional use of a connection, a Request message should contain an **IOP::ServiceContext** structure in its Request header, which indicates that this GIOP connection is bi-directional. The service context may provide additional information that the server may need to invoke the callback object. To determine whether an ORB may support bi-directional GIOP new policies has been defined (Bi-directional GIOP policy on page 118).

Each mapping of GIOP to a particular transport should define a transport-specific bi-directional service context, and have an **IOP::ServiceId** allocated by the OMG. It is recommended that names for this service context follows the pattern *BiDir<protocolname>*, where <protocol name> identifies a mapping of GIOP to a transport protocol (e.g., for IIOP the name is **BiDirIIOP**). The service context for bi-directional IIOP is defined in Bi-directional IIOP on page 117.

The server receives the Request, which contains a bi-directional **IOP::ServiceContext**. If the server supports bi-directional connections for that protocol, it may now send invocations along the same connection to any object that supports the particular protocol and matches the particular location information found in the bi-directional service context. If the server does not support bi-directional connections for that protocol, the service context can be ignored.

The data encapsulated in the **BiDirIIOPServiceContext** structure (see below), which is identified by the **ServiceId BI_DIR_IIOP** as defined in Service Context on page 37, allows the ORB to determine whether it needs to open a new connection in order to invoke on an object. If a host and port pair in a listen_point list matches a host and port of an object to which it does not yet have a connection (a callback object newly received, for instance), rather than open a new connection, the server may re-use any of the connections on which the listen_point data was received.

A server talking to a client on a bi-directional GIOP connection can use any message type traditionally used by clients only, so it can use **Request**, **LocateRequest**, **CancelRequest**, **MessageError,** and **Fragment** (for a **Request** or **LocateRequest**). Similarly the client can use message types traditionally used only by servers: **Reply**, **LocateReply**, **MessageError**, **CloseConnection,** and **Fragment** (for a **Reply** or **LocateReply**).

CloseConnection messages are a special case however. Either ORB may send a **CloseConnection** message, but the conditions in Connection Management on page 108 apply.

Bi-directional GIOP connections modify the behavior of Request IDs. In the GIOP specification, Connection Management on page 108, it is noted that "Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection." This property of unambiguous association of requests and replies must be preserved while permitting each end to generate Request Ids for new requests independently. To ensure this, on a connection that is used bi-directionally in GIOP 1.2, and later, the connection originator shall assign only even valued Request IDs and the other side of the connection shall assign only odd valued Request IDs. This requirement applies to the full lifetime of the connection, even before a **BiDirIIOPServiceContext** is transmitted. A connection on which this regime of Request ID assignment is not used, shall never be used to transmit bi-directional GIOP 1.2, or later messages.

It should be noted that a single-threaded ORB needs to perform event checking on the connection, in case a **Request** from the other endpoint arrives in the window between it sending its own **Request** and receiving the corresponding reply; otherwise a client and server could send **Request**s simultaneously, resulting in deadlock. If the client cannot support event checking, it must not indicate that bi-directionality is supported. If the server cannot support event checking, it must not make callbacks along the same connection even if the connection indicates it is supported.

A server making a callback to a client cannot specify its own bi-directional service context – only the client can announce the connection's bi-directionality.

An important security issue should be observed in the use of bi-directional GIOP. In the absence of other security mechanisms, a malicious client may claim that its connection is Bi-Directional for use with any host and port it chooses. In particular it may specify the host and port of security sensitive objects not even resident on its host. All the client has to do is pass the host and port in the listen data service context and the server may then invoke a masquerading object instead. In general, and in the absence of other security mechanisms, a server that has accepted an incoming connection has no way to discover the identity or verify the integrity of the client that initiated the connection. If the server has doubts in the integrity of the client, it is recommended that bi-directional GIOP is not used.

9.9.1 Bi-directional IIOP

The **IOP::ServiceContext** used to support bi-directional IIOP contains a **BiDirIIOPServiceContext** structure as defined below:

```
// IDL
module IIOP {

    struct ListenPoint {
        string   host;
        unsigned short port;
    };

    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {
        ListenPointList listen_points;
    };
};
```

The data encapsulated in the **BiDirIIOPServiceContext** structure, which is identified by the ServiceId **BI_DIR_IIOP** as defined in Service Context on page 37, allows the ORB, which intends to open a new connection in order to invoke on an object, to look up its list of active client-initiated connections and examine the structures associated with them, if any. If a **host** and **port** pair in a **listen_points** list matches a host and port, which the ORB intends to open a connection to, rather than open a new connection to that **listen_point**, the server may re-use any of the connections that were initiated by the client on which the listen point data was received.

The **host** element of the structure should contain whatever values the client may use in the IORs it creates. The rules for **host** and **port** are identical to the rules for the IIOP IOR **ProfileBody_1_1 host** and **port** elements; see IIOP IOR Profiles on page 112. Note that if the server wishes to make a callback connection to the client in the standard way, it must use the values from the client object's IOR, not the values from this **BiDirIIOPServiceContext** structure; these values are only to be used for bi-directional GIOP support.

The **BI_DIR_IIOP** service context may be sent by a client at any point in a connection's lifetime. The **listen_points** specified therein must supplement any **listen_points** already sent on the connection, rather than replacing the existing points.

If a client supports a secure connection mechanism, such as SECIOP or IIOP/SSL, and sends a **BI_DIR_IIOP** service context over an insecure connection, the **host** and **port** endpoints listed in the **BI_DIR_IIOP** should not contain the details of the secure connection mechanism if insecure callbacks to the client's secure objects would be a violation of the client's security policy.

It is the ORB's responsibility to ensure that an IOR contains an appropriate address.

9.9.1.1 IIOP/SSL considerations

Bi-directional IIOP can operate over IIOP/SSL without defining any additions to the IIOP/SSL or the bi-directional GIOP mechanisms. However, if the client wants to authenticate the server when the client receives a callback this cannot be done unless the client has already authenticated the server. This has to be performed during the initial SSL handshake. It is not possible to do this at any point after the initial handshake without establishing a new SSL connection (which defeats the purpose of the bi-directional connections).

9.10 Bi-directional GIOP policy

In GIOP protocol versions 1.0 and 1.1, there are strict rules on which side of a connection can issue what type of messages (for example version 1.0 and 1.1 clients can not issue GIOP reply messages). However, as documented above, it is sensible to relax this restriction if the ORB supports this functionality and policies dictate that bi-directional connection are allowed. To indicate a bi-directional policy, the following is defined.

```
// Self contained module for Bi-directional GIOP policy

module BiDirPolicy {

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = 37;

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```

A **BidirectionalPolicyValue** of **NORMAL** states that the usual GIOP restrictions of who can send what GIOP messages apply (i.e., bi-directional connections are not allowed). A value of **BOTH** indicates that there is a relaxation in what party can issue what GIOP messages (i.e., bi-directional connections are supported). The default value of a **BidirectionalPolicy** is **NORMAL**.

In the absence of a **BidirectionalPolicy** being passed in the **PortableServer::POA::create_POA** operation, a **POA** will assume a policy value of **NORMAL**.

A client and a server **ORB** must each have a **BidirectionalPolicy** with a value of **BOTH** for bi-directional communication to take place.

To create a **BidirectionalPolicy**, the **ORB::create_policy** operation is used.

9.11 OMG IDL

This sub clause contains the OMG IDL for the GIOP and IIOP modules.

9.11.1 GIOP Module

```
module GIOP {      // IDL extended for version 1.1, 1.2, and later

    struct Version {
        octet       major;
        octet       minor;
    };

#if MAX_GIOP_MINOR_VERSION == 0
    // GIOP 1.0
    enum MsgType_1_0{   // rename from MsgType
        Request, Reply,     CancelRequest,
        LocateRequest,      LocateReply,
        CloseConnection,    MessageError
        };

#else
    // GIOP 1.1
    enum MsgType_1_1{
        Request,            Reply,          CancelRequest,
        LocateRequest,      LocateReply,
        CloseConnection,    MessageError,
        Fragment            // GIOP 1.1 addition
    };
#endif // MAX_GIOP_MINOR_VERSION

    // GIOP 1.0

    typedef char Magicn[4]
    struct MessageHeader_1_0 {// Renamed from MessageHeader
        Magicn                  magic;
        Version                 GIOP_version;
        boolean                 byte_order;
        octet                   message_type;
        unsigned long           message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        Magicn                  magic;
        Version                 GIOP_version;
        octet                   flags;          // GIOP 1.1 change
        octet                   message_type;
        unsigned long           message_size;
    };
```

```
// GIOP 1.2 and later
typedef MessageHeader_1_1 MessageHeader_1_2;
typedef MessageHeader_1_1 MessageHeader_1_3;

// GIOP 1.0

struct RequestHeader _1_0 {
    IOP::ServiceContextList   service_context;
    unsigned long             request_id;
    boolean                   response_expected;
    IOP::ObjectKey            object_key;
    string                    operation;
    CORBA::OctetSeq           requesting_principal;
};

// GIOP 1.1
typedef octet RequestReserved[3];
struct RequestHeader_1_1 {
    IOP::ServiceContextList   service_context;
    unsigned long             request_id;
    boolean                   response_expected;
    RequestReserved           reserved; // Added in GIOP 1.1
    IOP::ObjectKey            object_key;
    string                    operation;
    CORBA::OctetSeq           requesting_principal;
};

// GIOP 1.2, and later
typedef short                 AddressingDisposition;
const short                   KeyAddr = 0;
const short                   ProfileAddr = 1;
const short                   ReferenceAddr = 2;

struct IORAddressingInfo {
    unsigned long             selected_profile_index;
    IOP::IOR                  ior;
};

union TargetAddress switch (AddressingDisposition) {
    case KeyAddr:             IOP::ObjectKey object_key;
    case ProfileAddr:         IOP::TaggedProfile profile;
    case ReferenceAddr:       IORAddressingInfo ior;
};

struct RequestHeader_1_2 {
    unsigned long             request_id;

    octet                     response_flags;
    RequestReserved           reserved; // Added in GIOP 1.1
    TargetAddress             target;
    string                    operation;
```

```
        // requesting_principal not in GIOP 1.2 and later
        IOP::ServiceContextList    service_context;    // 1.2 change
    };

#if MAX_GIOP_MINOR_VERSION < 2

    // GIOP 1.0 and 1.1
    enum ReplyStatusType_1_0 {// Renamed from ReplyStatusType
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    // GIOP 1.0
    struct ReplyHeader_1_0 {// Renamed from ReplyHeader
        IOP::ServiceContextList    service_context;
        unsigned long              request_id;
        ReplyStatusType_1_0        reply_status;
    };

    // GIOP 1.1
    typedef ReplyHeader_1_0 ReplyHeader_1_1;
    // Same Header contents for 1.0 and 1.1

#endif // MAX_GIOP_VERSION_NUMBER

#if MAX_GIOP_MINOR_VERSION >= 2

    // GIOP 1.2, and later
    enum ReplyStatusType_1_2 {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD,
        LOCATION_FORWARD_PERM,      // new value for 1.2
        NEEDS_ADDRESSING_MODE       // new value for 1.2
    };

    struct ReplyHeader_1_2 {
        unsigned long                   request_id;
        ReplyStatusType_1_2             reply_status;
        IOP::ServiceContextList         service_context;    // 1.2 change
    };
    typedef ReplyHeader_1_2 ReplyHeader_1_3;

#endif // MAX_GIOP_VERSION_NUMBER

    struct SystemExceptionReplyBody {
        string                      exception_id;
        unsigned long               minor_code_value;
```

```
        unsigned long               completion_status;
    };

    struct CancelRequestHeader {
        unsigned long               request_id;
    };

    // GIOP 1.0
    struct LocateRequestHeader_1_0 {
                                    // Renamed LocationRequestHeader
        unsigned long               request_id;
        IOP::ObjectKey              object_key;
    };

    // GIOP 1.1
        typedef LocateRequestHeader_1_0 LocateRequestHeader_1_1;
        // Same Header contents for 1.0 and 1.1

    // GIOP 1.2 and later
        struct LocateRequestHeader_1_2 {
            unsigned long           request_id;
            TargetAddress           target;
    };
    typedef LocateRequestHeader_1_2 LocateRequestHeader_1_3;

#if MAX_GIOP_MINOR_VERSION < 2

    // GIOP 1.0 and 1.1
    enum LocateStatusType_1_0 {// Renamed from LocateStatusType
            UNKNOWN_OBJECT,
            OBJECT_HERE,
            OBJECT_FORWARD
    };

    // GIOP 1.0
        struct LocateReplyHeader_1_0 {
                                        // Renamed from LocateReplyHeader
            unsigned long               request_id;
            LocateStatusType_1_0        locate_status;
    };

    // GIOP 1.1
    typedef LocateReplyHeader_1_0 LocateReplyHeader_1_1;
    // same Header contents for 1.0 and 1.1

#else
    // GIOP 1.2, and later
    enum LocateStatusType_1_2 {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD,
```

```
        OBJECT_FORWARD_PERM,        // new value for GIOP 1.2
        LOC_SYSTEM_EXCEPTION,       // new value for GIOP 1.2
        LOC_NEEDS_ADDRESSING_MODE   // new value for GIOP 1.2
    };

    struct LocateReplyHeader_1_2 {
        unsigned long               request_id;
        LocateStatusType_1_2        locate_status;
    };
    typedef LocateReplyHeader_1_2 LocateReplyHeader_1_3;

#endif // MAX_GIOP_VERSION_NUMBER

    // GIOP 1.2, and later
    struct FragmentHeader_1_2 {
        unsigned long               request_id;
    };
    typedef FragmentHeader_1_2 FragmentHeader_1_3;
};
```

9.11.2 IIOP Module

```
module IIOP {  // IDL extended for version 1.1, 1.2, and later
    struct Version {
        octet                   major;
        octet                   minor;
    };

    struct ProfileBody_1_0 {// renamed from ProfileBody
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        IOP::ObjectKey              object_key;
    };

    struct ProfileBody_1_1 {// also used for 1.2, and later
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        IOP::ObjectKey              object_key;

        // Added in 1.1 unchanged for 1.2, and later
        IOP::TaggedComponentSeq components;
    };

    struct ListenPoint {
        string   host;
        unsigned short port;
    };
```

```
    typedef sequence<ListenPoint> ListenPointList;

    struct BiDirIIOPServiceContext {// BI_DIR_IIOP Service Context
        ListenPointList listen_points;
    };
};
```

9.11.3 BiDirPolicy Module

```
// Self contained module for Bi-directional GIOP policy

module BiDirPolicy {

    typedef unsigned short BidirectionalPolicyValue;
    const BidirectionalPolicyValue NORMAL = 0;
    const BidirectionalPolicyValue BOTH = 1;

    const CORBA::PolicyType BIDIRECTIONAL_POLICY_TYPE = 37;

    interface BidirectionalPolicy : CORBA::Policy {
        readonly attribute BidirectionalPolicyValue value;
    };
};
```

10 Secure Interoperability

10.1 Overview

This clause defines the CORBA Security Attribute Service (SAS) protocol and its use within the CSIv2 architecture to address the requirements of CORBA security for interoperable authentication, delegation, and privileges.

The SAS protocol is designed to exchange its protocol elements in the service context of GIOP request and reply messages that are communicated over a connection-based transport. The protocol is intended to be used in environments where transport layer security, such as that available via SSL/TLS or SECIOP, is used to provide message protection (that is, integrity and or confidentiality) and server-to-client authentication. The protocol provides client authentication, delegation, and privilege functionality that may be applied to overcome corresponding deficiencies in an underlying transport.[1] The SAS protocol facilitates interoperability by serving as the higher-level protocol under which secure transports may be unified.

The SAS protocol is divided into two layers:

- The authentication layer is used to perform client authentication where sufficient authentication could not be accomplished in the transport.
- The attribute layer may be used by a client to push (that is, deliver) security attributes (identity and privilege) to a target where they may be applied in access control decisions.

The attribute layer also provides a means for a client to assert identity attributes that differ from the client's authentication identity (as established in the transport and/or SAS authentication layers). This identity assertion capability is the foundation of a general-purpose impersonation mechanism that makes it possible for an intermediate to act on behalf of some identity other than itself. An intermediate's authority to act on behalf of another identity may be based on trust by the target in the intermediate, or on trust by the target in a privilege authority that endorses the intermediate to act as proxy for the asserted identity. Identity assertion may be used by an intermediate to assume the identity of its callers in its calls.

The SAS protocol is modeled after the Generic Security Service API (GSSAPI) token exchange paradigm. A client initiates a context exchange by including a protocol element in the service context of its request that instructs the target to initiate a security context. The target either rejects or accepts the context.[2] When a target rejects a context, the target will reject the request and return an exception that contains a SAS protocol element that identifies the reason the context was rejected. When a target accepts a context, the reply to the request will carry a SAS protocol element that indicates that the context was accepted.

The SAS protocol element sent to initiate a security context carries layer-specific security tokens as necessary to establish the SAS authentication-layer and attribute-layer functionality corresponding to the context. Standard token formats are employed to represent the layer-specific authentication and attribute tokens. If the context includes SAS authentication-layer functionality, the protocol element will contain a mechanism-specific GSSAPI initial context token that authenticates the client to the target. If the context includes attribute-layer privilege attributes (and possibly proxy

1. For example, the SSL/TLS protocol does not enforce client authentication. Moreover, in a given environment, certificate-based client authentication may not be feasible because clients often do not have a certificate.
2. In the GSSAPI protocol, a target can challenge a client for additional context-establishment information. This is not true of the SAS context protocol, which assumes that at most one message in each direction may be used to establish a context.

endorsements), they will be contained in an attribute certificate signed by a privilege authority and corresponding to the subject of the invocation. If the context includes an attribute-layer identity assertion, the asserted identity will be represented in a standard name form corresponding to the technology domain of the asserted identity.

The SAS protocol supports the establishment of both transient and reusable security contexts. Transient contexts, also known as stateless contexts, exist only for the duration of the GIOP request that was used to establish the context. Reusable contexts, also known as stateful contexts, endure until they are discarded, and can be referenced for use with subsequent requests. The SAS protocol includes a simple negotiation protocol that defines a least-common-denominator form of interoperability between implementations that support only transient contexts and those that support both transient and reusable forms.

10.1.1 Assumptions

The SAS protocol was designed under the following assumptions:

- Secure interoperability is predicated on the use of a common transport-layer security mechanism, such as that provided by SSL/TLS.[3]
- The transport layer provides message protection as necessary to protect GIOP input and output request arguments.
- The transport layer provides target-to-client authentication as necessary to identify the target for the purpose of ensuring that the target is the intended target.
- Transport-layer security can ensure that the client does not have to issue a preliminary request to establish a confidential association with the intended target.[4]
- To support clients that cannot authenticate using transport-layer security mechanisms, the SAS protocol shall provide for client authentication above the transport layer.
- To support the formation of security contexts using GIOP service context, the SAS protocol shall require at most one message in each direction to establish a security context.
- The protocol shall support security contexts that exist only for the duration of a single request/reply pair.
- The protocol shall support security contexts that can be reused for multiple request/reply pairs.
- Targets cannot rely on clients to manage the lifecycle of reusable security contexts accepted by the target.
- Clients that reuse security contexts shall be capable of processing replies that indicate that the context has been discarded by the target.

3. Transport security mechanisms include unprotected transports within trusted environments.
4. This assumption does not preclude the use of such mechanisms, but rather sustains the use of this protocol in environments where such mechanisms are not considered favorably.

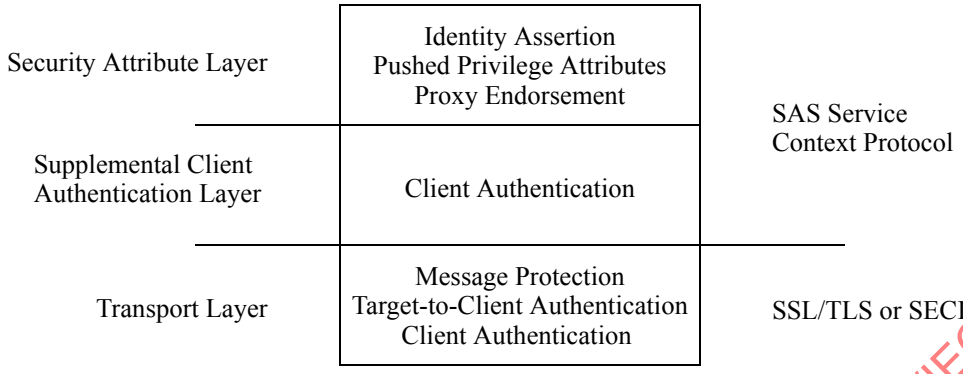


Figure 10.1 - CSIv2 Security Architecture

10.2 Protocol Message Definitions

10.2.1 The Security Attribute Service Context Element

This part of ISO/IEC 19500 defines a new GIOP service context element type, the security attribute service (SAS) element. The SAS context element may be used to associate any or all of the following contexts with GIOP request and reply messages:

- Identity context, to be accepted based on trust
- Authorization context, including authorization-based delegation context
- Client authentication context

A new **context_id** has been defined for the SAS element.

```
const ServiceId SecurityAttributeService = 15;
```

The **context_data** of a SAS element is an encapsulation octet stream containing a SAS message body marshaled according to the CDR encoding rules. The formats of the SAS message bodies are defined in the next sub clause.

```
struct ServiceContext {
    ServiceId context_id;
    sequence <octet> context_data;
};
```

At most one instance of this new service context element may be included in a GIOP request or reply.

10.2.2 SAS context_data Message Body Types

Four message types comprise the security attribute service context management protocol. Each security attribute service context element shall contain a message body that carries one of the following message body types:

- **EstablishContext**
  Sent by a client security service (CSS) to establish a security attribute service context.
- **ContextError**
  Sent by a target security service (TSS) to indicate errors that were encountered in context creation, in the message protocol, or in use of a context.
- **CompleteEstablishContext**
  Sent by a target security service (TSS) to indicate the outcome of a successful request to establish a security attribute service context.
- **MessageInContext**
  Sent by a client security service (CSS) to associate request messages with an existing stateful security attribute service context. This message may also be used to indicate that the context should be discarded after processing the request. Stateful contexts, also known as reusable contexts, endure until they are discarded, and can be referenced for use with subsequent requests.

A client security service (CSS) is the security service associated with the ORB that is used by the client to invoke the target object. A target security service (TSS) is the security service associated with the ORB that hosts the target object.

10.2.2.1 EstablishContext Message Format

An **EstablishContext** message is sent by a CSS to establish a SAS context with a TSS. The SAS context and the context identifier allocated by the CSS to refer to it are scoped to the transport layer connection or association over which the CSS and TSS are communicating. When an association is dismantled, all SAS contexts scoped to the connection shall be invalidated and may be discarded. The **EstablishContext** message contains the following fields:

- **client_context_id**
  The CSS allocated identifier for the security attribute service context. A stateless CSS shall set the **client_context_id** to 0, indicating to the TSS that it is stateless. A stateful CSS may allocate a nonzero client_context_id. See Stateful/Reusable Contexts on page 142 for a definition of the rules governing the use and allocation of context identifiers.
- **authorization_token**
  May be used by a CSS to "push" privilege information to a TSS. A CSS may use this token to send proxy privileges to a TSS as a means to enable the target to issue calls as the client.
- **identity_token**
  Carries a representation of the invocation identity for the call (that is, the identity under which the call is to be authorized). The **identity_token** carries a representation of the invocation identity in one of the following forms:
  - A typed mechanism-specific representation of a principal name.
  - A chain of identity certificates representing the subject and a chain of verifying authorities.
  - A distinguished name.
  - The anonymous principal identity (a type, not a name).

  An **identity_token** is used to assert a caller identity when that identity differs from the identity proven by authentication in the authentication layer(s). If the caller identity is intended to be the same as that established in the authentication layer(s), then it does not need to be asserted in an **identity_token**.
- **client_authentication_token**

Carries a mechanism-specific GSS initial context token that authenticates the client to the TSS. It contains a mechanism type identifier and the mechanism-specific evidence (that is, the authenticator) required by the TSS to authenticate the client.

When an initial context token contains private credentials, such as a password, this message may be safely sent only after a confidential connection with a trusted TSS has been established. The determination of when it is safe to send a client authentication token in an **EstablishContext** message shall be considered in the context of the CORBA location-binding paradigm for persistent objects (where an invocation may be "location forwarded" by a location daemon to the target object). This issue is considered in Client-Side Requirements and Location Binding on page 161.

When a TSS is unable to validate a security attribute service context, the TSS shall not dispatch on the target object method invocation. The TSS shall reply with a **ContextError** message that carries major and minor codes indicating the reason for the failure.

If an **EstablishContext** message contains an identity token, then it is the responsibility of the TSS to extract a principal identity from the identity token and determine if the identity established in the authentication layer(s) is trusted to assert the extracted identity. If so, the asserted identity is used as the caller identity in the target's authorization determination.

The processing of a request to establish a context that arrives on a one-way call shall be the same as an ordinary call, except that the TSS will not send an indication of the success (**CompleteEstablishContext**) or failure (**ContextError**) of the context validation.

10.2.2.2 ContextError Message Format

A **ContextError** message is sent by a TSS in response to an **EstablishContext** or **MessageInContext** message to indicate to the client that the TSS detected an error. CSS State Machine on page 146 defines the circumstances under which a TSS returns specific error values and exceptions. The **ContextError** message contains the following fields:

- **client_context_id**
  The value of the **client_context_id** that identifies the CSS context in the **EstablishContext** or **MessageInContext** message in response to which the **ContextError** is being returned.
- **major_status**
  The reason the TSS rejected the context.
- **minor_status**
  A more specific error code that further defines the reason for rejection in the context of the major status.
- **error_token**
  A GSS mechanism-specific error token. When an **EstablishContext** message is rejected because it contains a **client_authentication_token** (a GSS initial context token) that is invalidated by the TSS, then depending on the mechanism, the TSS may return a CDR encapsulation of a mechanism-specific GSS error token in this field. Not all GSS mechanisms produce error tokens in response to initial context token validation failures.

In all circumstances where a TSS returns a **ContextError**, the GIOP request that carried the rejected SAS context shall not be dispatched by the target ORB.

10.2.2.3 CompleteEstablishContext Message Format

A **CompleteEstablishContext** message is sent by a TSS in response to an **EstablishContext** message to indicate that the context was established. The **CompleteEstablishContext** message contains the following fields:

- **client_context_id**
  The CSS allocated identifier for the security attribute context. It is returned by the target so that a stateful CSS can link this message to the **EstablishContext** request. A TSS shall always return the value of the **client_context_id** it received in the **EstablishContext** message.
- **context_stateful**
  The value returned by the TSS to indicate whether or not the established context is stateful, and thus reusable. A stateless TSS shall always return false. A stateful TSS shall return true if the established context is reusable. Otherwise a stateful TSS shall return false.
- **final_context_token**
  The GSS mechanism-specific final context token that is returned by a TSS if the client requests mutual authentication. When a TSS accepts an EstablishContext message containing an initial context token that requires mutual authentication, the TSS shall return a mechanism-specific final context token. Not all GSS mechanisms support mutual authentication, and thus not all responses to initial context tokens may include final (or output) context tokens.[5]

  When a **CompleteEstablishContext** message contains a **final_context_token**, the token shall be applied (with **GSS_Init_sec_context**) to the client-side GSS state machine.

Two or more stateful SAS contexts are equivalent if they are established over the same transport layer connection or association, have the same non-zero **client_context_id** and have byte-equivalent identity, authorization, and authentication tokens.

A multithreaded CSS may issue multiple concurrent requests to establish (that is, with an **EstablishContext** message) an equivalent stateful SAS context.

A TSS shall not create a duplicate stateful SAS context in response to a request to establish a context that is equivalent to an existing context.

5. SAS layer authentication capabilities are designed to authenticate client to server where such authentication did not occur in the transport. The SAS protocol is predicated on server-to-client authentication having occurred in the transport layer, and in advance of the request. Server-to-client authentication in service context (which requires that the target return a **final_context_token**) is not the typical use model for SAS layer authentication capabilities.

A TSS shall return an exception containing a **ContextError** service context element if it receives a stateful **EstablishContext** message with a **client_context_id** that matches that of an existing context (established over the same transport layer connection or association) and for which any of the security tokens arriving in the message are not byte-equivalent to those recorded in the existing context. The request shall also be rejected. The exception and error values to be returned are defined in CSS State Machine on page 146.

Table 10.1- CompleteEstablishContext Message Semantics

client_context_id in EstablishContext Message	client_context_id in CompleteEstablishContext Message	context_stateful in CompleteEstablishContext Message	Semantic
0	0	False	Client requested stateless context.
N != 0	N	False	TSS is stateless or TSS did not choose to remember context. In either case, if the client attempts to reuse the context (via MessageInContext) it should expect to receive an error.
		True	Stateful TSS accepted reusable context.

10.2.2.4 MessageInContext Message Format

A **MessageInContext** message is used by a CSS that wishes to reuse an existing context with a request. A CSS may also use this message to release context that it has established with a stateful TSS. The **MessageInContext** message contains the following fields:

- **client_context_id**
  The nonzero context identifier allocated by the client in the **EstablishContext** message used to create the context.
- **discard_context**
  A boolean value that indicates whether the CSS wishes the TSS to discard the context after it processes the request. A value of true indicates that the CSS wishes the context to be discarded, a value of false, indicates that it does not. The purpose of the **discard_context** field is to allow a CSS to help a TSS manage the cleanup of reusable contexts.[6]

Any request message may be used to carry a **MessageInContext** message to a target. A TSS that receives a **MessageInContext** message shall complete the processing of the request before it discards the context (if **discard_context** is set to true).

A TSS may receive a **MessageInContext** message that refers to a context that does not exist at the TSS. This can occur either because the context never existed at the TSS or because it has been discarded by the TSS. In either case, the TSS shall return an exception containing a **ContextError** service context element with major and minor error codes indicating that the referenced context does not exist. The exception and error values to be returned are defined in CSS State Machine on page 146.

The processing of a **MessageInContext** message that arrives on a one-way call shall be the same as for an ordinary call, except that the TSS will not return a **ContextError** when the referenced context does not exist.

6. Stateful clients are under no obligation to manage TSS state, so their use of this message for that purpose is discretionary.

10.2.3 Authorization Token Format

The **authorization_token** field of the **EstablishContext** message of the Security Attribute Service context element is used to carry a sequence (0 or more) of typed representations of authorization data. The **AuthorizationElementType** defines the contents and encoding of the contents of **the_element** field.

The high order 20-bits of each **AuthorizationElementType** constant shall contain the Vendor Minor Codeset ID (VMCID) of the organization that defined the element type. The low order 12 bits shall contain the organization-scoped element type identifier. The high-order 20 bits of all element types defined by the OMG shall contain the VMCID allocated to the OMG (that is, 0x4F4D0).

Organizations must register their VMCIDs with the OMG before using them to define an **AuthorizationElementType**.

```
typedef unsigned long AuthorizationElementType;

typedef sequence <octet> AuthorizationElementContents;

struct AuthorizationElement {
    AuthorizationElementType        the_type;
    AuthorizationElementContents    the_element;
};
typedef sequence <AuthorizationElement> AuthorizationToken;

const AuthorizationElementType X509AttributeCertChain = OMGVMCID | 1;
```

This part of ISO/IEC 19500 has defined one element encoding type, an X509AttributeCertChain. For this type, the field **the_element** contains an octet stream containing an ASN.1 type composed of an X.509 **AttributeCertificate** and a sequence of 0 or more X.509 Certificates. The corresponding ASN.1 definition appears below:

```
VerifyingCertChain ::= SEQUENCE OF Certificate

AttributeCertChain ::= SEQUENCE {
    attributeCert    AttributeCertificate,
    certificateChain VerifyingCertChain,
}
```

The chain of identity certificates may be provided to certify the attribute certificate. Each certificate in the chain shall directly certify the one preceding it. The first certificate in the chain shall certify the attribute certificate. The ASN.1 representation of Certificate shall be as defined in [IETF RFC 2459]. The ASN.1 representation of **AttributeCertificate** shall be as defined in [IETF ID PKIXAC].

10.2.3.1 Extensions of the IETF AC Profile for CSIv2

The **extensions** field of the X.509 Attribute Certificates (AC) provides for the association of additional attributes with the holder or subject of the AC.

Each extension includes an **extnID** (an object identifier), an **extnValue** (an octet string), and a **critical** field (a boolean). The **extnID** identifies the extension, and the **extnValue** contains the value of the instance of the identified extension. The **critical** field indicates whether a certificate-using system shall reject the certificate if it does not recognize the extension. If the **critical** field is set to TRUE and the extension is not recognized (by its **extnID**), then the certificate shall be rejected. A non-critical extension that is not recognized may be ignored.

```
Extensions ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
}
```

[IETF ID PKIXAC] defines a profile for ACs that defines a collection of extensions that may be used in ACs that conform to the profile. An AC that includes any subset of these extensions conforms to the profile. An AC that includes any other critical extension does not conform to the profile. An AC that includes any other non-critical extension conforms to the profile.

The CSIv2 AC profile adds the Proxy Info extension to the collection of extensions defined by the IETF profile. This critical extension may be used to define who may act as proxy for the AC subject. Refer to [IETF ID PKIXAC] for the details of the format and semantics of the Proxy Info extension.

A TSS shall reject a security context that contains an authorization element of type **X509AttributeCertChain** that contains critical extensions or attributes not recognized by the TSS. In this case, the TSS shall return a **ContextError** service context element containing major and minor error codes indicating the evidence is invalid (that is, "Invalid evidence") as defined in ContextError Values and Exceptions on page 149.

10.2.4 Client Authentication Token Format

A CSIv2 client authentication token is a mechanism-specific GSS initial context token. It contains a mechanism type identifier (an object identifier) and the mechanism-specific evidence (that is, the authenticator) required to authenticate the client.

The following ASN.1 basic token definition describes the format of all GSSAPI initial context tokens. The definition of the inner context tokens is mechanism-specific.

```
-- basic Token Format
[APPLICATION 0] IMPLICIT SEQUENCE {
     thisMech MechType
          -- MechType is an Object Identifier
     innerContextToken ANY DEFINED BY thisMech
          -- contents mechanism specific
};
```

The client authentication token has been designed to accommodate the initial context token corresponding to any GSSAPI mechanism. Implementations are free to employ GSSAPI mechanisms other than those required for conformance to CSIv2, such as Kerberos.

The format of the mechanism OID in GSS initial context tokens is defined in [IETF RFC 2743] 3.1, "Mechanism-Independent Token Format," pp. 81-82.

10.2.4.1 Username Password GSS Mechanism (GSSUP)

This part of ISO/IEC 19500 defines a GSSAPI mechanism to support the delivery of authentication secrets above the transport such that they may be applied by a TSS to authenticate clients at shared secret authentication systems.

The GSSUP mechanism assumes that transport layer security, such as that provided by SSL/TLS, will be used to achieve confidentiality and trust in server, such that the contents of the initial context token do not have to be protected against exposures that occur as the result of networking.

The object identifier allocated for the GSSUP mechanism is defined as follows:

{ iso-itu-t (2) international-organization (23) omg (130) security (1) authentication (1) gssup-mechanism (1) }

10.2.4.1.1 GSSUP Initial Context Token

For the GSSUP mechanism, only an inner context token corresponding to the initial context token is defined.

The format of a GSSUP initial context token shall be as defined in [IETF RFC 2743] 3.1, "Mechanism-Independent Token Format," pp. 81-82. This GSSToken shall contain an ASN.1 tag followed by a token length, an authentication mechanism identifier, and a CDR encapsulation containing a GSSUP inner context token as defined by the type **GSSUP::InitialContextToken** in Module GSSUP - Username/Password GSSAPI Token Formats on page 174 (and repeated below).

```
// GSSUP::InitialContextToken

struct InitialContextToken {
    CSI::UTF8String username;
    CSI::UTF8String password;
    CSI::GSS_NT_ExportedName target_name;
};
```

The **target_name** field of the **GSSUP::InitialContextToken** contains the name of the authentication domain in which the client is authenticating. This field aids the TSS in processing the authentication should the TSS support several authentication domains. A CSS shall fill the **target_name** field of the **GSSUP::InitialContextToken** with the contents of the **target_name** field of the **CSIIOP::AS_ContextSec** structure of the chosen CSI mechanism.

The format of the name passed in the **username** field depends on the authentication domain. If the mechanism identifier of the target domain is GSSUP, then the format of the username shall be a Scoped-Username (with **name_value**) as defined in Scoped-Username GSS Name Form on page 136.

10.2.4.1.2 GSSUP Mechanism-Specific Error Token

The GSSUP mechanism-specific error token contains a GSSUP fatal error code.

```
typedef unsigned long ErrorCode;

// GSSUP Mechanism-Specific Error Token
struct ErrorToken {
    ErrorCode error_code;
};
```

The following fatal error codes are defined by the GSSUP mechanism:

```
// The context validator has chosen not to reveal the GSSUP
// specific cause of the failure.
const ErrorCode GSS_UP_S_G_UNSPECIFIED = 1;
```

```
// The user identified in the username field of the
// GSSUP::InitialContextToken is unknown to the target.
const ErrorCode GSS_UP_S_G_NOUSER = 2;

// The password supplied in the GSSUP::InitialContextToken was
// incorrect.
const ErrorCode GSS_UP_S_G_BAD_PASSWORD = 3;

// The target_name supplied in the GSSUP::InitialContextToken does
// not match a target_name in a mechanism definition of the target.
const ErrorCode GSS_UP_S_G_BAD_TARGET = 4;
```

A TSS is under no obligation to return a GSSUP error token; however, returning this token may facilitate the transition of the client-side GSS state machine through error processing. Accordingly, a TSS may indicate that SAS context validation failed in GSSUP client authentication by returning a GSSUP error token in a SAS **ContextError** message. In this case, a TSS that chooses not to reveal specific information as to the cause of the failed GSSUP authentication shall return a status value of **GSS_UP_S_G_UNSPECIFIED**.

10.2.5 Identity Token Format

An identity token is used in an **EstablishContext** message to carry a "spoken for" or asserted identity. The following table lists the five identity token types and defines the type of identity value that may be carried by each of the token types.

In addition to the identity token types described in the following table, the **IdentityTokenType** as defined in Module CSI - Common Secure Interoperability on page 175 provides for the definition of additional CSIv2 identity token types through the default selector of the **IdentityToken** union type. Additional standard identity token types shall only be defined by the OMG. All **IdentityTokenType** constants shall be a power of 2.

Table 10.2 - Identity Token Types

IdentityTokenType (Union Discriminator)	Meaning
ITTAbsent	Identity token is absent; the message conveys no representation of identity assertion.
ITTAnonymous	Identity token is being used to assert a valueless representation of an unauthenticated caller.
ITTPrincipalName	Identity token contains an octet stream containing a GSS mechanism-independent exported name object as defined in [IETF RFC 2743].
ITTDistinguishedName	Identity token contains an octet stream containing an ASN.1 encoding of an X.501 distinguished name.
ITTX509CertChain	Identity token contains an octet stream containing an ASN.1 encoding of a chain of X.509 identity certificates.

Identity tokens of type **ITTX509CertChain** contain an ASN.1 encoding of a sequence of 1 or more X.509 certificates. The asserted identity may be extracted as a distinguished name from the subject field of the first certificate. Subsequent certificates shall directly certify the certificate they follow. The ASN.1 encoding of identity tokens of this type is defined as follows:

```
CertificateChain ::= SEQUENCE SIZE (1..MAX) OF Certificate
```

Interpretation of identity tokens that carry a GSS mechanism-independent exported name object (that is, an identity token type of **ITTPrincipalName**) is dependent on support for GSS mechanism-specific name manipulation functionality.

When a TSS rejects a request because it carries an identity token constructed using an identity type or naming mechanism that is not supported by the target, the TSS shall return a **ContextError** service context element containing major and minor status codes indicating the mechanism was invalid.

Asserting entities may choose to overcome limitations in a target's supported mechanisms by mapping GSS mechanism-specific identities to distinguished names or certificates. The specifics of such mapping mechanisms are outside the scope of this text.

10.2.5.1 GSS Exported Name Object Form for GSSUP Mechanism

The mechanism OID within the exported name object shall be that of the GSSUP mechanism.

{ iso-itu-t (2) international-organization (23) omg (130) security (1) authentication (1) gssup-mechanism (1) }

The name component within the exported name object shall be a contiguous string conforming to the syntax of the scoped-username GSS name form. The encoding of GSS mechanism-independent exported name objects is defined in [IETF RFC 2743].

10.2.5.2 Scoped-Username GSS Name Form

The scoped-username GSS name form is defined as follows, where **name_value** and **name_scope** contain a sequence of 1 or more UTF8 encoded characters.

scoped-username ::= name_value | name_value@name_scope | @name_scope

The '@' character shall be used to delimit **name_value** from **name_scope**. All non-delimiter instances of '@' and all non-quoting instances of '\' shall be quoted with an immediately-preceding '\'. Except for these cases, the quoting character, '\', shall not be emitted within a scoped-username.

The Object Identifier corresponding to the GSS scoped-username name form is:

{ iso-itu-t (2) international-organization (23) omg (130) security (1) naming (2) scoped-username(1) }

The identity token for **ITTPrincipalName**, **ITTDistinguishedName**, **ITTX509CertChain** should contain their respective ASN.1 encodings of the name directly. However, the token may contain a CDR encapsulation of the octet stream that contains the ASN.1 encoding of the name. The TSS shall distinguish the difference by the first octet of the field. The values of 0x00 or 0x01 shall indicate that the field contains a CDR encapsulation. Any other value indicates the field for these identity token types contains the ASN.1 encoded value. For instance, the ASN.1 encoding for **ITTPrincipalName** starts with 0x04, and **ITTDistinguishedName** and **ITTX509CertChain** each start with 0x30. The TSS shall accept both the CDR encapsulation form and the direct ASN.1 encoding for these identity token types.

10.2.6 Principal Names and Distinguished Names

Principal names are carried in **EstablishContext** messages of the SAS protocol, where they may appear in the **identity_token** (the **ITTPrincipalName** discriminated type of an **IdentityTokenType**) or in the **client_authentication_token**, which is a GSS initial context token.

Principal names are also present in the compound mechanisms defined within a **TAG_CSI_SEC_MECH_LIST** tagged component within IORs. The **target_name** field of the **AS_ContextSec** structure may contain a sequence of principal names corresponding to the authentication identities of the target (see struct AS_ContextSec on page 156). A principal name may be used as one variant of the **ServiceSpecificName** form used to identify one of the **privilege_authorities** within the **SAS_ContextSec** structure of a compound mechanism definition within a target IOR (see struct SAS_ContextSec on page 157).

The principal names appearing in initial context tokens are in mechanism-specific; that is, internal form, and may be converted to GSS mechanism-independent exported name object format; that is, an external form by calling a mechanism-specific implementation of **GSS_Export_name**. The inverse translation is performed by a mechanism-specific implementation of **GSS_Import_name**. A mechanism-specific implementation of **GSS_Display_name** allows its caller to convert an internal name representation into a printable form with an associated mechanism type identifier.[7]

The principal names in identity tokens — those in the **target_name** field of **AS_ContextSec** structures and those in the **privilege_authorities** field of **SAS_ContextSec** structures — are in external form (**GSS_NT_ExportedName**), and may be converted to internal form by calling the appropriate mechanism-specific **GSS_import_name** function.

Distinguished names may appear within an identity token, either as an asserted identity or indirectly as the subject distinguished name within an asserted X.509 Identity Certificate. Distinguished names may also be derived from the underlying transport authentication layer if client authentication is done using SSL certificates. Distinguished names may also be used as a form of GeneralName in the GeneralNames variant of the ServiceSpecificName type. The **ServiceSpecificName** type is used to identify **privilege_authorities** within the **SAS_ContextSec** structure of a compound mechanism definition within a target IOR.

10.3 Security Attribute Service Protocol

10.3.1 Compound Mechanisms

The SAS protocol combines common authorization (security attribute) functionality with client authentication functionality and is intended to be used in conjunction with a transport-layer security mechanism, so that there may be as many as three protocol layers of security functionality. This sub clause describes the semantics of the compound security mechanisms that may be realized using this interoperability architecture.

The three protocol layers build on top of each other. The transport layer is at the bottom. The client authentication functionality of the SAS protocol provides a way to layer additional client authentication functionality above the transport layer. The common authorization functionality provides a way to layer security attribute functionality above the authentication layers. Any or all of the layers may be absent.

A target describes in its IORs the CSI compound security mechanisms it supports. Each mechanism defines a combination of layer-specific security functionality supported by the target, as defined in TAG_CSI_SEC_MECH_LIST on page 155.

The mechanisms a client uses to interact with a target shall be compatible with the target's capabilities and sufficient to satisfy its requirements.

7. As defined in IETF RFC 2743 on page 174, "Generic Security Service Application Program Interface Version 2, Update 1", J. Linn, January 2000.

10.3.1.1 Context Validation

A target indicates its requirements for client authentication in its IORs. The layers at which a CSS authenticates to a TSS shall satisfy the requirements established by the target (see the description in Target Security Configuration on page 151). When a CSS attempts to authenticate with a TSS using the client authentication functionality of the SAS context layer protocol (by including a **client_authentication_token** in an **EstablishContext** message), the authentication context established in the TSS will reflect the result of the service context authentication (after having satisfied the target's requirement for transport level authentication, if any).

If the service context authentication fails, the following shall happen:

- The request shall be rejected, whether or not authentication is required by the target.
- An exception containing a **ContextError** service context element shall be returned to the CSS. The **ContextError** service context element shall contain major and minor status codes indicating that client authentication failed.

If the request does not include a **client_authentication_token**, the client authentication identity is derived from the transport layer.

When a request includes an identity token, the TSS shall determine if the identity established as the client authentication identity is trusted to assert the identity represented in the identity token.

A TSS that does not support authorization-token-based delegation (see Conformance Levels on page 162) shall evaluate trust by applying the client authentication identity and the asserted identity to trust rules stored at the target. We call the evaluation of trust based on rules of the target a backward trust evaluation.

When a TSS that supports authorization-token-based delegation receives a request that includes both an identity token and an authorization token with embedded proxy attributes, the TSS shall evaluate trust by determining whether the proxy attributes were established (that is, signed) by a privilege authority acceptable to the target and whether the client authentication identity is included in the identities named in the proxy attributes. We call the evaluation of trust based on rules provided by the caller a forward trust evaluation. A TSS shall not accept requests that failed a forward trust evaluation based on a backward trust evaluation.

A TSS shall determine that a trusted identity established in the authentication layer(s) is trusted to assert exactly the same identity (in terms of identifier value and identification mechanism) in an identity token.

In either case of forward or backward trust evaluation, if trust is established, the context is considered correctly formed. Otherwise, the TSS shall reject the request by returning an exception containing a **ContextError** service context element. The **ContextError** element shall contain major and minor status codes indicating that the evidence was invalid.

If a request includes an authorization token but does not include an identity token, the TSS shall ensure that the access identity named in the authorization token is the same as the client authentication identity. If the request includes an identity token, the TSS shall ensure that the access identity is the same as the identity in the identity token. A TSS that supports authorization-token-based privilege attributes shall reject any request that does not satisfy this constraint and return an exception containing a **ContextError** service context element. The **ContextError** element shall contain major and minor status codes indicating that the evidence was invalid.

When a request includes an authorization token, it is the responsibility of the TSS to determine if the target trusts the authorities that signed the privileges in the token. A TSS that supports authorization-token-based privilege attributes shall reject any request with an authorization token that contains privilege information signed by an authority that is not trusted by the target. In this case, the TSS shall return an exception containing a **ContextError** service context element. The **ContextError** element shall contain major and minor status codes indicating that the evidence was invalid.

10.3.1.2 Legend for Request Principal Interpretations

This sub clause serves as a key to the invocation scenarios represented in Table 10.3 on page 140, Table 10.4 on page 140, and Table 10.5 on page 141. The three tables describe the interpretation of security context information arriving at a target object from a calling object, object 2, that may have been called by another object, object 1. The authentication identity of object 2, as seen by the target object, may have been established in the transport layer, or the SAS context layer, or both. If the authentication identity was established at the transport layer it is referred to as $P2^A$. If the authentication identity was established at the SAS context layer, it is referred to as $P2^B$. The authentication identity seen by object 2 when it is called by another object (that is, object 1) is referred to as P1, the authentication identity of object 1. No distinction is made between the transport and SAS layer authentication identities of object 1 as seen by object 2. Object 1 may also call object 2 anonymously.

P1 is also used to represent a non-anonymous identity that may be asserted by object 2 when it calls the target object. When object 2 calls the target object, it may include an asserted identity in the form of an identity token in its SAS layer context. The asserted identity may be the anonymous identity or, a non-anonymous identity (represented by P1). When object 2 asserts an identity to the target object, it may (or may not) establish proof of its own identity by authenticating at either or both of the transport ($P2^A$), or SAS ($P2^B$) layers. When the target object receives a request made with an asserted identity, the target object will determine if it trusts the client authentication identity (that of object 2, or P2) acting as proxy for the asserted identity (that of object 1, or P1).

When object 2 asserts a non-anonymous identity to the target object, it may include with its request a SAS layer authorization token containing PACs. Each PAC may include an attribute that assigns proxy to a collection of identities that are endorsed by the authority that created the PAC to assert the identity to which the privileges in the PAC apply. When the target object receives a request made with an asserted identity and an authorization token containing proxy rules, the target object will use the proxy rules to determine if it may trust the client authentication identity ($P2^A$ or $P2^B$) as proxy for the asserted identity(P1).

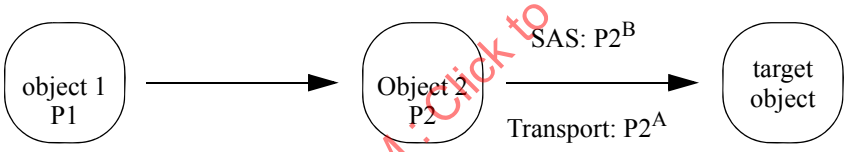


Figure 10.2 - Invocation Scenarios

10.3.1.3 Anonymous Identity Assertion

The anonymous identity is used to represent an unauthenticated entity. To assert an anonymous caller identity, a CSS (perhaps acting as an intermediate) shall include a SAS context element containing an **EstablishContext** message with an **identity_token** containing the anonymous **IdentityTokenType** in its request.

10.3.1.4 Presumed Trust

Presumed trust is a special case of the evaluation of identity assertions by a TSS. In presumed trust, a TSS accepts identity assertions based on the fact of their occurrence and without consideration of the authentication identity of the asserting entity. The presumption is that communications are constrained such that only trusted entities are capable of asserting an identity to the TSS.

10.3.1.5 Failed Trust Evaluations

Table 10.3 shows the circumstances under which the interpretation of caller credentials by a TSS results in a failed trust evaluation. None of these circumstances correspond to presumed trust, where trust evaluations are not performed (and therefore cannot fail).

Table 10.3 - Conditions under which Trust Evaluation Fails

Transport Client Principal	SAS Client Authentication Principal	SAS Identity Token Identity	Does Target Trust P2, or Is P2 Named as Proxy in Authorization Elements?
None	None	P1	Not Applicable
None	$P2^B$	P1	No (with respect to $P2^{B)}$
$P2^A$	None	P1	No (with respect to $P2^{A)}$
$P2^A$	$P2^B$	P1	No (with respect to $P2^B$)

A failed trust evaluation shall result in the request being rejected with an indication that client authentication failed.

10.3.1.6 Request Principal Interpretations

The entries in Table 10.4 describe the interpretation of client credentials by a TSS after an incoming call has satisfied the target's security requirements and has been validated by the TSS.

Table 10.4 - TSS Interpretation of Client Credentials After Validation

Transport Client Principal	SAS Client Authentication Principal	SAS Identity Token Identity	Client Principal is Trusted	Invocation Principal	Scenario
None	None	Absent	Not applicable	Anonymous	Unauthenticated
None	$P2^B$	Absent	Not applicable	P2	Client authentication
$P2^A$	None	Absent	Not applicable	P2	Client authentication
$P2^A$	$P2^B$ (by rule 1[a])	Absent	Not applicable	$P2^B$	Client authentication
None	None	P1	Yes if rule 2[b]	P1	identity assertion
None	$P2^B$	P1	Yes if rule 2 or rule 3[c]	P1	identity assertion
$P2^A$	None	P1	Yes if rule 2 or rule 3	P1	identity assertion
$P2^A$	$P2^B$ (by rule 1)	P1	Yes if rule 2 or rule 3	P1	identity assertion
None	None	Anonymous	Yes if rule 4[d]	Anonymous	assertion of anonymous
None	$P2^B$	Anonymous	Yes if rule 4	Anonymous	assertion of anonymous
$P2^A$	None	Anonymous	Yes if rule 4	Anonymous	assertion of anonymous
$P2^A$	$P2^B$ (by rule 1)	Anonymous	Yes if rule 4	Anonymous	assertion of anonymous
none	No SAS Message		Not Applicable	Anonymous	Unauthenticated
P2	No SAS Message		Not Applicable	P2	Client authentication

a. Rule 1: TSS trusts $P2^A$ to use authenticator for $P2^B$ is implied by $P2^B$ having been authenticated.

b. Rule 2: TSS presumes trust in transport to accept None, $P2^A$, or $P2^B$ speaking for P1.

c. Rule 3: TSS trusts $P2^A$, or $P2^B$ to speak for P1.

d. Rule 4: TSS trusts None, $P2^A$, or $P2^B$ to speak for Anonymous. A TSS shall support the configuration of rule 4, such that Anonymous identity assertions are accepted independent of authentication of the asserter.

The entries in Table 10.5 describe additional TSS interpretation rules to support delegation. These rules have been separated from those in Table 10.4 on page 140, because they describe functionality required of implementations that conform to a higher level of secure interoperability as defined in Conformance Level 2 on page 163. The entries in Table 10.5 correspond to invocations that carry an identity token and an authorization token with embedded delegation token (that is, a proxy endorsement attribute) in an EstablishContext service context element. Invocations that do not carry all of these tokens are represented in Table 10.4.

An authorization token may contain authorization elements that contain proxy statements, which endorse principals to proxy for other entities. Table 10.5 describes delegation scenarios in which endorsements from the issuer of the authorization element authorize the authenticated identity, which is $P2^A$ or $P2^B$, to proxy for the asserted identity. In this table, the column "Proxies Named in Authorization Element" defines the identities who are endorsed by the authorization element to proxy for P1, the asserted identity and the subject of the authorization element. The value "Any" indicates that the authorization element contains a blanket endorsement, such that as far as its issuer is concerned, any identity may proxy for P1. The outcomes described in Table 10.5 assume that the TSS trusts the issuer of the authorization element to endorse principals to proxy for others.

Table 10.5 - Additional TSS Rules to Support Delegation

Transport Client Principal	SAS Client Authentication Principal	SAS Identity Token Identity	Proxies Named in Authorization Element	Invocation Principal	Scenario
None	$P2^B$	P1	Any	P1	Delegation
$P2^A$	None	P1	Any	P1	Delegation
$P2^A$	$P2^B$	P1	Any	P1	Delegation
None	$P2^B$	P1	Restricted to set including $P2^B$	P1	Restricted delegation
$P2^A$	None	P1	Restricted to set including $P2^A$	P1	Restricted delegation
$P2^A$	$P2^B$	P1	Restricted to set including $P2^B$	P1	Restricted delegation

10.3.2 Session Semantics

This sub clause describes the negotiation of security contexts between a CSS and a TSS. A TSS is said to be stateless if it does not operate in the mode of accepting reusable (that is, stateful) security contexts. A TSS that accepts reusable security contexts is said to be stateful. A CSS is said to be stateless if it operates in the mode of establishing transient, non-reusable (that is, stateless) security contexts. A CSS that issues requests to establish reusable security contexts is said to be stateful.

10.3.2.1 Negotiation of Statefulness

A client initiates a stateless interaction by specifying a **client_context_id** of 0. A client issues a request to establish a stateful context by including a nonzero **client_context_id** in an **EstablishContext** message.

When a stateless TSS receives a request to establish a stateful session, the TSS shall attempt to validate the security tokens bound to the request. If the validation fails, an exception containing an appropriate **ContextError** service context element shall be returned to the client. If the validation succeeds, the TSS shall negotiate to stateless by responding with a **CompleteEstablishContext** message with **context_stateful** set to false.

A client that initiates a stateful interaction shall be capable of accepting that the target negotiated the context to stateless.

10.3.2.2 Stateful/Reusable Contexts

Each transport layer session defines a context identifier number *scope*. The CSS selects context identifiers for use within a scope.

A CSS may use the **EstablishContext** message to issue multiple concurrent requests to establish a stateful security context within a scope.

To avoid duplicate sessions, when the stateful **EstablishContext** requests sent within a scope carry equivalent security contexts, the CSS shall assign to them the same nonzero **client_context_id**.

Within a scope, a TSS shall reject any request to establish a stateful context that carries a different security context from an established context with the same **client_context_id**. In this case, an exception containing a **ContextError** service context element shall be returned to the caller.

Two security contexts are equivalent if all of the authentication, identity, and authorization tokens match both in existence and in value. Token values shall be evaluated for equivalence by comparing the corresponding byte sequences used to carry the tokens in **EstablishContext** messages.

When a target that supports stateful contexts receives a request to establish a stateful context, the TSS shall attempt to validate the security tokens in the **EstablishContext** element. If the validation succeeds, the request shall be accepted, and the reply (if there is one) shall carry a **CompleteEstablishContext** element that indicates (that is, **context_stateful** = true) that the context is available at the TSS for the caller's reuse. If the validation fails, an exception containing an appropriate **ContextError** service context element shall be returned to the caller.

A TSS that accepts stateful contexts shall bear the responsibility for managing the lifecycle of these sessions. Clients that reuse stateful contexts shall capable of processing replies that indicate that an established stateful context has been unilaterally discarded by the TSS.

A TSS shall not establish a stateful context in response to a request to establish a stateless context (that is, one with a **client_context_id** of zero).

A TSS that supports stateful contexts may negotiate a request to establish a stateful context to a stateless context in order to preserve resources. It may do so only if it does not already have an established matching stateful context.

Conversely, a stateful TSS that has negotiated a request to stateless may respond statefully to a subsequent context with the same (non-zero) **client_context_id**.

10.3.2.2.1 Relationship to Transport-Layer

A SAS context shall not persist beyond the lifetime of the transport-layer secure association over which it was established.

Stateful SAS contexts are not compatible with transports that do not make the relationship between the connection and the association transparent.

10.3.3 TSS State Machine

The TSS state machine is defined in the state diagram, Figure 10.3 on page 144 and in the TSS state table, Table 10.6 on page 145. Each TSS call thread shall operate independently with respect to this state machine. Where necessary, thread synchronization at shared state shall be handled in the actions called by this state machine.

An ORB must not invoke the TSS state machine if the target object does not exist at the ORB. The TSS state machine has no capacity to reject or forward[8] a request because the target object does not exist, and must rely on the ORB to only invoke the TSS when the target object exists at the ORB.

In response to a one-way call, a TSS shall not perform any of the send actions described by the state machine.

The shaded rows in Table 10.6 on page 145 indicate transitions and states that do not exist in a stateless implementation of the SAS protocol.

The state names, function names, and function signatures that appear in the state diagram and the state table are not prescriptive.

8. A TSS uses the LOCATION_FORWARD status to return an IOR containing up-to-date security mechanism configuration for an existing object.

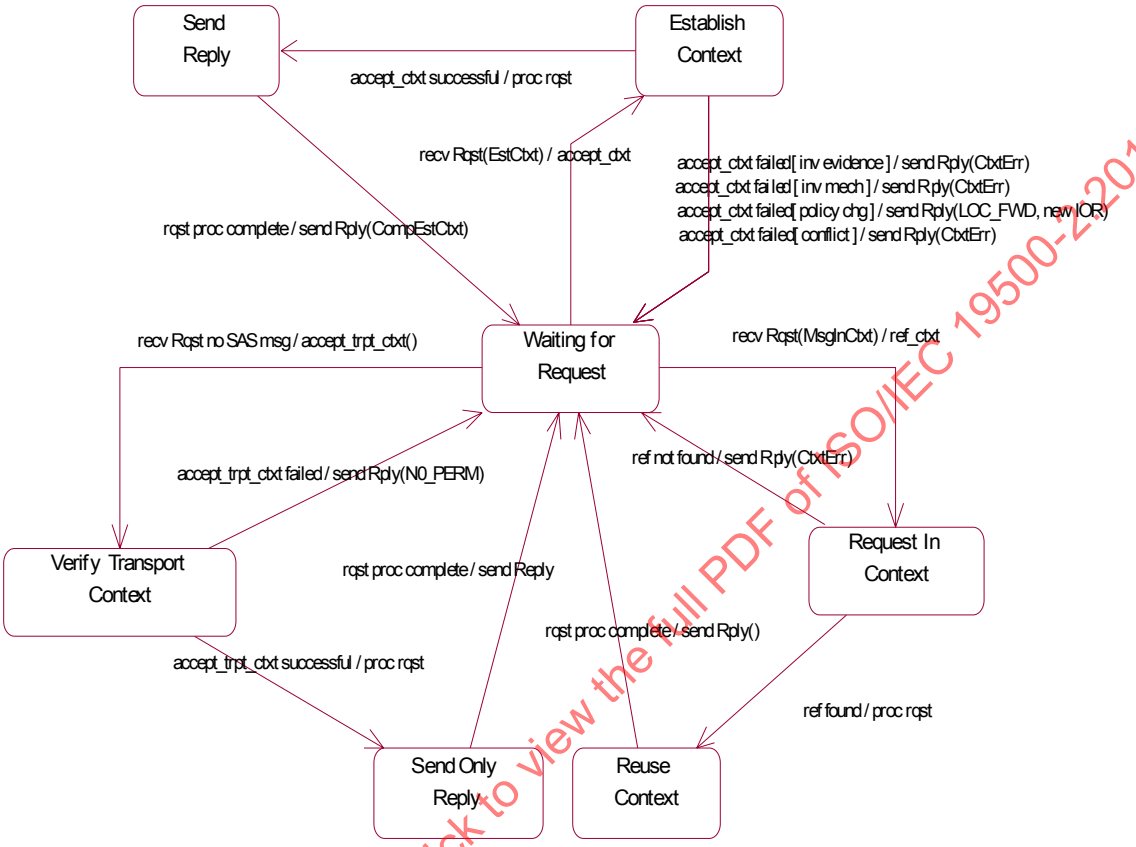


Figure 10.3 - TSS State Machine

Table 10.6 -TSS State Table

	State	Event	Action	New State
1	Waiting for Request	receive request without SAS message	accept_transport_context()	Verify Transport Context
		receive Request + EstablishContext {client_context_id = N, tokens}	accept_context(tokens, N, Out stateful)	Establish Context
		receive Request + MessageInContext {client_context_id = N, discard_context = D}	reference_context(N)	Request In Context
2	Verify Transport Context	accept_transport_context() returned success	process request	Send Only Reply
		accept_transport_context() returned failure	send exception (NO_PERMISSION)	Waiting for Request
3	Send Only Reply	request processing completed	send Reply	Waiting for Request
4	Send Reply	request processing completed	send Reply + CompleteEstablishContext { N, stateful}	Waiting For Request
5	Establish Context	accept_context (tokens, N, Out stateful) returned success	process request	Send Reply
		accept_context (tokens, N, Out stateful) returned failure (invalid evidence)	send exception + ContextError (invalid evidence)	Waiting for Request
		accept_context (tokens, N, Out stateful) returned failure (invalid mechanism)	send exception + ContextError (invalid mechanism)	Waiting for Request
		accept_context (tokens, N, Out stateful) returned failure (policy change)	send Reply + LOCATION_FORWARD status + updated IOR	Waiting for Request
		accept_context (tokens, N, Out stateful) returned failure (conflicting evidence)	send exception + ContextError (conflicting evidence)	Waiting for Request
6	Request in Context	reference_context(N) returned reference	process request	Reuse Context
		reference_context(N) returned empty reference	send exception + ContextError (context does not exist)	Waiting for Request
7	Reuse Context	request processing completed	send Reply if (D) discard_context(N)	Waiting for Request

10.3.3.1 TSS State Machine Actions

This sub clause defines the intended semantics of the actions appearing in the TSS state machine. As noted above, the function names and function signatures are not prescriptive.

- **accept_context** (tokens, N, Out stateful)

  This action validates the security context captured in the tokens including ensuring that they are compatible with the mechanisms supported by the target object. If a context is not validated, **accept_context** returns error codes that describe the reason the context was rejected.

  When called by a stateless TSS, **accept_context** always returns false in the output argument "**stateful**." When called by a stateful TSS, **accept_context** may (depending on the effective policy of the target object) attempt to record state corresponding to the context. If state for the identified context already exists and the

received tokens are not equivalent to those captured in the existing context, **accept_context** shall reject the context. If the context state either already existed, or was recorded, **accept_context** returns true in the output argument "**stateful**." An implementation of **accept_context** shall implement the error semantics defined in the following table.

Table 10.7- Accept Context Error Semantics

Semantic	Returned Error Code
Tokens match mechanism definition of target object but could not be validated.	Invalid evidence
Context has non-zero client_context_id that matches that of an exiting context but tokens are not equivalent to those used to establish the existing context.	Conflicting evidence
The mechanism configuration of the target object has changed and request indicates that CSS is not aware of the current mechanism configuration.	Policy change
The mechanism configuration of the target object has not changed, and request is not consistent with target mechanism configuration.	Invalid mechanism

When **accept_context** returns any of **Invalid evidence, Conflicting evidence,** or **Invalid mechanism** the TSS shall reject the request and send a NO_PERMISSION exception containing a **ContextError** service context element with error codes as defined in Table 10.9 on page 150. When **accept_context** returns **Policy change**, the TSS action shall reject the request and return a reply with status **LOCATION_FORWARD** and containing a new IOR for the target object that contains an up-to-date representation of the target's security mechanism configuration.

- **accept_transport_context()**
  This action validates that a request that arrives without a SAS protocol message; that is, **EstablishContext** or **MessageInContext** satisfies the CSIv2 security requirements of the target object. This routine returns true if the transport layer security context (including none) over which the request was delivered satisfies the security requirements of the target object. Otherwise, **accept_transport_context** returns false. When **accept_transport_context** returns false, the TSS shall reject the request and send a NO_PERMISSION exception.
- **reference_context (N)**
  If there is an existing context with **client_context_id = N**, **reference_context** returns a reference to it. Otherwise, **reference_context** returns an empty reference.
- **discard_context (N)**
  If context **N** exists and it is not needed to complete the processing of another thread, **discard_context** causes the context to be deleted.

10.3.4 CSS State Machine

A proposed implementation of the CSS state machine is defined in the state diagram, Figure 10.4 on page 147, and in the CSS state table, Table 10.8 on page 148. Each CSS call thread shall operate independently with respect to this state machine. Where necessary, thread synchronization at shared state shall be handled in the actions called by this state machine.

When a CSS processes a one-way call, it returns to the caller and sets its next state to done, as no response will be sent by the TSS. The shaded rows in the state table indicate transitions and states that need not exist in a stateless CSS client side implementation.

The state names, function names, and function signatures that appear in the state diagram and state table are not prescriptive.

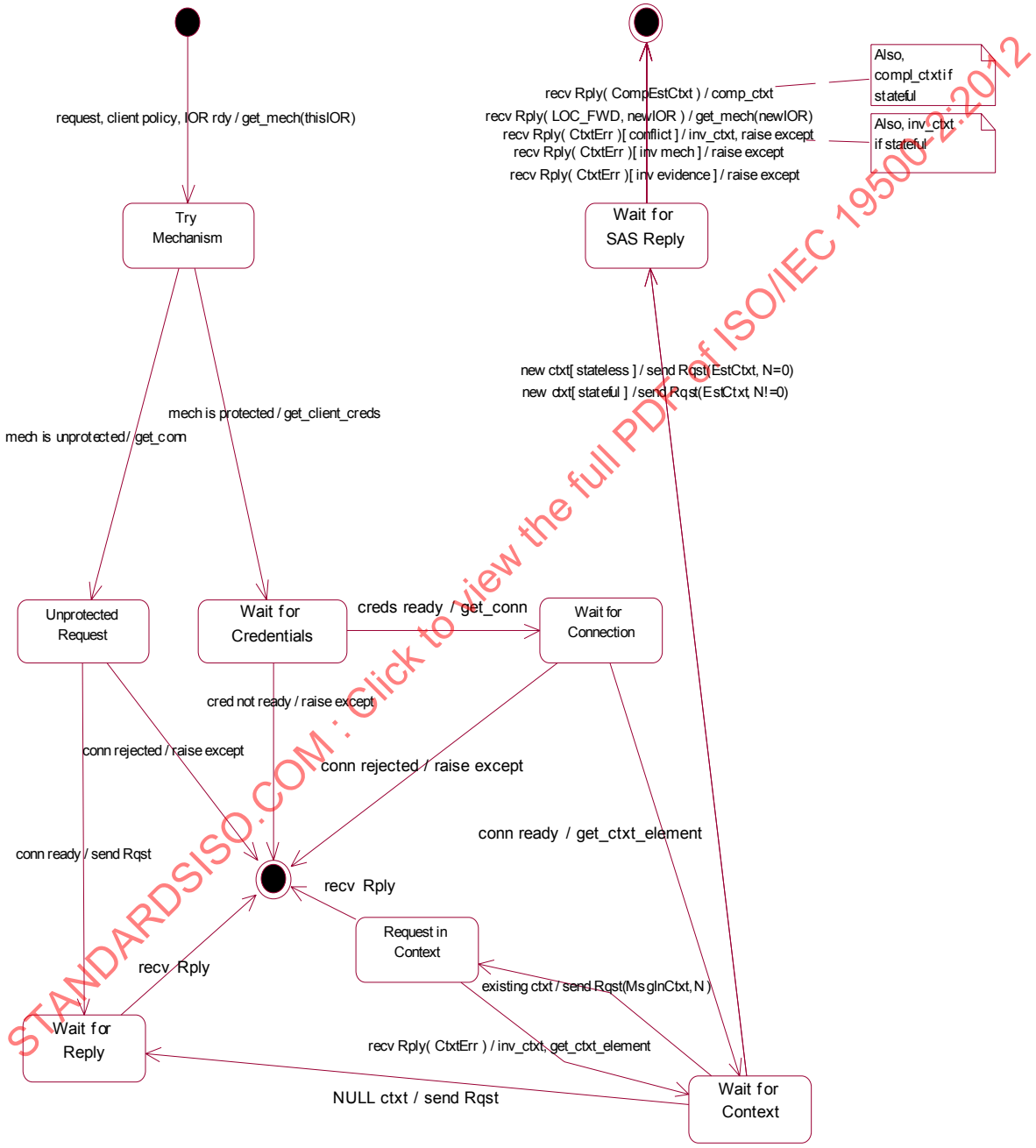


Figure 10.4 - CSS State Machine

Table 10.8 - CSS State Table

	State	Event	Action	New State
1	start	Request + client policy + IOR ready to send	get_mechanism (policy, thisIOR, Out mech)	Try Mechanism
2	Try Mechanism	the selected mechanism is unprotected	get_connection (mech, Out c)	Unprotected Request
		the selected mechanism is protected	get_client_creds (policy, mech, Out creds)	Wait for Credentials
3	Unprotected Request	connection ready	send request	Wait for Reply
		connection rejected	raise exception and return to caller[a]	done
4	Wait for Reply	receive reply	return to caller	done
5	Wait for Credentials	client credentials ready	get_connection (policy, mech, creds, Out c)	Wait for Connection
		necessary credentials not obtained	raise exception and return to caller[b]	done
6	Wait for Connection	connection ready	get_context_element (c, policy, creds, mech, Out element)	Wait for Context
		connection rejected	raise exception and return to caller[c]	done
7	Wait for Context	get_context_element returned EstablishContext {N = 0, tokens}	send Request + EstablishContext {client_context_id = N = 0, tokens}	Wait for SAS Reply
		get_context_element returned EstablishContext {N != 0, tokens}	send Request + EstablishContext {client_context_id = N != 0, tokens}	Wait for SAS Reply
		get_context_element returned NULL	send request	Wait for Reply
		get_context_element returned MessageInContext {N != 0, D}	send Request + MessageInContext {client_context_id = N != 0, D}	Request In Context
8	Wait for SAS Reply	receive exception + ContextError (invalid evidence)	raise exception and return to caller[d]	done
		receive exception + ContextError (invalid mechanism)	raise exception and return to caller	done
		receive exception + ContextError (conflicting evidence)	invalidate_context (c, N)	done
			raise exception and return to caller	
		receive Reply + LOCATION_FORWARD status + updated IOR	return to caller	done
		receive Reply + CompleteEstablishContext {N, context_stateful}	complete_context (c, N, context_stateful)	done
			return to caller	
9	Request in Context	receive exception + ContextError (context does not exist)	invalidate_context (c, N) get_context_element (c, policy, creds, mech, Out element)	Wait for Context
		receive Reply	return to caller	done

a. A CSS may do next mechanism processing, in which case it might call get_next_mechanism(policy,thisIOR) and transition to state Try Mechanism.
b. Same note as 1.
c. Same note as 1.
d. A CSS may re-collect authentication evidence and try again, in which case it might call get_client_creds(policy, mech, Out creds) and transition to state Wait for Credentials.

10.3.4.1 CSS State Machine Actions

This sub clause defines the intended semantics of the actions appearing in the CSS state machine. As noted above the function names and function signatures are not prescriptive. The descriptions appearing in the following sub clauses are provided to facilitate understanding of the proposed implementation of the CSS state machine.

- **get_mechanism** (policy, IOR, Out mech)
  Select from the **IOR** a **mechanism** definition that satisfies the client policy.
- **get_client_creds** (policy, mech, Out creds)
  Get the client **credentials** as necessary to satisfy the client **policy** and the target policy in the **mechanism**.
- **get_connection** (mech, Out c)
  Open a connection based on the port information in the **mechanism** argument.
- **get_connection** (policy, mech, creds, Out c)
  Open a secure connection based on the client **policy**, the target policy in the **mechanism** argument, and using the client credentials in the **creds** argument.
- **get_context_element** (c, policy, creds, mech, Out element)
  In the scope of connection **c**, use the client **creds** to create a SAS protocol context element that satisfies the client **policy** and the target policy in the **mechanism**. If the CSS supports reusable contexts, and the client policy is to establish a reusable context, the CSS allocates a **client_context_id**, and initializes a context element in the context table of the connection. A NULL context element may be returned by **get_context_element** when the target mechanism definition either does not support or require SAS layer security functionality, and the client establishes a policy not to use such functionality unless required to do so.
- **invalidate_context** (c, N)
  Mark context **N** in connection scope **c** as invalid such that no more requests may (re)use it.
- **complete_context** (c, N, context_stateful)
  This action applies the contents of a returned **CompleteEstablishContext** message to context **N**, in connection scope **c,** to change its state to completed. In a stateful CSS, **get_context_element** will not return a **MessageInContext** element until **complete_context** is called with **context_stateful** true.

10.3.5 ContextError Values and Exceptions

Table 10.9 defines the circumstances under which error values and exceptions shall be returned by a TSS. The state and event columns contain states and events appearing in Table 10.6.

Table 10.9- ContextError Codes and Exceptions

State	Event	Semantic	Major	Minor	Exception
Establish Context	accept_context returned failure	Invalid evidence	1	1	NO_PERMISSION
		Invalid mechanism	2	1	NO_PERMISSION
		Conflicting evidence	3	1	NO_PERMISSION
Request In Context	reference_context (N) returned false	No Context	4	1	NO_PERMISSION

10.4 Transport Security Mechanisms

10.4.1 Transport Layer Interoperability

The secure interoperability architecture that is defined by this part of ISO/IEC 19500 partitions secure interoperability into three layers: the transport layer, authentication above the transport layer, and the secure attribute layer. This text defines secure interoperability that uses transport-layer security for message protection and authentication of the target to the client.

10.4.2 Transport Mechanism Configuration

The configuration of transport-layer security mechanisms is specified in IORs. Support for CSI is indicated within an IOR profile by the presence of at most one **TAG_CSI_SEC_MECH_LIST** tagged component that defines the mechanism configuration pertaining to the profile. This component contains a list of one or more **CompoundSecMech** structures, each of which defines the layer-specific security mechanisms that comprise a compound mechanism that is supported by the target. This part of ISO/IEC 19500 does not define support for CSI mechanisms in multiple-component IOR profiles.

Each **CompoundSecMech** structure contains a **transport_mech** field that defines the transport-layer security mechanism of the compound mechanism. A compound mechanism that does not implement security functionality at the transport layer shall contain the **TAG_NULL_TAG** component in its **transport_mech** field. Otherwise, the **transport_mech** field shall contain a tagged component that defines a transport protocol and its configuration. TAG_TLS_SEC_TRANS on page 153 and TAG_SECIOP_SEC_TRANS on page 155 define valid transport-layer components that can be used in the **transport_mech** field.

10.4.2.1 Recommended SSL/TLS Ciphersuites

This part of ISO/IEC 19500 recommends that implementations support the following ciphersuites in addition to the mandatory ciphersuites identified in [IETF RFC 2246]. Of these additional ciphersuites, those which use weak encryption keys are only recommended for use in environments where strong encryption of SAS protocol elements (including GSSUP authenticators) and request arguments is not required. Some of the recommended ciphersuites are known to be encumbered by licensing constraints.

- TLS_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_MD5
- TLS_DHE_DSS_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC4_40_MD5

- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

10.5 Interoperable Object References

10.5.1 Target Security Configuration

A target that supports unprotected IIOP invocations shall specify in the corresponding **TAG_INTERNET_IOP** profile a nonzero port number at which the target will accept unprotected invocations.[9] A target that supports only protected IIOP invocations shall specify a port number of 0 (zero) in the corresponding **TAG_INTERNET_IOP** profile. A target may support both protected and unprotected IIOP invocations at the same port, but it is not required to do so.

```
struct IOR {
    string type_id;
    sequence <TaggedProfile> profiles = {
        ProfileId tag = TAG_INTERNET_IOP;
        struct ProfileBody_1_1 profile_data = {
            Version iiop_version;
            string host;
            unsigned short port;
            sequence <octet> object_key;
            sequence <IOP::TaggedComponent> components;
        };
    };
};
```

A target that supports protected invocations shall describe in a **CompoundSecMech** structure the characteristics of each of the alternative compound security mechanisms that it supports. The **CompoundSecMech** structure shall be included in a list of such structures in the body of a **TAG_CSI_SEC_MECH_LIST** tagged component.

```
sequence <IOP::TaggedComponent> components = {
    IOP::TaggedComponent {
        ComponentId tag = TAG_CSI_SEC_MECH_LIST;
        sequence <octet> component_data = {
            CSIIOP::CompoundSecMechList = {
                boolean stateful;
                CompoundSecMechanisms mechanism_list = {
                    CompoundSecMech;
                };
            };
        };
    };
};
```

9. The OMG has registered port numbers for IIOP (683) and IIOP/SSL (684) with IANA. Although the existence of these reservations does not prescribe their use, it may be useful to recognize these port numbers as defaults for the corresponding protocols.

The order of occurrence of the alternative compound mechanism definitions in a **TAG_CSI_SEC_MECH_LIST** component indicates the target's mechanism preference. The target prefers mechanism definitions occurring earlier in the list. An IOR profile shall contain at most one **TAG_CSI_SEC_MECH_LIST** tagged component. An IOR profile that contains multiple **TAG_CSI_SEC_MECH_LIST** tagged components is malformed and should be rejected by a client implementation.

10.5.1.1 AssociationOptions Type

The **AssociationOptions** type is an unsigned short bit mask containing the logical OR of the configured options. The properties of security mechanisms are defined in an IOR in terms of the association options supported and required by the target. A CSS shall be able to interpret the association options defined in Table 10.10.

Table 10.10 - Association Options

Association Option	target_supports	target_requires
Integrity	Target supports integrity protected messages	Target requires integrity protected messages .
Confidentiality	Target supports privacy protected messages	Target requires privacy protected messages.
EstablishTrustInTarget	Target can authenticate to a client	Not applicable. This bit should never be set, and should be ignored by CSS.
EstablishTrustInClient	Target can authenticate a client	Target requires client authentication.
IdentityAssertion	Target accepts asserted caller identities based on trust in the authentication identity of the asserting entity. Target can evaluate trust based on trust rules of the target. If DelegationByClient is set, target can also evaluate trust when provided with a delegation token (that is, a proxy attribute contained in an authorization token).[a]	Not applicable. This bit should never be set, and should be ignored by CSS.
DelegationByClient	When it occurs in conjunction with support for IdentityAssertion, this bit indicates that target can evaluate trust in an asserting entity based on a delegation token.[b]	Target requires that CSS provide a delegation token that endorses the target as proxy for the client.[c]

a. A target policy that accepts only identity assertions based on forward trust cannot be communicated in an IOR (although it can be enforced).
b. If an incoming request includes an identity token and a delegation token, the request shall be rejected if the delegation token does not endorse the asserting entity (see Section 10.3.1.1, Context Validation, on page 138)
c. A target with DelegationByClient set in **target_requires** shall also have this bit set in **target_supports**. As noted in the table, this has an impact on the target' s identity assertion policy (if any).

The representation of supported options is used by a client to determine if a mechanism is capable of supporting the client's security requirements. The supported association options shall be a superset of those required by the target.

When the **IdentityAssertion** bit is set in **target_supports**, it indicates that the target accepts asserted caller identities based on trust in the authentication identity of the asserting entity. When the **DelegationByClient** bit is not set, the target will evaluate trust based on rules of the target (that is, a backward trust evaluation). When the IdentityAssertion and **DelegationByClient** bits are set, they indicate that the target is also capable of evaluating trust in an asserting entity based on trust rules delivered in an authorization token (that is, a forward trust evaluation). A target that can perform a forward trust evaluation does so when trust rules are delivered in an authorization token. Otherwise a backward trust evaluation is performed.

When the **DelegationByClient** bit is set in **target_requires**, it indicates that the target requires a delegation token to complete the processing of a request. Such circumstances will occur when a target, acting as an intermediate, attempts to issue a request as its caller and sanctioned by the delegation token delivered by its caller.

The rules for interpreting asserted identities in the presence or absence of a delegation token (that is, a proxy attribute contained in an authorization token) are as defined in Context Validation on page 138.

The security mechanism configuration in an IOR being used by a CSS may (as the result of target policy administration) no longer represent the actual security mechanism configuration of the target object.

10.5.1.1.1 Alternative Transport Association Options

Implementations that choose to employ the service context protocol defined in this part of ISO/IEC 19500 to achieve interoperability over an alternative secure transport (one other than SSL/TLS) may also be required to support the message protection options defined in Table 10.11.

Table 10.11 - Alternative Transport Association Options

Association Option	target_supports	target_requires
DetectReplay	Target can detect replay of requests (and request fragments).	Target requires security associations to detect replay.
DetectMisordering	Target can detect sequence errors of request (and request fragments).	Target requires security associations to detect message sequence errors.

10.5.1.2 Transport Address

The TransportAddress structure indicates an INTERNET address where the TSS is listening for connection requests.

```
struct TransportAddress {
    string host_name;
    unsigned short port;
};

typedef sequence <TransportAddress> TransportAddressList;
```

The **host_name** field identifies the Internet host to which connection requests will be made. The **host_name** field shall not contain an empty string. The **host_name** field shall contain a host name or an IP address in standard numerical address (e.g., dotted-decimal) form.

The **port** field contains the TCP/IP port number (at the specified host) where the TSS is listening for connection requests. The port number shall not be zero.

10.5.1.3 TAG_TLS_SEC_TRANS

An instance of the **TAG_TLS_SEC_TRANS** component may occur in the **transport_mech** field within a **CompoundSecMech** structure in a **TAG_CSI_SEC_MECH_LIST** component.

When an instance of the **TAG_TLS_SEC_TRANS** component occurs in the **transport_mech** field of the **CompoundSecMech** structure, it defines the sequence of transport addresses at which the target will be listening for SSL/TLS protected invocations. The supported (**target_supports**) and required (**target_requires**) association options defined in the component shall define the transport level security characteristics of the target at the given addresses.

```
const IOP::ComponentId TAG_TLS_SEC_TRANS = 36;

struct TLS_SEC_TRANS {
    AssociationOptions target_supports;
    AssociationOptions target_requires;
    TransportAddressList addresses;
};
```

The **addresses** field provides a shorthand for defining multiple security mechanisms that differ only in their transport addresses. The **addresses** field shall contain at least one address.

Table 10.12, Table 10.13, Table 10.14, and Table 10.15 describe the association option semantics relating to the **TAG_TLS_SEC_TRANS** tagged component that shall be interpreted by a CSS and enforced by a TSS. The **IdentityAssertion** and **DelegationByClient** association options shall not occur in an instance of this component.

Table 10.12 - Integrity Semantics

Integrity	Semantic
Not supported	None of the ciphersuites supported by the target designate a MAC algorithm.
Supported	Target supports one or more ciphersuites that designate a MAC algorithm.
Required	All the ciphersuites supported by the target designate a MAC algorithm.

Table 10.13 - Confidentiality Semantics

Confidentiality	Semantic
Not supported	None of the ciphersuites supported by the target designate a bulk encryption algorithm[a].
Supported	Target supports one or more ciphersuites that designate a bulk encryption algorithm.
Required	All the ciphersuites supported by the target designate a bulk encryption algorithm.

a. Bulk encryption algorithms include both block and stream ciphers.

Table 10.14 - EstablishTrustInTarget Semantics

EstablishTrustInTarget	Semantic
Not supported	None of the ciphersuites supported by the target designate a key exchange algorithm that will authenticate the target to the client.
Supported	Target supports one or more ciphersuites that designate a key exchange algorithm that will authenticate the target to the client.
Required	Not applicable. This bit should never be set, and should be ignored by CSS.

Table 10.15 - EstablishTrustInClient Semantics

EstablishTrustInClient	Semantic
Not supported	Target does not support client authentication during the handshake. Moreover, target provides no opportunity for client to authenticate in the handshake (that is, target does not send certificate request message).
Supported	Target provides client with an opportunity to authenticate in handshake. Target will accept connection if client does not authenticate.
Required	Target accepts connections only from clients who successfully authenticate in the handshake.

10.5.1.4 TAG_SECIOP_SEC_TRANS

A tagged component with the **TAG_SECIOP_SEC_TRANS** tag is a valid component for the **transport_mech** field of the **CompoundSecMech** structure. The presence of this component indicates the generic use of the SECIOP protocol as a secure transport underneath the CSI mechanisms. A component tagged with this value shall contain the CDR encoding of the **SECIOP_SEC_TRANS** structure.

The **SECIOP_SEC_TRANS** structure defines the transport addresses for SECIOP messages, the association options pertaining to the particular GSS mechanism being supported, the GSS mechanism identifier, and the target's GSS exported name.

```
const IOP::ComponentId TAG_SECIOP_SEC_TRANS = 35;

struct SECIOP_SEC_TRANS {
    AssociationOptions target_supports;
    AssociationOptions target_requires;
    CSI::OID mech_oid;
    CSI::GSS_NT_ExportedName target_name;
    TransportAddressList addresses;
};
```

The **addresses** field provides a shorthand for defining multiple security mechanisms that differ only in their transport addresses. The **addresses** field shall contain at least one address.

Table 10.12, Table 10.13, Table 10.14, and Table 10.15 also describe the association option semantics relating to the **TAG_SECIOP_SEC_TRANS** tagged component that shall be interpreted by a CSS and enforced by a TSS.

10.5.1.5 TAG_CSI_SEC_MECH_LIST

This new tagged component, **TAG_CSI_SEC_MECH_LIST**, is used to describe support in the target for a sequence of one or more compound security mechanisms represented in the **mechanism_list** field of a **CompoundSecMechList** structure. The mechanism descriptions in the **mechanism_list** occur in decreasing order of target preference.

```
const IOP::ComponentId TAG_CSI_SEC_MECH_LIST = 33;

struct CompoundSecMech {
    AssociationOptions target_requires;
    IOP::TaggedComponent transport_mech;
    AS_ContextSec as_context_mech;
    SAS_ContextSec sas_context_mech;
```

```
};

typedef sequence <CompoundSecMech> CompoundSecMechanisms;

struct CompoundSecMechList {
    boolean stateful;
    CompoundSecMechanisms mechanism_list;
};
```

The **CompoundSecMech** structure is used to describe support in the target for a compound security mechanism that may include security functionality that is realized in the transport and/or security functionality realized above the transport in service context. Where a compound security mechanism implements security functionality in the transport layer, the transport functionality shall be represented in a transport-specific component (for example, **TAG_TLS_SEC_TRANS**) contained in the **transport_mech** field of the **CompoundSecMech** structure. Where a compound security mechanism implements client authentication functionality in service context, the mechanism shall be represented in an **AS_ContextSec** structure contained in the **as_context_mech** field of the **CompoundSecMech** structure. Where a compound security mechanism supports identity assertion or supports authorization attributes delivered in service context, the mechanism shall be represented in a **SAS_ContextSec** structure contained in the **sas_context_mech** field of the **CompoundSecMech** structure.

At least one of the **transport_mech**, **as_context_mech**, or **sas_context_mech** fields shall be configured. The **TAG_NULL_TAG** component shall be used in the **transport_mech** field to indicate that a mechanism does not implement security functionality at the transport layer. A value of "no bits set" in the **target_supports** field of either the **as_context_mech** or **sas_context_mech** fields shall be used to indicate that the mechanism does not implement security functionality at the corresponding layer.

The **target_requires** field of the **CompoundSecMech** structure is used to designate a required outcome that shall be satisfied by one or more supporting (but not requiring) layers. The **target_requires** field also represents all the options required independently by the various layers as defined within the mechanism.

Each compound mechanism defines a combination of layer-specific functionality that is supported by the target. A target's mechanism configuration is the sum of the combinations defined in the individual mechanisms.

A value of TRUE in the **stateful** field of the **CompoundSecMechList** structure indicates that the target supports the establishment of stateful or reusable SAS contexts. This field is provided to assist clients in their selection of a target that supports stateful contexts. It is also provided to sustain implementations that serialize stateful context establishment on the client side as a means to conserve precious server-side authentication capacity.[10]

A TSS shall set the **stateful** bit to FALSE in the **CompoundSecMechList** structure of IORs corresponding to target objects at which it will not accept reusable security contexts.

10.5.1.5.1 struct AS_ContextSec

The **AS_ContextSec** structure is used in the **as_context_mech** field within a **CompoundSecMech** structure in a **TAG_CSI_SEC_MECH_LIST** component to describe the client authentication functionality that the target expects to be layered above the transport in service context by means of the **client_authentication_token** of the **EstablishContext** element of the SAS protocol.

10. This serialization is only done when an attempt is being made to establish a stateful context.

```
struct AS_ContextSec{
    AssociationOptions target_supports;
    AssociationOptions target_requires;
    CSI::OID client_authentication_mech;
    CSI::GSS_NT_ExportedName target_name;
};
```

A value of "no bits set" in the **target_supports** field indicates that the mechanism does not implement client authentication functionality above the transport in service context. In this case, the values present in any of the other fields in this structure are irrelevant.

If the **target_supports** field indicates that the mechanism supports client authentication in service context, then the **client_authentication_mech** field shall contain a GSS OID that identifies the GSS mechanism that the compound mechanism supports for client authentication above the transport.

The target uses the **target_name** field to make its security name and or authentication domain available to clients. This information may be required by the client to obtain or construct (depending on the mechanism) a suitable initial context token.

Table 10.16 describes the association options that are supported by conforming implementations.

Table 10.16 - EstablishTrustInClient Semantics

	EstablishTrustInClient	Semantic
1	Not supported	Target does not support client authentication in service context (at this compound mechanism).
2	Supported	Target supports client authentication in service context. If a CSS does not send an initial context token (in an EstablishContext service context element), then the caller identity is obtained from the transport.
3	Required	Target requires client authentication in service context. The CSS may have also authenticated in the transport, but the caller identity is obtained from the service context layer.

When a compound mechanism that implements client authentication functionality above the transport also contains a transport mechanism (in the **transport_mech** field), any required association options configured in the transport component shall be interpreted as a prerequisite to satisfying the requirements of the client authentication mechanism.

struct SAS_ContextSec

The **SAS_ContextSec** structure is used in the **sas_context_mech** field within a **CompoundSecMech** structure in a **TAG_CSI_SEC_MECH_LIST** component to describe the security functionality that the target expects to be layered above the transport in service context by means of the **identity_token** and **authorization_token** of the **EstablishContext** element of the SAS service context protocol. The security functionality represented by this structure is configured as association options in the **target_supports** and **target_requires** fields.

```
// The high order 20-bits of each ServiceConfigurationSyntax constant shall contain the Vendor Minor
// Codeset ID (VMCID) of the organization that defined the syntax. The low order 12 bits shall contain the
// organization-scoped syntax identifier. The high-order 20 bits of all syntaxes defined by the OMG shall
// contain the VMCID allocated to the OMG (that is, 0x4F4D0).

typedef unsigned long ServiceConfigurationSyntax;
```

```
const ServiceConfigurationSyntax SCS_GeneralNames = CSI::OMGVMCID | 0;
const ServiceConfigurationSyntax SCS_GSSExportedName = CSI::OMGVMCID | 1;

typedef sequence <octet> ServiceSpecificName;

// The name field of the ServiceConfiguration structure identifies a privilege authority in the format
// identified in the syntax field. If the syntax is SCS_GeneralNames, the name field contains an ASN.1 (BER)
// SEQUENCE[1..MAX] OF GeneralName, as defined by the type GeneralNames in [IETF RFC 2459]. If the
// syntax is SCS_GSSExportedName, the name field contains a GSS exported name encoded according to
// the rules in [IETF RFC 2743] 3.2, "Mechanism-Independent Exported Name Object Format," p. 84.

struct ServiceConfiguration {
    ServiceConfigurationSyntax syntax;
    ServiceSpecificName name;
};

typedef sequence <ServiceConfiguration> ServiceConfigurationList;

struct SAS_ContextSec{
    AssociationOptions target_supports;
    AssociationOptions target_requires;
    ServiceConfigurationList privilege_authorities;
    CSI::OIDList supported_naming_mechanisms;
    CSI::IdentityTokenType supported_identity_types;
};
```

The **privilege_authorities** field contains a sequence of zero or more **ServiceConfiguration** elements. A non-empty sequence indicates that the target supports the CSS delivery of an **AuthorizationToken**, which is delivered in the **EstablishContext** message. A CSS shall not be required to look beyond the first element of this sequence unless required by the first element.

The **syntax** field within the **ServiceConfiguration** element identifies the format used to represent the authority. Two alternative formats are currently defined: an ASN.1 encoding of the **GeneralNames** (as defined in [IETF RFC 2459]) which identify a privilege authority, or a GSS exported name (as defined in [IETF RFC 2743] 3.2) encoding of the name of a privilege authority.

The high order 20-bits of each **ServiceConfigurationSyntax** constant shall contain the Vendor Minor Codeset ID (VMCID) of the organization that defined the **syntax**. The low order 12 bits shall contain the organization-scoped syntax identifier. The high-order 20 bits of all syntaxes defined by the OMG shall contain the VMCID allocated to the OMG (that is, 0x4F4D0).

Organizations must register their VMCIDs with the OMG before using them to define a **ServiceConfigurationSyntax**.

The **supported_naming_mechanisms** field contains a list of GSS mechanism OIDs. A TSS shall set the value of this field to contain the GSS mechanism OIDs for which the target supports identity assertions using an identity token of type **ITTPrincipalName**. The Identity token types are defined in Identity Token Format on page 135.

The value of the **supported_identity_types** field shall be the bitmapped representation of the set of identity token types supported by the target. A target always supports ITTAbsent.

The value in **supported_identity_types** shall be non-zero if and only if the **IdentityAssertion** bit is non-zero in target_supports. The bit corresponding to the **ITTPrincipalName** identity token type shall be non-zero in **supported_identity_types** if and only if the value in **supported_naming_mechanisms** contains at least one element.

Table 10.17 describes the combinations of association options that are supported by conforming implementations. Each combination in the table describes the attribute layer functionality of a target that may be defined in a mechanism definition. A target that defines multiple mechanisms may support multiple combinations.

A compound mechanism definition with the **DelegationByClient** bit set shall include the name of at least one authority in the **privilege_authorities** field.

When a compound mechanism configuration that defines SAS attribute layer functionality also defines client authentication layer or transport layer functionality, any required association options configured in these other layers shall be interpreted as a prerequisite to satisfying the requirements of the functionality defined in the attribute layer

Table 10.17 - Attribute Layer Association Option Combinations

	DelegationByClient	IdentityAssertion	Semantic
1	Not supported	Not supported	Target does not support identity assertion (that is, identity tokens in the EstablishContext message of the SAS protocol). The caller identity will be obtained from the authentication layer(s).
2	Not supported	Supported	Target evaluates asserted caller identities based on trust rules of the target. In the absence of an asserted identity, the caller identity will be obtained from the authentication layer(s).
3	Supported	Not supported	Target accepts delegation tokens that indicate who has been endorsed to assert an identity. Target does not accept asserted caller identities. The caller identity will be obtained from the authentication layer(s).
4	Supported	Supported	Target accepts delegation tokens that indicate who has been endorsed to assert an identity. Target evaluates asserted caller identities based on trust rules of the target or based on endorsements in a delegation token. In the absence of an asserted identity, the caller identity will be obtained from the authentication layer(s).
5	Required	Not supported	Same as 3, with the addition that target requires a delegation token that endorses the target as proxy for the calle.
6	Required	Supported	Same as 4, with the addition that target requires a delegation token that endorses the target as proxy for the caller.

10.5.1.6 TAG_NULL_TAG

This new tagged component is used in the **transport_mech** field of a **CompoundSecMech** structure to indicate that the compound mechanism does not implement security functionality at the transport layer.

```
// The body of the TAG_NULL_TAG component is a sequence of octets of
// length 0.
const IOP::ComponentId TAG_NULL_TAG = 34;
```

10.5.2 Client-side Mechanism Selection

A client should evaluate the compound security mechanism definitions contained within the **CompoundSecMechList** in the **TAG_CSI_SEC_MECH_LIST** component in an IOR to select a mechanism that supports the options required by the client.

The options supported by a compound mechanism are the union (the logical OR) of the options supported by the **transport_mech**, **as_context_mech**, and **sas_context_mech** fields of the **CompoundSecMech** structure.

The following table defines the semantics defined by the union of association options in compound mechanism definitions. Association options for server to client authentication and message protection add additional semantics that are not represented in the table.

Table 10.18- Interpretation of Compound Mechanism Association Options

	Semantic	EstablishTrustInClient		IdentityAssertion	DelegationByClient	
		Supported	Required	Supported	Supported	Required
1	No client identification.				Don't care[b]	
2	Presumed trust.			X		
3	Authentication optional.	X			Don't care	
4	Authentication optional, assertion supported.	X		X		
5	Authentication Required.	X	X		Don't care	
6	Authentication Required, assertion supported.	X	X	X		
7	Presumed trust including support for provided target restrictions.			X	X	
8	Authentication optional, assertion supported including forward trust rules.	X		X	X	
9	Authentication required, assertion supported including forward trust rules.	X	X	X	X	
10	Presumed Trust including support for provided target restrictions, delegation token required which implies assertion required[a].			X	X	X

Table 10.18- Interpretation of Compound Mechanism Association Options

	Semantic	EstablishTrustInClient		IdentityAssertion	DelegationByClient	
		Supported	Required	Supported	Supported	Required
11	Authentication optional, assertion supported including forward trust rules, delegation token required which implies either client authentication or assertion required.	X		X	X	X
12	Authentication required, delegation token required.	X	X		X	X
13	Authentication required, assertion supported including forward trust rules, delegation token required.	X	X	X	X	X

a. If a delegation token is required, a non-anonymous client identity shall be established so that it can be endorsed by the delegation token. This same rule applies to row 11, and explains why there is no row that supports client authentication and requires a delegation token.

b. If DelegationByClient is supported, a delegation token may be provided, but it is not required to process the request

10.5.3 Client-Side Requirements and Location Binding

The primary assumption of this interoperability protocol is that transport layer security can ensure that it is not necessary to issue a preliminary request to establish a confidential association with the intended target.

In order to sustain this assumption, trust in target and a confidential transport shall be established prior to issuing any call that may contain arguments (including object keys) or service context elements that the client considers confidential. A CSS acting on behalf of a client may trust a target to locate an object (process a locate request) without having to trust the target with confidential arguments (other than object keys) or service context elements. For example, a CSS may have established a confidential connection to an address it learned from an IOR, and may then determine if the client trusts the target with its request arguments and any associated service context elements. If the client does not trust the target with its request, the CSS may send a locate request.[11] If the locate reply contains a new address, the CSS may establish a new confidential connection, evaluate the level of trust the client has in the new target, and determine whether it can issue the client's request to the target. If in response to the request, the CSS receives a location forward, it will establish another confidential connection with the new address and repeat its trust determination.

Compound security mechanisms appearing in IORs leading to a location daemon should not require clients to authenticate using the username/password mechanism if doing so would cause an overly trusting caller to share its password with an untrusted location daemon.

The way in which a location daemon derives an IOR for a target object is not prescribed by this part of ISO/IEC 19500.

11. CSS can use the Object::validate_connection operation to get the ORB to issue a locate request.

10.5.3.1 Comments on Establishing Trust in Client

A client that does not have the artifacts necessary to provide evidence of its authenticity over at least one of the transports supported by it and its target should search the IOR for a security mechanism definition that does not require client authentication to occur in a transport mechanism.

10.5.4 Server Side Consideration

If the target requires client authentication, and the transport does not provide that authentication, then the target should always respond with OBJECT_HERE to **LocateRequest** messages and defer the real forwarding response until it receives a GIOP Request message.

10.6 Conformance Levels

10.6.1 Conformance Level 0

Level 0 defines the base level of secure interoperability that all implementations are required to support. Level 0 requires support for SSL/TLS protected connections. Level 0 implementations are also required to support username/password client authentication and identity assertion by using the service context protocol defined in this part of ISO/IEC 19500.

10.6.1.1 Transport-Layer Requirements

Implementations shall support the Security Attribute Service (SAS) protocol within the service context lists of GIOP request and reply messages exchanged over SSL 3.0 and TLS 1.0 protected connections.

Implementations shall also support the SAS protocol within the service context lists of GIOP request and reply messages over unprotected transports defined within IIOP.[12]

10.6.1.1.1 Required Ciphersuites

Conforming implementations are required to support both SSL 3.0 and TLS 1.0 and the mandatory TLS 1.0 ciphersuites identified in [IETF RFC 2246]. Conforming implementations are also required to support the SSL 3.0 ciphersuites corresponding to the mandatory TLS 1.0 ciphersuites.

An additional set of recommended ciphersuites is identified in Recommended SSL/TLS Ciphersuites on page 150.

10.6.1.2 Service Context Protocol Requirements

All implementations shall support the Security Attribute Service (SAS) context element protocol in the manner described in the following sub clauses.

10.6.1.2.1 Stateless Mode

All implementations shall support the stateless CSS and stateless TSS modes of operation as defined in Session Semantics on page 141, and in the protocol message definitions appearing in SAS context_data Message Body Types on page 127.

12. SAS protocol elements should only be sent over unprotected transports within trusted environments.

10.6.1.2.2 Client Authentication Tokens and Mechanisms

All implementations shall support the username password (GSSUP) mechanism for client authentication as defined in Username Password GSS Mechanism (GSSUP) on page 133.

10.6.1.2.3 Identity Tokens and Identity Assertion

All implementations shall support the identity assertion functionality defined in Context Validation on page 138 and the identity token formats and functionality defined in Identity Token Format on page 135.

All implementations shall support GSSUP mechanism specific identity tokens of type **ITTPrincipalName**.

10.6.1.2.4 Authorization Tokens (not required)

At this level of conformance, implementations are not required to be capable of including an authorization token in the SAS protocol elements they send or of interpreting such tokens if they are included in received SAS protocol elements.

The format of authorization tokens is defined in Authorization Token Format on page 132.

10.6.1.3 Interoperable Object References (IORs)

The security mechanism configuration of CSIv2 target objects, shall be as defined in Target Security Configuration on page 151, with the exception that Level 0 implementations are not required to support the **DelegationByClient** functionality described in AssociationOptions Type on page 152.

10.6.2 Conformance Level 1

Level 1 adds the following additional requirements to those of Level 0.

10.6.2.1 Authorization Tokens

Level 1 implementations shall support the push model for privilege attributes.

Level 1 requires that a CSS provide clients with an ability to include an authorization token, as defined in Authorization Token Format on page 132, in SAS EstablishContext protocol messages.

Level 1 requires that a TSS be capable of evaluating its support for a received authorization token according to the rules defined in Extensions of the IETF AC Profile for CSIv2 on page 132.

A Level 1 TSS shall recognize the standard attributes and extensions defined in the attribute certificate profile defined in [IETF ID PKIXAC].

Level 1 requires that a target object that supports pushed privilege attributes include in its IORs the names of the privilege authorities trusted by the target object (as defined in struct SAS_ContextSec on page 157).

10.6.3 Conformance Level 2

Level 2 adds to Level 1 the following additional requirements.

10.6.3.1 Authorization-Token-Based Delegation

Level 2 adds to Level 1 a requirement that implementations support the authorization-token-based delegation mechanism implemented by the SAS protocol.

A Level 2 TSS shall be capable of evaluating proxy rules arriving in an authorization token to determine whether an asserting entity has been endorsed (by the authority which vouched for the privilege attributes in the authorization token) to assert the identity to which the privilege attributes pertain. The semantics of the relationship between the identity token and authorization token shall be as defined in Context Validation on page 138.

A Level 2 TSS shall recognize the Extensions of the IETF AC Profile for CSIv2 on page 132" (that is, the Proxy Info extension) as defined on that page.

Level 2 requires that a target object that accepts identity assertions based on endorsements in authorization tokens represent this support in its IORs as defined in Table 10.17.

Level 2 requires that a target object that requires an endorsement to act as proxy for its callers represent this requirement in its IORs as defined in Table 10.17.

10.6.4 Stateful Conformance

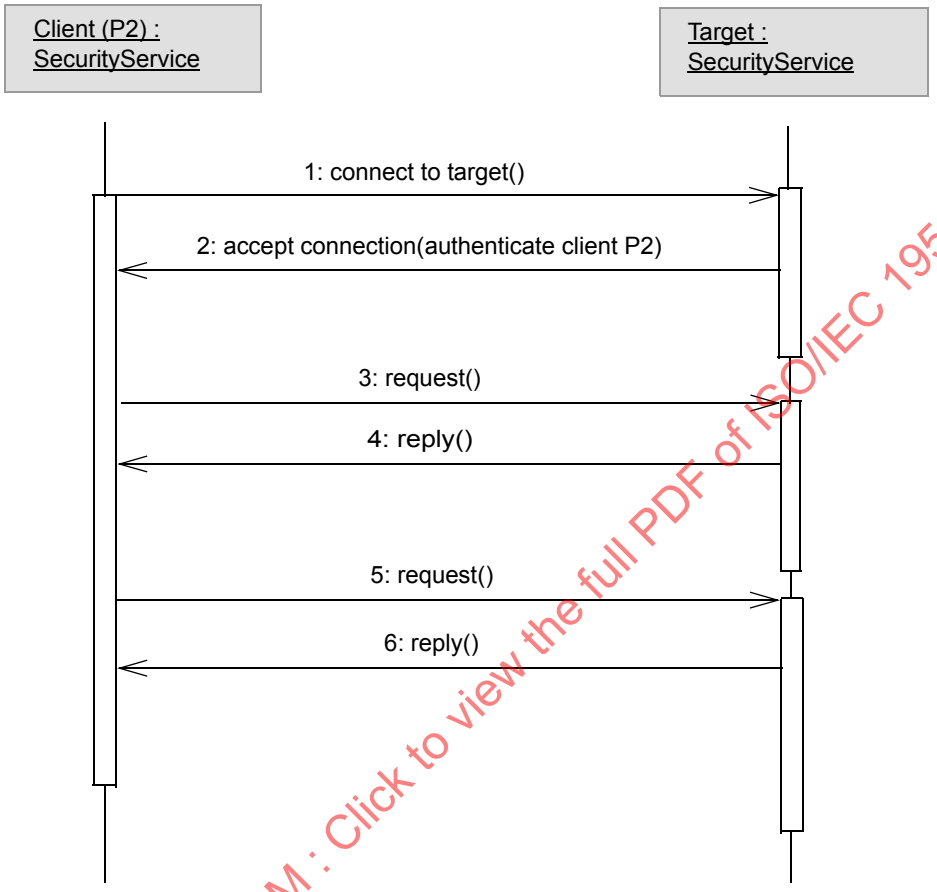
Implementations are differentiated not only by the conformance levels described in the preceding sub clauses but also by whether or not they support stateful security contexts.

For an implementation to claim stateful conformance, it shall implement the stateless and stateful functionality as defined in Session Semantics on page 141 and in SAS context_data Message Body Types on page 127.

10.7 Sample Message Flows and Scenarios

This appendix contains sequence diagrams and sample IORs for a set of scenarios selected to illustrate the interoperability protocols defined in this part of ISO/IEC 19500. The sample IORs are expressed in pseudocode.

10.7.1 Confidentiality, Trust in Server, and Trust in Client Established in the Connection



1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue. Client (P2) authenticates to TSS in the handshake.
3. Send request (with no security service context element).
4. Receive reply (with no security service context element).
5. Same as 3.
6. Same as 4.

10.7.1.1 Sample IOR Configuration

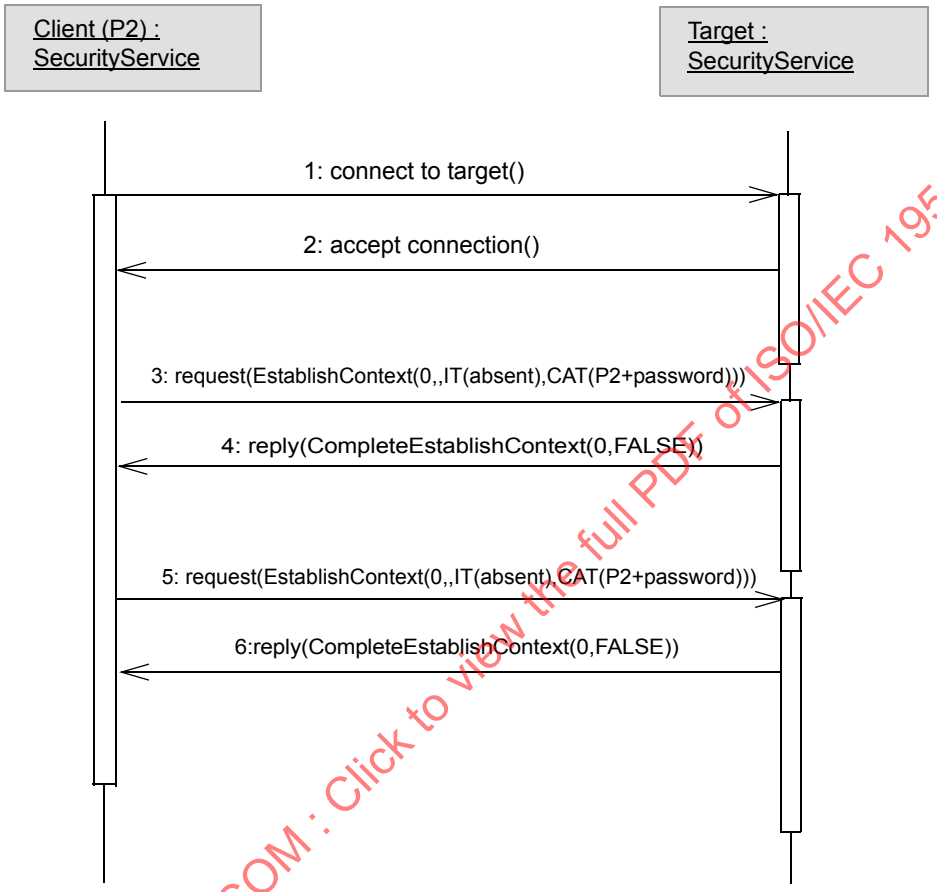
The following sample IOR was designed to address the related scenario.

CompoundSecMechList{
stateful = FALSE;
mechanism_list = {

```
        CompoundSecMec {
            target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
            transport_mech = TAG_TLS_SEC_TRANS {
                target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                    EstablishTrustInTarget};
                target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
                addresses = {
                    TransportAddress {
                        host_name = x;
                        port = y;
                    };
                };
            };
            as_context_mech = {
                target_supports = {};
                ...
            };
            sas_context_mech = {
                target_supports = {};
                ...
            };
        };
    };
};
```

Note that based on the ciphersuites listed in Required Ciphersuites on page 162 and the rules for target_supports and target_requires appearing in the tables in TAG_TLS_SEC_TRANS on page 153, all target IORs should include {Integrity, Confidentiality, EstablishTrustInTarget} in **target_supports** and at least {Integrity, Confidentiality} in **target_requires**. This statement applies to all the sample IORs corresponding to all the scenarios described in this sub clause.

10.7.2 Confidentiality and Trust in Server Established in the Connection - Stateless Trust in Client Established in Service Context



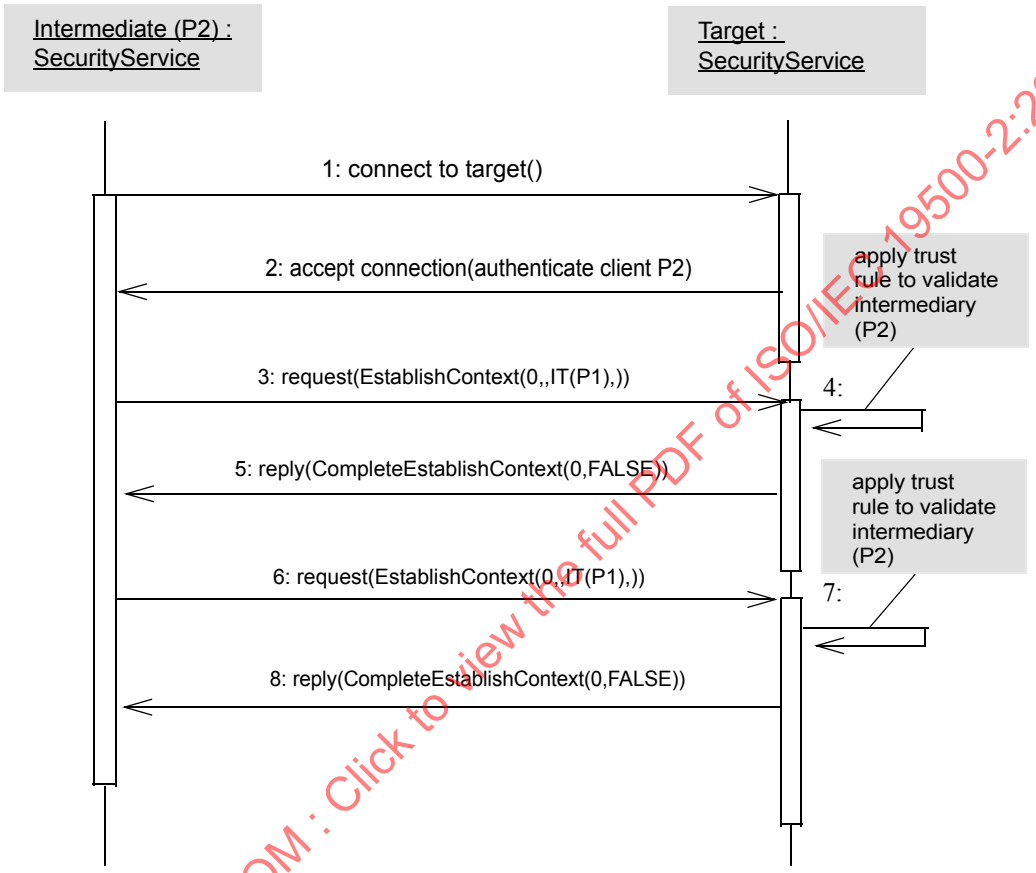
1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue.
3. Send request (with stateless security service context element containing a **client_authentication_token**).
4. Receive reply with **CompleteEstablishContext** service context element indicating context (and request) was accepted.
5. Same as 3.
6. Same as 4.

10.7.2.1 Sample IOR Configuration

The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList{
    stateful = FALSE;
    mechanism_list = {
        CompoundSecMec {
            target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
            transport_mech = TAG_TLS_SEC_TRANS {
                target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                    EstablishTrustInTarget};
                target_requires = {Integrity, Confidentiality};
                addresses = {
                    TransportAddress {
                        host_name = x;
                        port = y;
                    };
                };
            };
            as_context_mech = {
                target_supports = {EstablishTrustInClient};
                target_requires = {EstablishTrustInClient};
                client_authentication_mech = GSSUPMechOID;
                target_name = (GSSUPMechOID + name_scope);
            };
            sas_context_mech = {
                target_supports = {};
                ...
            };
        };
    };
};
```

10.7.3 Confidentiality, Trust in Server, and Trust in Client Established in the Connection Stateless Trust Association Established in Service Context



1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue. Client (P2) authenticates to TSS in the handshake.
3. Send request (with stateless security service context element containing spoken for identity (P1) in **identity_token**).
4. TSS validates that target trusts P2 to speak for P1.
5. Receive reply with **CompleteEstablishContext** service context element indicating context (and request) was accepted.
6. Same as 3.
7. Same as 4.
8. Same as 5.

10.7.3.1 Sample IOR Configuration

The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList {
    stateful = FALSE;
    mechanism_list = {
        CompoundSecMec {
            target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
            transport_mech = TAG_TLS_SEC_TRANS {
                target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                    EstablishTrustInTarget};
                target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
                addresses = {
                    TransportAddress {
                        host_name = x;
                        port = y;
                    };
                };
            };
            as_context_mech = {
                target_supports = {};
                ...
            };
            sas_context_mech = {
                target_supports = {IdentityAssertion};
                target_requires = {};
                privilege_authorities = {};
                supported_naming_mechanisms = {GSSUPMechOID};
                supported_identity_types = {ITTPrincipalName};
            };
        };
    };
};
```

10.7.3.2 Validating the Trusted Server

If trust is not presumed, then the TSS shall evaluate the trustworthiness of the speaking for identity (i.e., the client identity established in the authentication layer(s) - P2 in the preceding example) in order to determine if it is authorized to speak for the spoken for identity (i.e., the non-anonymous identity represented as P1 in the identity token in the preceding example).

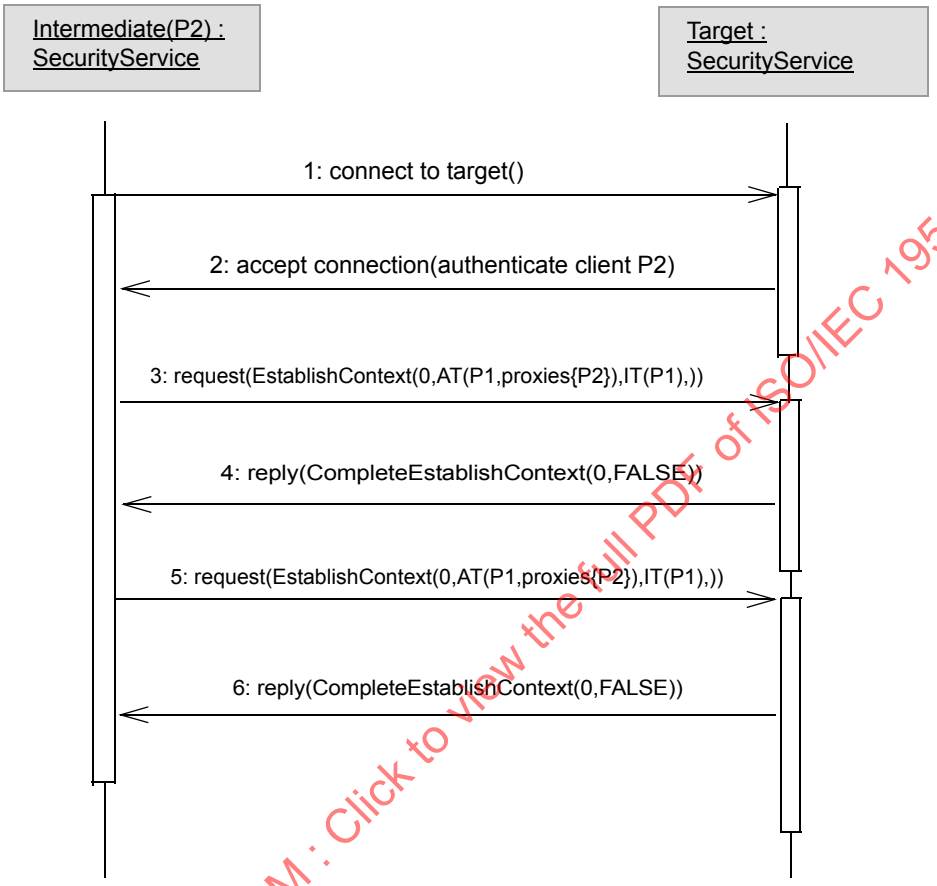
10.7.3.3 Presuming the Security of the Connection

There are variants of this scenario where either no security is established in the connection, or the connection is used to establish confidentiality only, and/or trust in the target only. These cases all fall under what is referred to as a presumed trust association. Where the security of the connection and the party using it is presumed, the TSS will not validate the trustworthiness of the speaking-for identity.

```
CompoundSecMechList {
    stateful = FALSE;
    mechanism_list = {
```

```
        CompoundSecMec {
            target_requires = {Integrity, Confidentiality};
            transport_mech = TAG_TLS_SEC_TRANS {
                target_supports = {Integrity, Confidentiality, EstablishTrustInTarget};
                target_requires = {Integrity, Confidentiality};
                addresses = {
                    TransportAddress {
                        host_name = x;
                        port = y;
                    };
                };
            };
            as_context_mech = {
                target_supports = {};
                ...
            };
            sas_context_mech = {
                target_supports = {IdentityAssertion};
                target_requires = {};
                privilege_authorities = {};
                supported_naming_mechanisms = {GSSUPMechOID};
                supported_identity_types = {ITTPrincipalName};
            };
        };
    };
};
```

10.7.4 Confidentiality, Trust in Server, and Trust in Client Established in the Connection - Stateless Forward Trust Association Established in Service Context



1. Initiate SSL/TLS connection to TSS.
2. SSL/TLS connection and ciphersuite negotiation accepted by both CSS and TSS. CSS evaluates its trust in target authentication identity and decides to continue. Intermediate (P2) authenticates to TSS in the handshake.
3. Send request with stateless security service context element containing spoken for identity (P1) in **identity_token**, and trust rule from P1 in **authorization_token** delegating proxy to P2.
4. Receive reply with **CompleteEstablishContext** service context element indicating context (and request) was accepted.
5. Same as 3.
6. Same as 4.

10.7.4.1 Sample IOR Configuration

The following sample IOR was designed to address the related scenario.

```
CompoundSecMechList {
    stateful = FALSE;
    mechanism_list = {
        CompoundSecMec {
            target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
            transport_mech = TAG_TLS_SEC_TRANS {
                target_supports = {Integrity, Confidentiality, EstablishTrustInClient,
                    EstablishTrustInTarget};
                target_requires = {Integrity, Confidentiality, EstablishTrustInClient};
                addresses = {
                    TransportAddress {
                        host_name = x;
                        port = y;
                    };
                };
            };
            as_context_mech = {
                target_supports = {};
                ...
            };
            sas_context_mech = {
                target_supports = {IdentityAssertion, DelegationByClient};
                target_requires = {};
                privilege_authorities = {
                    ServiceConfigurationSyntax {
                        syntax = s;
                        name = n;
                    };
                };
                supported_naming_mechanisms = {GSSUPMechOID};
                supported_identity_types = {ITTPrincipalName};
            };
        };
    };
};
```

10.8 References

CORBASEC

CORBA Security Service, Revision 1.2, http://www.omg.org/docs/ptc/98-01-02

CORBA Security Service, Revision 1.5, http://www.omg.org/docs/ptc/98-12-03

CORBA Security Service, Revision 1.7, http://www.omg.org/docs/ptc/99-12-03

IETF ID PKIXAC

An Internet Attribute Certificate Profile for Authorization, <draft-ietf-pkix-ac509prof-05.txt>, S. Farrell, Baltimore Technologies, R. Housley, SPYRUS, August 2000.

IETF RFC 2246

The TLS Protocol Version 1.0, T. Dierks, C. Allen, January 1999.

IETF RFC 2459

Internet X.509 Public Key Infrastructure Certificate and CRL Profile, R Housley, W. Ford, W. Polk, and D. Solo, January 1999.

IETF RFC 2743

Generic Security Service Application Program Interface Version 2, Update 1, J. Linn, January 2000.

X.501-93

ITU-T Recommendation X.501: Information Technology - Open Systems Interconnection - The Directory: Models, 1993.

10.9 IDL

10.9.1 Module GSSUP - Username/Password GSSAPI Token Formats

```
#ifndef _GSSUP_IDL_
#define _GSSUP_IDL_

import ::CSI;

module GSSUP {
    typeprefix GSSUP “omg.org”;

    // The GSS Object Identifier allocated for the
    // username/password mechanism is defined below.
    //
    //    { iso-itu-t (2) international-organization (23) omg (130)
    //          security (1) authentication (1) gssup-mechanism (1) }

    const CSI::StringOID GSSUPMechOID = "oid:2.23.130.1.1.1";

    // The following structure defines the inner contents of the
    // username password initial context token. This structure is
    // CDR encapsulated and appended at the end of the
    // username/password GSS (initial context) Token.

    struct InitialContextToken {
        CSI::UTF8String username;
        CSI::UTF8String password;
        CSI::GSS_NT_ExportedName target_name;
    };
    typedef unsigned long ErrorCode;

    // GSSUP Mechanism-Specific Error Token
```

```
    struct ErrorToken {
        ErrorCode error_code;
    };
    // The context validator has chosen not to reveal the GSSUP
    // specific cause of the failure.
    const ErrorCode GSS_UP_S_G_UNSPECIFIED = 1;

    // The user identified in the username field of the
    // GSSUP::InitialContextToken is unknown to the target.
    const ErrorCode GSS_UP_S_G_NOUSER = 2;

    // The password supplied in the GSSUP::InitialContextToken was
    // incorrect.
    const ErrorCode GSS_UP_S_G_BAD_PASSWORD = 3;

    // The target_name supplied in the GSSUP::InitialContextToken does
    // not match a target_name in a mechanism definition of the target.
    const ErrorCode GSS_UP_S_G_BAD_TARGET = 4;

}; // GSSUP

#endif
```

10.9.2 Module CSI - Common Secure Interoperability

```
#ifndef _CSI_IDL_
#define _CSI_IDL_

module CSI {
    typeprefix CSI “omg.org”;

    // The OMG VMCID; same value as CORBA::OMGVMCID. Do not change ever.

    const unsigned long OMGVMCID = 0x4F4D0;

    // An X509CertificateChain contains an ASN.1 BER encoded SEQUENCE
    // [1..MAX] OF X.509 certificates in a sequence of octets. The
    // subject’s certificate shall come first in the list. Each following
    // certificate shall directly certify the one preceding it. The ASN.1
    // representation of Certificate is as defined in [IETF RFC 2459].

    typedef sequence <octet> X509CertificateChain;

    // an X.501 type name or Distinguished Name in a sequence of
    // octets containing the ASN.1 encoding.

    typedef sequence <octet> X501DistinguishedName;

    // UTF-8 Encoding of String
```

```
typedef sequence <octet> UTF8String;

// ASN.1 Encoding of an OBJECT IDENTIFIER

typedef sequence <octet> OID;

typedef sequence <OID> OIDList;

// A sequence of octets containing a GSStoken. Initial context tokens are
// ASN.1 encoded as defined in [IETF RFC 2743]  3.1,
// "Mechanism-Independent token Format", pp. 81-82. Initial context tokens
// contain an ASN.1 tag followed by a token length, a mechanism identifier,
// and a mechanism-specific token (i.e. a GSSUP::InitialContextToken). The
// encoding of all other GSS tokens (e.g. error tokens and final context
// tokens) is mechanism dependent.

typedef sequence <octet> GSSToken;

// An encoding of a GSS Mechanism-Independent Exported Name Object as
// defined in [IETF RFC 2743]  3.2, "GSS Mechanism-Independent
// Exported Name Object Format," p. 84.
typedef sequence <octet> GSS_NT_ExportedName;
typedef sequence <GSS_NT_ExportedName> GSS_NT_ExportedNameList;

// The MsgType enumeration defines the complete set of service context
// message types used by the CSI context management protocols, including
// those message types pertaining only to the stateful application of the
// protocols (to insure proper alignment of the identifiers between
// stateless and stateful implementations). Specifically, the
// MTMessageInContext is not sent by stateless clients (although it may
// be received by stateless targets).

typedef short MsgType;

const MsgType MTEstablishContext = 0;
const MsgType MTCompleteEstablishContext = 1;
const MsgType MTContextError = 4;
const MsgType MTMessageInContext = 5;

// The ContextId type is used carry session identifiers. A stateless
// application of the service context protocol is indicated by a session
// identifier value of 0.

typedef unsigned long long ContextId;

// The AuthorizationElementType defines the contents and encoding of
// the_element field of the AuthorizationElement.

// The high order 20-bits of each AuthorizationElementType constant
// shall contain the Vendor Minor Codeset ID (VMCID) of the
// organization that defined the element type. The low order 12 bits
```

```
// shall contain the organization-scoped element type identifier. The
// high-order 20 bits of all element types defined by the OMG shall
// contain the VMCID allocated to the OMG (that is, 0x4F4D0).

typedef unsigned long AuthorizationElementType;

// An AuthorizationElementType of X509AttributeCertChain indicates that
// the_element field of the AuthorizationElement contains an ASN.1 BER
// SEQUENCE composed of an (X.509) AttributeCertificate followed by a
// SEQUENCE OF (X.509) Certificate.
// The chain of identity certificates is provided
// to certify the attribute certificate. Each certificate in the chain
// shall directly certify the one preceding it. The first certificate
// in the chain shall certify the attribute certificate. The ASN.1
// representation of (X.509) Certificate is as defined in [IETF RFC 2459].
// The ASN.1 representation of (X.509) AtributeCertificate is as defined
// in [IETF ID PKIXAC].

const AuthorizationElementType X509AttributeCertChain = OMGVMCID | 1;

typedef sequence <octet> AuthorizationElementContents;

// The AuthorizationElement contains one element of an authorization token.
// Each element of an authorization token is logically a PAC.

struct AuthorizationElement {
    AuthorizationElementType   the_type;
    AuthorizationElementContents   the_element;
};

// The AuthorizationToken is made up of a sequence of
// AuthorizationElements

typedef sequence <AuthorizationElement> AuthorizationToken;

typedef unsigned long IdentityTokenType;

// Additional standard identity token types shall only be defined by the
// OMG. All IdentityTokenType constants shall be a power of 2.

const IdentityTokenType ITTAbsent = 0;
const IdentityTokenType ITTAnonymous = 1;
const IdentityTokenType ITTPrincipalName = 2;
const IdentityTokenType ITTX509CertChain = 4;
const IdentityTokenType ITTDistinguishedName = 8;
typedef sequence <octet> IdentityExtension;

union IdentityToken switch ( IdentityTokenType ) {
    case ITTAbsent: boolean absent;
    case ITTAnonymous: boolean anonymous;
    case ITTPrincipalName: GSS_NT_ExportedName principal_name;
```

```
        case ITTX509CertChain: X509CertificateChain certificate_chain;
        case ITTDistinguishedName: X501DistinguishedName dn;
        default: IdentityExtension id;
    };

    struct EstablishContext {
        ContextId client_context_id;
        AuthorizationToken authorization_token;
        IdentityToken identity_token;
        GSSToken client_authentication_token;
    };

    struct CompleteEstablishContext {
        ContextId client_context_id;
        boolean context_stateful;
        GSSToken final_context_token;
    };

    struct ContextError {
        ContextId client_context_id;
        long major_status;
        long minor_status;
        GSSToken error_token;
    };

    // Not sent by stateless clients. If received by a stateless server, a
    // ContextError message should be returned, indicating the session does
    // not exist.

    struct MessageInContext {
        ContextId client_context_id;
        boolean discard_context;
    };

    union SASContextBody switch ( MsgType ) {
        case MTEstablishContext: EstablishContext establish_msg;
        case MTCompleteEstablishContext: CompleteEstablishContext complete_msg;
        case MTContextError: ContextError error_msg;
        case MTMessageInContext: MessageInContext in_context_msg;
    };

    // The following type represents the string representation of an ASN.1
    // OBJECT IDENTIFIER (OID). OIDs are represented by the string "oid:"
    // followed by the integer base 10 representation of the OID separated
    // by dots. For example, the OID corresponding to the OMG is represented
    // as: "oid:2.23.130"

    typedef string StringOID;

    // The GSS Object Identifier for the KRB5 mechanism is:
    // { iso(1) member-body(2) United States(840) mit(113554) infosys(1)
```

```
    // gssapi(2) krb5(2) }

    const StringOID KRB5MechOID = "oid:1.2.840.113554.1.2.2";

    // The GSS Object Identifier for name objects of the Mechanism-independent
    // Exported Name Object type is:
    // { iso(1) org(3) dod(6) internet(1) security(5) nametypes(6)
    // gss-api-exported-name(4) }

    const StringOID GSS_NT_Export_Name_OID = "oid:1.3.6.1.5.6.4";

    // The GSS Object Identifier for the scoped-username name form is:
    // { iso-itu-t (2) international-organization (23) omg (130) security (1)
    // naming (2) scoped-username(1) }

    const StringOID GSS_NT_Scoped_Username_OID = "oid:2.23.130.1.2.1";

}; // CSI
#endif
```

10.9.3 Module CSIIOP - CSIv2 IOR Component Tag Definitions

```
#ifndef _CSIIOP_IDL_
#define _CSIIOP_IDL_

import ::IOP;
import ::CSI;

module CSIIOP {
    typeprefix CIIOP “omg.org”;
    // Association options
    typedef unsigned short AssociationOptions;
    const AssociationOptions NoProtection = 1;
    const AssociationOptions Integrity = 2;
    const AssociationOptions Confidentiality = 4;
    const AssociationOptions DetectReplay = 8;
    const AssociationOptions DetectMisordering = 16;
    const AssociationOptions EstablishTrustInTarget = 32;
    const AssociationOptions EstablishTrustInClient = 64;
    const AssociationOptions NoDelegation = 128;
    const AssociationOptions SimpleDelegation = 256;
    const AssociationOptions CompositeDelegation = 512;
    const AssociationOptions IdentityAssertion = 1024;
    const AssociationOptions DelegationByClient = 2048;

    // The high order 20-bits of each ServiceConfigurationSyntax constant
    // shall contain the Vendor Minor Codeset ID (VMCID) of the
    // organization that defined the syntax. The low order 12 bits shall
    // contain the organization-scoped syntax identifier. The high-order 20
    // bits of all syntaxes defined by the OMG shall contain the VMCID
```

```
// allocated to the OMG (that is, 0x4F4D0).

typedef unsigned long ServiceConfigurationSyntax;

const ServiceConfigurationSyntax SCS_GeneralNames = CSI::OMGVMCID | 0;
const ServiceConfigurationSyntax SCS_GSSExportedName = CSI::OMGVMCID | 1;

typedef sequence <octet> ServiceSpecificName;

// The name field of the ServiceConfiguration structure identifies a
// privilege authority in the format identified in the syntax field. If the
// syntax is SCS_GeneralNames, the name field contains an ASN.1 (BER)
// SEQUENCE [1..MAX] OF GeneralName, as defined by the type GeneralNames in
// [IETF RFC 2459]. If the syntax is SCS_GSSExportedName, the name field
// contains a GSS exported name encoded according to the rules in
// [IETF RFC 2743] 3.2, "Mechanism-Independent Exported Name
// Object Format," p. 84.

struct ServiceConfiguration {
    ServiceConfigurationSyntax syntax;
    ServiceSpecificName name;
};

typedef sequence <ServiceConfiguration> ServiceConfigurationList;

// The body of the TAG_NULL_TAG component is a sequence of octets of
// length 0.

// type used to define AS layer functionality within a compound mechanism
// definition

struct AS_ContextSec {
    AssociationOptions target_supports;
    AssociationOptions target_requires;
    CSI::OID client_authentication_mech;
    CSI::GSS_NT_ExportedName target_name;
};

// type used to define SAS layer functionality within a compound mechanism
// definition

struct SAS_ContextSec {
    AssociationOptions target_supports;
    AssociationOptions target_requires;
    ServiceConfigurationList privilege_authorities;
    CSI::OIDList supported_naming_mechanisms;
    CSI::IdentityTokenType supported_identity_types;
};

// type used in the body of a TAG_CSI_SEC_MECH_LIST component to
// describe a compound mechanism
```