ORB 2.0 RFP Submission

Universal Networked Objects

BNR Europe Limited
Expersoft Corporation
IBM Corporation
ICL, plc
IONA Technologies
SunSoft, Inc.

September 28, 1994
OMG TC Document 94.9.32

Table of Contents

ORB Interoperability Architecture

Support for Inter-ORB Bridges

Internet Inter-ORB Protocol

Appendices

Preface 1

1.1 Cosubmitting Companies

The following companies are pleased to jointly submit this revised specification in response to the OMG ORB 2.0 Interoperability and Initialization RFP (OMG TC Document 93.9.15):

BNR
Expersoft
IBM
ICL
IONA
SunSoft

1.2 Guide to the Submission

This document consists of three main sections that together specify a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The proposed approach is universal in that its elements can be combined in a variety of ways to satisfy a very broad range of product and customer needs. These elements are:

- ORB Interoperability Architecture
- Support for Inter-ORB Bridges
- Internet Inter-ORB Protocol (IIOP)

All the submitting companies are sponsoring these specifications.

1.3 Summary of Key Features

ORB Interoperability Architecture

- Provides a conceptual framework for connecting arbitrary ORBs
- Describes bridging approaches both internal and external to ORBs
- Identifies architectural optimizations ensuring efficient bridging
- Allows for a variety of Inter-ORB Protocols (IOPs)

Support for Inter-ORB Bridges

- Enables requests from one ORB to be translated to requests on another
- Defines a new *Dynamic Skeleton Interface* (DSI), the server side analogue of the existing CORBA 1.2 *Dynamic Invocation Interface* (DII)
- Resolves problems in existing CORBA 1.2 support for dynamic invocation and type checking
- Provides support for managing tables keyed by object references

Internet Inter-ORB Protocol (IIOP)

- Based on TCP, the ubiquitous Internet protocol
- Simple to implement, with low runtime costs
- Can be used internally to ORBs as well as when connecting them
- Makes minimal assumptions about architecture of ORBs using it

1.4 Submission Contact Points

All questions about the joint submission should be directed to:

John Nichol
BNR Europe Limited
London Road
Harlow Essex CM17 9NA
UK
phone: +44 279 403 689
email: J.R.Nichol@bnr.co.uk

David Curtis
Expersoft Corporation
6620 Mesa Ridge Road, Suite 100
San Diego, CA 92121
USA
phone: +1 619 546 4100
email: curtis@expersoft.com

Don Vines
IBM Corporation
11400 Burnet Road
ZIP 1002
Austin, TX 78758
USA
phone: +1 512 823 7988
email: dvines@austin.ibm.com

Nic Holt
ICL (UK) Limited
Wenlock Way
West Gordon
Manchester
UK
phone: +44 61 223 1301
email: n.p.holt@uk03.wins.icl.co.uk

Oisin Hurley
IONA Technologies Ltd
8-34 Percy Place
Dublin 4
Ireland
phone: +353 1 668 6522
email: ohurley@iona.ie

Geoff Lewis
SunSoft, Inc
2550 Garcia Avenue
Mountain View, CA 94043
USA
phone: +1 415 336 2839
email: Geoffrey.Lewis@Eng.Sun.COM

Introduction and Overview 2

2.1 Introduction

This document consists of three main parts that together specify a comprehensive, flexible approach to supporting networks of objects that are distributed across and managed by multiple, heterogeneous CORBA-compliant ORBs. The proposed approach to "interORBability" is universal in that its elements can be combined in a variety of ways to satisfy a very broad range of product and customer needs. These elements are:

- **ORB Interoperability Architecture**
- **Support for Inter-ORB Bridges**
- **Internet Inter-ORB Protocol**

In addition, the architecture accommodates additional **environment-specific inter-ORB protocols** that are optimized for particular environments such as DCE.

The approach to ORB interoperability proposed in this submission conforms to and builds on the CORBA 1.2 specification and addresses the technical requirements identified in the *ORB 2.0 Interoperability RFP.*

2.2 Structure of the Document

Chapter 3 describes the proposed ORB Interoperability Architecture.

Chapters 4 to 7 specifies the proposed Inter-ORB Bridge Support. This includes extensions required to CORBA 1.2 in the following areas:

- Support for a Dynamic Skeleton Interface.
- Full support for dynamic operation invocation (client and server sides).

- Support for the efficient management of tables of object references. Specifically, object reference identity operations and mechanisms for the discovery of object destruction.

In the addition, the following are described:

- Clarifications important to interoperability relating to the CORBA 1.2 Interface Repository and the results of the ORB 2.0 *Interface Repository RFP*. Specifically, clarifications relating to the support for interface type equivalence checking, type codes, and IDL attributes.
- Requirements on IDL language mappings for full dynamic invocation support required for “request-level” bridging. Specifically, requirements on the support for all data types inside *any* and named values and full access to exceptions and other data types returned from DII operation invocations.

Chapter 8 specifies the proposed Internet Inter-ORB Protocol.

Appendix A codifies, but does not propose for standardization at this time, five frequently requested (and implemented) extensions to the set of OMG IDL primitive data types. While these extensions are not *required* for interoperability, they would simplify certain aspects and are recommended for future consideration by the ORB Task Force.

Appendix B summarizes all the IDL and pseudo IDL interfaces in the submission.

Appendix C identifies issues relating to interoperability that should be taken into account in the future evolution of the CORBA specification.

2.3 Elements of Universal Networked Objects

2.3.1 ORB Interoperability Architecture

The ORB Interoperability Architecture provides a conceptual framework within which ORB components relevant to interoperability are positioned and in which compliance points for interoperability are identified. 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 described below forms the common basis for broad-scope mediated bridging. The Inter-ORB Bridge Support described below can be used to implement both immediate bridges and to build “half-bridges” to mediated bridge domains.

2.3.2 Inter-ORB Bridge Support

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, ISVs, 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, e.g. Microsoft's Common Object Model (COM). The ease of doing this will depend on the extent that those systems conform to the CORBA Object Model.

2.3.3 Internet Inter-ORB Protocol

The Internet Inter-ORB Protocol (Internet IOP, or IIOP) element specifies a standardized interoperability protocol for the Internet. The Internet IOP provides "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 Internet IOP is purpose built for ORB to ORB interactions and is designed to work directly over TCP/IP. It does not require or rely on the use of higher level RPC mechanisms. The protocol is simple ("as simple as possible, but not simpler") and scalable and relatively easy to implement. It is designed to allow portable implementations with small memory footprints and reasonable performance and with minimal dependencies on supporting software other than the underlying transport layer.

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.

While the Internet IOP is initially defined for use over TCP/IP, the Internet IOP message formats are designed to be used with any transport layer that meets certain minimal assumptions. It is envisaged that in the future there could be versions of the Internet IOP for OSI, SNA, Netware (IPX), etc. While versions of the inter-ORB protocol running on different transports would not be directly interoperable, their commonality would allow easy and efficient bridging between such networking domains.

2.3.4 Environment-Specific Inter-ORB Protocols

This submission also makes provision for an open ended set of Environment-Specific Inter-ORB Protocols (ESIOPs, *pronounced E-SOPs*). 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 Internet IOP and ORB domains that use a particular ESIOP.

In the future, it is expected that specifications of interesting ESIOPs would be independently submitted to OMG for adoption sponsored by companies who plan commercial support. The first ESIOP could, for example, be specified for DCE environments.

2.4 Examples of Interoperability Solutions

The elements specified in this submission can be combined in a variety of ways to satisfy particular product and customer needs:

2.4.1 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 for bridges to be built between it and other ORBs that use environment-specific or proprietary protocols. To accomplish this, ORB A uses the Internet IOP and provides inter-ORB bridge support.

2.4.2 Example 2

ORB product B is designed to provide highly optimized, very high speed support for objects located on a single machine e.g. to support 1000s of Fresco GUI objects operated on at near function-call speeds. In addition, it is expected that 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 inter-ORB bridge support allowing it to be hooked into any appropriate "networked" ORB.

2.4.3 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 arbitrary other 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 Internet IOP to enable interoperability with other compatible ORBs.

2.5 Meaning of CORBA2 Compliance

CORBA2 (or CORBA 2.0) has been used in various senses: as the umbrella for a collection of enhancements to CORBA 1.2 including C++ and ORB interoperability; as the label for the ORB 2.0 Task Force RFP roadmap and adopted specs; and, most narrowly, as the name of the product of the ORB interoperability RFP.

In order to be clear on what it will mean to be compliant with the proposed ORB interoperability specifications it is necessary to have a clearer picture of what it means to be compliant with CORBA2 in general. The following presents a specific proposal for this and explains in this context what it will mean to be compliant with respect to ORB interoperability.

2.5.1 The CORBA2 family of component specifications

We propose that the label CORBA2 be used to denote the family of *component* specifications that together represent the next major revision of CORBA 1.x. Additional specifications would be added to this family in the future (e.g. an IDL COBOL mapping).

2.5.2 CORBA2 Common Object Request Environment

First, clearly there exists a "core CORBA" specification that every ORB should support. After the adoption of specifications as a result of the ORB 2.0 RFPs issued in 1994, this is likely to look something like:

CORBA2/Core =

- CORBA 1.2,
- minus IDL C mapping (see below),
- plus extensions required to support RFP2 Object Services
- plus Interface Repository extensions resulting from the *ORB 2.0 Interface Repository RFP*
- plus ORB Initialization resulting from the *ORB 2.0 Initialization RFP*
- plus Inter-ORB Bridge Support

Anything that doesn't make sense for *all* kinds of ORB implementations to support (e.g. singlenode as well as networked ORBs) would be moved out of the CORBA/Core.

2.5.3 CORBA2 IDL Language Mappings

Second, each IDL language mapping would be stand-alone i.e. no vendor is obliged to support any given language mapping, but if they do then they must comply with the OMG spec:

- CORBA2/C++
- CORBA2/SmallTalk
- CORBA2/C

C is also in this list, since, for example, an ORB product supporting only C++ is perfectly conceivable.

2.5.4 CORBA2 Internet Inter-ORB Protocol

Third, for basic interoperability between ORBs designed to be networked with other compatible ORBs (as opposed to bridged or stand-alone) there would be the Internet IOP specification:

- CORBA2/Internet IOP

2.5.5 CORBA2 Environment-Specific Inter-ORB Protocols

Fourth, for optimized interoperability between ORBs running in particular environments, a set of ESIOPs, the first of which could be based on DCE:

- CORBA2/DCE IOP

2.5.6 Compliance Branding

Thus, like the OMG Object Model, there would be “core CORBA” and a set of components. A given ORB product would support some set of these components depending on the needs of the vendor’s chosen market and the nature of their product. Note there are never two ways of doing any one thing. If a vendor supports C++ their ORB must comply with CORBA2/C++.

The proposed initial CORBA2 family of component specifications might therefore consist of:

- CORBA2/Core
- CORBA2/C
- CORBA2/C++
- CORBA2/SmallTalk
- CORBA2/Internet IOP
- CORBA2/DCE IOP

There may also be a need for one more component to be a container for things that fall out of the core as not being appropriate for all ORBs to support. Each component specification would be independently revised. For example, there might be CORBA2/Core 2.0 and CORBA2/Internet IOP 1.0.

The CORBA2 *specification* components would also be *compliance* components. When compliance branding machinery is in place, each of the CORBA2 components would have associated with it a *separate branding stamp*.

2.5.7 ORB Categories

Three broad categories of CORBA2-compliant ORBs are likely to emerge:

- Stand-alone ORBs

  These singlenode ORBs would support the CORBA2/Core and some set of CORBA2 language mappings. If needed, interoperability with other ORBs could be provided via bridges.

- Networked ORBs

  These ORBs would support the CORBA2/Core, some set of CORBA2 language mappings, and would provide multinode operation using either:
  - the Internet IOP,
  - an ESIOP, such as the DCE IOP, or
  - a proprietary protocol.

  Interoperability between different networked ORB domains would be provided via bridges.

2.6 Motivating Factors

2.6.1 ORB implementation diversity

As anticipated by the CORBA specification, there are today many different ORB products which address a variety of different 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 protocols, 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 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.

2.6.2 ORB boundaries have other motivations

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 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 it 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, etc., management of the state in an ORB (object reference location and translation information, interface repositories, per-object data, etc.) might also be done by creating “sub-ORBs”.

2.6.3 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 implementer 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 a 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.

2.7 Interoperability Design Goals

2.7.1 A General Approach to ORB Interoperability

Because of the diversity in ORB implementations, multiple approaches to interoperability are required. Options identified in the CORBA 1.2 specification 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 it 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 objrefs created by one ORB to another ORB without losing functionality.

The CORBA 1.2 specification also states that "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."

We propose an architecture and flexible framework within which a wide variety of ORB implementations can be made to interoperate and which includes both bridging and common protocol elements.

2.7.2 Design Goals

The following goals drove our approach to interoperability:

- The architecture and specifications should allow high performance, small footprint, lightweight interoperability solutions.
- The design should scale, should be not 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.

2.7.3 Non-Goals

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

- Support for security.
- Support for future ORB services.

2.8 Resolution of Technical Requirements

This section describes how our proposal addresses the general and specific technical requirements stated in the *ORB 2.0 Interoperability RFP*.

2.8.1 Relationship to CORBA 1.2

The ORB Interoperability Architecture builds on Common Object Request Broker Architecture (Chapter 3 of CORBA 1.2) by adding the notion of ORB Services, and the domains they define. It defines the problem of ORB interoperability in terms of bridging between those domains, and defines several ways in which those bridges can be constructed, both internal (in-line) and external (request-level) to ORBs.

The CORBA 1.2 Chapter on interoperability anticipates construction of bridges, and shared protocols, among other interoperability solutions. This submission is compatible with the approaches outlined there.

APIs included in this submission include compatible extensions to CORBA 1.2 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.
- The Interface Repository has been clarified, so that it fully supports dynamic access to all ORB functionality.
- APIs for managing object references have been defined, building on the support identified for the OMG Relationship Service.
- Language binding requirements, for full dynamic access to ORB functionality, have been identified.

2.8.2 Maximum implementation flexibility

This response has as a goal the preservation of implementation flexibility for ORBs. The interfaces defined here, including both the Internet IOP and the DSI, need not be intrusive to existing ORBs.

Bridges and common protocols may need to be placed at many points in a distributed system, for example collocated with the client, or the server, or at some intermediate location. This submission provides flexibility to ORB users as well as implementers in where in their systems each of these is placed. This allows a wide range of trade-offs in areas such as performance, security, administration, resource utilization, and usability.

2.8.3 Conformance to Existing Standards

The proposed ORB Interoperability Architecture is compatible with the concepts and architecture of the ISO Reference Model for Open Distributed Computing (RM-ODP).

2.8.4 Ability for two vendors' ORBs to interoperate without each having prior knowledge of the other's implementation

Two basic strategies are supported by this submission: bridging, and using the Internet IOP.

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

The Internet IOP may be used in bridging two (or more) ORBs by implementing "half bridges" which communicates to each other using the Internet IOP. This approach works both for stand-alone ORBs, and for networked ones which use an ESIOP.

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

2.8.5 Ability to support all ORB functionality

All ORB functionality defined in CORBA 1.2, and in the Object Transaction Service, can be supported through both bridges and the Internet IOP.

2.8.6 Preservation of content and semantics of ORB-specific information across ORB boundaries

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.

2.9 Proof of Concept

The architectural components of this submission reflect implementation experience from all of the submitters, as applied in released products.

The bridging components, as specified here, have been successfully prototyped.

Moreover, there have been several successful public demonstrations of the use of interface-specific and in-line bridges between ORB products. In particular, at the Object World show in July 1994, four ORB vendors bridged their ORBs in a public “hands-on” demonstration of a trading application called CORBAcards. Over several days of the show, hundreds of users executed thousands of trades between the bridged ORBs.

The Internet IOP is based on ORB protocol experience from all the submitters, and has elements from ORB protocols that are in current use.

ORB Interoperability Architecture 3

This chapter provides the architectural framework used in the rest of this submission.

3.1 Introduction

OMG Document 93-9-15 (CORBA 2.0 Interoperability & Initialization 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 (see "Resolution of Technical Requirements" on page 13) 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 (e.g. 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.

3.1.1 Domains

The CORBA Object Model (section 2 of CORBA 1.2) identifies various distribution transparencies which 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.

3.1.2 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.

3.1.3 Compliance Points

The architecture identifies two compliance points for ORB interoperability:

- standard APIs provided by an ORB to enable the construction of request level inter-ORB bridges;
- an *Internet Inter-ORB Protocol* (Internet IOP, or IIOP), which can be supported at application or ORB level.

These two 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.

3.2 ORBs and ORB Services

The ORB Core is defined in CORBA 1.2 (section 3.1.1) as "that part of the ORB which provides the basic representation of objects and the communication of requests." The ORB Core[1] 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 1.2.

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 incorporated directly into an ORB's core. It is an aim of this submission 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.

3.2.1 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 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 whilst 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 which 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.

1. Some figures in this submission use "ORB Kernel" as equivalent to ORB Core.

3.2.2 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.

- 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;
- 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.
- The server object performs the requested operation.
- 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 on 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.

3.2.3 Selection of ORB Services

The ORB Services used are determined by:

- Static properties of both client and server objects - e.g. whether a server is replicated;
- Dynamic attributes determined by a particular invocation context - e.g. 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 ways 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.

3.3 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.

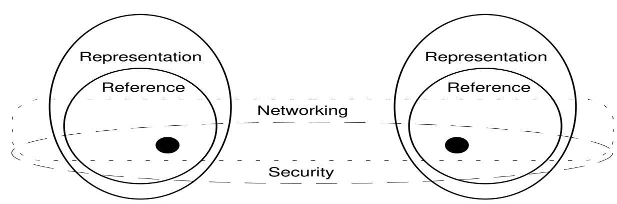

Figure 3-1 Different kinds of domains can coexist.

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 modelling concept which can simplify the analysis and description of complex systems. There may be many types of domains, for example, management domains, naming domains, language domains, technology domains.

3.3.1 Definition of a Domain

Domains allow partitioning of systems into collections of components which 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 modelled 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 and set up by their administrators.

3.3.2 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 which 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. In this submission, we assume that both domains are strictly compliant with the CORBA Object Model. This includes the use of OMG-IDL when defining interfaces, and several clarifications to CORBA 1.2 that we have found necessary (see "Interface Repositories" on page 51, and also "Extensions Needed to CORBA 1.2" on page 59). Variances from this model could easily compromise some aspects of interoperability.

3.4 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 submission focuses on bridging in only one direction. This is purely to simplify discussions, and does not imply that unidirectional connectivity satisfies basic interoperability requirements.

3.4.1 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. (Note that this is all the CORBA 1.2 specification 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).

3.4.2 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.

3.4.3 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 subsections.

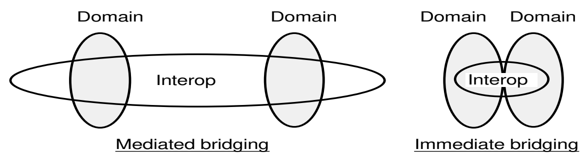

Figure 3-2 Two Bridging techniques that differ with respect to use of an intermediate form agreed on between the two domains.

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.

Some observations on mediated bridging:

a. The scope of agreement of a common form can range from a private agreement between two particular ORB/domain implementations to a universal standard;

b. There can be more than one common form, each oriented or optimized for a different purpose;

c. If there is more than one possible common form, then selection of 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);

d. Engineering of this approach can range from in-line specifically compiled (c.f. stubs) or generic library (e.g. encryption routines) code, to intermediate bridges to the common form.

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.

Some observations on immediate bridging:

a. 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;

b. This approach is often applicable when crossing domain boundaries which 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.

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 which 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.

Bridging level

As noted at the start of this section, 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 public CORBA APIs, including the newly specified 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 IDL-defined operation parameters, not implicit ones.) Rather, the precedent set with the Object 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.

3.4.4 Policy Mediated Bridging

An assumption made through most of this submission 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.

3.4.5 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.

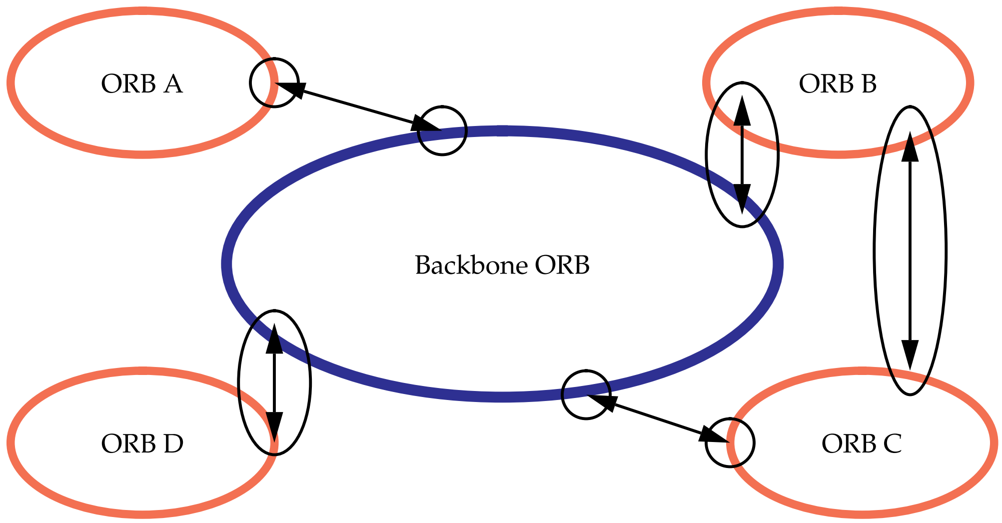

Figure 3-3 An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

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 linear in the number of protocols, rather than combinatoric.)

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.

3.5 Object Addressing

The CORBA Object Model, in section 2.2.2, 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 section of this submission 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.

3.5.1 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.

3.5.2 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 that new "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*[2]: 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. In the CORBA 1.2 document, this is called "Reference Embedding".

2. *Reference Encapsulation:* The bridge can avoid holding any state at all by conceptually catenating a domain identifier to the object name. Thus if a 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*.

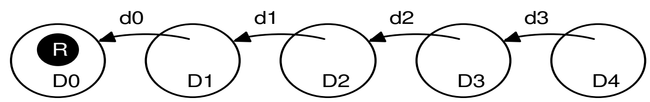

Figure 3-4 Reference encapsulation *adds domain information* during bridging.

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 kind of domain-based route identifier into the proxy, and leaving the original reference in like reference encapsulation.

   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)*.

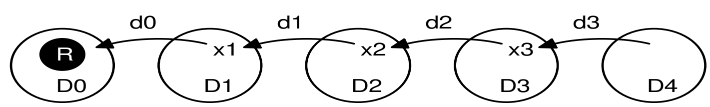

Figure 3-5 Domain Reference Translation *substitutes domain references* during bridging.

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 if a reference *R*, originating in domain *D0* would be identified in other domains as *D0.R*.

Some observations about these approaches to inter-domain reference handling:

a. Naive application of reference encapsulation could lead to arbitrarily large references. A "topology service" (noted further in Appendix C.1.5, "Topology Service", on page 107) could optimize cycles within any given encapsulated reference, and eliminate the appearance of references to local objects as alien references.

b. 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.

c. With the addition of the general purpose APIs in this submission (see "Managing Object References" on page 59) 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.

d. 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.

e. Reference canonicalization can also be performed with managed hierarchical namespaces such as those now in use on the Internet, and X.500 naming.

3.6 An Information Model for Object References

This section 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 "Internet IOP" on page 63.

3.6.1 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 19, several different ORB Services might be involved in an invocation, and providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

3.6.2 Interoperable Object References: IORs

To provide the information above, an "Interoperable Object Reference", or 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 briding object reference domains, within bridges.

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

```
module IOR {
    //
    // Standard Protocol Profile tag values
    //
    typedef unsigned long           ProfileId;
    const ProfileId                 TAG_INTERNET_IOP = 0;
    const ProfileId                 TAG_MULTIPLE_COMPONENTS = 1;

    struct TaggedProfile {
        ProfileId                   tag;
        sequence <octet>            profile_data;
    };

    //
    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.
    //
    struct IOR {
        string                      type_id;
        sequence <TaggedProfile>    profiles;
    };

    //
    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedComponent.
    //
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId                 tag;
        sequence <octet>            component_data;
    };
    typedef sequence <TaggedComponent> MultipleComponentProfile;
};
```

Object references have one *tagged profile* per protocol supported. Those profiles encapsulate all the basic information that protocol needs to identify an object. Any single profile holds enough information to drive a complete invocation using that protocol; the content and structure of those profile entries are wholly specified by that protocol. A bridge between two domains may need to know the detailed content of the profile for those domains' profiles, depending on the technique it uses to bridge the domains[3].

Each profile has a unique numeric tag, assigned by OMG. The ones defined here are for the IIOP (see "Internet IOP" on page 63) and for use in "multi component protocol profiles". 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, indicating ORB services accessible using that protocol. ORB services are assigned component identifiers in a namespace that is distinct from the profile identifiers[4].

Null object references are indicated by an empty set of profiles, and by a null type ID.

The type ID is provided to allow ORBs to preserve strong typing; it is further explained in "Interface, Operation, and Exception Equivalence" on page 51. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability submission, 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.

3.6.3 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.

3.6.4 Stringified Object References

According to CORBA, 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:

3. Based on topology and policy information available to it, a bridge may find it prudent to add or remove some profiles as it forwards an object reference. For example, a bridge acting as a firewall might remove all profiles except ones pointing to proxies it maintains. Other bridges might just add such profiles, letting clients which understand the profiles make routing choices.

4. Assigning of a unique profile or component numeric tag does not indicate that a profile has any status as a "standard", only that it will not be confused with other profiles.

- 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:

```
<oref>          ::= <prefix> <hex_Octets>
<prefix>        ::= "IOR:"
<hex_Octets>    ::= <hex_Octet> {<hex_Octet>}*
<hex_Octet>     ::= <hexDigit> <hexDigit>
<hexDigit>      ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
<digit>         ::= "0" | "1" | "2" | "3" | "4" | "5" |
                     "6" | "7" | "8" | "9"
<a>             ::= "a" | "A"
<b>             ::= "b" | "B"
<c>             ::= "c" | "C"
<d>             ::= "d" | "D"
<e>             ::= "e" | "E"
<f>             ::= "f" | "F"
```

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. (See "Common Data Representation: CDR Transfer Syntax" on page 66 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.

Building Bridges 4

This chapter provides an implementation-oriented conceptual framework for the construction of bridges to provide interoperability between ORBs.

It focuses on the *request level bridges* that this submission facilitates, although there is also recognition that ORBs may be modified to bridge internally.

4.1 In-Line and Request-Level Bridging

As described earlier, 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 nonstandard 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:

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

Request level bridges which 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

1. Special initialization support in OMG TC Document 94.9.46 allows two object referencing domains (e.g. two protocols) to be exposed to application programmers to support construction of this style bridge.

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 "Internet IOP" on page 63 can be used in either manner. In addition, this submission provides for "Environment Specific" inter-ORB protocols (ESIOP), allowing for alternative mediation mechanisms.

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

4.1.1 In-line bridging

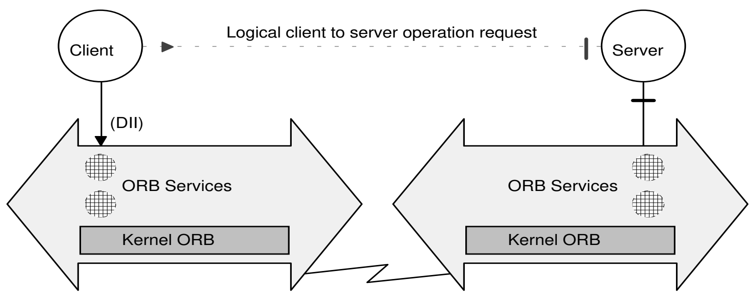

Figure 4-1 In-Line bridges are built using *ORB internal APIs*.

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 ORB(s);
- as additional or alternative stub and skeleton code.

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, and involves just as fundamental a set of modifications to an ORB.

4.1.2 Request-level bridging

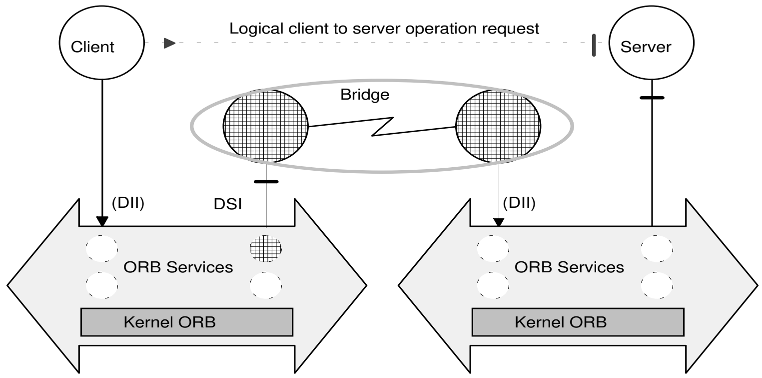

Figure 4-2 Request-Level bridges are built using *public ORB APIs*.

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.

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 requirement that the 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. (See “Requirements for a Language Mapping” on page 62.)

4.1.3 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 IDL-defined data as their mediating data format.

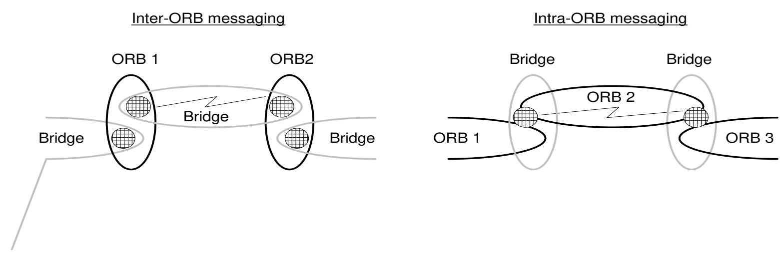

Figure 4-3 When the two ORBs are collocated in a bridge execution environment, network communications will be purely intra-ORB. If the ORBs are not collocated, such communications must go between ORBs.

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.

4.2 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 28.

- Reference translation approaches are possible with CORBA 1.2 APIs, though they benefit strongly from optimizations enabled by the interfaces defined in "Managing Object References" on page 59. 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 which support such encapsulation.

Note that from the perspective of clients, they only ever 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 thought 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 can not be proxies; see "Interface, Operation, and Exception Equivalence" on page 51 for more information.

4.3 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 use of generic bridges.

This submission defines a new interface, DSI, and various other enhancements to CORBA's dynamic typing infrastructure, specifically to facilitate the construction of fully functional generic bridges. That is, without those enhancements, portable request level bridges will not be able to bridge all CORBA operations; ORB-specific programming interfaces would need to be used. See "Dynamic Skeleton Interface: DSI" on page 41, "Interface Repositories" on page 51, and "Requirements for a Language Mapping" on page 62 for more information on these features.

4.4 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. See the appendix "Issues for Future CORBA Work" on page 106 for further information.

4.5 Bootstrapping Bridges

A particularly useful policy for setting up bridges is to create a pair of proxies for two COS Naming Service naming contexts (one in each ORB) and then install those proxies as naming contexts in the other ORB's naming service. 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.

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

Dynamic Skeleton Interface: DSI 5

This chapter is a normative portion of this specification; it describes a new dynamic interface for use in constructing request level bridges.

The *Dynamic Skeleton Interface* (DSI) is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. This contrasts with the type-specific, OMG IDL-based skeletons in CORBA 1.2, but serves the same architectural role.

DSI is the server side's analogue of the client side's *Dynamic Invocation Interface* (DII). Just as the implementation of an object cannot distinguish whether its client is using type-specific stubs or the DII, the client who invokes an object cannot determine whether the implementation is using a type-specific skeleton or the DSI to connect the implementation to the ORB.

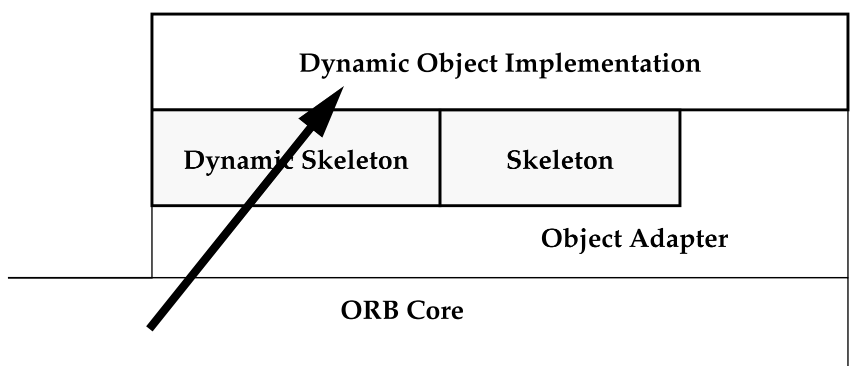

Figure 5-1 Requests are delivered through skeletons, including dynamic ones

DSI, like DII, has many applications beyond interoperability solutions. Uses include interactive software development tools based on interpreters, debuggers and monitors that wish to dynamically interpose on objects, and support for dynamically-typed languages such as LISP.

5.1 Overview of the Dynamic Skeleton Interface

The basic idea of the DSI is to implement all requests on a particular object by having the ORB invoke the same upcall routine, a *Dynamic Implementation Routine* (DIR). Since in any given language binding all DIRs have the same signature, a single DIR could be used as the implementation for many objects, with different interfaces.

The DIR is passed all the explicit operation parameters, as well as an indication of the object that was invoked and the operation that was requested, are encoded in the `request` parameter. The DIR can use the invoked object, its object adapter, and the interface repository to learn more about the particular object and invocation. It can access and operate on individual parameters. It can make the same use of an object adapter as other object implementations.

The Dynamic Skeleton Interface could be supported by any object adapter. Like type-specific skeletons, the DSI might have object adapter-specific details. We will describe a DSI interface for the Basic Object Adapter (BOA) and show how it is mapped into C and C++.

5.2 Explicit Request State: ServerRequest Pseudo-Object

The `ServerRequest` pseudo-object captures the explicit state of a request for the DSI, analogous to the Request pseudo-object in the DII. The following shows how it provides access to the information

```
module CORBA {
    pseudo interface ServerRequest
    {
        Identifier              op_name ();
        OperationDef            op_def ();
        Context                 ctx ();
        void                    params (inout NVList parms);
        NamedValue              result ();
    };
}
```

The `target` object of the invocation is provided by the language binding for the DIR. In the context of a bridge, it will typically be a proxy for an object in some other ORB.

The `op_name` operation returns the name of the operation being invoked; according to OMG IDL's rules, these names must be unique among all operations supported by this object's "most-derived" interface. As a convenience, the `op_def` operation may be used in preference to searching through the interface repository to acquire the definition of that operation.[1]

When the operation is not an attribute access, `ctx` will return the context information defined in OMG IDL for operation (if any). Otherwise, this context is empty.

Operation parameters will be retrieved with `params`. They appear in the NVList in the order in which they appear in the OMG IDL specification (left to right). This holds the "in", "out" and "inout" values.

The `result` operation is used to find where to store any return value for the call. Reporting of exceptions (which preclude use of result and out/inout values in `params`) is a function of the language mapping.

See each language binding for a description of the memory management aspects of these parameters.

5.3 Implicit Request State: ORB Extensions

CORBA 1.2 requires no state beyond the explicit state described above in the `ServerRequest` pseudo-object. However, various ORB services may need to provide some additional context to an ORB. This is to allow bridges to account for the semantics of those services as they bridge the service domains. An example of one solution to this problem is the Object Transaction Service, OMG TC Document 94.8.4.

The general provision of such implicit state, both to support true layering of object services and to allow request level bridges to manipulate it in the same way as explicit request state, is an area for future CORBA work, as noted in an appendix to this submission. For more information, see "Implicit Context" on page 107, and "Service-specific Implicit Context" on page 108

5.4 Dynamic Skeleton Interface: Language Mapping

Like other CORBA interfaces, the Dynamic Skeleton Interface must ultimately be mapped into particular programming languages. Because DSI is defined in terms of a pseudo-object, special attention must be paid to it in the language mapping. We show here how to map it into C and C++, supporting the CORBA's Basic Object Adapter (BOA).

1. Note that we require the interface repository to provide OperationDef objects for attribute get and set operations, and similarly to have operation names for those operations.

5.4.1 Language Mapping Issues

There is no requirement that a `ServerRequest` pseudo-object be usable as a general argument in OMG IDL operations, or listed in "orb.idl".

The client side memory management rules normally applied to pseudo-objects do not strictly apply to a ServerRequest's handling of operation parameters. Instead, the memory associated with parameters follows the memory management rules applied to data passed from skeletons into statically typed implementation routines, and vice versa.

In some language mappings, exceptions need special treatment. This is because the normal mapping for exceptions may require static knowledge of exception types. An example is the use of C++ exceptions, which require special run time typing information which can only be generated by a C++ compiler. Accordingly, CORBA's dynamic APIs (not just DSI, but also DII) needs a way to report exceptions that doesn't require such compile-time support: the DIR needs to be able to provide the TypeCode for an exception as it reports the exception.

Finally, note that these APIs have been specified to support a performance model whereby the ORB doesn't implicitly consult an interface repository (i.e. perform any remote object invocations, potentially slowing down a bridge) in order to handle an invocation: all the typing information is *provided to* the `ServerRequest` pseudo-object by an application. The ORB is allowed to verify that such information is correct, but such checking is not required.

5.4.2 C Mapping for ServerRequest

In the C mapping, a ServerRequest is a pseudo object in the CORBA module which supports the following operations:

```
CORBA_Identifier   CORBA_ServerRequest_op_name (
                       CORBA_ServerRequest req,
                       CORBA_Environment *env
                   );
```

This function returns the name of the operation being performed, as shown in the operation's OMG IDL specification.

```
CORBA_OperationDef CORBA_ServerRequest_op_def (
                       CORBA_ServerRequest req,
                       CORBA_Environment *env
                   );
```

This function returns an interface repository entry corresponding to the operation being performed. (This is provided purely as a convenience function, since it can always be found using the `InterfaceDef` for the target object and the operation name returned by `CORBA_ServerRequest_op_name`.)

```
CORBA_ContextRef   CORBA_ServerRequest_ctx (
                       CORBA_ServerRequest req,
                       CORBA_Environment *env
                   );
```

This function may be used to determine any context values passed as part of the operation. Context will only be available to the extent defined in the operation's OMG IDL definition; for example, attribute operations have none.

```
void               CORBA_ServerRequest_params (
                       CORBA_ServerRequest req,
                       CORBA_NVList parameters,
                       CORBA_Environment *env
                   );
```

This function is used to retrieve parameters from the ServerRequest, and to find the addresses used to pass pointers to result values to the ORB. It must always be called by each DIR, even when there are no parameters.

The caller passes ownership of the `parameters` NVList to the ORB. Before this routine is called, that NVList should be initialized with the TypeCodes for each of the parameters to the operation being implemented: *in*, *out*, and *inout* parameters inclusive. When the call returns, the `parameters` NVList is still usable by the DIR, and all *in* and *inout* parameters will have been unmarshaled. Pointers to those parameter values will at that point also be accessible through the `parameters` NVList.

The implementation routine will then process the call, producing any result values. If the DIR does not need to report an exception, it will replace pointers to *inout* values in parameters with the values to be returned, and assign pointers to *out* values in that NVList appropriately as well. When the DIR returns, all the parameter memory is freed as appropriate, and the NVList itself is freed by the ORB.

```
void               CORBA_ServerRequest_result (
                       CORBA_ServerRequest req,
                       CORBA_Any value,
                       CORBA_Environment *env
                   );
```

This function is used to report any result `value` for an operation; if the operation has no result, it must not be called. It also must not be called before the params have been retrieved, or if an exception is being reported.

```
void               CORBA_ServerRequest_exception (
                       CORBA_ServerRequest req,
                       CORBA_exception_type major,
```

```
                CORBA_String exception_id,
                CORBA_Any value,
                CORBA_Environment *env
            );
```

This function is used to report exceptions, both user and system, to the client who made the original invocation. The parameters are as follows:

- `major` – indicates whether the exception is a user exception or system exception
- `exception_id` – this is the logical ID for the exception
- `value` – this is the value of the exception, including a TypeCode.

5.4.3 C Mapping for BOA's Dynamic Implementation Routine

In C, a DIR is a function with this signature:

```
typedef void (*DynamicImplementationRoutine) (/* C */
    CORBA_Object                    target,
    CORBA_ServerRequest             request,
    CORBA_Environment               *env
);
```

Such a function will be invoked by the BOA when an invocation is received on an object reference whose implementation has registered a dynamic skeleton.

- `target` is the name object reference to which the invocation is directed.
- `request` is the ServerRequest used to access explicit parameters and report results (and exceptions).
- `env` may be passed to `CORBA_BOA_get_principal` if desired.

Unlike other BOA object implementations, the `CORBA_BOA_set_exception` API is not used. Instead, `CORBA_ServerRequest_exception` is used; this provides the TypeCode for the exception to the ORB, so that it does not need to consult the interface repository (or rely on compiled stubs) in order to marshal the exception value.

5.4.4 C++ Mapping for ServerRequest

The ServerRequest pseudo-object maps to a C++ class in the CORBA namespace which supports the following operations and signatures:

```
class ServerRequest {              // C++
 public:
    Identifier         op_name () throw (SystemException);
    OperationDef_ptr   op_def () throw (SystemException);
    Context_ptr        ctx () throw (SystemException);
    void               params (NVList_ptr parameters)
```

```
                              throw (SystemException);
        void              result (any *value)
                              throw (SystemException);
        void              exception (const String id, any *value)
                              throw (SystemException);
    };
```

Note that as with the rest of the (proposed) C++ mapping, ORB implementations are free to make such operations virtual and modify the inheritance as needed.

All of these operations follow the normal memory management rules for data passed into skeletons by the ORB. That is, the DIR is not allowed to modify or change the object reference returned by the `op_def()` member function, the string returned by `op_name()`, `in` parameters in the NVList, or the context returned by `ctx()`. Similarly, data allocated by the DIR and handed to the ORB (the NVList parameters, any result value, and exception values) is freed by the ORB rather than by the DIR.

Handling Operation Parameters and Results

The `ServerRequest` provides parameter values when the DIR invokes the `params()` operation. The `NVList` provided by the DIR to the ORB includes the `TypeCodes` (inside a `NamedValue`) for all parameters, including `out` ones (their values are null pointers at first), for the operation. This allows the ORB to verify that the correct parameter types have been provided before filling their values in, but does not require it to do so. It also relieves the ORB of all responsibility to consult the interface repository, promoting high performance implementations.

The `NVList` provided to the ORB then becomes owned by the ORB. It will not be deallocated until after the DIR returns. This allows the DIR to pass the `out` values, including the return side of `inout` values, to the ORB by modifying the NVList after `params()` has been called.

In order to guarantee that the ORB could always verify parameter lists, and to detect errors such as omitted parameters, Dynamic Implementation Routines are always required to call `params()`, even when the DIR believes that no parameters are used by the operation. When the DIR believes no parameters are used by the operation, it passes an empty `NVList`.

The `ServerRequest` will not send a response to the invocation until the DIR returns. If a return value is required, the `result()` operation must be invoked to provide that value to the ORB. Where no return value is required, this need not be invoked.

The `params()` and `result()` operations may be called only once, and in that exact order.

Sample Usage

In typical usage, the DIR will receive an upcall. It will determine the operation signature, either by using `op_name()` to consult a private cache of `OperationDef` information, or by using `op_def()`. This allows it to create an `NVList` and fill in the `TypeCodes` for all the operation's parameters: the `in` values, `out` values, and `inout` values. Then the DIR would call `params()` with that NVList. At this point, the value pointers for all `in` and `inout` (the input side only) parameters within that `NVList` would now be valid.

The DIR would then perform the work for the request, using the target object reference to determine to which real object the request relates. It would then store the value pointers for `out` and `inout` parameters into the NVList, and report any `result`() data. It would then return from the DIR upcall, signifying to the ORB that it could send any response message. The ORB would then free the data allocated by the DIR (in the NVList and in the result) after it to the client.

Reporting Exceptions

To report an exception, rather than provide return values, the DIR provides the exception value inside an Any, and passes that to `exception()` along with the logical ID for that exception. As with result data, the data would be freed by the ORB after the DIR returns. (The DIR can't in general throw exceptions, since in order to "throw" or "catch", C++ systems require type information that can only be generated at compile time. DSI, like DII, can not rely on such compile-time support.)

All exceptions are presented as structures embedded in an Any. This is required since the use of C++ catch/throw for user-defined exceptions relies on data generated by a C++ compiler, which will not be available to general bridges (which are constructed without any OMG IDL compiler support).

The `exception()` routine may be called only once, after `params()` is called. It may not be called if `result()` has been called.

5.4.5 C++ Mapping for BOA's Dynamic Implementation Routine

In C++ server side mappings, implementation objects are C++ objects. To use the DSI, an object implements a class in the BOA namespace that has a single member function with this signature:

```
class DynamicImplementation       // C++
{
 public:
    virtual void  invoke (
        CORBA::ServerRequestRef    request,
        CORBA::Environment         &env
    ) throw (
        // NO exceptions... uses ServerRequest::exception()
    ) = 0;
...
};
```

The env parameter is used in the `BOA::get_principal()` operation. Note that, as with the rest of the C++ mapping, the implementation inherits this interface, and may support other methods as well.

As with other C++ based operation implementations, two functions are accessible within the body of methods: `_this()`, returning an object reference (`Object_ptr`) for the target object, and `_boa()`, returning a `BOA_ptr` to the appropriate BOA. The method code may not assume where these two routines are defined.

5.5 Registering Dynamic Implementation Routines

Although it is not portably specified by the CORBA 1.2 specification, any ORB and its BOA implementation must have some way of connecting type-specific skeletons to the methods that implement the operations. The DSI uses the same mechanism.

A typical ORB/BOA implementation defines an operation, perhaps used when the object is activated, which specifies the methods to be used for a particular implementation class, e.g., for C:

```
BOA_setimpl (BOA, ImplementationDef, MethodList, skeleton);
```

The MethodList would be the DIR; the skeleton could be a Dynamic Skeleton, which would construct a ServerRequest object and invoke the DIR with it.

Whatever mechanism, whether at link time, run time, etc., is used to bind ordinary implementations to type-specific skeletons would be used to allow binding dynamic implementations to dynamic skeletons. Such bindings could be maintained on a per-object, per-interface, per-class, or other basis.

Interface Repositories 6

This submission assumes as a base that the agreements described in OMG TC Document 94-8-3 will be realized in some form in the eventual Interface Repositories specification to be adopted by the OMG. This chapter draws attention to several Interface Repository issues affecting interoperability, including solutions for all such issues.

Two specific issues still need final resolution, as planned by the Interface Repositories team according to 94-8-3:

- Defining a single shared logical type ID space that can be used by ORBs being bridged;
- Specifying an API through which a structurally equivalent `TypeCode` may be constructed from another, using only the structural information provided by the `kind()` and `parameter()` operations.

With respect to interoperability, the key role that interface repositories serve is to enable generic bridges the full use of all operations and data types through the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). Both sides of a bridge require full access to each ORB's interface repositories, and use information found there to manipulate `CORBA::Request` and `CORBA::ServerRequest` pseudo-objects.

6.1 Interface, Operation, and Exception Equivalence

There are three kinds of type-related identifier which CORBA 1.2 in effect requires to be processed through a request level bridge. These are identifiers for interfaces, operations, and exceptions.

Document 94-8-3 states that Interface Repositories will expose these logical IDs, so that if two repositories use the same ID value, that ID denotes the same entity. These identifiers are called “Repository IDs”, reusing an existing abstraction in a more general manner. (The phrases “logical IDs” and “type IDs” are also used to denote these IDs.)

- *Interface IDs*... inside TypeCodes, interface IDs identify the “most derived” interface supported by the object reference being described. These also appear in InterfaceDef objects returned by `Object::get_interface()`, and potentially in other cases (e.g. as general parameters).
- *Operation IDs*... identify the operation being requested, and is exposed at the API level (e.g. DII, DSI) in the form of an Operation Name[1]. That name is currently required by the OMG IDL language to be unambiguous over the scope of all operations inherited by an object’s interface.
- *Exception IDs*... are exposed when exceptions are reported, and are used to distinguish which exception is reported and what data is provided with it. These IDs also appear in `ExceptionDef` objects in interface repositories.

The issue surrounding these IDs may be illustrated using exception IDs. Two different ORBs are allowed by CORBA 1.2 to use different formats to identify their exception IDs. If a request level bridge were to take an exception ID reported from one ORB and give it to the other when it forwards that exception to the original client, that client would not be able to understand the exception being reported even though the server obeyed the full OMG IDL specification for the operation involved.

Briefly, there are two possible configurations. In one, the domain of these type identifiers encompasses both the ORBs being bridged. In the other configuration, each ORB is its own type identifier domain.

6.1.1 Single Type Domain

In the simpler and preferable model, ORBs to be bridged will share the same type identifiers. The logical repository IDs used by the repositories (and OMG IDL compiler output) in each ORB will mean the same thing, and be directly comparable.

A case of particular interest is that objects in one ORB can directly use and access interface repository entries from another, using normal proxy mechanisms. A proxy object in one ORB could have even been created using (a proxy for) an InterfaceDef from another ORB.

Note that simple schemes for creating a single type ID space exist, such as using the fully scoped name of an identifier from the OMG IDL definitions text. These are adequate for small scale interoperation. However, such simple

1. Operation IDs also show up “on the wire” in the Common Inter-ORB Protocol,

schemes fail as systems scale up and evolve; managed schemes are required, which involve the use of tools to assign and reassign identifiers in conjunction with the OMG IDL definitions text.

6.1.2 Multiple Type Domains

If two ORBs choose to use different identifiers internally, then they need to be translated at the ORB boundary. It may potentially be troublesome to allow the logical repository IDs from one ORB to reach the other without translation[2], as general arguments.

In such a configuration, bridges should not create proxies for objects residing in interface repositories. Instead of mapping remote objects to proxies maintained by the bridge, the remote interface repository objects will need to be mapped to objects in “local” interface repositories.

6.1.3 Drawbacks of Multiple Type Domains

Clearly, constructing bridges to ORBs which use different identifier schemes will involve maintaining tables that establish equivalence of the various logical repository IDs, and between comparable entries in the two different repositories. It will also involve creating entries in both ORBs’ repositories when new interfaces are installed, because those interfaces might be exposed across a bridge. Such administrative overhead is undesirable.

Multiple type identifiers also significantly complicate the existence of any kind of binary interface for applications. This is because stubs and skeletons need to incorporate these identifiers. Taking an application, with its stubs and skeletons as prepared for one ORB, and using it with another ORB would not be possible without some tool which could modify the precompiled stubs and skeletons to use the “locally appropriate” type identifiers.

6.1.4 Type Identifiers: Solution, and Interim Solution

Document 94-8-3 describes two issues with respect to type identifiers: the representation of those identifiers, and the way in which they are generated and assigned.

With respect to this first issue, in this submission we continue the CORBA 1.2 choice for the representation of `RepositoryId` values: they are strings. Binary values can easily be encapsulated in strings, if needed, so this should not be problematic; ORBs could use binary values internally if desired, only using a string form for an external representation.

2. Note that this is a general issue with namespaces managed by applications. Such scoping issues commonly show up when scaling many applications. Solutions are well recognized, and involve using identifiers for name contexts. By convention, some contexts are “well known”.

More problematic is the exact scheme for allocating the single type ID space. This is an area under active discussion. Pending final resolution of this issue, we include the following solution for assignment of type identifiers:

- Type IDs will be explicitly assigned, using unspecified administrative interfaces to the OMG IDL compiler and/or Interface Repositories.
- In the default case, when no administrative statement about type ID values has been made, the type ID values for interfaces and exceptions are a string consisting of two components:
  - The three prefix characters "`IDL`", which are intended to distinguish type identifiers assigned under this scheme from those allocated under others;
  - A single colon ("`:`") character, to separate this prefix from the rest of the type identifier;
  - The scoped name of the interface (or exception) as it is expressed in IDL, with forward slash ("`/`") characters separating individual nested scoped name components, and an initial slash used to indicate the global scope.

6.2 Equivalence Checking Operation

To facilitate maintaining type-safety for object references over the scope of an ORB, we propose the following API:

```
module CORBA {
    pseudo interface Object {
        ... existing CORBA::Object operations, plus:
        boolean     is_a (in string logical_type_id);
    };
};
```

The `logical_type_id` is a string denoting a shared type identifier (RepositoryId). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the "most derived" type of that object.

This operation exposes to application programmers functionality that must already exist in ORBs which support "type safe narrow", and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

6.3 TypeCodes

The CORBA 1.2 specification includes typecodes as part of the interface repositories chapter, though they are otherwise unrelated to such repositories. This submission includes clarifications needed to ensure that the dynamic invocation infrastructure (DII, DSI, Any,...) is fully capable of supporting request level bridges.

6.3.1 TypeCode Creation

When bridging two ORBs, TypeCode pseudo-object parameters passed out of one ORB (by themselves, or inside of an “Any”) will not generally be understood by the other ORB. New TypeCodes will need to be created.

Note – The Interface Repositories team is planning on defining an API which can be used to create TypeCodes. Portable generic bridges require such an API, though ORB-specific solutions (such as building TypeCodes from the octet form defined by the IIOP) can be used when portability is a minor concern.

That API should support efficient construction of TypeCodes which are equivalent, requiring input of no more than the structural type information exposed through the `TypeCode::kind()` and `TypeCode::parameter()` operations. It is the structural type information in these TypeCodes which drives the dynamic construction of intra-ORB messages using the DII and DSI, and the incidental construction of TypeCodes should not be a factor affecting the performance of bridges. In particular, it should not be necessary to consult any interface repositories when constructing a TypeCode.

6.3.2 Multidimensional Arrays

CORBA 1.2’s typecodes assume only unidimensional arrays, because there is only one TypeCode parameter indicating the size of the array. However, OMG IDL allows multiple dimensions, each of different sizes.

To resolve this problem, TypeCodes for multidimensional arrays are allowed to have more than two parameters. As written in CORBA 1.2, the first parameter is the type of the array element, and the second parameter is the bound of the first dimension. The clarification provided here is to have a third parameter be the bound of the second dimension, and so forth, so that the typecode for an N-dimensional array has N+1 parameters (rather than just two).

6.3.3 TypeCodes for Exceptions

CORBA 1.2 does not specify how the TypeCode for an exception looks; there is no `CORBA::tk_exception` enum value, or other way to describe them. This leaves unspecified what Interface Repositories will record for exceptions’ TypeCodes, and hence what information an ORB will have available to drive the marshaling of exceptions.

To resolve this problem, exceptions will use TypeCodes of type `CORBA::tk_struct`, as if the OMG IDL text used the `struct` keyword instead of the `exception` keyword.

6.3.4 Integer TypeCode Parameters

CORBA 1.2 describes some typecode parameters as having “integer” values. However, OMG IDL has no such data type.

To resolve this problem, such “integer” parameters are instead represented as “unsigned long”.

6.3.5 TypeCodes Encode Structural Information

The state of TypeCode definition CORBA 1.2 poses a significant problem for interoperability. It left underspecified many nuances of the definition of TypeCodes, in particular the extent to which they reliably carry nonstructural information. As a result, ORB implementors have made divergent assumptions regarding the intent and functionality of TypeCodes. To provide meaningful interoperability, this specification defines key characteristics of Typecodes

The interpretation used consistently throughout this specification is that only that structural information in TypeCodes is significant. Adopting this proposal includes accepting that:

- Individual TypeCodes primarily convey structural information, not type identity. Type identity and equivalence are complex concepts which are not yet well defined in CORBA systems.
- TypeCodes are primarily to allow the ORB to marshal and unmarshal data whose structure is not known at compile time.
- TypeCodes are not associated with Interface Repositories in any significant way. Specifically, propagation and construction of TypeCodes does not need to involve Interface Repositories.
- There is no semantic significance associated with the names associated with struct, union, enum, or other TypeCodes.

Note that the `tk_objref` TypeCode parameter, an interface ID as discussed in "Interface, Operation, and Exception Equivalence" on 51, is deemed to be structural information.

6.4 OMG IDL Attributes

The Interface Repositories defined in CORBA 1.2 do not fully support attributes, because of two omissions that prevent their use from DII and DSI.

6.4.1 Operation Names for Attribute Get, Set

In CORBA 1.2, there is no language independent way for a DII user to get or set attribute values. This is because no operation name is specified. Lack of such definition precludes portable software, including request level bridges, from using DII (or DSI) from using object attributes.

To resolve this problem, the interface repositories will guarantee that for an OMG IDL attribute “foo”, the operation name of “_get_foo” be used to indicate the attribute “get” operation, and “_set_foo” be used to indicate any “set” operation on that attribute.

6.4.2 OperationDef Entries for Attribute Get, Set

In OMG IDL, attribute declarations are declared to be a shorthand for a “get” operation and, for writable attributes, a “set” operation. The “get” operation returns a value of the attribute’s type, and the “set” operation takes a single `in` parameter of that type. However, interface repositories in CORBA 1.2 provide no way to get the `OperationDef` entries corresponding to those operations.

To resolve this problem, `OperationDef` entries for attribute “get” (and, as appropriate, “set”) operations are available in Interface Repositories. These will be contained in the relevant `AttributeDef` entries.

Extensions Needed to CORBA 1.2 7

This chapter is normative: all ORBs must support the interfaces and requirements included here.

In addition to proposing improved dynamic typing support by adding the Dynamic Skeleton Interface (section 5) and certain Interface Repository extensions (section 6) as already identified, this submission further proposes extensions to CORBA 1.2 by adding support for efficient management of tables of object references (e.g. for supporting reference translation) and placing further requirements on language mappings for dynamic typing support.

Note that this section incorporates, with additional material and with some minor changes, object reference identity functionality from OMG TC Document 94.5.5, *Joint Submission: Relationship Service Specification*. Those two interfaces have been incorporated into `CORBA::Object` with name changes, clarifications, and some generalization.

7.1 Managing Object References

CORBA1.2 provides no facilities to efficiently manage the large numbers of object references that facilities such as bridges, event channels, and relationship services need to support. Two basic problems are recognized:

- There is no way to detect that objects no longer exist, so that records for them (e.g. proxies) could be removed.
- There is no notion of object reference identity that can be used to determine whether records associated with an object reference already exist.

Similar problems also appear in other areas, so this section presents general solutions which have applicability beyond interoperability. We believe that these APIs are sufficient to address the basic object reference management problems that bridges need to solve, and will not be onerous for any current ORB implementation to provide.

7.1.1 Discovering Object Destruction

We define an additional exception to allow applications to discover that an object to which they hold a reference is no longer in existence, and an operation to let applications test for object existence without performing object invocations.

Exceptions for Object Non-Existence

This standard system exception is raised whenever an invocation on a deleted object was performed. It is an authoritative "hard" fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate "final recovery" style procedures.

```
module CORBA {
    exception OBJECT_NOT_EXIST {// new system exception
        unsigned long                  minor;
        completion_status              completed;
    };
};
```

Bridges forward this exception to clients, also destroying any records they may hold (for example, proxy objects used in reference translation). The clients could in turn purge any of their own data structures.

Probing for Object Non-Existence

The `Object::non_existent` attribute may be used to test whether an object (e.g. a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising `CORBA::OBJECT_NOT_EXIST`) if the ORB knows authoritatively that the object does not exist, and otherwise it returns false.

```
module CORBA {
    pseudo interface Object {
        readonly attribute boolean    non_existent ();
        ...
    };
};
```

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their "idle time" to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

7.1.2 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

Hashing: Object Identifiers

Object references are associated with ORB-internal identifiers which may indirectly be accessed by applications using the `hash()` operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

```
module CORBA {
    pseudo interface Object {
        unsigned long      hash (in unsigned long maximum);
        ...
    };
}
```

The `maximum` parameter to the `hash` operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given “real” object. Those proxies would not necessarily hash to the same value.

Equivalence Testing

The `Object::is_equivalent()` operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns `TRUE` if the target object reference is known to be equivalent to the other object reference passed as its parameter, and `FALSE` otherwise.

```
module CORBA {
    pseudo interface Object {
        boolean        is_equivalent (in Object other_object);
        ...
    };
}
```

If two object references are identical, they are equivalent. Two different object references which in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a `FALSE` return from `Object::is_equivalent` should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects.

A typical application use of this operation is be to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to “flatten” graphs that represent cyclical relationships between objects. Some might do this as they construct the table, while others might do during idle time.

7.2 Requirements for a Language Mapping

We extend the requirements listed in CORBA 1.2 for all language mappings to ensure that generic request level bridges can be built atop all ORBs without requiring vendor-specific extensions.

- A language mapping must include supporting all data types inside `Any` values and in `NamedValue` parameters to the DII (and DSI). That is, the “minimal types for `Any`” listed in TBL.20 of CORBA 1.2 is replaced by “All data types expressible in OMG IDL must be supported as `Any` values”.
- Language mappings for the DII must provide full access to exceptions and other data types returned by operation invocations, without requiring compiler generated support to examine their constituents.

For example, users of the DII must be able to locate object references in a user defined exception structure such as this one, without having compile-time knowledge of this exception type:

```
union state switch (boolean) {    // IDL: example
    case TRUE:      long value;
    case FALSE:     Object prober;
};
exception state_check {
    sequence <state>   states;
};
```

Internet IOP 8

This chapter is normative. It specifies an Inter-ORB Protocol (IOP) for ORB interoperability, which runs directly over the Internet. It should be supported by conforming ORB products regardless of other aspects of their implementation. Such support need not involve using it internally; conforming ORBs may also provide bridges to this protocol.

8.1 Goals of the Internet IOP

The Internet IOP (IIOP) supports protocol level ORB interoperability in a general, low-cost manner. The following objectives were pursued vigorously in IIOP's design:

- *Widest possible availability* – The IIOP is 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 IIOP 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 IIOP should support ORBs, and networks of bridged ORBs, to the size of today's Internet, and beyond.
- *Low cos*t – Adding support for 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, IIOP message formats are designed to be used with any transport layer that meets a minimal set of assumptions; specifically, the IIOP is designed to be implemented on other connection-oriented transport protocols.

- *Architectural neutrality* – The IIOP specification makes minimal assumptions about the architecture of agents that will support it. The IIOP specification treats ORBs as opaque entities with unknown architectures.

The approach a particular ORB takes to providing support for the 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.

8.2 Internet IOP Overview

The IIOP 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).
- *Internet IOP Message Formats*. IIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- *Internet IOP Message Transport*. IIOP messages are transferred over TCP/IP connections. The IIOP specification describes how agents open connections and use them to transfer IIOP messages.

8.2.1 Common Data Representation (CDR)

CDR is a transfer syntax, mapping from data types defined in OMG IDL into 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 IIOP 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 IIOP 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 extends OMG IDL specifications to define common representations for data types whose representations were undefined or implementation-dependent under CORBA 1.2.

8.2.2 Internet IOP Message Overview

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

- *Few, simple messages* – With only seven message formats, the IIOP supports full CORBA functionality between ORBs, with extended capabilities supporting object location services, dynamic migration, and efficient management of communication resources. IIOP 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. IIOP 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* – IIOP 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`).

IIOP also supports passing service-specific context, such as the transaction context defined by the Object Transaction Service (OMG TC Document 94.8.84). This mechanism is designed to support any service that requires service related context to be implicitly passed with requests.

8.2.3 Internet IOP Message Transfer

In this specification, IIOP messages are exchanged over TCP/IP connections. The general specification, however, is designed operate over any connection-oriented transport protocol that meets a minimal set of assumptions (described in "Internet IOP Message Transport" on page 87).

IIOP defines how agents initiate and manage TCP/IP connections. IIOP uses TCP/IP connections in the following ways:

- *Asymmetrical connection usage* – The IIOP 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. The server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction allows the IIOP specification to be much simpler and avoids certain race conditions.
- *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, IIOP message ordering constraints are minimal. IIOP 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* – IIOP defines messages for request cancellation orderly connection shutdown. These features allow ORBs to reclaim and reuse idle connection resources.

8.3 Common Data Representation: CDR Transfer Syntax

CDR is the format in which the IIOP 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 67.

IIOP defines two distinct kinds of octet streams, messages and encapsulations. Messages are the basic units of information exchange in IIOP, described in detail in "Internet IOP Message Formats" on page 78.

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 subsequently marshaled 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 IIOP, they are clearly noted.

8.3.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see "Internet IOP Message Formats" on page 78) 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 71. 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.

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 8-1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 8-1 Alignment requirements for OMG IDL primitive data types

TYPE	OCTET ALIGNMENT
char	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
float	4
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 an IIOP message header (see "Internet IOP Message Header" on page 78) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See "Encapsulation" on page 71).

Integer Data Types

Figure 8-1 illustrates the representations for OMG IDL integer data types, including the following data types:

- short
- unsigned short
- long
- unsigned long

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

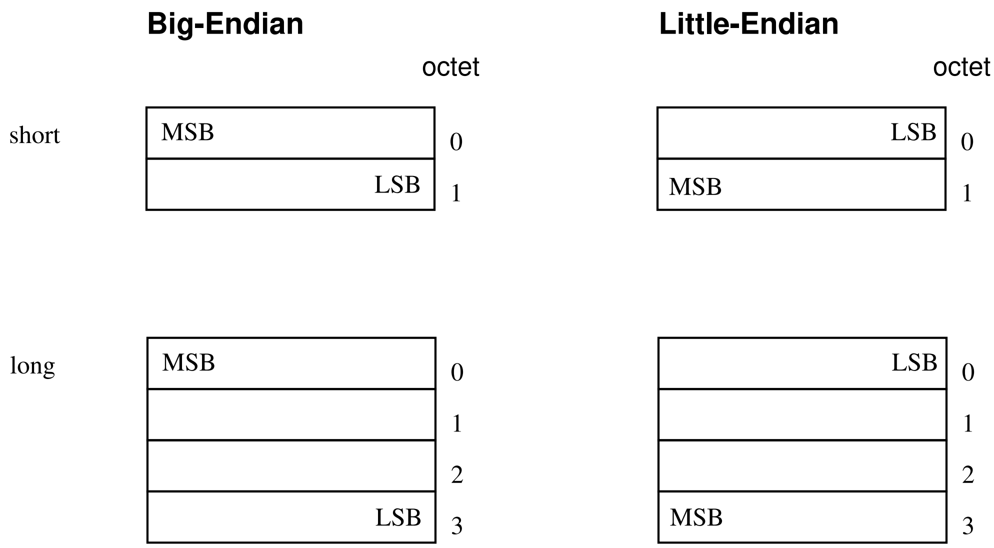

Figure 8-1 Sizes and bit ordering in big-endian and little-endian encodings of IDL integer data types, both signed and unsigned.

Floating Point Data Types

Figure 8-2 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.

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

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:

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

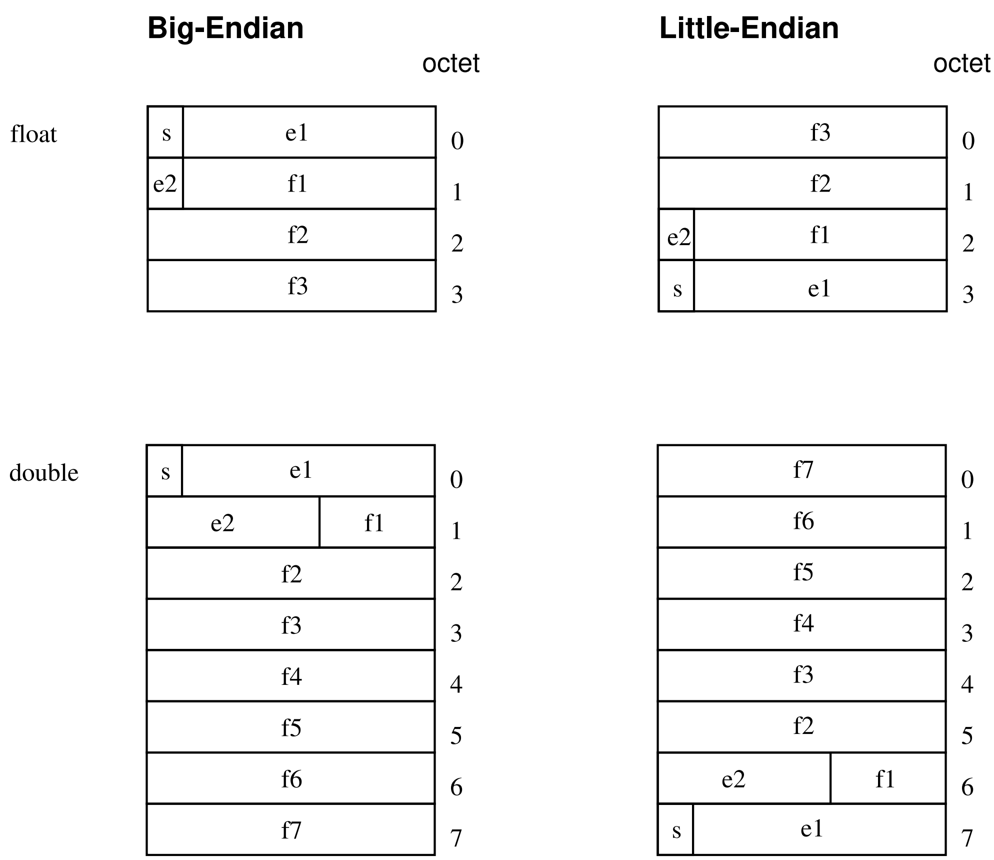

Figure 8-2 Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single and double precision floating point numbers.

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 specification, octets may be considered as unsigned 8-bit integer values.

Boolean

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

Character Types

OMG IDL characters are represented single octets, encoded as defined by ISO Latin-1 (8859.1).

8.3.2 OMG IDL Constructed Types

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

Alignment

Constructed type 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. IIOP 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.

Struct

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

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.

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.

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.

String

Strings are encoded as an unsigned long containing the length of the string, followed by the individual characters in the string, encoded in ISO Latin-1 (8859.1). The length (initial unsigned long) and string representation include a terminating null character, so that conventional string handling library routines (e.g., `strcpy`) may be used in the encoded message buffer.

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 are take ascending numeric values, in order of declaration from left to right.

8.3.3 Encapsulation

As described above, OMG IDL data types may be independently marshaled into encapsulation octet streams. The octet stream is represented as the OMG IDL type `sequence<octet>`, which may be subsequently included in an IIOP message or nested in another encapsulation.

The IIOP explicitly uses encapsulations in three places: `TypeCodes` (see "TypeCode" on page 72), the IIOP protocol profile inside an IOR (see "Object Reference" on page 76), and in service-specific context (see "Service Context" on page 79). In addition, some ORBs may use choose to use an encapsulation to hold `Principal` identification information (see "Principal" on page 76), the `object_key` (see "Object Reference" on page 76), 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 IIOP message headers (see "Internet IOP Message Header" on page 78). 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 71). 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 IIOP messages and nested encapsulations.[2]

8.3.4 Pseudo-object types

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

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 8-2. The enum value column 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 section 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.
- Ones with *simple parameter lists* are encoded as the `TCKind` enum value followed by the parameter value(s), encoded as indicated in Table 8-2. A "simple" parameter list has a fixed number of fixed length entries, or a single parameter which is has its length encoded first. Currently, only the `TCKind` values `tk_objref` or `tk_string` have such parameter lists.

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. The IIOP protocol itself does not require encapsulation of such data.

- 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 71) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 8-2.

The third column of Table 8-2 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 75. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 8-2. The ordering and meaning of parameters is a superset of those given in CORBA 1.2, TBL 16 on page 144; 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`, `tk_array`) 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 etc., before the next tuple is encoded (first, then second, etc.).

Note that the tuples identifying struct and enum members, and array bounds, 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]

Encoded Identifiers and Names

The interface ID parameter of the `tk_objref` TypeCode is the interface logical identifier, described in "Interface, Operation, and Exception Equivalence" on page 51. If two interface ID strings have equal values, the interfaces supported by the object references are identical.

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 8-2 TypeCode enum values, parameter list types, and parameters

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	simple	string (interface ID)
`tk_struct`	15	complex	string (name), ulong (count) {string (member name), TypeCode (member type)}
`tk_union`	16	complex	string(name), TypeCode (discriminant type), long (default used), ulong (count) {*discriminant type*[1] (label value), string (member name), TypeCode (member type)}
`tk_enum`	17	complex	string (name), ulong (count) {string (member name)}
`tk_string`	18	simple	ulong (max length[2])
`tk_sequence`	19	complex	TypeCode (element type), ulong (bounds)
`tk_array`	20	complex	TypeCode (element type), ulong (count) {ulong (dimension size)}
`- none -`	0xffffffff	simple	long (indirection[3])

1. The type of union label values is determined by the second parameter, discriminant type.
2. For unbounded strings, this value is zero.
3. See "Indirection: Recursive and Repeated TypeCodes" on page 75.

The *member name* parameters in `tk_struct`, `tk_union`, and `tk_enum`

TypeCodes, as well as the initial *name* string in those three TypeCodes, are not specified by (or significant in) IIOP. Agents should not make assumptions about type equivalence based on these name values; only the structural information is significant. If provided, the strings should be the simple, unscoped names supplied in the OMG IDL definition text.

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.

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.
- Only the second (and subsequent) references to a given 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. This indirection may well cross encapsulation boundaries, but this is not problematic because of first constraint identified above. Because of the second constraint, the value of the offset will always be negative.

The encoding of such an indirection is as a TypeCode with an "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. (As an example, that this means offset of negative four (-4) is illegal, because it be self-indirecting.)

Any

Any values are encoded as a TypeCode (encoded as described above) followed by the encoded value.

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 (as described in CORBA 1.2) serving to identify callers (and potential callers). This specification does not define any inter-ORB security mechanisms, or prescribe any usage of `Principal` values.

By representing Principal values as `sequence<octet>`, IIOP 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.

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.

Exception

Exceptions are encoded as a string followed by exception members, if any. The string contains the exception logical identifier, described in "Interface, Operation, and Exception Equivalence" on page 51. Exception members (if any) are encoded in the same manner as a struct.

Note – A catalog of minor codes for CORBA's System Exceptions will be provided, based on implementers agreements. Such an agreement is needed to support complete interoperability, since otherwise applications could distinguish ORBs based on the diagnostics they report, and could not reliably assign meanings to system exceptions reported to them.

8.3.5 Object Reference

Object references are encoded in OMG IDL as described in "Interoperable Object References: IORs" on page 30. IIOP protocol profiles, identifying individual objects accessible through IIOP, have the following form:

```
module IIOP {
    struct Version {
        char                major;
        char                minor;
    };

    struct ProfileBody {
        Version             iiop_version;
        string              host;
        unsigned short      port;
        sequence <octet>    object_key;
    };
};
```

An instance of the `IIOP::ProfileBody` type is marshaled into an encapsulation octet stream. This encapsulation (a `sequence<octet>`) becomes the `profile_data` member of the `IOR::TaggedProfile` structure representing the IIOP profile in an IOR, and the `tag` has the value `TAG_INTERNET_IOP` (as defined earlier).

The members of `IIOP::ProfileBody` 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 specification is one (1); the minor version is zero (0).
- *host*– identifies the Internet host to which IIOP 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*– This member 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*– The `object_key` member 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.

Note that host addresses are restricted in this version of IIOP to be Class A, B, or C Internet addresses. That is, Class D (multicast) addresses are not allowed. Such addresses are reserved for use in future versions of IIOP.

Also, note that at this time no “well known” port number has been allocated, so individual agents will need to assign previously unused ports as part of their installation procedures. IIOP supports multiple such agents per host.

8.4 Internet IOP Message Formats

In describing IIOP messages, it is necessary to define client and server roles. For the purpose of this discussion, a client is the agent that opens a connection (see more details in "Connection Management" on page 88) and originates requests. A server is an agent that accepts connections and receives requests.

IIOP message types are summarized in Table 8-3, 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 IIOP message headers.

Table 8-3 IIOP Message Types and originators

Message Type	Originator	Value
`Request`	Client	0
`Reply`	Server	1
`CancelRequest`	Client	2
`LocateRequest`	Client	3
`LocateReply`	Server	4
`CloseConnection`	Server	5
`MessageError`	Both	6

8.4.1 Internet IOP Message Header

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

```
module IIOP {
    enum MsgType {
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError
    };

    struct MessageHeader {
        char           magic [4];
        Version        iiop_version;
        boolean        byte_order;
        octet          message_type;
        unsigned long  message_size;
    };
};
```

The message header clearly identifies IIOP messages, but is defined to be byte-ordering independent, since the header itself defines the byte ordering of subsequent message elements. The members of the header are:

- *magic* – identifies IIOP messages. The value of this member is always the four (upper case) characters “IIOP”, encoded in ISO Latin-1 (8859.1).
- *iiop_version* – contains the version number of the IIOP protocol being used in the message, as defined earlier in "Object Reference" on page 76.
- *byte_order*– 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.
- *message_type*– indicates the type of the message, according to Table 8-3; these correspond to enum values of type MsgType.
- *message_size*– contains the length of the message following the message header, in octets. This count includes any alignment gaps.

8.4.2 Service Context

Emerging specifications for object services occasionally require service-specific context information to be passed implicitly with requests and replies. The IIOP defines a mechanism for identifying and passing this service-specific context information as “hidden” parameters. This 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.

IIOP does not describe any service-specific information. It only describes a mechanism for transmitting it in the most general way possible.

Each object service requiring implicit service-specific context to be passed through IIOP 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 service context data is described by the following OMG IDL and commentary:

```
module IIOP {
    typedef unsigned long             ServiceID;

    struct ServiceContext {
        ServiceID          context_id;
        sequence <octet>   context_data;
    };
    typedef sequence <ServiceContext> ServiceContextList;

    const ServiceID                   TransactionService = 0;
};
```

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 `IIOP::ServiceContext`. (See "Encapsulation" on page 71.) 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).

Note – The only *ServiceID* currently defined is *TransactionService*, for a CDR encapsulation of the *TSInteroperation::PropogationContext* defined in section 5.22 of OMG document 94.8.4.

8.4.3 Request Message

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

Request messages have three elements, encoded in this order:

- A IIOP message header,
- A *RequestHeader*, and
- The Request Body.

Request Header

The request header is specified as follows:

```
module IIOP {
    struct RequestHeader {
        ServiceContextList service_context;
        unsigned long      request_id;
        boolean            response_expected;
        sequence <octet>   object_key;
        string             operation;
        Principal          requesting_principal;
    };
};
```

The members have the following definitions:

- *service_context* – This contains ORB service data being passed from the client to the server, encoded as described in "Service Context" on page 79.
- *request_id*– The `request_id` member 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 84).
- *response_expected*– This member is set to `TRUE` if the request is expected to have an associated reply. The value is `FALSE` if the operation is defined as oneway, or if the operation is invoked with the DII and the invocation flags include the `INV_NO_RESPONSE` flag.
- *object_key*– This member identifies the object which is the target of the invocation. It is the `object_key` field from the `IIOP::ProfileBody` struct, 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.
- *operation*– This string contains the name of 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 defined in CORBA 1.2, the operation names are `_interface` and `_implementation`[4]. Each of these returns object references of a type defined in CORBA 1.2. This specification defines two more operations which need to be passed in IIOP, `_is_a` and `_not_exists`. (See "Extensions to “orb.idl”" on page 100 for call syntax.)

4. Since CORBA::Object::get_implementation is a null interface, clients must narrow the object reference they get to some ORB-specific kind of ImplementationDef.

- *requesting_principal*– This member contains a value identifying the requesting principal. It is provided to support the `BOA::get_principal` operation.

Request Body

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 76. This item is only included if the operation's OMG IDL definition includes a context expression, and only includes context members as defined in that expression.

As a clarifying example, the request body for the following OMG IDL operation

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

would be equivalent to this structure:

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

8.4.4 Reply Message

Reply messages are sent in response to Request messages. Replies include inout and out parameters, operation results, and may include exception values. In addition, Reply messages may provide object location information. Replies flow from server to client.

Reply messages have three elements, encoded in this order:

- A IIOP message header,
- A *ReplyHeader*, and
- The reply body.

Reply Header

The reply header is defined as follows:

```
module IIOP {
    enum ReplyStatusType {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    struct ReplyHeader {
        ServiceContextList service_context;
        unsigned long      request_id;
        ReplyStatusType    reply_status;
    };
};
```

The members have the following definitions:

- *service_context*– contains ORB service data being passed from the server to the client, encoded as described in "Service Context" on page 79.
- *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 else directs the client to reissue the request to an object at some other location.

Reply Body

The reply body format is controlled by the value of reply_status. There are three 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 service context is followed by the exception that is raised by the operation, encoded as described in "Exception" on page 76. (Only the user defined exceptions listed in the operation's OMG IDL definition may be raised.)

- If the `reply_status` value is `LOCATION_FORWARD`, then the service context is followed by an object reference (IOR) encoded as described in "Object Reference" on page 76. The client ORB is responsible re-sending the original request to that (different) object. This resending is transparent to the client program making the request.

As a clarifying example, the reply body for a successful response (the value of `reply_status` is `NO_EXCEPTION`) to the *Request* example shown on page 82 would be equivalent to the following structure:

```
    struct example_reply {
        double      return_value;   // return value
        string      str;            // leftmost inout or out parameter
        Principal   p;              // ... to the rightmost
    };
```

Note that the `object_key` field in the `IIOP::ProfileBody` 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 IIOP profile may not have the same value as the `object_key` in the IIOP profile of the original object reference. For more detail on location forwarding, see "Object Location" on page 90.

8.4.5 CancelRequest Message

CancelRequest messages may be sent 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 IIOP message header, and
- A *CancelRequestHeader*.

Cancel Request Header

The cancel request header is defined as follows:

```
module IIOP {
    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.

8.4.6 LocateRequest Message

LocateRequest messages may be sent from a client to a server to determine the following regarding a specified object reference: *(a)* whether the object reference is valid, `(b)` whether the current server is capable of directly receiving requests for the object reference, and if not, *(c)* 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 IIOP message header, and
- A *LocateRequestHeader*

LocateRequest Header.

The LocateRequest header is defined as follows:

```
module IIOP {
    struct LocateRequestHeader {
        unsigned long       request_id;
        sequence <octet>    object_key;
    };
};
```

The members are defined as follows:

- *request_id* – This is used to associate *LocateReply* messages with *LocateRequest* ones. The client (requester) is responsible for generating values; see "Request Message" on page 80 for the applicable rules.
- *object_key* – This identifies the object being located. 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. This value is only meaningful to the server and is not interpreted or modified by the client.

See "Object Location" on page 90 for details on the usage of *LocateRequest*.

8.4.7 LocateReply Message

LocateReply messages are sent from servers to clients in response to *LocateRequest* messages.

A LocateReply message has three elements, encoded in this order:

- A IIOP message header,
- A *LocateReplyHeader*, and
- The locate reply body.

Locate Reply Header

The locate reply header is defined as follows:

```
module IIOP {
    enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    struct LocateReplyHeader {
        unsigned long       request_id;
        LocateStatusType    locate_status;
    };
};
```

The members have the following definitions:

- *request_id* – The `request_id` member 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.
  - `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` – a *LocateReply* body exists.

LocateReply Body

The body is empty unless the `LocateStatus` value is `OBJECT_FORWARD`, in which case the body contains an object reference (IOR) that may be used as the target for requests to the object specified in the LocateRequest message.

8.4.8 CloseConnection Message

CloseConnection messages are sent only by servers. 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 awaiting replies will never be processed, and may safely be reissued (on another connection).

The *CloseConnection* message consists only of the IIOP message header, identifying the message type.

For details on the usage of *CloseConnection* messages, see "Connection Management" on page 88.

8.4.9 MessageError Message

The *MessageError* message is sent in response to any IIOP message whose version number or message type is unknown to the recipient, or any message is 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 IIOP message header, identifying the message type.

8.5 Internet IOP Message Transport

The IIOP is designed to be implementable on a wide range of transport protocols. The IIOP definition makes the following assumptions regarding transport behavior:

- The transport is connection-oriented. IIOP 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. (However, see "Connection Management" on page 88 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.

8.5.1 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.

Connections are not symmetrical. Only clients can send *Request*, *LocateRequest*, and *CancelRequest* messages over a connection. Only a server can send *Reply, LocateReply* and *CloseConnection* messages over a connection. Either client or server can send *MessageError* messages.

Only IIOP messages are sent over IIOP 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.

Connection Closure

Connections can be closed in two ways: orderly shutdown, or abortive disconnect. 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. If there are pending requests when a client shuts down a connection, the server should consider all such requests canceled. 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 receives an *CloseConnection* message from the server, it should assume that any outstanding messages (i.e., without replies) were received after the server sent the *CloseConnection* message, were not processed, and may be safely resent on a new connection.

After reliably issuing a *CloseConnection* message, the server may close the connection.

If a client detects connection closure without receiving a *CloseConnection* message, it should assume an abortive disconnect has occurred, and treat the condition as an error. Specifically, it should report COMM_FAILURE exceptions for all pending requests on the connection, with completion_status values set to COMPLETED_MAYBE.

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 re-using 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.

8.5.2 Message Ordering

Only the client (connection originator) may send *Request*, *LocateRequest*, and *CancelRequest* messages. Connections are not fully symmetrical.

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.

8.5.3 Baseline Transport Protocol

The baseline transport specified for IIOP 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.

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 "Object Reference" on page 76. 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 a object's services must initiate a connection with the address specified in the IOR, with a connect request.

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).

5. Postel, J., "Transmission Control Protocol – DARPA Internet Program Protocol Specification", RFC-793, Information Sciences Institute, September 1981

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.

After sending (or receiving) a *CloseConnection* message, both client or server must close the TCP/IP connection.

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.

8.6 Object Location

The IIOP 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*, etc.). 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 IIOP'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 return *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 *LocationRequest* 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 (obviously) be prepared to accept *LocateReply* messages.

A client should not make any assumptions about the longevity of object addresses returned by location forwarding mechanisms. Once a connection based on location forwarding information is closed, subsequent attempts to send requests to the same object should start with the original address specified in the initial object reference.

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.

New OMG IDL Primitive Types A

This appendix is not normative. It codifies enhancements to CORBA 1.2 for the support of several new fundamental OMG IDL data types. While these extensions are not required for interoperability, they are recommended for future consideration by the ORB Task Force.

Certain data types represent commonly implemented or requested extensions to the OMG Interface Definition Language (IDL). In order to define the semantics associated with those data types in all ORBs, and hence to ensure their interoperability, we describe the way to define and use those data types. They are in two categories: new arithmetic types, and support for wide characters.

Adding such support to CORBA 1.2 also involves changes to TypeCodes, language mappings, and the IIOP. These are described below.

A.1 New Arithmetic Types

We define extended integer and floating point data types, reflecting common usage. No new keywords are defined in order to support these new arithmetic data types.

A.1.1 IDL Grammar Productions

The following IDL grammar productions are modified by the addition of the text shown in *italics*:

(38) <floating_pt_type>	**::= "float"**
	| "double"
	| "long" "double"
(40) <signed_int>	**::= <signed_long_int>**
	| <signed_short_int>

| <signed_longlong_int>

(43) <unsigned_int> ::= <unsigned_long_int>
| <unsigned_short_int>
| <signed_longlong_int>

The following new IDL productions are defined:

(40.5) <signed_longlong_int> ::= “long” “long”
(43.5) <unsigned_longlong_int> ::= “unsigned” “long” “long”

A.1.2 Type Declaration

The `long long` data type represents signed integers in the range $-2^{63}..2^{63}-1$, while the `unsigned long long` data type represents unsigned integers in the range $0..2^{64}-1$.

The long double type represents an IEEE extended precision floating point number. In particular, it uses 15 bits of biased exponent and 112 bits of fraction. (This format is sometimes called “quad precision”.)

A.2 Wide Character Support

We define character and string types, based on the 16-bit Unicode Version 1.1 character set. This is the international standard ISO/IEC 10646-1:1993(E); Coded Representation Form=UCS-2; Subset=300; Implementation Level=3.

A.2.1 Keywords

Two new identifiers are reserved as keywords, and may not be used otherwise. These identifiers are `wchar` and `wstring`.

A.2.2 Literals

Since IDL definitions are always produced in the ISO Latin-1 (8859.1) character set, literals for wide characters and strings of wide characters may not be directly entered into IDL definitions text.

However, numeric forms of wide character constants may be used.

A.2.3 IDL Grammar Productions

The following IDL grammar productions are modified by the addition of the lines shown in *italics*:

(31) <base_type_spec> ::= <floating_pt_type>
| <integer_type>
| <char_type>

	| <wide_char_type>
	| <boolean_type>
	| <octet_type>
	| <any_type>
(32) <template_type_spec>	**::= <sequence_type>**
	| <string_type>
	| <wide_string_type>

The following new IDL productions are defined:

(46.5) <wide_char_type>	**::= “wchar”**
(62.5) <wide_string_type>	**::= “wstring”**
	| “wstring” “<“ <positive_int_const> “>”

A.2.4 Type Declaration

IDL defines a `wchar` type consisting of 16 bit quantities. These characters are in the Unicode 1.1 character set, and in the encoding described above. During transmission, and in particular language bindings, the representation of these wide characters may change but the meaning remains the same.

The `wstring` data type is an ordered set of zero or more `wchar` values, except 0x0000. It is similar to a sequence of `wchar`, differing in the same ways that the string data type differs from a sequence of *char*. An upper bound to the size of that set of characters may be defined.

A.3 TypeCode Extensions

New `CORBA::TCKind` values are defined as shown here, with TypeCode parameters as described.

```
module CORBA {
    ...
    enum TCKind {
        ... existing CORBA 1.2 TCKind enums, followed by:
        tk_longlong, tk_ulonglong, tk_longdouble,
        tk_wchar, tk_wstring
    };
    ...
};
```

TypeCodes with these four TCKind values have no parameters, and describe data values as follows:

- `tk_longlong`... a 64 bit two's complement signed integer.
- `tk_ulonglong`... a 64 bit two's complement unsigned integer
- `tk_longdouble`... a 128 bit IEEE extended precision floating point number
- `tk_wchar`... ISO/IEC 10646-1:1993(E), UCS-2 (Unicode 1.1)

TypeCodes with this TCKind value have a single parameter, describing the maximum length of a string.

- `tk_wstring`... a sequence of nonzero wchar values. the sequence may be bounded, or of arbitrary length.

A.4 Requirements for a Language Mapping

All IDL language mappings are required to support these new primitive types.

A.5 IIOP Extensions

These extended IDL primitive types require extensions only to the Common Data Representation, CDR. Those CDR extensions are described here.

Since IDL specifications using these extensions are incompatible with older ones not using them, all IIOP messages using these extensions will use the minor version number one (1) rather than the minor version number zero (0). (See "Object Reference" on page 76 for more information about the major and minor version numbers in the IIOP.)

A.5.1 CDR Primitive Encodings

The numeric types and wide character types are all CDR primitives.

Alignment

As with all CDR data types, these new data types consist of integral numbers of octets, and must be aligned on their natural boundaries. Those alignment restrictions are summarized in Table A-1.

Table A-1 Alignment requirements for extended IDL primitive data types

Type	Octet Alignment
`wchar`	2
`long long`	8
`unsigned long long`	8
`long double`	16

Note that the addition of the “long double” data type adds a new possible alignment value (16 octets) to CDR.

Integer Data Types

Figure A-1 illustrates the representations for the extended integer types, in both big-endian and little-endian mode. Signed types (`long long`) are represented as two’s complement numbers; unsigned ones (`unsigned long long`) are represented as unsigned binary numbers.

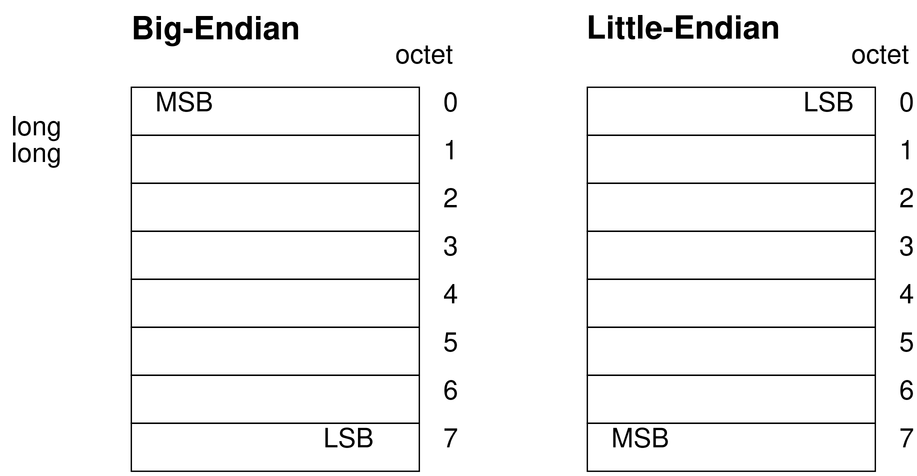

Figure A-1 Sizes and bit ordering in big and little endian encodings of the “long long” extended IDL integer data types, both signed and unsigned

Floating Point Data Type

Figure A-2 illustrates the representations for the extended floating point type, in both big-endian and little-endian mode. As with IDL’s float and double data types, these follow in detail the IEEE standard.

For these extended precision floating point values (sometimes called "quad precision"), the exponent (*e1-e2* in Figure A-2) is 15 bits long, where the 7 bits in *e1* are most significant. The exponent is represented as excess 16383. The fractional mantissa (*f1-f14*) is 112 bits long, with *f1* being the most significant. There is a single sign bit. That is, a normalized value is described by:

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

Big-Endian	octet	Little-Endian	octet
s | e1	0	f14	0
e2	1	f13	1
f1	2	f12	2
f2	3	f11	3
f3	4	f10	4
f4	5	f9	5
f5	6	f8	6
f6	7	f7	7
f7	8	f6	8
f8	9	f5	9
f9	10	f4	10
f10	11	f3	11
f11	12	f2	12
f12	13	f1	13
f13	14	e2	14
f14	15	e1 | s	15

Figure A-2 Sizes and bit ordering in big-endian and little-endian representations of the extended IDL floating point data type.

Wide Character Data Type

The wide character data type is described in ISO/IEC 1064601:1993(E), UCS-2 as a 16 bit numeric code. These numeric codes are encoded exactly like CDR "short" data types are, in both big-endian and little-endian representations.

A.5.2 Wide Character Strings

The wide string data type is a constructed CDR type. It is encoded as an IDL `sequence<short>`, including a terminating null (0x0000) character in the encoded count. Such a terminating null character is not counted in any bounds; that is, the length of the sequence as encoded in CDR can be one greater than the maximum allowed by the IDL definition of a bounded-length wide character string.

A.5.3 CDR TypeCode Support

In order to support these new extended IDL primitive data types, CDR's typecodes are extended as summarized below in Table A-2, which is an

Table A-2 TypeCode Representations for Extended IDL Primitive Types

TCKind	Integer Value	Type	Parameters
`tk_longlong`	21	empty	– none –
`tk_ulonglong`	22	empty	– none –
`tk_longdouble`	23	empty	– none –
`tk_wchar`	24	empty	– none –
`tk_wstring`	25	simple	ulong (max length)

extension to Table 8-2 on page 74. Note that only the `tk_wstring` typecode requires a parameter; that parameter is the upper bound on the length of such a wide character string, as defined in IDL. Its simple parameter list is not encapsulated when values of this `TypeCode` are encoded.

IDL for New Facilities B

This appendix is not normative. It gathers together in a single place all of the IDL and pseudo-IDL text used elsewhere in this document. The text in those locations is normative, rather than what is given here; the purpose of this appendix is to provide a summary of those extensions.

B.1 Extensions to “orb.idl”

The following summarizes extensions to the “orb.idl” file; these are detailed in"Interface, Operation, and Exception Equivalence" on page 51 and on "Managing Object References" on page 59.

```
module CORBA {
    exception OBJECT_NOT_EXIST {      // new system exception
        unsigned long      minor;
        completion_status  completed;
    };

    pseudo interface Object {
        // ... all existing operations, and:
        boolean            is_a (in string logical_type_id);
        readonly attribute boolean
                           non_existent ();
        unsigned long      hash (in unsigned long maximum);
        boolean            is_equivalent (in Object other_object);
    };
};
```

Note that the `ServerRequest` pseudo-object is not listed in *orb.idl*, but does appear in language mapping support.

B.2 ior.idl

The following summarizes a standardized representation of object references that may be used when bridging domain boundaries. (See "Interoperable Object References: IORs" on page 30.)

Note – These IDL definitions are only used to describe data structures for ORB internal mechanisms. They do not appear in any user level programming API. Their inclusion in any standard header file is unnecessary.

```
module IOR {
    //
    // Standard Protocol Profile tag values
    //
    typedef unsigned long             ProfileId;
    const ProfileId                   TAG_INTERNET_IOP = 0;
    const ProfileId                   TAG_MULTIPLE_COMPONENTS = 1;

    struct TaggedProfile {
        ProfileId                     tag;
        sequence <octet>              profile_data;
    };

    //
    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.
    //
    struct IOR {
        string                        type_id;
        sequence <TaggedProfile>      profiles;
    };
```

```
    //
    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedComponent.
    //
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId                  tag;
        sequence <octet>             component_data;
    };
    typedef sequence <TaggedComponent>MultipleComponentProfile;
};
```

B.3 iiop.idl

The following summarizes the IDL definitions used in the *Internet Inter-ORB Protocol* (IIOP; see "Internet IOP" on page 63).

Note – These IDL definitions are only used to describe data structures for ORB internal mechanisms. They do not appear in any user level programming API. Their inclusion in any standard header file is unnecessary.

```
module IIOP {
    //
    // An ObjectLocator is encapsulated in a IIOP profile
    // within an IOR.
    //
    struct Version { char major, minor; };
    struct ObjectLocator {
        Version            iiop_version;
        string             host;
        unsigned short     port;
        sequence <octet>   object_key;
    };
```

```
//
// All IIOP messages include a header and message type.
//
enum MsgType {
    Request,Reply, CancelRequest,
    LocateRequest,LocateReply,
    CloseConnection,MessageError
};
struct MessageHeader {
    char                magic [4];
    Version             iiop_version;
    char                byte_order;
    char                message_type;
    unsigned long       message_size;
};

//
// Support for Implicit ORB Service Context
//
typedef unsigned long ServiceID;
struct ServiceContext {
    ServiceID           context_id;
    sequence <octet>    context_data;
};
typedef sequence <ServiceContext> ServiceContextList;

const ServiceID         TransactionService = 0;
```

```
//
// Request, Reply headers
//
struct RequestHeader {
    ServiceContextList service_info;
    unsigned long      request_id;
    boolean            response_expected;
    sequence <octet>   object_key;
    string             operation;
    Principal          requesting_principal;
};

enum ReplyStatusType {
    NO_EXCEPTION,
    USER_EXCEPTION,
    SYSTEM_EXCEPTION,
    LOCATION_FORWARD
};

struct ReplyHeader {
    ServiceContextList service_info;
    unsigned long      request_id;
    ReplyStatusType    reply_status;
};

//
// Cancellation -- applies both to Requests and LocateRequests.
//
struct CancelRequestHeader {
    unsigned long      request_id;
};
```

```
    //
    // Location service support -- request and reply messages
    //
    struct LocateRequestHeader {
        unsigned long      request_id;
        sequence <octet>   object_key;
    };

    enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    struct LocateReplyHeader {
        unsigned long      request_id;
        LocateStatusType   locate_status;
    };

    //
    // CloseConnection and MessageError messages have no
    // data, consisting only of a header.
    //
};
```

Note that the encodings of IDL primitive data types are not expressed here in IDL, only the message formats themselves can be (largely) declared in IDL.

Issues for Future CORBA Work C

This appendix is not normative. It summarizes issues identified during the preparation of this submission, which need to be taken into account in the future evolution of CORBA.

C.1 Desirable CORBA Extensions

There are several areas where we have noted deficiencies in the CORBA interfaces which seem to cause deficiencies in the portability of application software, including in particular request level bridges, using ORBs.

C.1.1 IDL extensions

In addition to the new primitive data types discussed in "New OMG IDL Primitive Types" on page 92, it appears some convention would be useful to help assign and manage type IDs.

C.1.2 Implementation Repository

As noted in "Internet IOP" on page 63, clients using the `CORBA::Object::get_implementation()` operation must narrow this object reference to an ORB-specific type before using it. This is at worst awkward, and in general renders use of `CORBA::ImplementationDef` object references wholly nonportable.

Some standard interface should be defined for use by clients. Such an interface might allow clients to “break the transparencies” provided by the ORB.

C.1.3 BOA Portability

The programming interface provided by the BOA is insufficient to enable the construction of fully portable applications, including fully portable request level bridges.

A minimal portable specification would define how implementations connect themselves and their skeletons (both static and dynamic) to the BOA, and a threading model.

C.1.4 Implicit Context

It is felt that there may be utility in defining a general API though which services can securely provide implicit context to ORBs, so that new object services can be built which use implicit context to achieve new quality of service goals. This API needs to work equally well for both static (stub and skeleton based) and dynamic (DII, DSI) styles of object invocation, and in particular must allow bridges to fully transform the contents of any such implicit context during the transport of requests between domains.

C.1.5 Topology Service

As more domains, in particular object reference domains, are bridged and the connectivity of ORBs becomes richer, it may become more desirable to define what are variously called location, resolution and topology services. Such services will, when given an object name or reference, return optimized addressing and routing information for that object. This allows route optimization even when non-global or non-uniform addressing is used in the overall configuration of ORBs. It could be thought of as an OMG global object addressing service.

C.2 Constraints on CORBA Extensions

The OMG standards suite is strongly interrelated, and extensions in any one area could in theory compromise another. Here we identify several issues that future standardization efforts ought to consider, so to ensure continued ORB interoperability in the face of evolved specifications.

C.2.1 CORBA Object Model

The fundamental way that CORBA promotes interoperability is to define a single object model. So long as systems provide exactly that object model, potentially with controlled, standardized extensions, these different ORBs may safely and reliably be bridged.

However, that object model was originally not well specified in several areas. Notable zones of ambiguity surrounded the use of type identifiers, which are exposed in TypeCodes (e.g. for object references) and in exception IDs. In a similar way, the object model accessed through IDL-generated stubs and skeletons was richer than that accessible through the dynamic interfaces.

This specification has resolved the issues that affect interoperability, well enough that all of the IDL functionality is equally accessible through the static and dynamic APIs. However, future changes should be made with extreme care so as not to compromise interoperability.

C.2.2 Pseudo-Objects

If subsequent OMG specifications define pseudo-objects that may be passed as general arguments, or passed implicitly during requests, the specification should define a pseudo-object state in IDL. That IDL pseudo-object state will describe how the pseudo-object is encoded in CDR. Internal pseudo-object representations (i.e. ones that may not be passed as general arguments, or implicitly) are implementation-dependent, determined by the ORB.

C.2.3 Service-specific Implicit Context

The IIOP describes a mechanism for passing encapsulated service-specific context information in requests and replies. This mechanism requires that services define the context in IDL, and that each such service is allocated a unique service ID used to describe the context information in requests.

C.2.4 TypeCodes

If new data types are defined in ORB extensions, the specifications must describe how the TypeCodes for the data types are encoded in CDR. The TypeCode encodings must be consistent with the general forms described in "TypeCode" on page 72. Specifically, TypeCodes with multiple variable- length parameters, nested TypeCodes, or a variable number of parameters must encapsulate the parameters in octet sequences.

C.2.5 Primitives

If ORB extensions define new primitive data types, the specifications must define CDR encodings and TypeCodes for the new data types. The specifications should be generally consistent with CDR’s existing representations and alignment rules.

C.2.6 Protocol Extensions

The IIOP is designed to be extensible. Extensions that are intended to be required as part of basic ORB interoperability should be defined as new IIOP version numbers. Other extensions may be defined as optional alternatives, such as IIOP extensions for message fragmentation, or optional support for multiple floating-point formats. These optional extensions should be defined as new profiles in IORs with unique profile identifiers.