General Inter-ORB Protocol 13

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

Contents

This chapter contains the following sections.

13.1 Goals of the General Inter-ORB Protocol

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

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

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

13.2 GIOP Overview

The GIOP specification consists of the following elements:

- *The Common Data Representation (CDR) definition*. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for "on-the-wire" transfer between ORBs and Inter-ORB bridges (agents).
- *GIOP Message Formats*. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- *GIOP Transport Assumptions*. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

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

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

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

The complete OMG IDL specifications for the GIOP and IIOP are shown in Section 13.8, "OMG IDL," on page 13-37. Fragments of the specification are used throughout this chapter as necessary.

13.2.1 Common Data Representation (CDR)

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

- ***Variable byte ordering*** - Machines with a common byte order may exchange messages without byte swapping. When communicating machines have different byte order, the message originator determines the message byte order, and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message (and CDR encapsulation) contains a flag that indicates the appropriate byte order.
- ***Aligned primitive types*** - Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.
- ***Complete OMG IDL Mapping*** - CDR describes representations for all OMG IDL data types, including transferable pseudo-objects such as TypeCodes. Where necessary, CDR defines representations for data types whose representations are undefined or implementation-dependent in the CORBA Core specifications.

13.2.2 GIOP Message Overview

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

- ***Few, simple messages.*** With only seven message formats, the GIOP supports full CORBA functionality between ORBs, with extended capabilities supporting object location services, dynamic migration, and efficient management of communication resources. GIOP semantics require no format or binding negotiations. In most cases, clients can send requests to objects immediately upon opening a connection.
- ***Dynamic object location.*** Many ORBs' architectures allow an object implementation to be activated at different locations during its lifetime, and may allow objects to migrate dynamically. GIOP messages provide support for object location and migration, without requiring ORBs to implement such mechanisms when unnecessary or inappropriate to an ORB's architecture.

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

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

13.2.3 GIOP Message Transfer

The GIOP specification is designed to operate over any connection-oriented transport protocol that meets a minimal set of assumptions (described in "GIOP Message Transport" on page 13-30). GIOP uses underlying transport connections in the following ways:

- ***Asymmetrical connection usage*** - The GIOP defines two distinct roles with respect to connections, client and server. The client side of a connection originates the connection, and sends object requests over the connection. The server side receives requests and sends replies. The server side of a connection may not send object requests. This restriction allows the GIOP 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, GIOP message ordering constraints are minimal. GIOP is designed to allow overlapping asynchronous requests; it does not dictate the relative ordering of requests or replies. Unique request/reply identifiers provide proper correlation of related messages. Implementations are free to impose any internal message ordering constraints required by their ORB architectures.
- ***Connection management*** - GIOP defines messages for request cancellation and orderly connection shutdown. These features allow ORBs to reclaim and reuse idle connection resources.

13.3 CDR Transfer Syntax

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

An octet stream is an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport. For the purposes of this discussion, an octet stream is an arbitrarily long (but finite) sequence of eight-bit values (octets) with a well-defined beginning. The octets in the stream are numbered from *0* to *n-1*, where *n* is the size of the stream. The numeric position of an octet in the stream is called its *index*. Octet indices are used to calculate alignment boundaries, as described in "Alignment" on page 13-5.

GIOP defines two distinct kinds of octet streams, messages and encapsulations. Messages are the basic units of information exchange in GIOP, described in detail in "GIOP Message Formats" on page 13-19.

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

13.3.1 Primitive Types

Primitive data types are specified for both big-endian and little-endian orderings. The message formats (see "GIOP Message Formats" on page 13-19) 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 13-12. 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.

The transfer syntax for an IDL wide character depends on whether the transmission code set (TCS-W, which is determined via the process described in "Code Set Conversion" on page 11-22) is byte-oriented or non-byte-oriented:

- Byte-oriented (e.g., SJIS). Each wide character is represented as one or more octets, as defined by the selected TCS-W.
- Non-byte-oriented (e.g., Unicode UTF-16). Each wide character is represented as one or more codepoints. A codepoint is the same as "Coded-Character data element," or "CC data element" in ISO terminology. Each codepoint is encoded using a fixed number of bits as determined by the selected TCS-W.

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 13-1 gives alignment boundaries for CDR/OMG-IDL primitive types.

Table 13-1 Alignment requirements for OMG IDL primitive data types

TYPE	OCTET ALIGNMENT
char	1
wchar	1, 2, or 4, depending on code set
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enum	4

Alignment is defined above as being relative to the beginning of an octet stream. The first octet of the stream is octet index zero (0); any data type may be stored starting at this index. Such octet streams begin at the start of an GIOP message header (see “GIOP Message Header” on page 13-19) and at the beginning of an encapsulation, even if the encapsulation itself is nested in another encapsulation. (See “Encapsulation” on page 13-12).

Integer Data Types

Figure 13-1 on page 13-7 illustrates the representations for OMG IDL integer data types, including the following data types:

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

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

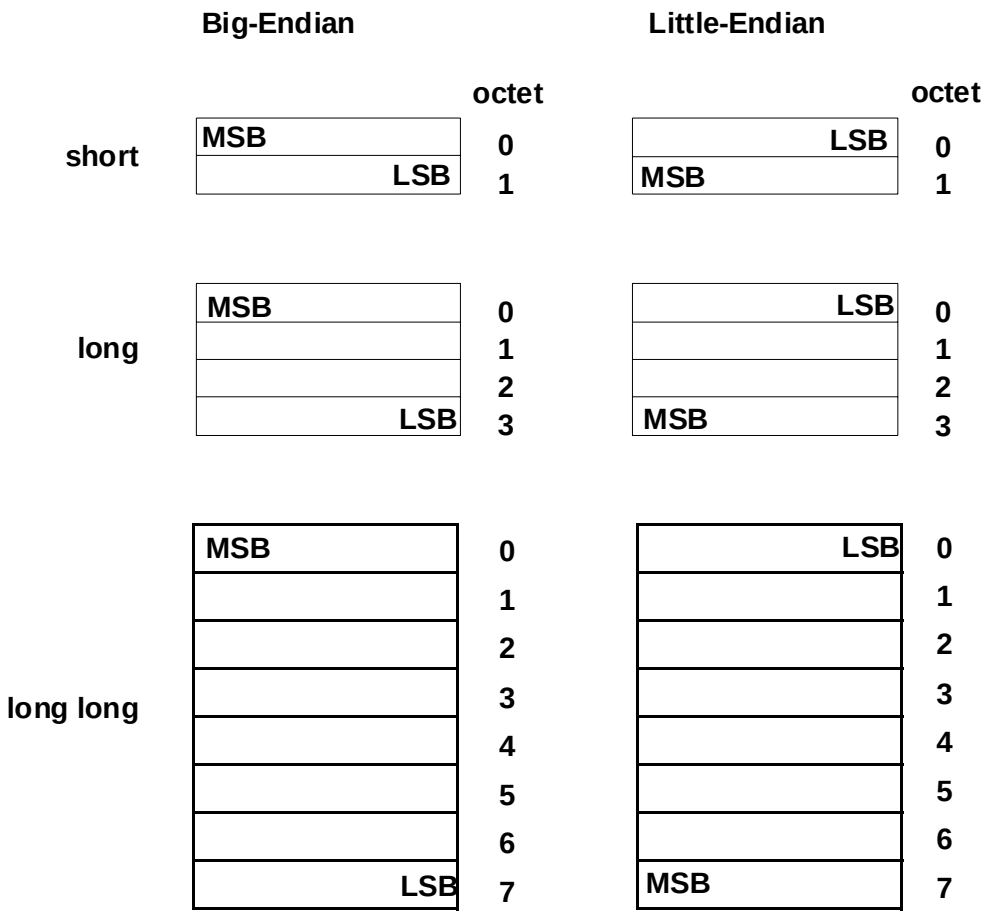

Figure 13-1 Sizes and bit ordering in big-endian and little-endian encodings of OMG IDL integer data types, both signed and unsigned.

Floating Point Data Types

Figure 13-2 on page 13-9 illustrates the representation of floating point numbers. These exactly follow the IEEE standard formats for floating point numbers[1], selected parts of which are abstracted here for explanatory purposes. The diagram shows three different components for floating points numbers, the sign bit (s), the exponent (e) and the fractional part (f) of the mantissa. The sign bit has values of 0 or 1, representing positive and negative numbers, respectively.

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

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

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

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

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

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

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

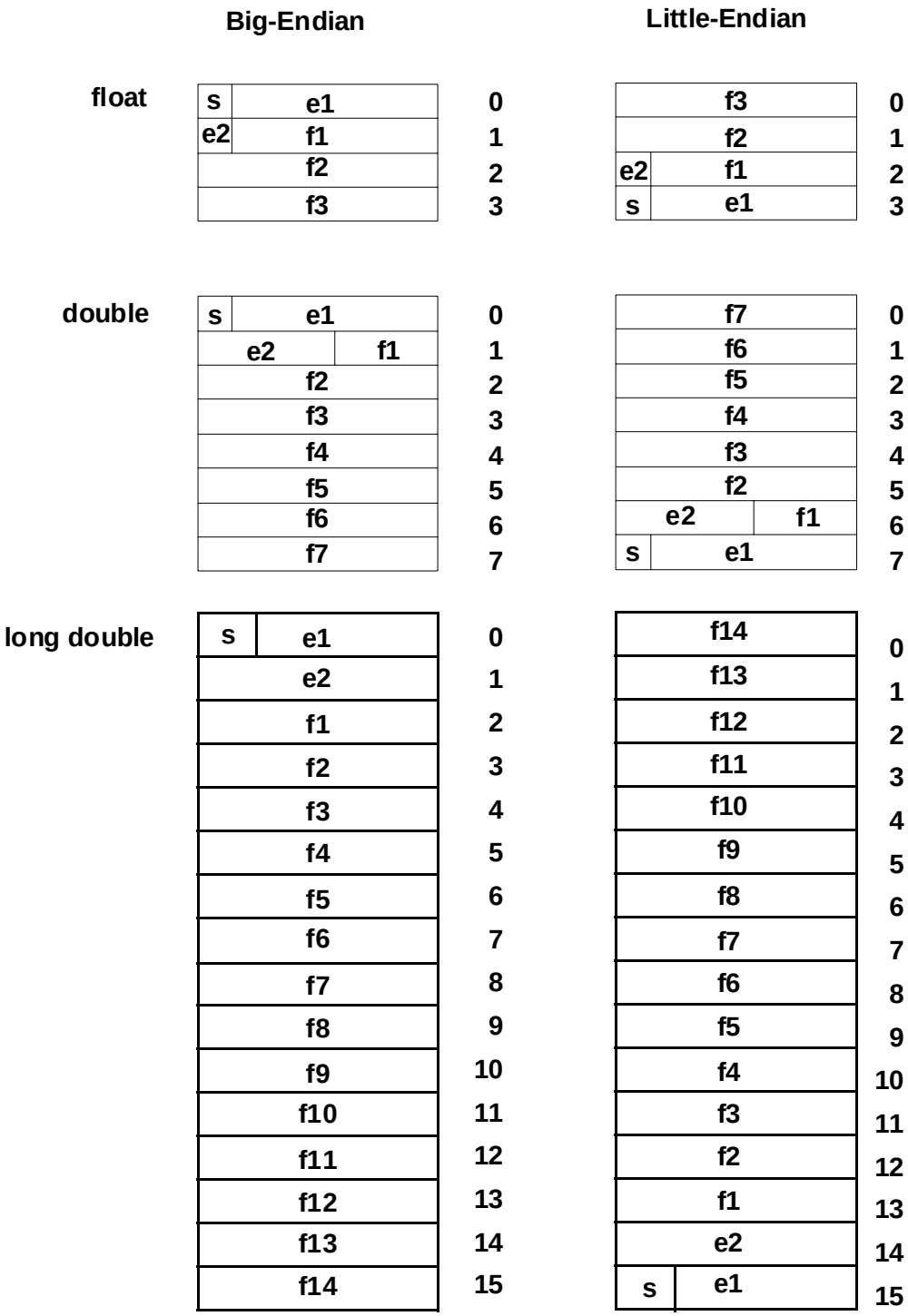

Figure 13-2 Sizes and bit ordering in big-endian and little-endian representations of OMG IDL single, double precision, and double extended floating point numbers.

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

An IDL character is represented as a single octet; the code set used for transmission of character data (e.g., TCS-C) between a particular client and server ORBs is determined via the process described in Section 11.7, “Code Set Conversion,” on page 11-22. Note that multi-byte characters will require the use of an array of IDL **char** variables.

The transfer syntax for an IDL wide character depends on whether the transmission code set (TCS-W, which is determined via the process described in “Code Set Conversion” on page 11-22) is byte-oriented or non-byte-oriented:

- Byte-oriented (e.g., SJIS). Each wide character is represented as one or more octets, as defined by the selected TCS-W.
- Non-byte-oriented (e.g., Unicode UTF-16). Each wide character is represented as one or more codepoints. A codepoint is the same as “Coded-Character data element,” or “CC data element” in ISO terminology. Each codepoint is encoded using a fixed number of bits as determined by the selected TCS-W. The OSF Character and Code Set Registry may be examined using the interfaces in Appendix 10B on page 10-37 to determine the maximum length (max_bytes) of any character codepoint. For example, if the TCS-W is ISO 10646 UCS-2 (Universal Character Set containing 2 bytes), then wide characters are represented as **unsigned shorts**. For ISO 10646 UCS-4, they are represented as **unsigned longs**.

13.3.2 OMG IDL Constructed Types

Constructed types are built 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. GIOP assumes that agents will usually construct structured data types by copying primitive data between the marshaled buffer and the appropriate in-memory data structure layout for the language mapping implementation involved.

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.

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.

Strings and Wide Strings

A string is encoded as an **unsigned long** indicating the length of the string in octets, followed by the string value in single- or multi-byte form represented as a sequence of octets. Both the string length and contents include a terminating null.

A wide string is encoded as an **unsigned long** indicating the length of the string in octets or unsigned integers (determined by the transfer syntax for **wchar**) followed by the individual wide characters. Both the string length and contents include a terminating null. The terminating null character for a **wstring** is also a wide character.

Fixed-Point Decimal Type

The IDL **fixed** type has no alignment restrictions, and is represented as shown in Figure 13-3 on page 13-12. Each octet contains (up to) two decimal digits. If the **fixed** type has an odd number of decimal digits, then the representation begins with the first

(most significant) digit — d0 in the figure. Otherwise, this first half-octet is all zero, and the first digit is in the second half-octet — d1 in the figure. The sign configuration, in the last half-octet of the representation, is 0xD for negative numbers and 0xC for positive and zero values.

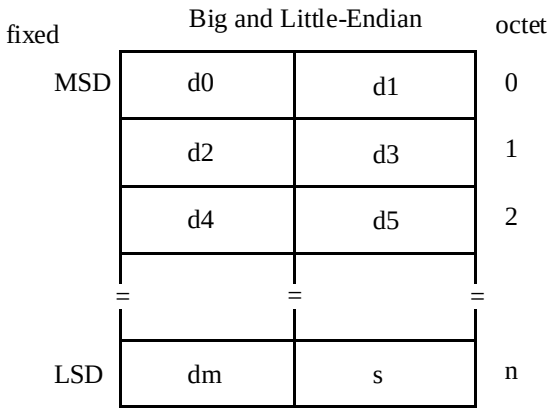

Figure 13-3 IDL Fixed Type Representation

13.3.3 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 a GIOP message or nested in another encapsulation.

The GIOP and IIOP explicitly use encapsulations in three places: *TypeCodes* (see "TypeCode" on page 13-13), the IIOP protocol profile inside an IOR (see "Object References" on page 13-18), and in service-specific context (see "Object Service Context" on page 11-20). In addition, some ORBs may choose to use an encapsulation to hold **Principal** identification information (see "Principal" on page 13-18), the **object_key** (see "IIOP IOR Profiles" on page 13-34), or in other places that a **sequence<octet>** data type is in use.

When encapsulating OMG IDL data types, the first octet in the stream (index 0) contains a boolean value indicating the byte ordering of the encapsulated data. If the value is FALSE (0), the encapsulated data is encoded in big-endian order; if TRUE (1), the data is encoded in little-endian order, exactly like the byte order flag in GIOP message headers (see "GIOP Message Header" on page 13-19). 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 13-11). The length value is not part of the encapsulation's octet stream, and does not affect alignment of data within the encapsulation.

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

13.3.4 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 13-2. 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.
- Typecodes with *simple parameter lists* are encoded as the **TCKind** enum value followed by the parameter value(s), encoded as indicated in Table 13-2. 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** value **tk_string** has such a parameter list.
- 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 13-12) containing the encapsulated, marshaled parameters. The order of these parameters is shown in the fourth column of Table 13-2.

---

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 GIOP protocol itself does not require encapsulation of such data.

The third column of Table 13-2 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 13-17. This indirection does not need to be exposed to application programmers.

TypeCode Parameter Notation

TypeCode parameters are specified in the fourth column of Table 13-2. The ordering and meaning of parameters is a superset of those given in the Interface Repository specification (Chapter 8); 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_except**) contain a counted sequence of tuples.

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

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

Table 13-2 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_longlong`	23	empty	-none-
`tk_ushort`	4	empty	– none –

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

Table 13-2 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
`tk_ulong`	5	empty	– none –
`tk_ulonglong`	24	empty	-none-
`tk_fixed`	28	simple	ushort(digits), short(scale)
`tk_float`	6	empty	– none –
`tk_double`	7	empty	– none –
`tk_longdouble`	25	empty	-none-
`tk_boolean`	8	empty	– none –
`tk_char`	9	empty	– none –
`tk_wchar`	26	empty	-none-
`tk_octet`	10	empty	– none –
`tk_any`	11	empty	– none –
`tk_TypeCode`	12	empty	– none –
`tk_Principal`	13	empty	– none –
`tk_objref`	14	complex	string (repository ID), string(name)
`tk_struct`	15	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
`tk_union`	16	complex	string (repository ID), string(name), TypeCode (discriminant type), long (default used), ulong (count) {*discriminant type*[1] (label value), string (member name), TypeCode (member type)}
`tk_enum`	17	complex	string (repository ID), string (name), ulong (count) {string (member name)}
`tk_string`	18	simple	ulong (max length[2])
`tk_wstring`	27	simple	ulong(max length or zero if unbounded)

Table 13-2 TypeCode enum values, parameter list types, and parameters

TCKind	Integer Value	Type	Parameters
`tk_sequence`	19	complex	TypeCode (element type), ulong (max length[3])
`tk_array`	20	complex	TypeCode (element type), ulong (length)
`tk_alias`	21	complex	string (repository ID), string (name), TypeCode
`tk_except`	22	complex	string (repository ID), string (name), ulong (count) {string (member name), TypeCode (member type)}
`– none –`	0xffffffff	simple	long (indirection[4])

1. The type of union label values is determined by the second parameter, discriminant type.

2. For unbounded strings, this value is zero.

3. For unbounded sequences, this value is zero.

4. See "Indirection: Recursive and Repeated TypeCodes" on page 13-17.

Encoded Identifiers and Names

The Repository ID parameters in **tk_objref, tk_struct, tk_union, tk_enum, tk_alias**, and **tk_except** TypeCodes are Interface Repository `RepositoryId` values, whose format is described in the specification of the Interface Repository. RepositoryId values are required for **tk_objref** and **tk_except** TypeCodes; for other TypeCodes they are optional and are encoded as empty strings if omitted.

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

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.

TypeCodes for Multi-Dimensional Arrays

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

Indirection: Recursive and Repeated TypeCodes

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

CDR provides a constrained indirection to resolve this problem:

- The indirection applies only to TypeCodes nested within some “top level” TypeCode. Indirected TypeCodes are not “freestanding,” but only exist inside some other encoded TypeCode.
- 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 and points to the TCKind value for the typecode. (Note that the byte order of the TCKind value can be determined by its encoded value.) This indirection may well cross encapsulation boundaries, but this is not problematic because of the first constraint identified above. Because of the second constraint, the value of the offset will always be negative.

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

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

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 RepositoryId for the exception, as defined in the Interface Repository chapter. Exception members (if any) are encoded in the same manner as a struct.

If an ORB receives a non-standard system exception that it does not support, the exception shall be mapped to **UNKNOWN**.

13.3.5 Object References

Object references are encoded in OMG IDL (as described in “Object Addressing” on page 11-11). IOR profiles contain transport-specific addressing information, so there is no general-purpose IOR profile format defined for GIOP. Instead, this specification describes the general information model for GIOP profiles and provides a specific format for the IIOP (see “IIOP IOR Profiles” on page 13-34).

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

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

13.4 GIOP Message Formats

In describing GIOP 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 13-30) and originates requests. A server is an agent that accepts connections and receives requests.

GIOP message types are summarized in Table 13-3, which lists the message type names, whether the message is originated by client, server, or both, and the value used to identify the message type in GIOP message headers.

Table 13-3 GIOP Message Types and originators

Message Type	Originator	Value	GIOP Versions
`Request`	Client	0	1.0, 1.1
`Reply`	Server	1	1.0, 1.1
`CancelRequest`	Client	2	1.0, 1.1
`LocateRequest`	Client	3	1.0, 1.1
`LocateReply`	Server	4	1.0, 1.1
`CloseConnec-tion`	Server	5	1.0, 1.1
`MessageError`	Both	6	1.0, 1.1
`Fragment`	Both	7	1.1

13.4.1 GIOP Message Header

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

```
module GIOP { // IDL extended for version 1.1

    struct Version {
        octet       major;
        octet       minor;
    };

    #ifndef GIOP_1_1
    // GIOP 1.0
    enum MsgType_1_0 { // Renamed from MsgType
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError
    };
```

```
    #else
    // GIOP 1.1
    enum MsgType_1_1 {
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError,
        Fragment                        // GIOP 1.1 addition
    };
    #endif

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

    // GIOP 1.1
    struct MessageHeader_1_1 {
        char            magic [4];
        Version         GIOP_version;
        octet           flags;          // GIOP 1.1 change
        octet           message_type;
        unsigned long   message_size;
    };
};
```

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

- **magic** identifies GIOP messages. The value of this member is always the four (upper case) characters “GIOP,” encoded in ISO Latin-1 (8859.1).
- **GIOP_version** contains the version number of the GIOP protocol being used in the message. The version number applies to the transport-independent elements of this specification (i.e., the CDR and message formats) which constitute the GIOP. This is not equivalent to the IIOP version number (as described in “Object References” on page 13-18) though it has the same structure. The major GIOP version number of this specification is one (1); the minor versions are zero (0) and one (1).
- **byte_order** (in GIOP 1.0 only) indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of FALSE (0) indicates big-endian byte ordering, and TRUE (1) indicates little-endian byte ordering.

- **flags** (in GIOP 1.1) is an 8 bit octet. The least significant ibt indicates the byte ordering used in subsequent elements of the message (including **message_size**). A value of FALSE (0) indicates big-endian byte ordering, and TRUE (1) indicates little-endian byte ordering.

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

  The most significant 6 bits are reserved. All these 6 bits must have value 0 for GIOP version 1.1.

- **message_type** indicates the type of the message, according to Table 13-3; these correspond to enum values of type **MsgType**.

- **message_size** contains the number of octets in the message following the message header, encoded using the byte order specified in the byte order bit (the least significant bit) in the **flags** field (or using the bute_order field in GIOP 1.0). It refers to the size of the message body, not including the 12 byte message header. This count includes any alignment gaps. The use of a message size of 0 with a Request, LocateRequest, Reply, or LocateReply message is reserved for future use.

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 GIOP message header
- A Request Header
- The Request Body

Request Header

The request header is specified as follows:

```
module GIOP {                    // IDL extended for version 1.1

    // GIOP 1.0
    struct RequestHeader_1_0 { // Renamed from RequestHeader
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        sequence <octet>            object_key;
        string                      operation;
        Principal                   requesting_principal;
    };

    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        octet                       reserved[3]; // Added in GIOP 1.1
        sequence <octet>            object_key;
        string                      operation;
        Principal                   requesting_principal;
    };
};
```

The members have the following definitions:

- **service_context** contains ORB service data being passed from the client to the server, encoded as described in “Object Service Context” on page 11-20.
- **request_id** is used to associate reply messages with request messages (including LocateRequest messages). The client (requester) is responsible for generating values so that ambiguity is eliminated; specifically, a client must not re-use request_id values during a connection if: *(a)* the previous request containing that ID is still pending, or *(b)* if the previous request containing that ID was canceled and no reply was received. (See the semantics of the “CancelRequest Message” on page 13-26).
- **response_expected** is set to TRUE if a reply message is expected for this request. If the operation is not defined as oneway, and the request is not invoked via the DII with the INV_NO_RESPONSE flag set, the **response_expected** flag must be set to TRUE.

  If the operation is defined as oneway, or the request is invoked via the DII with the INV_NO_RESPONSE flag set, the **response_expected** flag may be set to TRUE or FALSE. Asking for a reply gives the client ORB an opportunity to receive LOCATION_FORWARD responses and replies that might indicate system exceptions. When this flag is set to TRUE for a oneway operation, receipt of a reply does not imply that the operation has necessarily completed.

- **reserved** is always set to 0 in GIOP 1.1. These three octets are reserved for future use.
- **object_key** identifies the object which is the target of the invocation. It is the **object_key** field from the transport-specific GIOP profile (e.g., from the encapsulated IIOP profile of the IOR for the target object). This value is only meaningful to the server and is not interpreted or modified by the client.
- **operation** is the IDL identifier naming, within the context of the interface (not a fully qualified scoped name), the operation being invoked. In the case of attribute accessors, the names are `_get_<attribute>` and `_set_<attribute>`. The case of the operation or attribute name must match the case of the operation name specified in the OMG IDL source for the interface being used.

  In the case of `CORBA::Object` operations that are defined in the CORBA Core ("Object Reference Operations" on page 4-4) and that correspond to GIOP request messages, the operation names are `_interface`, `_implementation`[4], `_is_a` and `_not_existent`.
- **requesting_principal** 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 13-18. 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.

For 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
};
```

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.

13.4.2 Reply Message

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

Reply messages have three elements, encoded in this order:

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

Reply Header

The reply header is defined as follows:

```
module GIOP {                                   // IDL
    enum ReplyStatusType {
    NO_EXCEPTION,
    USER_EXCEPTION,
    SYSTEM_EXCEPTION,
    LOCATION_FORWARD
    };

    struct ReplyHeader {
    IOP::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 “GIOP Message Transfer” on page 13-4.
- **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:

1 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.)

2 If the **reply_status** value is **USER_EXCEPTION** or **SYSTEM_EXCEPTION**, then the body contains the exception that was raised by the operation, encoded as described in "Exception" on page 13-18. (Only the user defined exceptions listed in the operation's OMG IDL definition may be raised.)

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

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

The high order 20 bits of **minor_code_value** contain a 20-bit "vendor minor codeset ID"(**VMCID**); the low order 12 bits contain a minor code. A vendor (or group of vendors) wishing to define a specific set of system exception minor codes should obtain a unique **VMCID** from the OMG, and then define up to 4096 minor codes for each system exception. Any vendor may use the special **VMCID** of zero (0) without previous reservation, but minor code assignments in this codeset may conflict with other vendor's assignments, and use of the zero **VMCID** is officially deprecated.

3 If the **reply_status** value is **LOCATION_FORWARD**, then the body contains an object reference (IOR) encoded as described in "Object References" on page 13-18. The client ORB is responsible for re-sending the original request to that (different) object. This resending is transparent to the client program making the request.

For example, the reply body for a successful response (the value of **reply_status** is **NO_EXCEPTION**) to the Request example shown on page 13-23 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 any specific GIOP profile is server-relative, not absolute. Specifically, when a new object reference is received in a **LOCATION_FORWARD** Reply or in a LocateReply message, the **object_key** field embedded in the new object reference's GIOP profile may not have the same value as the **object_key** in the GIOP profile of the original object reference. For details on location forwarding, see "Object Location" on page 13-32.

13.4.3 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 GIOP message header
- A CancelRequestHeader

Cancel Request Header

The cancel request header is defined as follows:

```
module GIOP {                           // IDL
    struct CancelRequestHeader {
    unsigned long           request_id;
    };
};
```

The **request_id** member identifies the **Request** or **LocateRequest** message to which the cancel applies. This value is the same as the **request_id** value specified in the original Request or LocateRequest message.

When a client issues a cancel request message, it serves in an advisory capacity only. The server is not required to acknowledge the cancellation, and may subsequently send the corresponding reply. The client should have no expectation about whether a reply (including an exceptional one) arrives.

13.4.4 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 GIOP message header
- A LocateRequestHeader

LocateRequest Header.

The **LocateRequest** header is defined as follows:

```
module GIOP {                                   // IDL
    struct LocateRequestHeader {
    unsigned long           request_id;
    sequence <octet>        object_key;
    };
};
```

The members are defined as follows:

- **request_id** is used to associate *LocateReply* messages with *LocateRequest* ones. The client (requester) is responsible for generating values; see “Request Message” on page 13-21 for the applicable rules.
- **object_key** identifies the object being located. In an IIOP context, this value is obtained from the **object_key** field from the encapsulated **IIOP::ProfileBody** in the IIOP profile of the IOR for the target object. When GIOP is mapped to other transports, their IOR profiles must also contain an appropriate corresponding value. This value is only meaningful to the server and is not interpreted or modified by the client.

See “Object Location” on page 13-32 for details on the use of **LocateRequest.**

13.4.5 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 GIOP message header
- A LocateReplyHeader
- The locate reply body

Locate Reply Header

The locate reply header is defined as follows:

```
module GIOP {                          // IDL
    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** - is used to associate replies with requests. This member contains the `same request_id` value as the corresponding *LocateRequest* message.
- **locate_status -** the value of this member is used to determine whether a **LocateReply** body exists. Values are:
  - **UNKNOWN_OBJECT -** the object specified in the corresponding **LocateRequest** message is unknown to the server; no body exists.
  - **OBJECT_HERE -** this server (the originator of the **LocateReply** message) can directly receive requests for the specified object; no body exists.
  - **OBJECT_FORWARD -** 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.

13.4.6 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 are awaiting replies will never be processed, and may safely be reissued (on another connection).

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

For details on the usage of **CloseConnection** messages, see "Connection Management" on page 13-30.

13.4.7 MessageError Message

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

13.4.8 Fragment Message

This message is added in GIOP 1.1.

The **Fragment** message is sent following a previous request or response message that has the more fragments bit set to TRUE in the **flags** field.

All of the GIOP messages begin with a GIOP header. One of the fields of this header is the **message_size** field, a 32-bit unsigned number giving the number of bytes in the message following the header. Unfortunately, when actually constructing a GIOP **Request** or **Reply** message, it is sometimes impractical or undesirable to ascertain the total size of the message at the stage of message construction where the message header has to be written. GIOP 1.1 provides an alternative indication of the size of the message, for use in those cases.

A **Request** or **Reply** message can be broken into multiple fragments. The first fragment is a regular message (e.g., **Request** or **Reply**) with the more fragments bit in the **flags** field set to TRUE. This initial fragment can be followed by one or more messages using the fragment messages. The last fragment shall have the more fragment bit in the flag field set to FALSE.

A **CancelRequest** message may be sent by the client before the final fragment of the message being sent. In this case, the server should assume no more fragments will follow.

A primitive data type of 8 bytes or smaller should never be broken across two fragments.

13.5 GIOP Message Transport

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

- The transport is connection-oriented. GIOP uses connections to define the scope and extent of request IDs.
- The transport is reliable. Specifically, the transport guarantees that bytes are delivered in the order they are sent, at most once, and that some positive acknowledgment of delivery is available.
- The transport can be viewed as a byte stream. No arbitrary message size limitations, fragmentation, or alignments are enforced.
- The transport provides some reasonable notification of disorderly connection loss. If the peer process aborts, the peer host crashes, or network connectivity is lost, a connection owner should receive some notification of this condition.
- The transport's model for initiating connections can be mapped onto the general connection model of TCP/IP. Specifically, an agent (described herein as a server) publishes a known network address in an IOR, which is used by the client when initiating a connection.

The server does not actively initiate connections, but is prepared to accept requests to connect (i.e., it *listens* for connections in TCP/IP terms). Another agent that knows the address (called a client) can attempt to initiate connections by sending *connect* requests to the address. The listening server may *accept* the request, forming a new, unique connection with the client, or it may *reject* the request (e.g., due to lack of resources). Once a connection is open, either side may *close* the connection. (See "Connection Management" on page 13-30 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.

13.5.1 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 GIOP messages are sent over GIOP connections.

Request IDs must unambiguously associate replies with requests within the scope and lifetime of a connection. Request IDs may be re-used if there is no possibility that the previous request using the ID may still have a pending reply. Note that cancellation does not guarantee no reply will be sent. It is the responsibility of the client to generate and assign request IDs. Request IDs must be unique among both *Request* and *LocateRequest* messages.

Connection Closure

Connections can be closed in two ways: orderly shutdown, or abortive disconnect. Orderly shutdown is initiated by servers reliably 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. Some transport protocols (not including TCP) do not provide an “orderly disconnect” capability, guaranteeing reliable delivery of the last message sent. When GIOP is used with such protocols, an additional handshake needs to be provided to guarantee that both ends of the connection understand the disposition of any outstanding GIOP requests.

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.

13.5.2 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.

13.6 Object Location

The GIOP is defined to support object migration and location services without dictating the existence of specific ORB architectures or features. The protocol features are based on the following observations:

A given transport address does not necessarily correspond to any specific ORB architectural component (such as an object adapter, object server process, Inter-ORB bridge, and so forth). It merely implies the existence of some agent with which a connection may be opened, and to which requests may be sent.

The “agent” (owner of the server side of a connection) may have one of the following roles with respect to a particular object reference:

- The agent may be able to accept object requests directly for the object and return replies. The agent may or may not own the actual object implementation; it may be an Inter-ORB bridge that transforms the request and passes it on to another process or ORB. From GIOP’s perspective, it is only important that requests can be sent directly to the agent.
- The agent may not be able to accept direct requests for any objects, but acts instead as a location service. Any Request messages sent to the agent would result in either exceptions or replies with LOCATION_FORWARD status, providing new addresses to which requests may be sent. Such agents would also respond to *LocateRequest* messages with appropriate LocateReply messages.
- The agent may directly respond to some requests (for certain objects) and provide forwarding locations for other objects.
- The agent may directly respond to requests for a particular object at one point in time, and provide a forwarding location at a later time (perhaps during the same connection).

Agents are not required to implement location forwarding mechanisms. An agent can be implemented with the policy that a connection either supports direct access to an object, or returns exceptions. Such an ORB (or inter-ORB bridge) always 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 shall 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, a client can attempt to reuse the forwarding information it has, but, if that fails, it shall restart the location process using the original address specified in the initial object reference.

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.

13.7 Internet Inter-ORB Protocol (IIOP)

The baseline transport specified for GIOP is TCP/IP[5]. Specific APIs for libraries supporting TCP/IP may vary, so this discussion is limited to an abstract view of TCP/IP and management of its connections. The mapping of GIOP message transfer to TCP/IP connections is called the Internet Inter-ORB Protocol (IIOP).

IIOP 1.0 is based on GIOP 1.0.

IIOP 1.1 can be based on either GIOP 1.0 or GIOP 1.1. An IIOP 1.1 client can either support both GIP 1.0 and 1.1, or GIOP 1.1 only. An IIOP 1.1 server must support both GIOP 1.0 and GIOP 1.1. An IIOP 1.1 server must be able to receive both GIOP 1.0 and GIOP 1.1 requests and reply using the same GIOP revision as invoked.

---

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

13.7.1 TCP/IP Connection Usage

Agents that are capable of accepting object requests or providing locations for objects (i.e., servers) publish TCP/IP addresses in IORs, as described in “IIOP IOR Profiles” on page 13-34. A TCP/IP address consists of an IP host address, typically represented by a host name, and a TCP port number. Servers must listen for connection requests.

A client needing an object’s services must initiate a connection with the address specified in the IOR, with a connect request.

The listening server may accept or reject the connection. In general, servers should accept connection requests if possible, but ORBs are free to establish any desired policy for connection acceptance (e.g., to enforce fairness or optimize resource usage).

Once a connection is accepted, the client may send **Request, LocateRequest**, or **CancelRequest** messages by writing to the TCP/IP socket it owns for the connection. The server may send **Reply, LocateReply**, and **CloseConnection** messages by writing to its TCP/IP connection.

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.

13.7.2 IIOP IOR Profiles

IIOP profiles, identifying individual objects accessible through the Internet Inter_ORB Protocol, have the following form:

```
module IIOP {                       // IDL extended for version 1.1
    struct Version {
        octet           major;
        octet           minor;
    };

    struct ProfileBody_1_0 { // renamed from ProfileBody
        Version             iiop_version;
        string              host;
        unsigned short      port;
        sequence <octet>    object_key;
    };

    struct ProfileBody_1_1 {
        Version             iiop_version;
```

```
        string              host;
        unsigned short      port;
        sequence <octet>    object_key;

        // Added in 1.1
        sequence <IOP::TaggedComponent> components;
    };
};
```

IIOP Profile version number:

- Indicates the IIOP protocol version.
- Major number can stay the same if the new changes are backward compatible.
- Clients with lower minor version can attempt to invoke objects with higher minor version number by using only the information defined in the lower minor version protocol (ignore the extra information).

Profiles supporting only IIOP version 1.0 use the **ProfileBody_1_0** structure, while those supporting IIOP version 1.1 use the **ProfileBody_1_1** structure. An instance of one of these structure types is marshaled into an encapsulation octet stream. This encapsulation (a **sequence <octet>**) becomes the **profile_data** member of the **IOP::TaggedProfile** structure representing the IIOP profile in an IOR, and the tag has the value **TAG_INTERNET_IOP** (as defined earlier).

If the major revision number is 1, and the minor revision number is greater than 0, then the length of the encapsulated profile may exceed the total size of components defined in this specification for profiles with minor revision number 0. ORBs that support only revision 1.0 IIOP profiles must ignore any data in the profile that occurs after the object_key. If the revision of the profile is 1.0, there shall be no extra data in the profile, i.e., the length of the encapsulated profile must agree with the total size of components defined for version 1.0.

The members of **IIOP::ProfileBody1_0** and **IOP::ProfileBody1_1** are defined as follows:

- **iiop_version** describes the version of IIOP that the agent at the specified address is prepared to receive. When an agent generates IIOP profiles specifying a particular version, it must be able to accept messages complying with the specified version or any previous minor version (i.e., any smaller version number). The major version number of this specification is 1; the minor version is 1. Compliant ORBs must generate version 1.1 profiles, and must accept any profile with a major version of 1, regardless of the minor version number. If the minor version number is 0, the encapsulation is fully described by the **ProfileBody_1_0** structure. If the minor version number is 1, the encapsulation is fully described by the **ProfileBody_1_1** structure. If the minor version number is greater than 1, then the length of the encapsulated profile may exceed the total size of components defined in this specification for profiles with minor version number 1. ORBs that support only version 1.1 IIOP profiles must ignore, but preserve, any data in the profile that occurs after the **components** member.

Note – This value is not equivalent to the GIOP version number specified in GIOP message headers. Transport-specific elements of the IIOP specification may change independently from the GIOP specification.

- **host** identifies the Internet host to which GIOP messages for the specified object may be sent. In order to promote a very large (Internet-wide) scope for the object reference, this will typically be the fully qualified domain name of the host, rather than an unqualified (or partially qualified) name. However, per Internet standards, the host string may also contain a host address expressed in standard "dotted decimal" form (e.g., "192.231.79.52").
- **port** contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.
- **object_key** is an opaque value supplied by the agent producing the IOR. This value will be used in request messages to identify the object to which the request is directed. An agent that generates an object key value must be able to map the value unambiguously onto the corresponding object when routing requests internally.
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. **TaggedComponent**s that apply to IIOP 1.1 are described below in section 13.7.3. Other components may be included to support enhanced versions of IIOP, to support ORB services such as security, and to support other GIOPs, ESIOPs, and proprietary protocols. If an implementation puts a non-standard component in an IOR, it cannot be assured that any or all non-standard component will remain in the IOR.

  The relationship between the IIOP protocol version and component support conformance requirements is as follows:
  - Each IIOP version specifies a set of standard components and the conformance rules for that version. These rules specify which components are mandatory presence, which are optional presence, and which can be dropped. A conformant implementation has to conform to these rules, and is not required to conform to more than these rules.
  - New components can be added, but they do not become part of the versions conformance rules.
  - When there is a need to specify conformance rules which include the new components, there will be a need to create a new IIOP version.

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

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

13.7.3 IIOP IOR Profile Components

The following components are part of the IIOP 1.1 conformance. All these components are optional presence in the IIOP profile and cannot be dropped from an IIOP 1.1 IOR.

- TAG_ORB_TYPE
- TAG_CODE_SETS
- TAG_SEC_NAME
- TAG_ASSOCIATION_OPTIONS
- TAG_GENERIC_SEC_MECH

13.8 OMG IDL

This section contains the OMG IDL for the GIOP and IIOP modules.

13.8.1 GIOP Module

```
module GIOP {                          // IDL extended for version 1.1

    struct Version {
        octet       major;
        octet       minor;
    };

    #ifndef GIOP_1_1
    // GIOP 1.0
    enum MsgType_1_0{   // rename from MsgType
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError
    };

    #else
    // GIOP 1.1
    enum MsgType_1_1{
        Request, Reply, CancelRequest,
        LocateRequest, LocateReply,
        CloseConnection, MessageError,
        Fragment                              // GIOP 1.1 addition
    };
    #endif

// GIOP 1.0
    struct MessageHeader_1_0 { // Renamed from MessageHeader
        char            magic [4];
        Version         GIOP_version;
        boolean         byte_order;
        octet           message_type;
```

```
        unsigned long       message_size;
    };

    // GIOP 1.1
    struct MessageHeader_1_1 {
        char                magic [4];
        Version             GIOP_version;
        octet               flags;              // GIOP 1.1 change
        octet               message_type;
        unsigned long       message_size;
    };

};// GIOP 1.0
    struct RequestHeader _1_0 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        sequence <octet>            object_key;
        string                      operation;
        Principal                   requesting_principal;
    };

    // GIOP 1.1
    struct RequestHeader_1_1 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        boolean                     response_expected;
        octet                       reserved[3]; // Added in GIOP 1.1
        sequence <octet>            object_key;
        string                      operation;
        Principal                   requesting_principal;
    };
    enum ReplyStatusType {
            NO_EXCEPTION,
            USER_EXCEPTION,
            SYSTEM_EXCEPTION,
            LOCATION_FORWARD
    };

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

    struct CancelRequestHeader {
            unsigned long                   request_id;
    };

    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;
    };
};
```

13.8.2 IIOP Module

```
module IIOP {                   // IDL extended for version 1.1

    struct Version {
        octet                   major;
        octet                   minor;
    };

    struct ProfileBody_1_0 { // renamed from ProfileBody
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        sequence <octet>        object_key;
    };
struct ProfileBody_1_1 {
        Version                 iiop_version;
        string                  host;
        unsigned short          port;
        sequence <octet>        object_key;
        sequence <IOP::TaggedComponent> components;
    };
};
```