Unreliable Multicast Inter-ORB Protocol Specification

This OMG document replaces the draft adopted specification (ptc/2001-10-18). It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by May 15, 2002.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on July 1, 2002. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.

OMG Final Adopted Specification
November 2001

Unreliable Multicast Inter-ORB Protocol 29

Note – from the OMG Technical Editor: Eventually, this specification will become part of the CORBA Core document. The chapter number is temporary and may change.

Contents

This chapter contains the following topics.

Topic	Page
Section I - Introduction	
“Purpose”	29-2
“MIOP Packet”	29-3
“Packet Collection”	29-3
“PacketHeader”	29-3
“Joining an IP/Multicast Group”	29-5
“Quality Of Service”	29-6
“Delivery Requirements”	29-6
Section II - MIOP Object Model	
“Definition”	29-6
“Unreliable IP/Multicast Profile Body (UIPMC_ProfileBody)”	29-7
“Group IOR”	29-9
“Adding Group Knowledge to PortableServer::POA”	29-11

Topic	Page
“MIOP Gateway”	29-15
“Multicast Group Manager”	29-15
“MIOP URL”	29-32
Section III - Request Issues	
“GIOP Request Message Compatibility”	29-33
“MIOP Request Efficiency”	29-34
“Client Use Cases”	29-35
“Server Use Cases”	29-36
Appendix A - “Conformance”	29-37
Appendix B - “Consolidated IDL”	29-38

Section I - Introduction

29.1 Purpose

The purpose of MIOP (Unreliable Multicast Inter-ORB Protocol) is to provide a common mechanism to deliver GIOP request and fragment messages via multicast. The default transport specified for MIOP is IP Multicast[1] through UDP/IP[2] which will provide the ability to perform connectionless multicast. This requires that IDL operations will have one-way semantics.

The initial version of MIOP will be designated 1.0. This version scheme will be independent of the GIOP version.

MIOP will not be dependent upon data that is contained in the GIOP header, request header, or fragment header; MIOP does not read or interpret GIOP messages. This will provide the capability for MIOP to be used for future and existing protocols as defined in by the OMG.

This specification mandates that implementers of this technology shall reuse the CDR marshalling.

The following sections describe the wire protocol and send/receive semantics of MIOP.

---

1. Deering, S. “Host Extensions for IP Multicasting” RFC 1112 Network Working Group, Stanford University August 1989
2. Postel,J. “User Datagram Protocol” RFC-768 Information Sciences Institute, August 28, 1980

29.2 MIOP Packet

An MIOP Packet is defined as the MIOP PacketHeader information, which is defined below, as well as the raw GIOP data (body) contained in the rest of the MIOP Packet. An MIOP Packet will be sent and later reassembled on the receiving side. MIOP Packets are the atomic pieces that comprise a Packet Collection which is discussed below.

29.3 Packet Collection

A Packet Collection is comprised of one or more MIOP Packets and is defined as complete, packaged, GIOP request/fragment message. Only GIOP request messages and associated request message fragments are allowed in an MIOP Packet in a Packet Collection.

The total data contained in a GIOP message (header, request/fragment header and body), determines the total number of MIOP Packets that need to be sent and subsequently reassembled by the receiver.

The number of packets that comprise a Packet Collection are dependent on the maximum size of the frame buffer supported by the hardware. Typically Ethernet supports 1518 bytes per frame, although UDP will allow up to 65536 bytes per frame if the physical layer can support it. If the Packet Collection cannot fit in the hardware specified frame size, the MIOP protocol requires that the GIOP message be broken up into packets that comprise a Packet Collection.

The MIOP sender will label the packet data so that the receiving MIOP layer can determine the number of packets in the MIOP message, as well as the position of each packet as a part of the Packet Collection (e.g., packet 3 of 20). In addition, the sender provides a unique signature for each Packet Collection to ensure that the receiver can properly reassemble the packets (i.e., make sure that 3 of 20 is not some other message's 3 of 20).

29.4 PacketHeader

The PacketHeader is the MIOP data structure that represents the state information for a single packet within the Packet Collection in MIOP. This data structure is used to send and receive packetized GIOP messages in the form of MIOP Packet Collections. This data structure precedes any associated GIOP data which has been packetized. All MIOP packets in a Packet Collection must start with a PacketHeader.

Therefore it is required that:

- MIOP senders insert a PacketHeader in front of each packet of GIOP request/fragment data;
- MIOP receivers read and strip off MIOP PacketHeaders and concatenate all the GIOP related data from the packets of a message before posting the GIOP request to the ORB; and
- The GIOP data in the MIOP Packet body must start on a eight byte boundary following the **UniqueId** field.

The following IDL defines the fields of the Packet Header. The text following the IDL will describe the field in detail.

```
module MIOP
{
  typedef sequence <octet, 252> UniqueId;

  struct PacketHeader_1_0
  {
     char                    magic[4];           // 4bytes
     octet                   hdr_version;        // 1 byte
     octet                   flags;              // 1 byte
     unsigned short          packet_length;      // 2 bytes
     unsigned long           packet_number;      // 4 bytes
     unsigned long           number_of_packets;  // 4bytes,
sub-total 16 bytes
     UniqueId                id;                 // body
data must begin on a
                                                 // 8 byte
boundary
  };

};
```

29.4.1 struct PacketHeader

The PacketHeader is a variable length data structure that provides the capability for packet reassembly on the receivers side. The sending side is responsible for properly filling in the values before it is sent.

29.4.1.1 Magic

The magic field is a four byte character array which will always be the literal value ‘MIOP.’ This field will provide a mechanism for determining if a multicast message is a CORBA MIOP message or some other unrelated datagram.

29.4.1.2 Header Version

This field is a 1 byte value that contains the major and minor versions of MIOP. The high-order four bits of the octet will be the major version and the low-order four bits are the minor version (e.g. decimal 16 will equal version 1.0).

29.4.1.3 Flags

The Flags field is a single octet that provides the value of the sending endian, and a stop message bit as well as unused, additional bits for future use.

The 1.0 version of MIOP specifies that the lowest order bit of the octet will contain the endian flag. The value zero for the endian flag indicates big-endian byte ordering and the value one indicates little-endian byte ordering.

The second lowest order bit will designate the stop message flag. This bit contains a value of one for the last packet sent in a Packet Collection; otherwise it must be zero.

The values of the six reserved bits must be set to zero for MIOP 1.0

29.4.1.4 Packet Length

This field is mandatory and is used to define the actual number of bytes delivered with an MIOP Packet in a Packet Collection. This value must be the same for all MIOP Packets with the exception of the last packet which may contain a value which is less than the length specified in the other packets in the Packet Collection. This value is constrained by the physical layer of the underlying transport. For instance, UDP allows as much as 65K or as little as 512 bytes for the size of a single UDP packet.

The value specified in this field will not take into account the length of the header.

29.4.1.5 Packet Number

This field is mandatory and is an unsigned long value that states the current packet number that is being delivered (e.g. packet 3 of 20 in the Packet Collection). The number of the first packet must be zero and the number of the last packet must be n-1.

29.4.1.6 Number of Packets

This field is optional but can be used in conjunction with the first instance of an MIOP Packet in the Packet Collection, Packet Length field, to perform optimizations for allocating the receive message buffer used in packet reassembly. The number-of-packets field is an unsigned long value that states the total number of packets that are to be delivered[3]. In the event that this field is not used it must be set to the value zero.

29.4.1.7 UniqueId

This variable length data structure provides the unique signature required for packet reassembly on the receiving side. This field ***must*** be identical for all packets associated with a Packet Collection. The data structure for this field is a bounded sequence of octets not to exceed 252 bytes of octet data[4]. The data contained in the body of the packet following the octet portion of the sequence must start on an eight byte boundary to maintain the integrity of the GIOP data in the packet. Implementations must provide as a default, uniqueness appropriate for the internet as a whole (e.g., GUID format of COM).

29.5 Joining an IP/Multicast Group

IP/Multicast only requires receivers to explicitly join an IP/Multicast group before they receive their first MIOP packet.

---

3. Based on a UDP packet of 512, and the maximum number of packets (4,294,967,296) packets, the maximum message size would be approximately 2GB. This size would include the IP header, UDP header, MIOP PacketHeader(s), and all other related data.
4. The unique id should be kept as small as possible for those applications that broadcast small GIOP messages.

29.6 Quality Of Service

29.6.1 Time-To-Live

When the implementation is using IP/Multicast, the socket option IP_MULTICAST_TTL, provides the capability for a UDP/IP datagram to be sent over more than one subnet. This value could be assigned statically or dynamically. This specification does not address how an application would set TTL.

29.6.2 Incomplete Receipt of a Packet Collection

In the event that an MIOP packet is received out of order in the context of its Packet Collection, the protocol should wait for the missing packets until the last packet is received. If the missing packets are not received by the time of the receipt of the last packet, the Packet Collection is in error and should be dropped.

The receiver should provide a mechanism to time out incomplete Packet Collections in the event that a partial collection has been received but no further packets are incoming. This will keep the protocol from expending resources and waiting indefinitely.

29.7 Delivery Requirements

The MIOP protocol requires that a single GIOP request message, including any fragment, must be sent as a single MIOP message. This does not imply that the entire GIOP request must be encoded before it is delivered to the MIOP layer. The MIOP protocol must be notified that the last portion of the GIOP message is being sent. This notification must also occur for non-fragmented GIOP requests. Upon receipt of the last portion or the whole GIOP message. MIOP will generate packets from the GIOP data and set the stop bit in the Packet Header flags for the last MIOP packet it creates. All MIOP packets associated with a GIOP request, including the fragments will have the same data image expressed in their UniqueId field of their Packet Header in a given Packet Collection.

Section II - MIOP Object Model

29.8 Definition

The current CORBA object model specifies that a single object reference will map to a single object implementation via an object key. The invocation semantics are both two-way and one-way with reliability requirements for the delivery and ordering of messages.

The proposed specification for the MIOP object model does not specify an object key but a group identifier that can be associated to multiple **PortableServer::ObjectId** values which in turn can be used to activate implementation objects. The delivery semantics of CORBA messages over MIOP are one-way with no reliability of message receipt. There are no requirements for MIOP to be reliable so therefore no group membership is

needed. Therefore it is entirely possible to have an application sending CORBA messages via MIOP with no receiving applications present (e.g., the unpopular radio station that no one listens to but broadcasts non-the-less).

An on-going theme in the MIOP object model is the behavior of object groups. An object group in MIOP consists of group identification information as well as network communication information. Object group technology is necessary for implementing the MIOP object model. Therefore, the role of object groups will be covered during the discussion of the following document sections.

The object model for MIOP will consist of the following topics:

- MIOP Unreliable IP/Multicast Profile Body;
- Group IOR;
- Portable Group Adapter (PGA);
- Gateway; and the
- Multicast Group Manager (MGM).

29.9 Unreliable IP/Multicast Profile Body (UIPMC_ProfileBody)

The **UIPMC_ProfileBody** contains all the information required to make an invocation on a servant object using IP/Multicast as a transport mechanism. The profile differs from the IIOP profile in that:

- the object key field is omitted; and
- the field expressing a host name is replaced by a multiple character field that will express a valid IPv4/IPv6 multicast address, or an alias name to a multicast address.

The absence of the object key in the UIPMC profile is due to the fact that an MIOP request will typically be delivered to multiple servants that are associated with an object group and are housed in applications built using different vendor's ORBs. This of course is different than the classical IIOP scenario in which an object key provides a mapping to a single servant.

The definition of the UIPMC profile is defined below.

```
module MIOP
{
    ...
    typedef GIOP::Version Version;
    typedef string Address;
    struct UIPMC_ProfileBody
    {
        Version         miop_version;
        Address         the_address;
        short           the_port;
        sequence<IOP::TaggedComponents> components;
    };
};
```

29.9.1 Version

The version field contains the major and minor version of the data structure. The version nomenclature will start at 1.0. This version field should be considered separate from the version of the MIOP PacketHeader.

29.9.2 Address

The Address field specifies either

- a class D address for IPv4 (e.g., 225.1.1.1);
- an IPv6 address (e.g., FF01:0:0:0:0:0:0:1 - all nodes address); or
- a alias to a multicast address.

29.9.3 Port

This field is an unsigned short value that contains the port value associated with the Address field.

29.9.4 Group Components

This field will contain any additional component profiles. It has the same semantics as in an IIOP ProfileBody. At least one of these components must contain a group component and additionally a component specifying a IIOP profile to be used for two-way operations and operations supporting the **CORBA::Object** OMA. The concepts of the components are discussed below.

```
module IOP {

    const ProfileId      TAG_UIPMC      = OMG_assigned;
    const ComponentId    TAG_GROUP      = OMG_assigned;
    const ComponnetId    TAG_GROUP_IIOP = OMG_assigned
};

module PortableGroup {

    typedef GIOP::Version Version;

    struct GroupInfo { // tag = TAG_GROUP;
        Version component_version;
        GroupDomainId group_domain_id;
        ObjectGroupId object_group_id;
        ObjectGroupRefVersion object_group_ref_version;
    };
    typedef sequence <octet> GroupIIOPProfile
};
```

Object Groups can have lifetimes that persist after senders and receivers are no longer invoking on the destination endpoints of the object group. In fact, object groups can exist with no associated, participating sender or receiver objects. They should therefore be created to be unique to avoid ambiguity.

29.9.4.1 GroupInfo

The fields associated with this tagged component are used to provide unique information to describe a group.

Version

The current version of the tagged component.

GroupDomainId

A simple string that applies scope to the **ObjectGroupId**. This was changed from the **FT::FTDomainId**.

ObjectGroupId

A 64 bit identifier that uniquely defines the group.

ObjectGroupRefVersion

This field is optional and will be set to zero for MIOP when it is not being used. An implementation may set the value if multiple versions of an object group reference exist.

29.9.4.2 GroupIIOPProfile

This data field will contain the profile data for an IIOP tagged profile. This IIOP profile will be used to support the OMA of **CORBA::Object**. This component is not required to complete the UIPMC profile.

29.10 Group IOR

A Group IOR will serve the purpose of providing the client with the means to invoke directly to:

- an IIOP gateway application in the event that the client is not multicast capable (via a standard IIOP profile); or
- MIOP aware servant objects that support the same interface and belong to the same multicast group (via a multicast profile; for example, **UIPMC_ProfileBody**).

The creator of the IOR, (typically the MGM but not necessarily), creates the number of profiles in the IOR based on the requirements of the object group participants. This suggests that one or both profiles could be present in the Group IOR. MIOP aware clients will directly target the UIPMC profile contained in the Group IOR. This would allow them to directly use the multicast capability as opposed to using a gateway. All the profiles are essentially optional but one of course must be present. The figure below details the contents of the Group IOR.

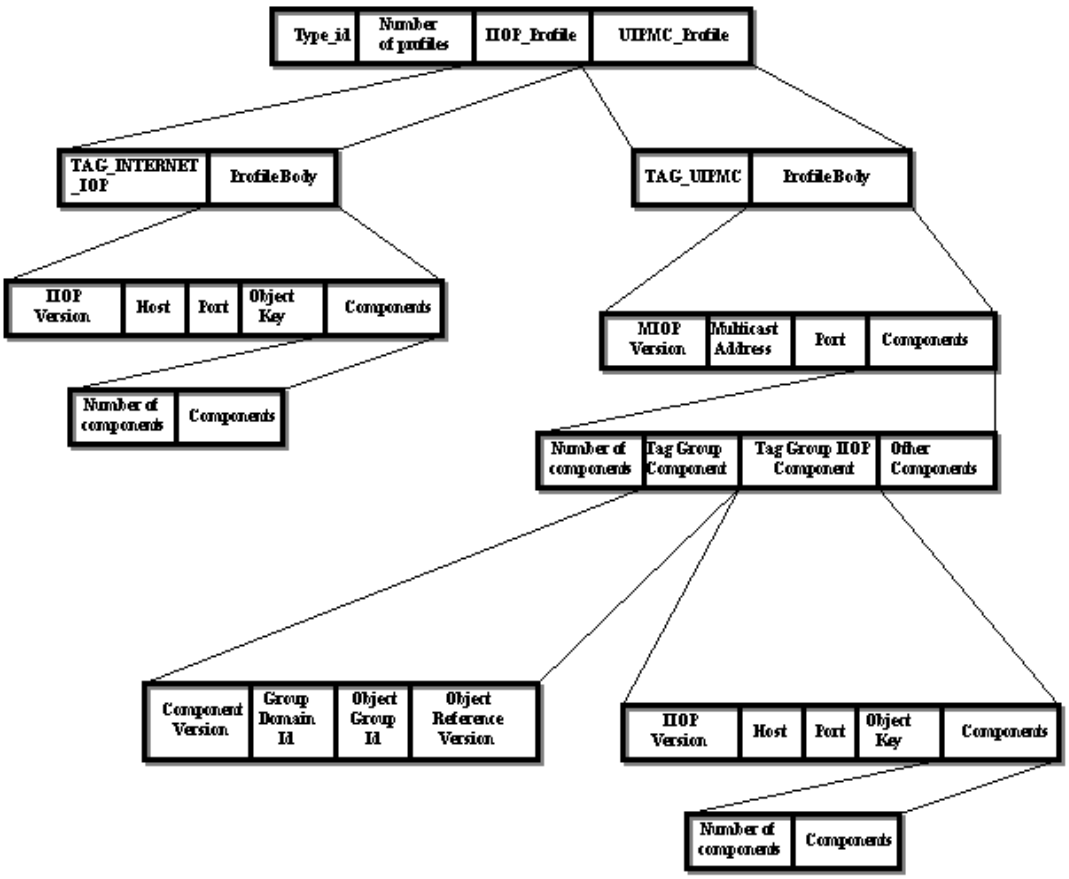

Figure 29-1 An example of the Group Interoperable Object Reference used for Unreliable Multicast.

29.10.1 Gateway Profile

The profile for the gateway is a standard IIOP profile. The gateway itself could be multicast aware, or like the CORBA EventService, simply multiplex requests to the servant objects via a conventional IIOP mechanism. Therefore no restrictions should be placed on servants which would not allow both standard GIOP over IIOP requests and MIOP requests to target the same servant object.

If the gateway supports GIOP version 1.2, the creator of the Group IOR should place the UIPMC Profile[5] information in the profile field of the union field for **GIOP::TargetAddress**.

5. It is assumed that other profiles created for MIOP will be handled in a similar fashion.

If the gateway supports GIOP version 1.0 or 1.1, the creator of the Group IOR should place the UIPMC Profile information in the object key field. This data will appear as an encapsulation preceded by the literal characters ‘MIOP.’

Providing the entire profile as part of the request header allows the gateway to use the destination endpoints for forwarding the request to its intended destinations via multicast.

29.10.2 UIPMC Profile

The UIPMC profile would directly support MIOP operations. This could be the only profile associated with an IOR for those situations where there are no gateway applications available. In addition, the creator of the group IOR can specify an IIOP profile to be placed as a tagged component of the UIPMC profile to support the OMA of **CORBA::Object**. The UIPMC profile differs from an IIOP profile in that the object key field is not present. The identify of the group is defined by the **PortableGroup::GroupInfo** data supplied in the components.

29.10.3 Unreliable Multicast Object Groups

Object groups represent a collection of participating objects, both invoking and receiving CORBA operations on common object group information, that take their identity from the information associated with the group definition. Typically this group information is stored in the Multicast Group Manager (MGM) when such an application is present. Often, though not in this specification, an object group’s information will define a group’s object membership; this specification ignores membership. It is assumed that future reliable multicast oriented specifications will address group membership.

For the scope of this specification, an object group will provide some unique identification of itself (id, name, type version) as well as the ability to disclose its destination endpoints.

29.11 Adding Group Knowledge to PortableServer::POA

This section will discuss how to extend the current model for the PortableServer::POA to take into account servant groups. Several operations will added to extend the POA and provide object group functionality. For those ORB vendors that do not wish to implement the group methods of the POA, these additional routines will be optional compliance points when implementing the POA specification and implementations should raise the CORBA system exception CORBA::NO_IMPLEMENT.

The UIPMC profile for the Group IOR does not define a universal object key; object keys are opaque structures that are vendor defined. In addition, an object key as defined in IIOP, is intended to specify a single object as opposed to an object group. The intent of this specification was to not change the semantics of object keys or define a common object key format; neither of which would be hardily welcomed by the vendor community. Instead, the ORB implementation for the Group Object Adapter should use the **PortableGroup::GroupInfo** in the UIPMC profile components field and an associated **PortableServer::ObjectId** to correctly dispatch the MIOP operation to the correct objects[6].

The operations discussed below provide a mechanism to map a well known multicast group reference associated with an object group to a standard **PortableServer::ObjectId**. This removes the need of having a common object key format. This also provides a mechanism to allow servant objects residing in different vendor's ORB applications to all receive the same messages. This of course is dependent on:

1. The servants are using the same group reference containing the definition for the same object group.
2. The destination endpoints in the group reference are being used for receiving requests.
3. The interface that the request is being invoked upon is identical, or derived from the parent interface by inheritance, to the one contained in the object group's definition.

There should be no restrictions on different object groups sharing the same destination endpoints.

29.11.1 New Operations

```
module PortableServer

    exception NotAGroupObject {};
    typedef sequence <ObjectId> IDs;

    interface POA {
        ...
        ObjectId
            create_id_for_reference(in CORBA::Object the_ref)
                raises (NotAGroupObject);

        IDs
            reference_to_ids (in CORBA::Object the_ref)
                raises (NotAGroupObject);

        void
            associate_reference_with_id
                (in CORBA::Object ref, in ObjectId oid)
                raises(NotAGroupObject);

        void
            disassociate_reference_with_id
                (in CORBA::Object ref, in ObjectId oid)
                raises(NotAGroupObject);
```

---

6. The plural is used here to address multiple objects all associated with the same object group within a single process space. This requirement is necessary to address location transparency. Multiple collocated objects that have associated themselves with the same object group, which are invoked upon locally, will need to receive the request.

```
    }; // end interface POA

}; // end module PortableServer
```

29.11.1.1 Group Object Adapter Operations

create_id_for_reference

The operation **create_id_for_reference()** takes as an argument a widened Group IOR and generates a unique **PortableServer::ObjectId** for that reference. This identifier returned by this routine is of the type **PortableServer::ObjectId**. This identifier is later associated with a servant via the standard API in the POA; that is, **activate_object_with_id()**.

```
ObjectId
    create_id_for_reference(in CORBA::Object the_ref)
        raises (NotAGroupObject);
```

Parameters

the_ref	A reference for the object group.

Return Value

A unique ObjectId.

Raises

NotAGroupObject	Raised if the object reference is not a group reference.

reference_to_ids

The operation **reference_to_ids()** takes as an argument a widened Group IOR and returns a sequence of object identifiers that are currently associated with the Group IOR.

```
IDs
    reference_to_ids (in CORBA::Object group_ref)
        raises (NotAGroupObject);
```

Parameters

the_ref	A reference for the object group.

Return Value

A sequence of servant object identifiers that are currently associated with the group.

Raises

NotAGroupObject	Raised if the object reference is not a group reference.

associate_reference_with_id

The operation takes a previously generated ObjectId and associates it with a group reference. Servants activated using this ObjectId will be candidates for receiving MIOP requests via the group information provided in the IOR. The operation silently ignores repeat/duplicate associations of a POA/ObjectId pair with the provided object reference.

```
void
    associate_reference_with_id
        (in CORBA::Object ref, in ObjectId oid)
        raises(NotAGroupObject);
```

Parameters

ref	A reference for the object group.
oid	A system or user generated ObjectId.

Return Value

None.

Raises

NotAGroupObject	Raised if the object reference is not a group reference.

disassociate_reference_with_id

The operation takes a previously generated ObjectId and removes the association it had with a group reference. Servants activated using this ObjectId will no longer receive MIOP requests via the group information provided in the IOR. The operation silently ignores disassociations that no longer or never existed.

```
void
    disassociate_reference_with_id
        (in CORBA::Object ref, in ObjectId oid)
        raises(NotAGroupObject);
```

Parameters

ref	A reference for the object group.
oid	A system or user generated ObjectId.

Return Value

None

Raises

NotAGroupObject	Raised if the object reference is not a group reference.

29.11.2 Invocation Scenarios

The routines listed above do not have the capability to modify or destroy an ObjectId that is generated by the POA or by the application programmer. They simply associate an ID to a reference either implicitly or explicitly. Therefore a call to **create_id_for_reference()** or **associate_id_with_reference()** followed by a call

to **disassociate_id_with_reference()** does nothing to the existence of the ObjectId and allows that same ObjectId to be reused again in a call to **associate_id_with_reference()**.

29.12 MIOP Gateway

A gateway may be used to provide access to all MIOP aware servants that are in the object group. Prior to the creation of the group IOR, the creator can specify the use of a gateway and creator of the group IOR can insert the IIOP profile of the gateway in the Group IOR.

An MIOP unaware client ORB uses the IIOP profile from the group IOR and establishes a connection to the gateway. The gateway in turn uses the destination endpoints specified by the UIPMC profile (placed in the object key pre GIOP 1.2, and in the TargetAddress profile field in GIOP 1.2) to forward GIOP request messages to the members of the object group. The gateway could use MIOP multicast or an IIOP mechanism similar to the CORBA EventService.

An MIOP client application using a gateway would be MIOP unaware. Like any normal IIOP sending application, it simply makes requests without regard to the UIPMC profile in the IOR.

29.13 Multicast Group Manager

The Multicast Group Manager serves the purpose creating and managing multicast object groups as well as managing multicast transport resources. Creation of the multicast group can result in the assignment of multicast destination endpoints on which senders multicast their messages and receivers accept them. Once an object group is created, the group reference can be stored in the Naming Service to be retrieved by applications interested in participating in the object group.

When the MGM is instructed to create a multicast object group, it may perform one of the following criteria:

- Just create the group with a no destination endpoints;
- Create the group and automatically allocate the endpoints; or
- Create the group and supply its own preferred destination endpoints.

When the endpoints of the multicast transport are specified, the MGM can create a completed group reference and publish the reference to the world. The newly created/updated group reference would contain any specific IIOP profiles to the gateway, as well as the UIPMC profile. Once a client application gets a group reference, it can then start multicasting to servant objects who are listening on those same destination endpoints. A receiving application would acquire a published IOR and associate this reference with an ObjectId and then activate a servant object with that id allowing the dispatching of request to the participating servant objects.

A single MIOP multicast address may be associated with more than one object group. This allows a process to listen to messages for more than one group on a single multicast address.

Objects that are recipients of multicasts have interfaces defined in IDL like any other objects. However, because multicast is unidirectional, the only operations that can be invoked on an interface are operations that have a void return type, in parameters only, and do not raise exceptions. These restrictions create a problem because all IDL interfaces inherit from **CORBA::Object** which contains operations that do not meet these restrictions. For example, **is_a()** and **non_existent()** are operations that have a return value.

Therefore the group IOR created by the MGM must be able to:

- Provide a common factory to manage object groups; and
- Provide the capability to be able to support the OMA of **CORBA::Object**.

The MGM's role in implementing the OMA will be discussed at the end of this section on the MGM. The following sections will discuss the IDL that comprises the MGM.

Since the implementation of the MGM is optional, an orb vendor should use the corbaloc mechanism to create a group reference with a UIPMC profile if the MGM is not available. Once this reference is properly created, it needs to be published in a conventional application determined place (e.g., file, CORBA Naming Service, etc.) so that participating applications can acquire the reference to send and receive multicast requests.

29.13.1 module PortableGroup

This section presents the IDL for the module PortableGroup. The module will be used for other specifications outside this specification that deal with Object Groups. The IDL from the discussion of the PGA above is not included in the full IDL definition of PortableGroup pending its full acceptance. The full IDL for module will be presented at the end of the specification in Appendix B.

29.13.1.1 Common Types

```
module PortableGroup {

    // Specification for Interoperable Object Group References
    typedef string GroupDomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;

    typedef GIOP::Version Version;

    struct GroupInfo { // tag = TAG_GROUP;
        Version component_version;
        GroupDomainId group_domain_id;
        ObjectGroupId object_group_id;
        ObjectGroupRefVersion object_group_ref_version;
    };
    typedef sequence <octet> GroupIIOPProfile

    // Specification of Common Types and Exceptions
```

```
// for GroupManagement
interface GenericFactory;

typedef CORBA::RepositoryId TypeId;
typedef Object ObjectGroup;
typedef CosNaming::Name Name;
typedef any Value;

struct Property {
    Name nam;
    Value val;
};

typedef sequence<Property> Properties;
typedef Name Location;
typedef sequence<Location> Locations;
typedef Properties Criteria;

struct FactoryInfo {
    GenericFactory the_factory;
    Location the_location;
    Criteria the_criteria;
};

typedef sequence<FactoryInfo> FactoryInfos;
typedef long MembershipStyleValue;

const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;

typedef unsigned short InitialNumberReplicasValue;
typedef unsigned short MinimumNumberReplicasValue;

exception InterfaceNotFound {};
exception ObjectGroupNotFound {};
exception MemberNotFound {};
exception ObjectNotFound {};
exception MemberAlreadyPresent {};
exception BadReplicationStyle {};
exception ObjectNotCreated {};
exception ObjectNotAdded {};
exception UnsupportedProperty {
    Name nam;
};

exception InvalidProperty {
    Name nam;
    Value val;
};

exception NoFactory {
```

```
        Location the_location;
        TypeId type_id;
    };

    exception InvalidCriteria {
        Criteria invalid_criteria;
    };

    exception CannotMeetCriteria {
        Criteria unmet_criteria;
    };
};
```

29.13.2 Identifiers for PortableGroup

The identifiers listed below are identical to those defined in fault tolerant CORBA.

GroupDomainId

The name of the Group Domain. This name provides additional scoping with a group identifier.

ObjectGroupId

Unique Id for the Object Group.

ObjectGroupRefVersion

The current version of the reference. It should start at version 1.0.

GroupInfo

The unique information used object group identification as well as being used for object dispatching in the Portable Group Adapter.

GroupIIOPProfile

This is additional component information that is defined as an IIOP profile. This profile is used to support the implicit two-way operations associated with **CORBA::Object**.

TypeId

Repository Id of the Group Object's supported interface.

ObjectGroup

A collection of information used to define how one contacts a group of related group participants. It is uniquely identified by the information in the **GroupInfo** and the destination endpoints used to contact the servants in the group.

Name

The name of a property - may be hierarchical.

Value

The value of a property - may be any valid IDL type.

Property

A Name/Value pair.

Properties

A sequence of **Property**.

Location

This type is not used in MIOP for unreliable multicast.

Locations

This type is not used for MIOP for unreliable multicast.

Criteria

An IDL rename of the type **Properties**.

FactoryInfo

This type is not used for MIOP for unreliable multicast.

FactoryInfos

This type is not used for MIOP for unreliable multicast.

MembershipStyleValue

This type is not used for MIOP for unreliable multicast.

29.13.3 Exceptions for PortableGroup

The behavior of the exceptions is the same as in the CORBA Fault Tolerant specification except where differences are noted.

InterfaceNotFound

The exception is not used by the MGM.

ObjectGroupNotFound

The object group cannot be found by the MGM based on the identifier that was provided.

MemberNotFound

The exception is not used by the MGM.

ObjectNotFound

The exception is raised if no group reference is associated with the object group.

MemberAlreadyPresent

The exception is not used by the MGM.

BadReplicationStyle

The exception is not used by the MGM.

ObjectNotCreated

The **GenericFactory** did not create the object.

ObjectNotAdded

The exception is not used by the MGM.

UnsupportedProperty

The property is not recognized or unsupported.

InvalidProperty

The property was either repeated or is in conflict with an existing property.

NoFactory

The factory cannot create an object with the id provided.

InvalidCriteria

The criteria provided was not understood by the factory.

CannotMeetCriteria

The criteria was understood but the factory is unable to support the criteria.

29.13.4 interface PropertyManager

```
// Specification of PropertyManager Interface
interface PropertyManager {

    void set_default_properties
        (in Properties props)
            raises (InvalidProperty, UnsupportedProperty);

    Properties get_default_properties();

    void remove_default_properties
        (in Properties props)
            raises (InvalidProperty, UnsupportedProperty);
```

```
void set_type_properties
    (in TypeId type_id, in Properties overrides)
        raises (InvalidProperty, UnsupportedProperty);

Properties get_type_properties(in TypeId type_id);

void remove_type_properties
    (in TypeId type_id, in Properties props)
        raises (InvalidProperty, UnsupportedProperty);

void set_properties_dynamically
    (in ObjectGroup object_group, in Properties overrides)
        raises
        (ObjectGroupNotFound,
         InvalidProperty,
         UnsupportedProperty);

Properties get_properties
    (in ObjectGroup object_group)
        raises(ObjectGroupNotFound);

};  // endPropertyManager
```

This interface was taken from the CORBA Fault Tolerant specification. It has been modified to be general to object groups

29.13.4.1 Operations for PropertyManager

set_default_properties

The method sets all the default properties in the object group domain for all object groups. The default property values are determined by the implementation.

```
void set_default_properties
    (in Properties props)
    raises (InvalidProperty, UnsupportedProperty);
```

Parameters

props	A sequence of properties that are to applied to all object groups within a object group domain.

Return Value

None.

Raises

InvalidProperty	If one or more of the properties in the sequence is not valid.
UnsupportedProperty	If one or more of the properties in the sequence is not supported.

get_default_properties

This method returns the default properties for the object groups within the object group domain.

Properties get_default_properties();

Parameters

None.

Return Value

The default properties that have been set for the object groups.

Raises

None.

remove_default_properties

This method removes the given default properties.

void remove_default_properties(in Properties props)
raises (InvalidProperty, UnsupportedProperty);

Parameters

props	The properties to be removed.

Return Value

None

Raises

InvalidProperty	If one or more of the properties in the sequence is not valid.
UnsupportedProperty	If one or more of the properties in the sequence is not supported.

set_type_properties

This method sets the properties that override the default properties of the object groups, with the given type identifier, that are created in the future.

void set_type_properties
(in TypeId type_id, in Properties overrides)

raises (InvalidProperty, UnsupportedProperty);

Parameters

type_id	The repository id for which the properties, that are to override the existing properties, are set.
overrides	The overriding properties.

Return Values

None

Raises

InvalidProperty	If one or more of the properties in the sequence is not valid.
UnsupportedProperty	If one or more of the properties in the sequence is not supported.

get_type_properties

This method returns the properties of the object groups, with the given type identifier, that are created in the future. These properties include the properties determined by **set_type_properties()**, as well as the default properties that are not overridden by **set_type_properties()**.

Properties get_type_properties(in TypeId type_id);

Parameters

type_id	The repository id for which the properties, that are to override the existing properties, are set.

Return Value

The overriding properties for the given type identifier.

Raises

None.

remove_type_properties

This method removes the given properties, with the given type identifier.

void remove_type_properties
(in TypeId type_id, in Properties props)
raises (InvalidProperty, UnsupportedProperty);

Parameters

type_id	The repository id for which the given properties are to be removed.
props	The properties to be removed.

Return Value

None.

Raises

InvalidProperty	If one or more of the properties in the sequence is not valid.
UnsupportedProperty	If one or more of the properties in the sequence is not supported.

set_properties_dynamically

This method sets the properties for the object group with the given reference dynamically while the application executes. The properties given as a parameter override the properties for the object when it was created which, in turn, override the properties for the given type which, in turn, override the default properties.

void set_properties_dynamically
(in ObjectGroup object_group, in Properties overrides)
raises(ObjectGroupNotFound, InvalidProperty, UnsupportedProperty);

Parameters

object_group	The reference of the object group for which the overriding properties are set.
overrides	The overriding properties.

Raises

ObjectGroupNotFound	If object group specified cannot be found.
InvalidProperty	If one or more of the properties in the sequence is invalid
UnsupportedProperty	If one or more of the properties in the sequence is not supported.

get_properties

This method returns the current properties of the given object group. These properties include those that are set dynamically, those that are set when the object group was created but are not overridden by **set_properties_dynamically()**, those that are set as properties of a type but are not overridden by **create_object()** and **set_properties_dyamically()**, and those that are set as defaults but are not overridden by **set_type_properties(**), **create_object(),** and **set_properties_dyamically()**.

Properties get_properties(in ObjectGroup object_group)
raises(ObjectGroupNotFound);

Parameters

object_group	The reference of the object group for which the properties are to be returned.

Return Value

The set of current properties for the object group with the given reference.

Raises

ObjectGroupNotFound	If the object group is not found.

29.13.5 interface ObjectGroupManager

```
// Specification of ObjectGroupManager Interface
interface ObjectGroupManager {

    ObjectGroup create_member
        (in ObjectGroup object_group,
         in Location the_location,
         in TypeId type_id,
         in Criteria the_criteria)
        raises
            (ObjectGroupNotFound,
             MemberAlreadyPresent,
             NoFactory,
             ObjectNotCreated,
             InvalidCriteria,
             CannotMeetCriteria);

    ObjectGroup add_member
        (in ObjectGroup object_group,
         in Location the_location,
         in Object member)
        raises
            (ObjectGroupNotFound,
             CORBA::INV_OBJREF,
             MemberAlreadyPresent,
             ObjectNotAdded);

    ObjectGroup remove_member
        (in ObjectGroup object_group,
         in Location the_location)
    raises
        (ObjectGroupNotFound, MemberNotFound);

    Locations locations_of_members
        (in ObjectGroup object_group) raises(ObjectGroupNotFound);

    ObjectGroupId get_object_group_id
        (in ObjectGroup object_group) raises(ObjectGroupNotFound);

    ObjectGroup get_object_group_ref
        (in ObjectGroup object_group) raises(ObjectGroupNotFound);
```

```
Object get_member_ref
    (in ObjectGroup object_group,
     in Location loc)
raises(ObjectGroupNotFound, MemberNotFound);

}; //  end ObjectGroupManager
```

This interface is largely unused by the MGM with the exception of two methods which will be discussed below. The MGM will use this interface to obtain the current reference and identifier of an object group. The routines that are not discussed in the table below all return the exception CORBA::NO_IMPLEMENT. The behavior of the operations in this interface is the same as in the CORBA Fault Tolerant specification except where differences are noted.

29.13.5.1 Operations for ObjectGroupManager

get_object_group_id

The method takes an object group reference as a parameter and returns the identifier of the object group.

```
ObjectGroupId get_object_group_id
    (in ObjectGroup object_group) raises (ObjectGroupNotFound);
```

Parameters

object_group — A reference for the object group.

Return Value

The identifier of the object group.

Raises

ObjectGroupNotFound	Raised if the object group is not found by the MGM.

get_object_group_ref

The method takes an object group reference as a parameter and returns the current reference of the object group. Any address changes or new allocations can be found by updating this reference.

```
ObjectGroupId get_object_group_ref
    (in ObjectGroup object_group) raises (ObjectGroupNotFound);
```

Parameters

object_group — A reference for the object group.

Return Value

The identifier of the object group.

Raises

ObjectGroupNotFound	Raised if the object group is not found by the MGM.

29.13.6 interface GenericFactory

```
// Specification of GenericFactory Interface
interface GenericFactory {
    typedef any FactoryCreationId;

    Object create_object
        (in TypeId type_id,
         in Criteria the_criteria,
         out FactoryCreationId factory_creation_id)
    raises
        (NoFactory,
         ObjectNotCreated,
         InvalidCriteria,
         InvalidProperty,
         CannotMeetCriteria);

    void delete_object
        (in FactoryCreationId factory_creation_id)
            raises (ObjectNotFound);

}; // end GenericFactory
```

This interface provides a generic create and destroy functionality for object groups. The call to **create_object()** will return a type Any that contains a **PortableGroup::ObjectGroupId**. The call to **destroy_object()** will remove the object group and the group reference from the factory. The behavior of the operations in this interface are the same as in the CORBA Fault Tolerant specification except where differences are noted.

29.13.6.1 Operations for GenericFactory

create_object

This routine creates a group reference from the type id and criteria list specified. It returns a group object identifier and an group object reference.

```
Object create_object
   (in TypeId type_id,
    in Criteria the_criteria,
    out FactoryCreationId factory_creation_id)
raises
   (NoFactory,
    ObjectNotCreated,
    InvalidCriteria,
```

```
    InvalidProperty,
    CannotMeetCriteria
  );
```

Parameters

object_group	A reference for the object group.
type_id	The repository id of the object to be created.
the_criteria	Additional information that is evaluated before the object is created. MIOP can use these to set the types and numbers of profiles in the Group IOR.
factory_creation_id	Unique value assigned by the factory and later used for deletion.

Return Value

The group object created by the factory.

Raises

NoFactory	The object cannot be created.
ObjectNotCreated	The object cannot be created.
InvalidCriteria	The criteria is not understood.
InvalidProperty	Invalid property was passed in the criteria.
CannotMeetCriteria	The application understands the criteria but is unable to process it

delete_object

This method deletes an object group, and all its available information, based on the type id specified.

```
void delete_object
    (in FactoryCreationId factory_creation_id) raises (ObjectNotFound);
```

Parameters

factory_creation_id	A identifier that was previously provided by a create call.

Return Value

None

Raises

ObjectNotFound	Raised if the object reference is not found by the MGM.

29.13.7 module MGM

```
module MGM {

    // Property values

    typedef long GroupCreationMode
    const GroupCreationMode CREATE_ADDRESS_DEFERED = 0;
    const GroupCreationMode CREATE_ADDRESS_GENERATED = 1;
    const GroupCreationMode CREATE_ADDRESS_SUPPLIED = 2;

    interface ObjectGroupFactory :
        PortableGroup::GenericFactory,
        PortableGroup::PropertyManager,
        PortableGroup::ObjectGroupManager {}
};
```

This module will encapsulate the specific properties of the MGM as well as the interface for **ObjectGroupFactory**.

29.13.8 MGM Properties

The following sections document the policies in the MGM. It is assumed that the implementers may add to the list of properties based on their specific protocol and application needs. The only protocol currently supported is IP/Multicast.

29.13.8.1 GroupCreationMode

Name org.omg.mgm.GroupCreationMode

Value CREATE_ADDRESS_DEFERED

CREATE_ADDRESS_GENERATED

CREATE_ADDRESS_SUPPLIED

The creation mode **CREATE_ADDRESS_DEFERED** will direct the creation of an object group without any multicast destination endpoints. The inclusion of properties that involve destination endpoints will cause the exception CannotMeetCriteria to be raised.

The creation mode **CREATE_ADDRESS_GENERATED** will direct the creation of an object group with MGM selected multicast destination endpoints. The inclusion of properties that involve destination endpoints will cause the exception CannotMeetCriteria to be raised.

The creation mode **CREATE_ADDRESS_SUPPLIED** will direct the creation of an object group with those destination endpoints which are specified in another property. The exclusion of properties that contain destination endpoints will cause the exception CannotMeetCriteria to be raised.

These properties can only be set at group creation time.

29.13.8.2 CreateSpecifyGateway

Name org.omg.mgm.CreateSpecifyGateway

Value The CORBA::Object of the gateway.

This property will register the MIOP gateway in the object group. This property can be set anytime.

29.13.8.3 SupportImplicitOperations

Name org.omg.mgm.SupportImplicitOperations

Value CORBA::Object.

This property will allow the MGM to create an IOR with a profile that supports the OMA of the object group. The IOR value could be one of the following:

- A object not associated with an MIOP gateway (application defined);
- An MIOP gateway; or
- The MGM.

If the value of the IOR is null, the MGM will assume it will be supporting the OMA. This property can be set anytime.

29.13.8.4 CreateIncludeGateway

Name org.omg.mgm.CreateIncludeGateway

Value CORBA::Boolean

If the value of the property is set to TRUE, the profile of the gateway is included in the group IOR. If the value of the property is set to FALSE, the profile of the gateway is excluded in the group IOR. This property can be set anytime.

29.13.8.5 ProtocolEndpointsIPPort

Name org.omg.mgm.ProtocolEndpointsIPPort

Value An unsigned short value designating a unique port.

This property can be set anytime.

29.13.8.6 ProtocolEndpointsIPAddress

Name org.omg.mgm.ProtocolEndpointsIPv4Address

Value A string designating an IPv4 or IPv6 multicast address.

This property can be set anytime.

29.13.8.7 GroupDomainId

Name org.omg.mgm.GroupDomainId

Value string

This value is used to scope the group identifier. If this property is not specified, the group domain identifier will default to "DefaultGroupDomain." This property can only be set once during the life of the object group. If an attempt is made to set this value after the default has been changed, the exception PortableGroup::InvalidProperty will be raised.

29.13.9 interface ObjectGroupFactory

This interface provides the capability to manage objects groups. It directly inherits the **ObjectGroupManager**, **PropertyManager**, and the **GenericFactory** interfaces. It completely reuses the specifications for its inherited interfaces.

29.13.10 Interoperable Object Group Reference Operations

To avoid breaking the CORBA object model, it is recommended that each group IOR's UIPMC profile contain the **PortableGroup::GroupIIOPProfile** tagged component which will profile the capability to invoke two-way CORBA::Object implicit operations. The methods for addressing these operations are discussed in the following sections.

is_a

This operation is unchanged. If the interface is understood by the client ORB, the call will return `true`. If the UIPMC profile was created without the **PortableGroup::GroupIIOPProfile** component, the client ORB should try to resolve the interface internally and only return false it cannot resolve the interface name internally. Otherwise if will use the IIOP profile in the **PortableGroup::GroupIIOPProfile** to try to resolve the call.

non_existent

For a group IOR, this operation always returns the value **true**.

validate_connection

For a group IOR, this operation always replies with **true** if the current policies are correct.

get_domain_managers

Similar considerations as for **is_a** here.

get_interface

Same considerations as for **is_a** and **get_domain_managers**.

is_nil

This operation would return **false** if at least one profile is present, otherwise it returns **true** if no profiles are present.

is_equivalent

Cases:

- If both references are non-group references the behavior is unchanged.
- If one reference is a group reference and the other is not a group reference, then the references are not equivalent.
- The number of profiles must be equal. If both references are group references then the field of the group components are compared and must be identical for all profiles that contain them.

hash

Follows the semantics of **is_equivalent**.

create_request

Unchanged.

get_policy

Unchanged.

set_policy_overrides

Unchanged.

Other Two Way Calls

If the group IOR contains only the UIPMC profile, the client ORB may use the tagged component **PortableGroup::GroupIIOPProfile**, if it exists, to process two-way calls on an interface that supports both two-way and one-ways calls.

If the group IOR contains both an IIOP gateway profile and the UIPMC profile, the sending ORB can choose to use the gateway IIOP profile even if it is MIOP aware. An MIOP unaware client would always use the IIOP gateway profile even in the existence of the UIPMC profile.

29.14 MIOP URL

This section provides a corbaloc URL definition of an MIOP profile. The following defines the syntax:

```
<corbaloc> = "corbaloc:"<obj_addr_list>["/"<key_string>]
<obj_addr_list> = [<obj_addr> ","]* <obj_addr>
<obj_addr> = <prot_addr>
<prot_addr> = <iiop_prot_addr> | <miop_prot_addr>
<miop_prot_addr> = <miop_prot_token><miop_addr>
<miop_prot_token> = "miop"
```

<iiop_prot_token> = “iiop”
<miop_addr> = <version><group_addr>[;<group_iiop>]
<version> = <major> "." <minor> "@" | empty_string
<group_addr> = <group_id>”/”<ip_multicast_addr>
<group_iiop> = <iiop_prot_token>”:”<version> <hostname>":"
<port> “/” <objecy_key>
<ip_multicast_addr> = <classD_IP_address> | <IPv6_address> ":" <port>
<classD_IP_address> = "224.0.0.0" - "239.255.255.255"
<port> = number (default to be defined)
<group_id> = <group component version>”-”<group_domain_id>”-”
<object_group_id>[“-”<object group reference version>]
<group component version> = <major> "." <minor>
<group_domain_id> = string
<object_group_id> = unsigned long long
<object group reference version> = unsigned long
<major> = number (default 1)
<minor> = number (default 0)

It would be written as follows below. The example URL does not use the Object Reference Version as defined in the **PortableGroup::GroupInfo**. Therefore this value must be 0 in the constructed profile. Not that both the multicast address and the group information are required. This example also supplies a **PortableGroup::GroupIIOPProfile** tagged component.

corbaloc:miop:1.0@1.0-MyLIttleDomin-1/225.1.1.8:5000;
iiop:1.1@oma_host:1234/object_key,
iiop:1.2@gateway_host:1234/object_key

Section III - Request Issues

29.15 GIOP Request Message Compatibility

Client ORBs will fall into two categories when invoking operations via MIOP:

1. MIOP aware clients; and

2. MIOP unaware clients.

If the client is MIOP aware it will use the UIPMC profile even if the IIOP gateway profile is present. In addition, the MIOP aware client may make use of the **PortableGroup::GroupIIOPProfile** tagged component to resolve **CORBA::Object** implicit operations and other interface specific two-way operations.

The MIOP unaware client will always use the gateway IIOP profile and ignore the UIPMC profile. All calls (one-way, two-way including implicit operations) will be send to the gateway application.

29.15.1 GIOP 1.2 Request Message

All requests involving MIOP operations will send the UIPMC profile along with the request. This will be placed in the **target** field of the GIOP 1.2 request header. This profile will be used by the object adapter to dispatch the request to the appropriate objects that support the interface within the confines of their object adapter. Some client ORBs may decide to set the destination endpoints to null or zero value since they are not required for message dispatching.

29.15.2 Object Key Support in Pre-GIOP 1.2

In order to achieve inter-ORB interoperability for MIOP, the notion of an object key had to be abandoned unless one was willing to define a common object key format for MIOP. For GIOP version 1.2, the `target` field can contain either an IOR, an object key or a tagged profile making it possible for the UIPMC profile can be sent along with the request. To insure consistency, the ORB must always send the UIPMC profile for GIOP version 1.2.

GIOP versions 1.0 and 1.1 do not have the flexibility of the **target** field in their request header. Support for these protocols must be negotiated through their **object_key** field. The encoding for pre-GIOP 1.2 versions shall be required to mark the first four octets of the sequence with 'MIOP.' ORB vendors on the receiving side that do not recognize the object key format will have to check the beginning of the sequence for the presence of the literal value 'MIOP.' The remainder of the sequence will contain the UIPMC profile as an encapsulation. This data can then be used by the object adapter to correctly dispatch the request after it is extracted from the **object_key** field.

29.16 MIOP Request Efficiency

There are two efficiency related scenarios for invoking requests via a UIPMC profile. Since the semantics of these requests are one-way with no remote status or exceptions, there is no way to detect the failure of a request message.

In one scenario there could be a registered object group with no members. Although this is not desirable, it is perfectly legal in MIOP since the notion of group membership is not enforced or discussed. The client in this case would be broadcasting messages to no recipients. There is nothing specified in this specification keep this from occurring. It will be the responsibility of the participating applications to make sure that group resources are cleaned up by one of the participants and that listening applications exist and are cooperating with the broadcasters.

A different scenario involves a client invoking requests on a non-existent object group or one that has been destroyed by a participating application. This scenario is detectable if the MGM is present. It is therefore advisable that the requesting objects periodically poll the MGM for the presence of the object group via two of the **CORBA::Object** implicit operations **is_a()** and **non_existent()**. These operations will return the values **false** and **true** respectively in the event that the object group no longer exists. Another alternative would be to invoke **MGM::ObjectGroupFactory::get_object_group_ref()**, which would raise an exception in the event that the object group had been deleted or could not be found.

29.17 Client Use Cases

The following sections will address specific scenarios that clients would potentially use to initiate their communication with a multicast object group.

29.17.1 Using the MGM

29.17.1.1 Creating/Finding an Object Group

The application responsible for creating object groups obtains the IOR of the **MGM::ObjectGroupManager**. The application responsible for creating object groups invokes **create_object()** and specifies a list of properties to create an object group. The IOR that is returned to the client potentially contains an IIOP profile for the MIOP gateway and a UIPMC profile for multicast object group. This IOR is made available to participating group applications via some applications determined mechanism such as the Naming Service.

29.17.2 No MGM is Present

The application responsible for creating object groups creates an object group reference via the corbaloc scheme. The application is responsible to make sure that the **TAG_GROUP** information in the profile's component field is unique. This IOR is made available to participating group applications via some applications determined mechanism such as the Naming Service.

29.17.3 Gateway Application is Used

The presence of a gateway is usually an indication that the sender or some subset of senders are not MIOP aware. MIOP unaware sending applications simply invoke on the gateway reference as they would any IIOP IOR and ignore the UIPMC profile.

29.17.4 Sender is MIOP Aware

The sending application potentially acquires a group reference with all possible profiles:

- a UIPMC profile with a **PortableGroup::GroupInfo** and a **PortableGroup::GroupIIOPProfile** component; and
- an IIOP profile of the gateway.

The client will choose the UIPMC profile and send its MIOP request via this profile.

29.17.5 Sender is MIOP Unaware

The sending application potentially acquires a group reference with all possible profiles:

- a UIPMC profile with a **PortableGroup::GroupInfo** and a **PortableGroup::GroupIIOPProfile** component; and
- an IIOP profile of the gateway.

The client will always choose the IIOP gateway profile and send its request via this profile and ignore the UIPMC profile.

29.18 Server Use Cases

The following sections will address specific scenarios that server ORBs would potentially use to initiate their communication with a multicast object group.

29.18.1 Using an Object Group

1. The receiving ORB acquires a group IOR from the Naming Service.
2. The server ORB associates the group reference with a **PortableServer::ObjectId** to provide a mechanism for dispatching requests to this servant object or collection of servant objects.
3. The server ORB will have to use the destination endpoints in the UIPMC profile to read messages from sending applications. This can be done ahead of time if the endpoints are well known or dynamically from the acquired group reference.

29.18.2 Gateway Application is Used

Since the gateway is acting as an advocate of the sending applications, there is no effect on the server applications. They would behave as defined in the previous section.

Appendix A Conformance

A.1 Summary of Optional Verses Mandatory Interfaces

An interface to an MIOP gateway should be considered an optional interface.

A.2 Proposed Compliance Points

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

A.3 Changes to Other OMG Specifications

This specification contains an extension to the PortableServer module and the POA interface as well as additions to module IOP.

```
module PortableServer
...
    exception NotAGroupObject {};
    typedef sequence <ObjectId> IDs;

    interface POA {
        ...
        ObjectId
            create_id_for_reference(in CORBA::Object the_ref)
            raises (NotAGroupObject);

        IDs
            reference_to_ids (in CORBA::Object the_ref)
                raises (NotAGroupObject);

        void
            associate_reference_with_id
                (in CORBA::Object ref, in ObjectId oid)
                raises(NotAGroupObject);

        void
            disassociate_reference_with_id
                (in CORBA::Object ref, in ObjectId oid)
                raises(NotAGroupObject);

    }; // end interface POA
...
}; // end module PortableServer
```

Appendix B Consolidated IDL

B.1 OMG IDL

```
#ifndef MIOP_IDL
#define MIOP_IDL

#include <IOP.idl>;
#include "GIOP.idl"
#pragma prefix "omg.org"
module MIOP
{
  typedef sequence <octet, 252> UniqueId;
  struct PacketHeader_1_0
  {
     char              magic[4];
     octet             hdr_version;
     octet             flags;
     unsigned short    packet_length;
     unsigned long     packet_number;
     unsigned long     number_of_packets;
     UniqueId          Id;
  };

     typedef GIOP::Version Version;

     typedef string Address;

     struct UIPMC_ProfileBody
     {
          Version         miop_version;
          Address         the_address;
          short           the_port;
          sequence<IOP::TaggedComponents> components;
     };
};
#endif


#ifndef _PortableGroup_IDL_
#define _PortableGroup_IDL_

#include "CosNaming.idl" // 98-10-19.idl
#include "IOP.idl" // from 98-03-01.idl
#include "GIOP.idl" // from 98-03-01.idl
#include "CORBA.idl" // from 98-03-01.idl
#pragma prefix "omg.org"

module IOP {
     const ProfileId          TAG_UIPMC          = OMG_assigned;
```

```
    const ComponentId    TAG_GROUP        = OMG_assigned;
    const ComponnetId    TAG_GROUP_IIOP = OMG_assigned
};

module PortableGroup {

    // Specification for Interoperable Object Group References
    typedef GIOP::Version Version;
    typedef string GroupDomainId;
    typedef unsigned long long ObjectGroupId;
    typedef unsigned long ObjectGroupRefVersion;

    struct TagGroupTaggedComponent { // tag = TAG_GROUP;
        GIOP::Version group_version;
        GroupDomainId group_domain_id;
        ObjectGroupId object_group_id;
        ObjectGroupRefVersion object_group_ref_version;
    };

    typedef sequence <octet> GroupIIOPProfile; // tag = TAG_GROUP_IIOP

    // Specification of Common Types and Exceptions
    // for GroupManagement
    interface GenericFactory;

    typedef CORBA::RepositoryId TypeId;
    typedef Object ObjectGroup;
    typedef CosNaming::Name Name;
    typedef any Value;

    struct Property {
        Name nam;
        Value val;
    };

    typedef sequence<Property> Properties;
    typedef Name Location;
    typedef sequence<Location> Locations;
    typedef Properties Criteria;

    struct FactoryInfo {
        GenericFactory the_factory;
        Location the_location;
        Criteria the_criteria;
    };

    typedef sequence<FactoryInfo> FactoryInfos;
    typedef long MembershipStyleValue;

    const MembershipStyleValue MEMB_APP_CTRL = 0;
    const MembershipStyleValue MEMB_INF_CTRL = 1;
```

```
typedef unsigned short InitialNumberReplicasValue;
typedef unsigned short MinimumNumberReplicasValue;

exception InterfaceNotFound {};
exception ObjectGroupNotFound {};
exception MemberNotFound {};
exception ObjectNotFound {};
exception MemberAlreadyPresent {};
exception BadReplicationStyle {};
exception ObjectNotCreated {};
exception ObjectNotAdded {};
exception UnsupportedProperty {
    Name nam;
};

exception InvalidProperty {
    Name nam;
    Value val;
};

exception NoFactory {
    Location the_location;
    TypeId type_id;
};

exception InvalidCriteria {
    Criteria invalid_criteria;
};

exception CannotMeetCriteria {
    Criteria unmet_criteria;
};

// Specification of PropertyManager Interface
interface PropertyManager {

    void set_default_properties
        (in Properties props)
            raises (InvalidProperty, UnsupportedProperty);

    Properties get_default_properties();

    void remove_default_properties
        (in Properties props)
            raises (InvalidProperty, UnsupportedProperty);

    void set_type_properties
        (in TypeId type_id, in Properties overrides)
            raises (InvalidProperty, UnsupportedProperty);
```

```
        Properties get_type_properties(in TypeId type_id);

        void remove_type_properties
            (in TypeId type_id, in Properties props)
                raises (InvalidProperty, UnsupportedProperty);

        void set_properties_dynamically
            (in ObjectGroup object_group, in Properties overrides)
                raises
                (ObjectGroupNotFound,
                 InvalidProperty,
                 UnsupportedProperty);

        Properties get_properties
            (in ObjectGroup object_group)
                raises(ObjectGroupNotFound);

    };  // endPropertyManager

// Specification of ObjectGroupManager Interface
    interface ObjectGroupManager {

        ObjectGroup create_member
            (in ObjectGroup object_group,
             in Location the_location,
             in TypeId type_id,
             in Criteria the_criteria)
            raises
                (ObjectGroupNotFound,
                 MemberAlreadyPresent,
                 NoFactory,
                 ObjectNotCreated,
                 InvalidCriteria,
                 CannotMeetCriteria);

        ObjectGroup add_member
            (in ObjectGroup object_group,
             in Location the_location,
             in Object member)
            raises
                (ObjectGroupNotFound,
                 CORBA::INV_OBJREF,
                 MemberAlreadyPresent,
                 ObjectNotAdded);

        ObjectGroup remove_member
            (in ObjectGroup object_group,
             in Location the_location)
        raises
            (ObjectGroupNotFound, MemberNotFound);
```

```
        Locations locations_of_members
            (in ObjectGroup object_group) raises(ObjectGroupNotFound);

        ObjectGroupId get_object_group_id
            (in ObjectGroup object_group) raises(ObjectGroupNotFound);

        ObjectGroup get_object_group_ref
            (in ObjectGroup object_group) raises(ObjectGroupNotFound);

        Object get_member_ref
            (in ObjectGroup object_group,
             in Location loc)
        raises(ObjectGroupNotFound, MemberNotFound);
    }; //  end ObjectGroupManager

    // Specification of GenericFactory Interface
    interface GenericFactory {
        typedef any FactoryCreationId;

        Object create_object
            (in TypeId type_id,
             in Criteria the_criteria,
             out FactoryCreationId factory_creation_id)
        raises
            (NoFactory,
             ObjectNotCreated,
             InvalidCriteria,
             InvalidProperty,
             CannotMeetCriteria);

        void delete_object
            (in FactoryCreationId factory_creation_id)
                raises (ObjectNotFound);
    }; // end GenericFactory

}; // end PortableGroup
#endif // for #ifndef _PortableGroup_IDL_


#ifdef _MGM_idl
#define _MGM_idl
#include “PortableGroup.idl”
module MGM {

    // Property values

    typedef long GroupCreationMode
    const GroupCreationMode CREATE_ADDRESS_DEFERED = 0;
    const GroupCreationMode CREATE_ADDRESS_GENERATED = 1;
    const GroupCreationMode CREATE_ADDRESS_SUPPLIED = 2;
```

```
        interface ObjectGroupFactory :
            PortableGroup::GenericFactory,
            PortableGroup::PropertyManager,
            PortableGroup::ObjectGroupManager {}

};
#endif // _MGM_idl
```