**Date**: May 2008

# GIOP Compression RFP initial submission

In response to OMG RFP mars/2007-06-10

mars/2008-xx-xx

**Contact:**

Juan A. Cáceres
Telefónica I+D
C/ Emilio Vargas 6,
28043 Madrid (Spain)
http://www.tid.es
caceres@tid.es

Johnny Willemsen
Remedy IT
Postbus 101
2650 Berken en Rodenrijs (The Netherlands)
http://www.remedy.nl
jwillemsen@remedy.nl

and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

# 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).

# Preface

## About this Specification

### Overview of this Specification

This RFP initial submission from Telefónica I+D and Remedy IT is based on GIOP Compression RFP initial submission presented by IONA and Remedy IT mars/2007-11-02.

### Intended Audience

CORBA vendors and users

### Organization of this Specification

<brief description of chapters and appendices>

## Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

**Helvetica/Arial - 10 pt. Bold:** OMG Interface Definition Language (OMG IDL) and syntax elements.

**`Courier - 10 pt. Bold:`** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

•Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

## Issues

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

## 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/](http://www.omg.org/).

## OMG Specifications

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

[http://www.omg.org/technology/documents/spec_catalog.htm](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. at:

OMG Headquarters

250 First Avenue
Suite 100
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

# 1 Scope

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

# 2 Conformance

This specification defines two conformance points. ORB implementations must support at least one of these conformance points:

- A default compressor based on the zlib compression algorithm
- A plug-in mechanisms to register compressors

ORB implementations may also optionally support other compressors based on others standard compression algorithms.

# 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 3.0.3 specification

# 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 objects that implement the CORBA::Policy interface. See CORBA 3.0.3, chapter 4.8.1

**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 Changes to Adopted OMG Specifications*

This specification adds the following to CORBA 3.0.3 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 Request/Reply message body with compressed data.

## *6.2 How to Read this Specification*

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

## *6.3 Acknowledgments*

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

## *6.4 Proof of Concept*

This submission describes the ZIOP pluggable protocol implemented lately in TIDorbC++ and TAO product (see www.cs.wustl.edu/~schmidt).

# 7 Description

Many CORBA applications are deployed in environments with restricted bandwidth. Examples of these environments include aviation and retail banking, where applications may operate over a wide area network. Although GIOP is performant when compared to other protocols, some further optimization is possible, by minimizing the amount of information passed across the network through compression.

## *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 that 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 data part of a GIOP Messages is compressed using a specific compression algorithm. For this purpose a new standard Message will 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 is 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 can deliver ZIOP through pluggable compressors or supply a standard and well known compression algorithm.

### 1.1.1 Compression Module Interfaces

The Compression module provides a set of interfaces to create and register entities which provides compression and decompression functionalities. These features can be used in stand-alone mode, to obtain compressed and decompressed

CORBA octet sequences, or internally by ORB to compress GIOP Messages when 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 can request specific Ids 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 in 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 thought the ORB services.

The Zlib compressor is provided by default and can be used easily as another CORBA feature. Also it is possible to implement a new custom compressor by extending the Compressor interface.

### 1.1.2 Compressor interface

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

```
// IDL
module Compression {
    exception CompressionException {
        unsigned long reason;
    };
    typdef 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 unsigned long compression_ratio;
    };
};
```

## 7.2.1.1 compress

This operation compresses data contained in a source buffer into the target buffer. If an error occurs during the compression, it throws CompressionException. The buffer can be an octet sequence or an ORB specific data type.

#### 7.2.1.2 decompress

This operation decompresses data contained in the source buffer into the target buffer. If an error occurs during the decompression, it throws CompressionException. The buffer can be an octet sequence or an ORB specific data type.

#### 7.2.1.3 compressor_factory

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

#### 7.2.1.4 compression_level

This attribute represents, for the specific algorithm, the compression level that will be applied using this Compressor. For ZIOP we define that a low value stands for a low compression, a high value for better compression.

#### 7.2.1.5 compressed_bytes

This attribute represents the total number of compressed bytes read and written by this compressor (i.e. the "target" side of Compressor::compress and the "source" side of Compressor::decompress operations). This information could be useful for statistical purposes.

#### 7.2.1.6 uncompressed_bytes

This attribute represents the total number of uncompressed bytes read and written by this compressor (i.e. the "source" side of Compressor::compress and the "target" side of Compressor::decompress operations). This information could be useful for statistical purposes.

#### 7.2.1.7 compression_ratio

This attribute represents the compression ratio achieved by this compressor. The ratio can be obtained with the following formula: 100 - (compressed_length / original_length)*100.

### 1.1.3 CompressorFactory Interface

The CompressorFactory interface allows the creation of a Compressor with a particular algorithm implementation Compressors having in account the different compression levels allowed.

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

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

#### 7.2.1.8 compressor_id

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

### 7.2.1.9 get_compressor

This operation creates a new Compressor instance with the given compression level.

## 1.1.4 CompressionManager Interface

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

```
// IDL
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();
};
```

### 7.2.1.10 register_factory

This operation registers a new CompressorFactory.

### 7.2.1.11 unregister_factory

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

### 7.2.1.12 get_factory

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

### 7.2.1.13 get_compressor

This operation creates a Compressor with the given compression_level from the CompressorFactory with the given CompressorId.

### 7.2.1.14 get_factories

This operation lists all registered CompressorFactories in the CompressionManager.

## *7.3 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::CompressorFactory_var cf =
    cm->get_factory(Compression::COMPRESSORID_ZLIB);

Compression::CompressionLevel clevel = 9;
Compression::Compressor_var compressor = cf->get_compressor(clevel);

CORBA::ULong 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, 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;
```

# ZIOP Protocol

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

## *7.4 ZIOP Messages*

A ZIOP message is a GIOP message which has set the value TRUE for the ZIOP bit at Flags field of the GIOP Header and has compressed the application data. This ZIOP bit must be selected and reserved by OMG between third and eight bit of Flags field of the GIOP 1.2 Header.

GIOP compression will be only applied to send or receive the following GIOP 1.2 or later messages: GIOPRequest and GIOPReply and includes fragmented messages.

A ZIOP message defines how the application data of the GIOP Messages is compressed: when the ZIOP bit is TRUE then the GIOPRequestBody or GIOPReplyBody  is replaced by the CompressionData structure, which contains the according RequestBody or ReplyBody compressed, and which is marshaled into the CDR encapsulation of the containing Message immediately following the Request/Reply Header.

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

To allow interoperability between a ZIOP and a non ZIOP party the client ORB 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.

2- The server ORB, reads the ZIOP bit at GIOP header and read the compressed GIOP Request uncompressing the GIOPRequestBody.

3- In the server side, if the GIOPReply message fulfills the compression policies, a compressor object is allocated and server ORB will generate a compressed GIOP Reply and will sent it to client.

4- The client ORB side will read the ZIOP bit at GIOP header message and then will continue reading the compressed GIOP Reply and uncompressing the GIOPReplyBody.

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.

## 7.5 ZIOP Message use

Client and server ORBs interchange available compression details through a set of new ZIOP CORBA Policies. These policies can be propagated as standards CORBA Policies in a ServiceContext into a GIOP Request and GIOP Reply messages. They can also be propagated into a IOR by using the Messaging propagation of QoS. This is described in detail in section 22.3 of the CORBA 3.0.3 specification.

ORB server side applications can 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. ORB client side 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 be agreed in GIOP Compression policies to be used. To allow this, each party must know if the other party support GIOP Compression and its preferences about compression before send to it a ZIOP message (GIOPRequest or GIOPReply compressed messages).

The server can 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 a new profile. The client can 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 could 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 server does not allow receipt of compressed GIOP Requests, then the client-side ORB should not send any GIOP compressed message. 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 can not respondto a client with a compressed GIOP Reply if the client does not support GIOP compression.

In this way, a client and server can 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.

At the moment the client changes the ZIOP Policies set, it will transfer these updated policies to the server.

## 7.6 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. The following interfaces are the new ZIOP policies:

### 1.1.5 CompressionEnablingPolicy interface

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

### 1.1.6 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 can be used. The **CompressorIdLevelListPolicy** contains a sequence that 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.

### 1.1.7 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  sentbefore the ORB will consider this as a ZIOP message. This policy is not client exposed..

### 1.1.8 CompressionMinRatioPolicy interface

This interface represents the ZIOP policy CompressionMinRatioPolicy. It has an unsigned long 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 uncompression process. The ratio can be obtained with the following formula: 100 - (compressed_length / original_length)*100. This policy is not client exposed.

## *7.7 Propagation of ZIOP Compression Policies*

ZIOP Compression policies are transferred using the Messaging QoS Profile Component which is defined in section 22.3 of the CORBA 3.0.3 specification.

## *7.8 ZIOP Usage Scenario*

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

### 1.1.1 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;
CORBA::ULong compression_min_ratio = 30;

CORBA::Any enabling_any, compressors_any,  low_value_any;
CORBA::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& pol) {
    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& sys) {
    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);
```

### 1.1.2 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;
CORBA::ULong min_compression_ratio = 40;

CORBA::Any enabling_any,  compressors_any, low_value_any;
CORBA::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& pol) {
    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

# Compression IDL

```
#pragma prefix “omg.org”
module Compression {
    exception CompressionException {
        unsigned long reason;
    };
    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_LZOP = 6;
    const CompressorId COMPRESSORID_RZIP = 7;
    const CompressorId COMPRESSORID_7X = 8;
    const CompressorId COMPRESSORID_XAR = 9;

    typedef unsigned long CompressionLevel;

    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 unsigned long 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

# ZIOP IDL

```
module ZIOP {
    struct CompressedData {
        Compression::CompressorId compressorid;
        unsigned long original_length;
        Compression::Buffer data;
    };

    typedef boolean CompressionEnablingPolicyValue;

    const CORBA::PolicyType COMPRESSION_ENABLING_POLICY_ID = “TBD by OMG”;

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

    const CORBA::PolicyType COMPRESSOR_ID_LEVEL_LIST_POLICY_ID = “TBD by OMG”;

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

    typedef unsigned long CompressionLowValuePolicyValue;

    const CORBA::PolicyType COMPRESSION_LOW_VALUE_POLICY_ID = “TBD by OMG”;

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


    typedef unsigned long CompressionMinRatioPolicyValue;

    const CORBA::PolicyType COMPRESSION_MIN_RATIO_POLICY_ID = “TBD by OMG”;

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