Date: May 2009

GIOP Compression

FTF Beta 2

OMG Document Number: ptc/2009-05-07
Standard document URL: http://www.omg.org/spec/ZIOP/Beta2

Source document: General Compression submission (mars/2008-12-20) and Beta1 (ptc/09-01-03)

This OMG document replaces the Beta 1 document (ptc/09-01-03). It is an OMG Adopted Beta Specification and is currently in the finalization phase. You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on July 10, 2009. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and Interface Specifications

- CORBAservices

- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. (as of January 16, 2006) at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

This specification defines a compression mechanism for the CORBA GIOP protocol. Such a mechanism provides a way for servers to publish objects which accept compressed requests and for clients to make compressed invocations. Pluggable compression algorithms could additionally be defined by clients.

2 Conformance

This specification defines an optional CORBA conformance point. In order to claim ZIOP compliance an ORB implementations must support the following conformance point:

- ZIOP - The ORB implements the ZIOP wire protocol and the ZIOP module policy interfaces for controlling it, with support for at least the zlib algorithm.

When an ORB claims ZIOP compliance it optionally can claim the following ZIOP compliance point:

- Pluggable compression - The ORB implements the Compression module interfaces, and the registered CompressorFactory instances are available for use by ZIOP.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- OMG CORBA 3.1 specification, formal/08-01-05

4 Terms and Definitions

For the purposes of this specification, the terms and definitions given in the normative reference and the following apply.

Policy

The term policy in this document describes CORBA objets that implement the CORBA::Policy interface. See CORBA 3.1, chapter 8.8.

Compressor

An entity which provides compression and decompression of octet sequences.

CompressionRatio

The numerical relation between compressed and original uncompressed sequences.

5 Symbols

List of symbols/abbreviations.

ZIOP - Zipped Inter-ORB protocol

GIOP - Generic Inter-ORB protocol

ORB - Object Request Broker

CORBA - Common Object Request Broker Architecture

IOR - Interoperable Object Reference

6 Additional Information

6.1 Overview of this Specification

This specification describes the compression and ZIOP additions to the CORBA specification. The intended audiences are CORBA vendors and users.

6.2 Changes to Adopted OMG Specifications

This specification adds the following to CORBA 3.1 specification:

- A set of new POA Policies: CompressionEnablingPolicy, CompressionIdLevelListPolicy, CompressionLowValuePolicy, CompressionMinRatioPolicy
- A new initial reference retrievable from the ORB's resolve_initial_references operation: CompressionManager
- A new ZIOP message with compressed data.

6.3 How to Read this Specification

The rest of this document contains the technical content of this specification.

6.4 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Dion Picco, IONA Technologies, The IONA building, Shelbourde Road, Dublin 4, Ireland, dion.picco@iona.com
- Johnny Willemsen, Remedy IT, Postbus 101, 2650 AC Berkel en Rodenrijs, The Netherlands, jwillemsen@remedy.nl
- Alvaro Vega García, Telefónica I+D, C/ Emilio Vargas 6, 28043 Madrid, Spain avega@tid.es

6.5 Proof of Concept

This specification describes the ZIOP pluggable protocol implemented in TIDorbJ and TIDorbC++ by Telefonica I+D and TAO by Remedy IT.

7 Description

7.1 Goal

CORBA is deployed in numerous areas where the bandwidth is restricted. Such environments may operate with antiquated network infrastructure, or the network infrastructure may be overloaded. In such environments, reducing the bandwidth used by each communication request made between a client and server is desirable.

An example of such an environment is aviation, where a relatively large amount of information (such a flight charts, passenger and route data) must be passed to a remote location in a timely manner. Another example would be retail banking, particularly in developing economies, where remote bank branches may be connected to a central server only over a dial-up modem connection.

A rise on the CPU overload is expected in this environment in order to reduce data length to be transmitted by the wire. It is needed to define a configurable way to indicate in which circumstances (source data length, compressed ratio obtained) compression is applied or not.

7.2 ZIOP Overview

The new ZIOP protocol is the result to apply compression to GIOP. ZIOP is the same as GIOP Compression. ZIOP is the way to introduce compression between CORBA parties with the aim to reduce the amount of data to be transmitted on the wire. In a CORBA communication which uses ZIOP protocol, the GIOP message is compressed using a specific compression algorithm. For this purpose a compressed message is be defined as ZIOP message.

A set of new compression CORBA Policies related with ZIOP are defined to activate and communicate to other ORBs the available compression functionalities.

The compression features will be provided to ZIOP protocol by some entities. The Compressor which will be in charge of basic compression and decompression operations. The CompressorFactory will create Compressors and then CompressorFactory will be registered by the CompressionManager interface.

ORB vendors may deliver ZIOP through pluggable compressors or support a standard and well known compression algorithm.

7.3 Compression Module Interfaces

The Compression module provides a set of interfaces to create and register entities which provides compression and decompression functionalities. These features may be used in stand-alone mode, to obtain compressed and decompressed CORBA octet sequences, or internally by the ORB to compress GIOP messages when the ZIOP protocol is enabled.

The Compressor interface is an abstraction which provides the basic mechanism to compress and decompress CORBA octet sequences. The compressor collects statistical information about its compression. A specific compressor is identified by its CompressorId. CompressorIds are maintained by the OMG, vendors and users must request specific CompressorIds for their own compressors.

The CompressorFactory interface is a factory to create different compressors using a particular algorithm depending on its compression level.

The CompressionManager interface is an ORB initial reference for register CompressorFactories depending on its compression algorithm.

All these entities, Compressor, CompressorFactory, and CompressionManager are local CORBA interfaces.

The Compression module provides the way to easily create custom compressors. The procedure involves two steps. First, the user provides an implementation of CompressorFactory and Compressor interfaces. Second, this new custom CompressorFactory must be registered in the CompressionManager to make it accessible through the ORB services.

The zlib compressor must be provided by default and may be used easily as another CORBA feature. Also it must be possible to implement a new custom compressor by implementing the Compressor interface.

7.3.1 Compressor interface

This interface is an abstraction of a specific algorithm for compression and decompression. All different algorithms implementations will support this common interface.

```
// IDL
module Compression {
    exception CompressionException {
        long reason;
        string description;
    };
    typedef CORBA::OctetSeq Buffer;
    typedef unsigned short CompressionLevel;
    typedef float CompressionRatio;
    local interface Compressor {
        void compress(
                in Buffer source,
                inout Buffer target)
            raises (CompressionException);
        void decompress(
                in Buffer source,
                inout Buffer target)
            raises (CompressionException);
        readonly attribute CompressorFactory compressor_factory;
        readonly attribute CompressionLevel compression_level;
        readonly attribute unsigned long long compressed_bytes;
        readonly attribute unsigned long long uncompressed_bytes;
        readonly attribute CompressionRatio compression_ratio;
    };
};
```

7.3.1.1 CompressionException

This exception is thrown when the compress or decompress fail. The reason can be used by the concrete compressor implementation to give some feedback on the technical reason of the failure. This could be used by application mode when they are aware of the concrete compressor and their list of possible reasons. Because there are a lot of different compression algorithms with each different possible error reasons we don’t want to attempt to list all possible error reasons. When the underlying compression algorithm has the possibility to retrieve an error string this will be put in the description field.

7.3.1.2 compress

This operation compresses the data contained in a source buffer into the target buffer. If an error occurs during the compression, it throws a CompressionException. The buffer is an octet sequence that could be extended with ORB specific operations.

7.3.1.3 decompress

This operation decompresses the data contained in the source buffer into the target buffer. If an error occurs during the decompression, it throws a CompressionException. The buffer is an octet sequence that could be extended with ORB specific operations.

7.3.1.4 compressor_factory

This attribute represents the object reference to CompressorFactory which created this Compressor.

7.3.1.5 compression_level

This attribute represents, for the specific algorithm, the compression level that will be applied using this Compressor. For ZIOP we define that 0 means no compression, 1 low compression, 9 the highest compression available.

7.3.1.6 compressed_bytes

This attribute represents the total number of compressed bytes written by this compressor during compression (i.e, the “target” argument of Compressor::compress). This information could be useful for statistical purposes.

7.3.1.7 uncompressed_bytes

This attribute represents the total number of uncompressed bytes read by this compressor during compression (i.e., the “source” argument of Compressor::compress). This information could be useful for statistical purposes.

7.3.1.8 compression_ratio

This attribute represents the compression ratio achieved by this compressor. The ratio must be obtained with the following formula: compressed_bytes / uncompressed_bytes.

7.3.2 CompressorFactory Interface

The CompressorFactory interface allows the retrieval of a Compressor with a particular algorithm implementation Compres-sors are retrieved for a specific compression level.

```
// IDL
local interface CompressorFactory {
    readonly attribute CompressorId compressor_id;

    Compressor get_compressor(in CompressionLevel compression_level);
};
```

7.3.2.1 compressor_id

This attribute represents the specific compression algorithm associated with this CompressorFactory. All Compressors retrieved from this factory use this algorithm.

7.3.2.2 get_compressor

This operation retrieves a Compressor instance with the given compression level. Calling this operation multiple times with the same compression level should return the same instance. The CompressorFactory is responsible for managing the lifetime of the Compressors. If a compression level > 9 is passed a BAD_PARAM exception with minor code 44 is raised.

7.3.3 CompressionManager Interface

This is the interface to register and unregister CompressorFactories objects with an ORB. It is obtained by resolving initial references: "CompressionManager."

```
// IDL
exception FactoryAlreadyRegistered {
};
exception UnknownCompressorId {
};
local interface CompressionManager {
    void register_factory(
            in CompressorFactory compressor_factory)
        raises (FactoryAlreadyRegistered);
    void unregister_factory(
            in CompressorId compressor_id)
        raises (UnknownCompressorId);
    CompressorFactory get_factory(
            in CompressorId compressor_id)
        raises (UnknownCompressorId);
    Compressor get_compressor(
            in CompressorId compressor_id,
            in CompressorLevel compression_level)
    raises (UnknownCompressorId);
    CompressorFactorySeq get_factories();
};
```

7.3.3.1 register_factory

This operation registers a new CompressorFactory.

7.3.3.2 unregister_factory

This operation unregisters a CompressorFactory with the given CompressorId from the CompressionManager.

7.3.3.3 get_factory

This operation retrieves a CompressorFactory with the given CompressorId from the CompressionManager.

7.3.3.4 get_compressor

This operation retrieves a Compressor with the given compression_level from the CompressorFactory with the given CompressorId. Calling this operation multiple times with the same compressor id and compression level should return the same instance. If a compression level > 9 is passed a BAD_PARAM exception with minor code 44 is raised.

7.3.3.5 get_factories

This operation lists all registered CompressorFactories in the CompressionManager.

7.3.4 Compression Usage Scenario

This subsection provides an example about how to use Compression facilities.

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

CORBA::Object_var cm_obj =
   orb->resolve_initial_references("CompressionManager");
Compression::CompressionManager_var cm =
   Compression::CompressionManager::_narrow(cm_obj);
Compression::Compressor_var compressor =
   cm->get_compressor (Compression::COMPRESSORID_ZLIB, 9);

CORBA::ULong const max_length = 65000;
Compression::Buffer source;
source.length(max_length);
for (CORBA::ULong i = 0; i < max_length; i++)
   source[i] = (CORBA::Octet)'A';

Compression::Buffer compressed;
Compression::Buffer uncompressed;

cout << "[Tester] source sequence length = " << source.length() << endl;
compressor->compress(source, compressed);
cout << "[Tester] compressed sequence length = " << compressed.length() <<
endl;
compressor->decompress(compressed, uncompressed);
cout << "[Tester] uncompressed sequence length = " << uncompressed.length() <<
endl;
```

7.4 ZIOP Protocol

ZIOP Protocol is a mechanism which in some particular circumstances applies compression to a GIOP message.

7.4.1 ZIOP Messages

A ZIOP message is a GIOP message which has ZIOP as first four magical bytes instead of the regular GIOP magical bytes.

GIOP compression can be applied to send or receive GIOP 1.2 and higher messages and includes fragmented messages.

```
// PIDL: ZIOP body in ZIOP Message
module ZIOP {
    struct CompressionData {
        Compression::CompressorId compressor;
        unsigned long original_length;
        Compression::Buffer data;
    };
};
```

A ZIOP message defines how the application data of the GIOP Messages is compressed: when the magic bytes are ZIOP then the data after the GIOP MessageHeader is replaced by the CompressionData structure, which contains the following items encoded in this order:

1. compressor: contains the identifier which indicates the compressor used for the current ZIOP message.
2. original_length: contains an unsigned long value which represents the GIOP body length of the current GIOP message without applying any compression.
3. data: is an octet sequence which contains the compressed message.

The length in the GIOP Header is updated to reflect the new message length, the other fields are unchanged as it is described below.

To allow interoperability between a ZIOP and a non ZIOP party the client that supports ZIOP will send only ZIOP messages to servers which have been declared to accept ZIOP messages.

At message level, the sequence of message exchange is as follows:

1. When client and server ORB support a compatible compression algorithm and if the message fulfills the compression policies (for example message size threshold) the message is compressed and the four magic start bytes are changed to ZIOP. The length field in the GIOP MessageHeader is updated, all other fields are unchanged.
2. The server ORB, reads the ZIOP header. It then takes the CompressionData struct and uncompresses the data. The other fields of the header and the uncompressed data can then be used as a regular GIOP message.
3. In the server side, if the GIOPReply message fulfills the compression policies, a compressor object is retrieved and server ORB will generate a compressed GIOP Reply and will sent it to client where the magic bytes in the header are set to ZIOP.
4. The client ORB side will read the ZIOP magic bytes message and then will continue reading the compressed GIOP Reply and decompressing the GIOPBody.

Both client and server only send ZIOP messages when it knows that the remote ORB supports ZIOP and it has a compatible compressor implementation, as is described in the following section.

7.4.2 ZIOP Message use

Client and server ORBs interchange available compression details through a set of new ZIOP CORBA Policies. These policies must be propagated as standards CORBA Policies in a ServiceContext into a GIOP Request and GIOP Reply messages. They may also be propagated into an IOR by using the Messaging propagation of QoS. Policies which values are transferred to the remote ORB are called ‘client-exposed’ policies. The Messaging propagation mechanism is described in detail in section 17.3 of the CORBA 3.1 specification.

ORB server side applications may set available compression algorithms via appointing ZIOP Policies list to the POA that will create object references which embed these policies into the IOR component. The client side ORB could send ZIOP messages defining similar Policies using PolicyContext interfaces, at ORB, thread or reference level.

As it was described before servers and clients must agree on which compression algorithm will be used. To allow this, each party must know if the other party supports ZIOP and its preferences about compression before send to it a ZIOP message.

The server must register the CORBA object in a POA that was created with ZIOP Policies. These ZIOP Policies will be transmitted as part of the IOR through the Messaging QoS Profile Component. The client may indicate through 'set_policies_overrides' over the remote CORBA object reference the ZIOP Polices which it has as preferences.

The client-side ORB will decide the compatible ZIOP Policies list which the ORB must use to send a GIOPRequest to the server. For this, the client-side ORB will extract the compression server preferences (ZIOP Policies) from a TaggedComponent of an IOR if it is present. The client will select a compression algorithm and send the application data compressed to the server. The client-side ORB will also create a Policy list with its compression policies and send them in the Request as a Messaging ServiceContext.

The server-side ORB will reply to the request taking into account the ZIOP Policies that it found in the ServiceContext of the ZIOP messaging and compare it with the ZIOP Policies of the POA object.

If the server does not allow receipt of compressed GIOP Requests, then the client-side ORB should not send any GIOP compressed messages. Instead, the client-side ORB will only send the ZIOP Policies values that the client supports in Messaging ServiceContext. In a similar way a server may not respond to a client with a compressed GIOP Reply if the client does not support GIOP compression.

In this way, a client and server may decide independently if compression could be used or not. There is no necessity to exchange CORBA messages between client and server to obtain the best set of ZIOP Policies to be applied in communication to get the optimal performance.

7.4.3 ZIOP Compression Policies

This module ZIOP provides all necessary elements to allow interchange of compressed GIOP messages between client and servers using mechanisms defined in Compression module. If a specific policy is not supplied, then an ORB default is used. The following interfaces are the ZIOP policies.

7.4.3.1 CompressionEnablingPolicy interface

This interface represents the ZIOP policy CompressionEnablingPolicy that has a boolean attribute indicating if compression is enabled or not by the tier. Only when this policy has been set to true ZIOP may be used by the ORB. This policy is client-exposed and both client and server must have set this policy to TRUE in order to enable ZIOP.

7.4.3.2 CompressorIdLevelListPolicy interface

This interface represents the ZIOP policy CompressorIdLevelListPolicy. It has a list of CompressorId/CompressionLevel attributes indicating the compression algorithms with their respective levels that may be used. The CompressorIdLevelListPolicy contains a sequence of structures and this sequence is ordered by preference priority. This policy is client-exposed, the client/server will take its own sequence and search for the first CompressorId that is also supported by the other tier. For this Compressor then the lowest CompressionLevel is selected.

7.4.3.3 CompressionLowValuePolicy interface

This interface represents the ZIOP policy CompressionLowValuePolicy. It has an unsigned long attribute indicating the minimum size of application data that has to be sent before the ORB will consider this as a ZIOP message. This policy is not client exposed.

7.4.3.4 CompressionMinRatioPolicy interface

This interface represents the ZIOP policy CompressionMinRatioPolicy. It has an float attribute indicating the minimum compression ratio that must be obtained at compression time to send with a compressed GIOP message. This policy tries to prevent the sending of compressed messages with few improvements about the original size in order to not overload the server with a useless decompression process. The ratio must be obtained with the following formula: compressed_length / original_length. This policy is not client exposed.

7.4.4 Propagation of ZIOP Compression Policies

ZIOP Compression policies are transferred using the Messaging QoS Profile Component which is defined in section 17.3 of the CORBA 3.1 specification. That section also describes the concept of client-exposed policies.

7.4.5 ZIOP Usage Scenario

This section describes a client-server communication through ZIOP protocol.

7.4.5.1 Client

```
CORBA::ORB_ptr orb = CORBA::ORB_init (argc, argv);

CORBA::Boolean compression_enabling = true;
Compression::CompressorId compressor_id = Compression::COMPRESSORID_ZLIB;
Compression::CompressorIdLevelList compressor_id_list(1);
compressor_list.length(1);
compressor_list[0].compressor_id = compressor_id;
compressor_list[0].level = 9
CORBA::ULong compression_low_value = 32000;
Compression::CompressionRatio compression_min_ratio = 0.30;

CORBA::Any enabling_any, compressors_any, low_value_any, min_ratio_any;

enabling_any    <<= CORBA::Any::from_boolean(compression_enabling);
compressors_any <<= compressor_list;
low_value_any   <<= compression_low_value;
min_ratio_any   <<= compression_min_ratio;

CORBA::PolicyList policies(4);
policies.length(4);

try {
   policies[0] = orb->create_policy(ZIOP::COMPRESSION_ENABLING_POLICY_ID,
                          enabling_any);
   policies[1] = orb->create_policy(ZIOP::COMPRESSOR_ID_LEVEL_LIST_POLICY_ID,
                          compressors_any);
   policies[2] = orb->create_policy(ZIOP::COMPRESSION_LOW_VALUE_POLICY_ID,
                        low_value_any);
   policies[3] = orb->create_policy(ZIOP::COMPRESSION_MIN_RATIO_POLICY_ID,
                        min_ratio_any);
} catch(const CORBA::PolicyError&) {
   policies.length(0);
}

CORBA::Object_var obj = orb->string_to_object(uri);
CORBA::Object_var obj2 = CORBA::Object::_nil();

try{
   obj2 = obj->_set_policy_overrides(policies, CORBA::ADD_OVERRIDE);
} catch(const CORBA::SystemException&) {
   obj2 = obj;
}
Echo::Test_var test_ref = Echo::Test::_narrow(obj.in ());
Echo::Test_var ziop_test_ref = Echo::Test::_narrow(obj2.in ());
CORBA::String_var str = test_ref->echo(message);
CORBA::String_var str = ziop_test_ref->echo(message);
```

7.4.5.2 Server

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
PortableServer::POAManager_var mgr = poa->the_POAManager();

CORBA::Boolean compression_enabling = true;
Compression::CompressorId compressor_id = Compression::COMPRESSORID_ZLIB;
Compression::CompressorIdLevelList compressor_id_list(1);
compressor_list.length(1);
compressor_list[0].id = compressor_id;
compressor_list[0].level = 5;
CORBA::ULong compression_low_value = 16384;
Compression::CompressionRatio min_compression_ratio = 0.40;

CORBA::Any enabling_any, compressors_any, low_value_any, min_ratio_any;

enabling_any    <<= CORBA::Any::from_boolean(compression_enabling);
compressors_any <<= compressor_list;
low_value_any   <<= low_value;
min_ratio_any   <<= min_ratio;

PortableServer::POA_var my_compress_poa = PortableServer::POA::_nil();

CORBA::PolicyList policies(4);
policies.length(4);

try {
   policies[0] = orb->create_policy(ZIOP::COMPRESSION_ENABLING_POLICY_ID,
                          compression_enabling_any);
   policies[1] = orb->create_policy(ZIOP::COMPRESSOR_ID_LEVEL_LIST_POLICY_ID,
                        compressors_any);
   policies[2] = orb->create_policy(ZIOP::COMPRESSION_LOW_VALUE_POLICY_ID,
                        low_value_any);
   policies[3] = orb->create_policy(ZIOP::MIN_COMPRESSION_RATIO_POLICY_ID,
                        min_ratio_any);
   my_compress_poa = poa->create_POA("My_Compress_Poa",
                        PortableServer::POA::_nil (), policies);
} catch(const CORBA::PolicyError&) {
   policies.length(0);
   my_compress_poa = poa->create_POA("My_Compress_Poa",
                     PortableServer>>POA::_nil (), policies);
}

PortableServer::POAManager_var my_compress_poa_mgr =
   my_compress_poa->the_POAManager();

my_compress_poa_mgr->activate();

PortableServer::ObjectId_var oid = my_compress_poa->activate_object (servant);

CORBA::Object_var ref = poa->id_to_reference (oid.in ());
```

Annex A

(normative)

Compression IDL

```
#pragma prefix "omg.org"
module Compression {
    exception CompressionException {
        long reason;
        string description;
    };
    exception FactoryAlreadyRegistered { };
    exception UnknownCompressorId { };

    typedef unsigned short CompressorId { };
    const CompressorId COMPRESSORID_NONE = 0;
    const CompressorId COMPRESSORID_GZIP = 1;
    const CompressorId COMPRESSORID_PKZIP = 2;
    const CompressorId COMPRESSORID_BZIP2 = 3;
    const CompressorId COMPRESSORID_ZLIB = 4;
    const CompressorId COMPRESSORID_LZMA = 5;
    const CompressorId COMPRESSORID_LZO = 6;
    const CompressorId COMPRESSORID_RZIP = 7;
    const CompressorId COMPRESSORID_7X = 8;
    const CompressorId COMPRESSORID_XAR = 9;

    typedef unsigned short CompressionLevel;
    typedef float CompressionRatio;

    struct CompressorIdLevel {
        CompressorId compressor_id;
        CompressionLevel compression_level;
    }
    typedef sequence <CompressorIdLevel> CompressorIdLevelList;

    typedef CORBA::OctetSeq Buffer;

    local interface Compressor {
        void compress(
            in Buffer source,
            inout Buffer target)
                raises (CompressionException);
        void decompress(
            in Buffer source,
            inout Buffer target)
                raises (CompressionException);
        readonly attribute CompressorFactory compressor_factory;
        readonly attribute CompressionLevel compression_level;
        readonly attribute unsigned long long compressed_bytes;
        readonly attribute unsigned long long uncompressed_bytes;
        readonly attribute CompressionRatio compression_ratio;
    };

    local interface CompressorFactory {
        readonly attribute CompressorId compressor_id;

        Compressor get_compressor(in CompressionLevel compression_level);
    };

    typedef sequence<CompressorFactory> CompressorFactorySeq;
    local interface CompressionManager {
        void register_factory(
            in CompressorFactory compressor_factory)
                raises (FactoryAlreadyRegistered);
        void unregister_factory(
            in CompressorId compressor_id)
                raises (UnknownCompressorId);
        CompressorFactory get_factory(
            in CompressorId compressor_id)
                raises (UnknownCompressorId);
        Compressor get_compressor(
            in CompressorId compressor_id,
            in CompressorLevel compression_level)
                raises (UnknownCompressorId);
        CompressorFactorySeq get_factories();
    };
};
```

Annex B
(normative)

ZIOP IDL

```
#pragma prefix “omg.org”
module ZIOP {
    struct CompressedData {
        Compression::CompressorId compressorid;
        unsigned long original_length;
        Compression::Buffer data;
    };

    typedef boolean CompressionEnablingPolicyValue;

    const CORBA::PolicyType COMPRESSION_ENABLING_POLICY_ID = 64;

    local interface CompressionEnablingPolicy: CORBA::Policy
    {
        readonly attribute CompressionEnablingPolicyValue compression_enabled;
    };

    const CORBA::PolicyType COMPRESSOR_ID_LEVEL_LIST_POLICY_ID = 65;

    local interface CompressionIdLevelListPolicy: CORBA::Policy
    {
        readonly attribute Compression::CompressorIdLevelList compressor_ids;
    };

    typedef unsigned long CompressionLowValuePolicyValue;

    const CORBA::PolicyType COMPRESSION_LOW_VALUE_POLICY_ID = 66;

    local interface CompressionLowValuePolicy: CORBA::Policy
    {
        readonly attribute CompressionLowValuePolicyValue low_value;
    };

    const CORBA::PolicyType COMPRESSION_MIN_RATIO_POLICY_ID = 67;

    local interface CompressionMinRatioPolicy: CORBA::Policy
    {
        readonly attribute Compression::CompressionRatio ratio;
    };
};
```