Horus CORBA Reference

Version 2.0 - Jan 2003

Marc Navarro
Dennis Koelma

Intelligent Sensory Information Systems
University of Amsterdam, Faculty of Science
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
koelma@science.uva.nl
http://www.science.uva.nl/~horus/

Contents

1 Corba **2**

1.1 Corba 2

1.2 Corba architecture 2

1.3 Interoperable object references (IOR) 4

1.4 Portable object adapter (POA) 5

1.5 Corba services 8

2 Client and server **10**

2.1 Client and server 10

2.2 Corba configuration 10

2.3 Horus server 11

2.4 C++ client 13

2.5 RGB image display data transfer 14

2.6 Data object management 18

3 Stubs and servants in C++ **19**

3.1 Stubs and servants in C++ 19

4 Stubs and servants in Java **22**

4.1 Stubs and servants in Java 22

5 Servant generator **24**

5.1 Servant Generator 24

5.2 GenType Objects 25

Chapter 1

Corba

1.1 Corba

This overview is based on the book "Advanced CORBA progamming with C++" by Michi Henning and Steve Vinoski.

Corba provides platform independent progamming interfaces and models for portable distributed object-oriented computing applications. In other words, Corba provides access to objects, i.e. sending requests to objects and receiving responses from these objects, independent of the language used for the implementation of the object, the operating system of the computer the object lives on, the location of the computer in the network, the type of network connecting the computers, and the implementation of the ORB managing the objects.

- **Corba architecture** (p. 2)
- **Interoperable object references (IOR)** (p. 4)
- **Portable object adapter (POA)** (p. 5)
- **Corba services** (p. 8)

1.2 Corba architecture

A Corba object is a virtual entity capable of being located by an ORB and having client requests invoked on it. It is virtual in the sense that it does not really exist unless it is made concrete by an implementation written in a programming language.

A request in an invocation of an operation on a Corba object by a client. Requests flow from a client to the target object in the server, and the target object sends the results back if the request requires one.

An object reference is a handle used to identify, locate, and address a Corba object. To clients, object references are opaque entities.

A servant is a programming language entity that implements one or more Corba objects. Servants are said to incarnate Corba objects because they provide bodies, or implementations, for those objects. Servants exist within the context of a server application. In C++, servants are object instances of a particular class.

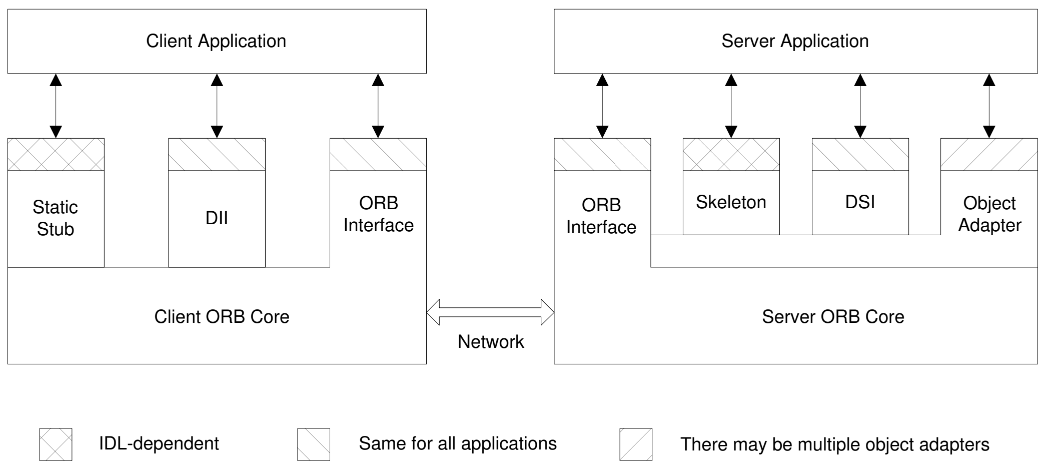

Figure 1.1: Common Object Request Broker Architecture (CORBA)

General request flow:

1. The client can choose to make requests either using static stubs (generated from the IDL definition, see below) or using the Dynamic Invocation Interface (DII). Either way, the client directs the request into the ORB core linked into its process.
2. The client ORB core transmits the request to the ORB core linked with the server application.
3. The server ORB core dispatches the request to the object adapter that created the target object.
4. The object adapter further dispatches the request to the servant that is implementing the target object. Like the client, the server can choose between static and dynamic dispatching mechanisms for its servants. It can rely on static skeletons generated from the object's IDL definition, or its servants can use the Dynamic Skeleton Interface (DSI).
5. After the servant carries out the request, it returns its response to the client application.

To invoke operations on a Corba object, a client must know the interface offered by the object. An object's interface is composed of the operations it supports and the types of data that can be passed to and from those operations. The interface is defined in IDL - the Interface Definition Language. IDL is not a programming language, so objects and applications cannot be implemented in IDL. The sole purpose of IDL is to allow object interfaces to be defined in a manner that is independent of any particular programming language.

Language mappings specify how IDL is translated into different programming languages. IDL is translated into stubs and skeletons that are compiled into applications. Compiling stubs and skeletons into an application gives it static knowledge of the programming language type and functions mapped from the IDL descriptions of remote objects. A stub is a client-side function that allows a request invocation to be made via a normal local function call. In C++, a Corba stub is a member function of a class. The local C++ object that supports stubs functions is often called a proxy because it represents the remote target object to the local application. Similarly, a skeleton is a server-side function that allows a request invocation received by a server to be dispatched to the appropriate servant.

In Corba, object adapters serve as the glue between servants and the ORB. As described by the Adapter design pattern, an object adapter is an object that adapts the interface of one object to a different interface expected by a caller. In other words, an object adapter is an interposed object that uses delegation to allow a caller to invoke requests on an object without knowing the object's true interface.

Corba object adapters fulfill three key requirements

1. They create object references, which allow clients to address objects.
2. They ensure that each target object is incarnated by a servant.
3. They take requests dispatched by a server-side ORB and further direct them to the servants incarnating each of the target objects.

The General Inter-ORB Protocol (GIOP) is an abstract protocol that specifies transfer syntax and a standard set of message formats to allow independently developed ORBs to communicate over any connection-oriented transport. The Internet Inter-ORB Protocol (IIOP) specifies how GIOP is implemented over Transmission Control Protocol/Internet Protocol (TCP/IP). In the remainder, we will focus on IIOP.

1.3 Interoperable object references (IOR)

ORB interoperability also requires standardized object reference formats. Object references are opaque to applications, but they contain information that ORBs need in order to establish communications between clients and target objects. The standard object reference format, called the Interoperable Object Reference (IOR), is flexible enough to store information for almost any inter-ORB protocol imaginable. An IOR identifies one or more suported protocols and, for each protocol, contains information specific to that protocol.

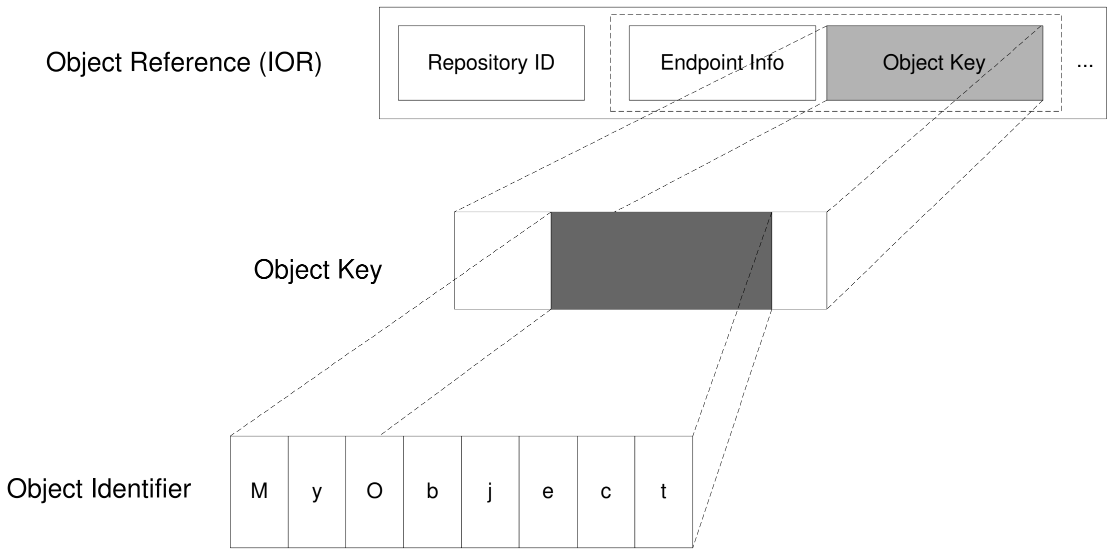

Figure 1.2: Object reference, key, and identifier

An IOR contains three major pieces of information

1. The repository ID is a string that identifies the most derived IDL type of the IOR at the time the IOR was created, for example "IDL:HxCorba/ImageRep:1.0". The repository ID allows you to locate a detailed description of the interface of the object in the Interface Repository.
2. The endpoint info field contains all information required by the ORB to establish a physical connection to the server implementing the object. The endpoint information specifies which protocol to use and contains physical addressing information appropriate for a particular transport. For example, for the IIOP it contains a host name and a TCP/IP port number. The addressing information may

directly contain the address and port number of the server (direct binding) but it is more likely to contain the address of an implementation repository that can be consulted to locate the correct server (indirect binding). This extra level of indirection permits server processes to migrate from machine to machine without breaking existing references held by clients. Corba also allows information for several different protocols and transports to be embedded in the reference.

3. The object key identifies the target object at the given host name and port combination. Exactly how this information is organized and used depends on the ORB. However, all ORBs allow the server, more specifically the POA, to embed an application-specific object identifier inside the object key when the server creates the reference. Object identifiers are specified using the `ObjectId` type, which is defined in the `PortableServer` module as a sequence of `octet`. This allows for virtually any type of data to be used to identify an object, for example, a string or a database key or a simple number.

1.4 Portable object adapter (POA)

The standard object adapter defined in the Corba specification is the Portable Object Adapter (POA). It provides features necessary to allow programming language servants to be portable among ORBs supplied by different vendors. A server application may contain multiple POA instances to support Corba objects with different characteristics or to support multiple servant implementation styles.

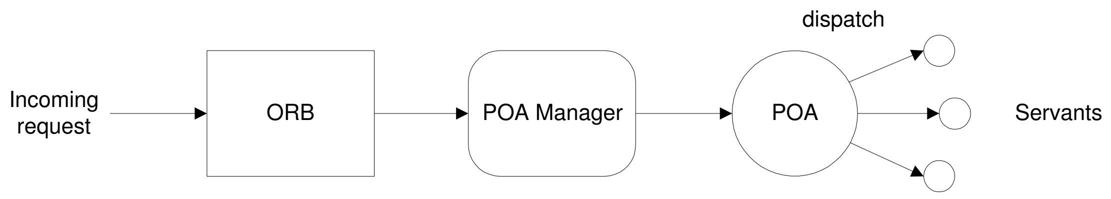

Figure 1.3: Incoming request

Conceptually, requests for Corba objects residing in the server application are sent from a client and arrive at the server ORB, which dispatches them to the POA in which the target object was created (dispatch is based on the object key). The POA then further dispatches the request to the servant incarnating the target object (using the object identifier).

A server cannot accept any incoming requests until it tells its ORB to start listening for them. In addition, because a single server application may contain multiple POAs, the flow of requests into each POA is controlled by a POAManager object associated with that POA. In addition to letting requests flow into a POA, a POAManager can queue requests for later dispatch or can discard them.

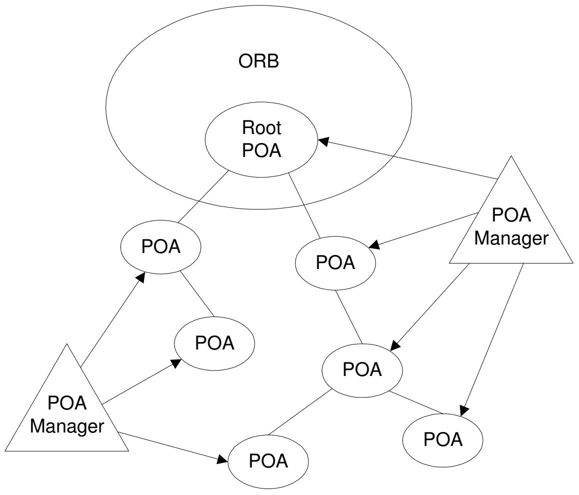

Figure 1.4: POA manager

In a server application, a POA is responsible for creating object references, activating objects, and dispatching requests made on objects to their respective servants. As such, the POA deals with three key entities : object references, object identifiers (a sequence of octets), and servants. How a POA handles these entities is determined via policies that are specified when a POA is created. All server applications have at least one POA, the Root POA, which has a standard set of policies.

LifespanPolicy (TRANSIENT, PERSISTENT) Controls the object life span. The default is TRANSIENT. The Root POA uses the default.

IdAssignmentPolicy (USER_ID, SYSTEM_ID) Controls object identification, i.e. generation of object identifiers. Typically, an application that uses persistent objects supplies its own identifiers (USER_-ID) because it uses them to keep track of where it stores the persistent state of the object. Applications that use transient objects usually let the POA create identifiers for them (SYSTEM_ID). The default is SYSTEM_ID. The Root POA uses the default.

IdUniquenessPolicy (UNIQUE_ID, MULTIPLE_ID) Allows a single servant to incarnate multiple Corba objects, or restricts servants to incarnating only a single object. Basically, the policy controls the entries in the Active Object Map of the POA. Each entry consists of an association between an ObjectId and a pointer to a servant. With UNIQUE_ID, each ObjectId must map to a different servant. With MULTIPLE_ID, multiple ObjectId's can map to a single servant. The default is UNIQUE_ID. The Root POA uses the default.

ImplicitActivationPolicy (IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION) Specifies that the POA should allow Corba objects to be created and activated implicitly or that only explicit Corba object creation and servant registration is allowed. Implicit activation is usually performed through a shortcut function supplied by a language mapping, such as the _this function provided by C++ skeleton classes. The default is NO_IMPLICIT_ACTIVATION. The Root POA uses IMPLICIT_-ACTIVATION.

RequestProcessingPolicyValue (USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER) Controls associations between objects and servants. An application containing many thousands of objects may want to incarnate only those objects that actually receive requests. It does this by registering a servant manager with the POA. Servant managers are local objects that are up-called by a POA if it receives an invocation on an object that has no associated servant. Depending upon the POA's value for the IdUniquenessPolicy, the servant manager can either provide the POA with a newly created servant or reuse an existing one. Either way, it returns the servant as the result of the up-call, which the POA uses to complete the request invocation. After the invocation completes, the POA either retains the association of the servant and the Corba object in its Active Object Map or throws the association away, meaning that the next invocation on the object will again require the services of the servant manager. Still anoher alternative is for applications to supply a default servant to a POA. A default servant incarnates all Corba objects for a POA. The default is USE_ACTIVE_OBJECT_MAP_ONLY. The Root POA uses the default.

ServantRetentionPolicy (RETAIN, NON_RETAIN) Specifies whether associations between objects and servants should be retained in the Active Object Map. The default is RETAIN. The Root POA uses the default.

ThreadPolicy (ORB_CTRL_MODEL, SINGLE_THREAD_MODEL) Specifies the threading model used for handling requests. The ORB-controlled model allows the underlying ORB implementation to choose an appropriate multithreading model, whereas the single-thread model guarantees that all requests for all objects in that POA will be dispatched sequentially. The default is ORB_CTRL_-MODEL. The Root POA uses the default.

A POA is created by invoking the create_POA operation on another POA. Because all server applications have a Root POA, its create_POA operation serves as the starting point for creating other POAs. The Root POA is obtained via ServantBase::_default_POA. A POA created using another POA becomes a child POA of the creating POA. Note, however, that this has no effect on the policies of the child POA. Policies are not inherited. During shutdown, child POAs are destroyed before their parents, meaning that the Root POA is the last POA to be destroyed.

The POA provides several options for creating objects and activating them by servant registration:

Object creation without creating any servants The POA interface provides two functions for this : create_reference and create_reference_with_id. Both create a Corba object (without creating a servant) using a RepositoryId that identifies the most derived IDL interface that the new object will support.

Servant registration The POA interface provides two functions for this : activate_object and activate_-object_with_id. With these operations, the user explicitly supplies a servant to incarnate the object being activated.

Servant managers There are two types of servant managers, i.e. interfaces to be implemented by the user supplied servant manager object (both interfaces are derived from the ServantManager interface):

ServantActivator (ServantRetentionPolicy == RETAIN) The interface supplies the incarnate and etherealize operations. When the POA receives a request for a target object that is not in its Active Object Map, it calls the incarnate function of the servant manager to supply the servant. The POA normally invokes the etherealize operation in response to an explicit object deactivation via deactivate_object (even if the servant for that object was not created by the servant activator) or in response to the deactication or destruction of the POA itself.

ServantLocator (ServantRetentionPolicy == NON_RETAIN) The interface supplies the preinvoke and postinvoke operations. The POA does not store object-to-servant associations in its Active Object Map, so it must invoke its ServantLocator for each incoming request. If first invokes preinvoke to obtain a servant to dispatch the request to. After the request returns, the POA invokes postinvoke.

Because servant managers are themselves Corba objects, they must be created and activated before they can be registered with a POA. The easiest way to create an object reference for a servant manager is to implicitly register its servant in the Root POA using _this(). The registration as a servant manager is done via the set_servant_manager function of the POA.

Default servants The POA interface provides the set_servant function for this. The default servant acts as a catch-all servant for those objects that do not have their own servants. Each object incarnated by the default servant must support the same interface. Because they incarnate multiple Corba objects, a key aspect is that a default servant must not hold object-specific state, so that must be contained within the ObjectId. Wihtin the context of a request dispatch, the server ORB allows to obtain the ObjectId of the target object and a reference to the POA that is dispatching the request via the PortableServer::Current interface. You obtain a reference to the POA Current by passing the string "POACurrrent" to ORB::resolve_initial_references.

Memory related problems with servants are best avoided by deriving the servant class from the RefCountServantBase class provided in the PortableServer namespace. The interface provides the _add_ref and _remove_ref functions. Unlike the do-nothing versions of the _add_ref and _remove_ref functions supplied by PortableServer::ServantBase, the versions provided by the RefCountServantBase class perform thread-safe reference counting for derived servant classes.

Like servants, POAs can be created on demand. POA activation occurs when a request arrives for an object in a descendant POA that has not yet been created or when the application searches a hierarchy of POAs using the POA::find_POA operation for a named POA that has not yet been created. The application registers an AdapterActivator with each POA that must activate its descendant POAs. Adapter activators are normal Corba objects, so they are incarnated via servants.

1.5 Corba services

Implementation repository

The process of opening a connection (between a client and a server) and associating an object reference with its servant is known as binding. ORBs typically support two binding modes : direct binding and indirect binding. Direct binding is supported by all ORBs. Indirect binding relies on an external location broker known as an implementation repository and is an optional component of Corba.

Whenever a server application creates an object reference, the server-side run time embeds information to support binding inside the object reference. Specifically, an IOR contains an IP address (or host name), TCP port number, and an object key. If a server inserts its own address and port number into a reference, the reference uses direct binding.

With indirect binding, the address and port number of the implementation repository are inserted into the object reference. The implemetation repository maintains a registry of known servers and records which server is currently running on which host and at which port number. The implementation repository basically forwards the request to a server or starts a new server to handle the request.

Naming service

The Naming Service provides a mapping from names to object references: given a name, the service returns an object reference stored under that name. A name-to-reference assiciation is called a name binding. The same object reference can be stored several times under different names, but each name identifies exactly one reference.

A naming context is an object that stores name bindings. In other words, each context object implements a table that maps names to object references. A name in the table can denote either an object reference to an application object or another context object in the Naming Service. This means that, like a file system, contexts can be connected to form hierarchies : contexts correspond to directories that store names to either directories (other contexts) or files (application objects).

A hierarchy of contexts and bindings is known as a naming graph. It is possible for the graph to have contexts that have no names. Such contexts are known as orphaned contexts. A naming graph has one or more distinguished contexts known as initial naming contexts. Initial naming contexts determine the points at which clients gain access to a naming graph.

ORB::resolve_initial_references allows you to partably obtain references that are crucial for bootstrapping your client or server. The parameter to the call determines which particular reference is returned. The OMG standardizes the set of well-known object identifiers. Currently, they are RootPOA, POACurrent, Interface-Repository, NameService, TradingService, SecurityCurrent, and TransactionCurrent. list_initial_references returns the list of object identifiers configured for your ORB.

If you call resolve_initial_references with an object identifier of NameService, the operation returns a reference to an object of type NamingContext. The returned context is the configured initial context of the Naming Service for the local ORB.

Chapter 2

Client and server

2.1 Client and server

- **Corba configuration** (p. 10)
- **Horus server** (p. 11)
- **C++ client** (p. 13)
- **RGB image display data transfer** (p. 14)
- **Data object management** (p. 18)

2.2 Corba configuration

Configuration of connections between Corba components (servers, clients, and Corba services) relies on a number of configuration files and environment variables. Assuming you installed Horus correctly, these are already setup.

The main configuration file is ${HXSRC}/HxCorba/cfg/orbacus.cfg. It looks something like this:

```
#   Copyright (c) 2000, University of Amsterdam, The Netherlands.
#   All rights reserved.
#
#   Author(s):
#   Marc Navarro            (mnavarro@wins.uva.nl)


org.omg.CORBA.ORBClass = com.ooc.CORBA.ORB
org.omg.CORBA.ORBSingletonClass = com.ooc.CORBA.ORBSingleton

# Naming Service
ooc.naming.port=8001
ooc.naming.database=NS.dat
ooc.naming.endpoint=iiop --port 8001

# Interface Repository (2.3)
ooc.ifr.port=8002
# From 4.1.0 on :
ooc.ifr.endpoint=iiop --port 8002

# Initial references
ooc.orb.service.NameService=corbaloc::localhost:8001/NameService
ooc.orb.service.InterfaceRepository=corbaloc::localhost:8002/DefaultRepository
ooc.orb.service.Constructor=corbaloc::localhost:8010/Constructor
ooc.orb.service.GlobalOps=corbaloc::localhost:8010/GlobalOps
ooc.orb.service.Registry=corbaloc::localhost:8010/Registry
ooc.orb.service.Test=corbaloc::localhost:8010/Test
ooc.orb.service.Configure=corbaloc::localhost:8010/Configure

ooc.orb.service.WebImageFactory=corbaloc::localhost:8011/WebImageFactory
ooc.orb.service.TVCapture=corbaloc::localhost:8020/TVCapture
ooc.orb.service.VideoPlayerFactory=corbaloc::localhost:8020/VideoPlayerFactory

ooc.orb.service.Database=corbaloc::carol.wins.uva.nl:8888/Oracle
```

The file specifies the default Horus setup to all Corba components. The Orbacus C++ ORB's access the file via the ORBACUS_CONFIG environment variable. Since Java applications do not have access to environment variables we use the ${HXSRC}/HxBin/hxproperties scripts to turn the content of orbacus.cfg into Java properties (Java applets do not have access to the script so they so something like a services.txt).

Of course, you can override the default values of orbacus.cfg via command line arguments, see the examples in this document and the Orbacus manual.

Interface Repository

The Interface Repository (started with `irserv`) runs at port 8002 as specified in orbacus.cfg. Typically, the Interface Repository is started via the ${HXSRC}/HxBin/ss(.bat) script with `ss ir`. On Windows, the Horus toolbar contains an icon to call the ss script.

Naming Service

The Naming Service (started with `nameserv`) runs at port 8001 as specified in orbacus.cfg. The ss script provides two ways to start the Naming Service : with (`ss nss`) and without (`ss ns`) creating a new database for its bindings.

Orbacus provides a command line tool (`nsadmin`) to list the content of the Naming Service and to add new bindings. For example, the following will list the Horus servers (see **Horus server** (p. 11)) registered in the Naming Service running on the local machine at port 8001 under **HxCorba/**Servers:

```
nsadmin -l HxCorba/Servers
```

To list the Horus servers registered in the Naming Service running on machine Mach2 at port 8001:

```
nsadmin -l HxCorba/Servers -ORBservice NameService corbaloc::Mach2:8001/NameService
```

To register the Horus server running on machine Mach2 at port 8010 in the Naming Service running on the local machine under the name Mach2Server:

```
nsadmin -b HxCorba/Servers/Mach2Server corbaloc::Mach2:8010/Constructor
```

Note that the context HxCorba/Servers should already exist. You can create it using:

```
nsadmin -c HxCorba
nsadmin -c HxCorba/Servers
```

2.3 Horus server

In this context, a Horus server basically refers to a Corba server that provides a **HxCorba::Constructor** object. The server is addressed via the C language API defined in **HxServerCAPI.h**.

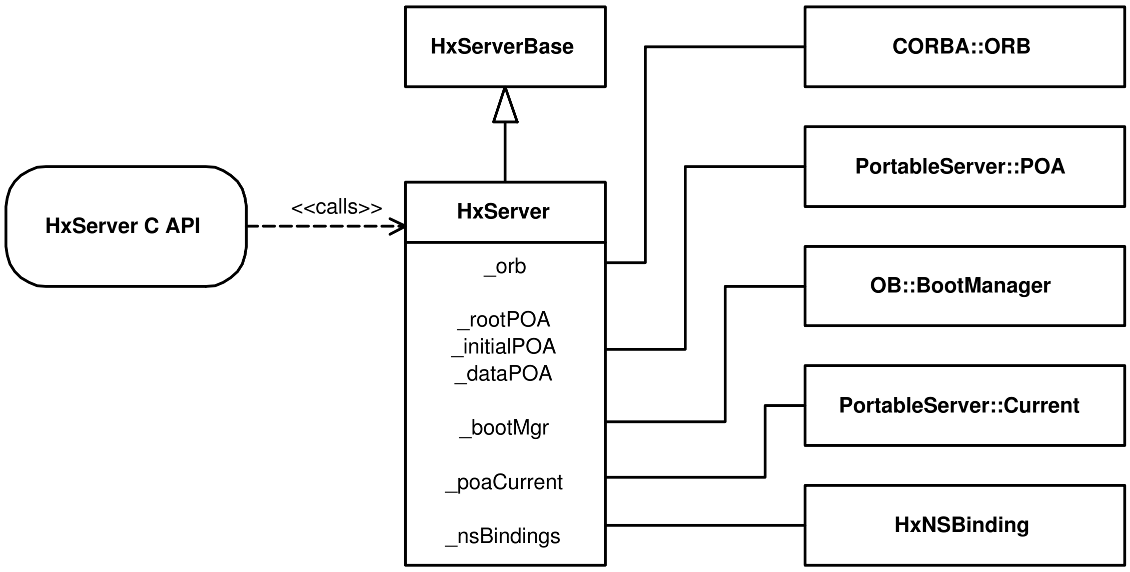

Figure 2.1: Corba Horus server

The heart of the server is **HxServer** (p. **??**). Initialization is done in **HxServer::init** (p. **??**). The most important steps are:

- Initialize **HxDataObjectManager** (p. **??**) by calling **HxDataObjectManager::init** (p. **??**)
- Initialize the ORB
- Create a POA for the initial objects and create the objects themselves by activating the corresponding servants (in **HxServer::addInitialObject** (p. **??**)). By giving the initial objects also to the Orbacus specific BootManager they are more easily accessible to clients (via corbaloc). The current list of initial objects is:
  - ”Constructor” - **HxConstructorServant** (p. **??**)
  - ”GlobalOps” - **HxGlobalOpsServant** (p. **??**)
  - ”Registry” - **HxRegistryServant** (p. **??**)
  - ”Test” - **HxTestServant** (p. **??**)
  - ”Configure” - **HxConfigureServant** (p. **??**)
- Set up **HxEnvCorba** (p. **??**)

The server also maintains a list of **HxNSBinding** (p. **??**)’s that record bindings of Horus servers in the Corba NamingService.

Stand-alone server application

A stand-alone Horus server is started by the HxServerApp program. The main function is defined in **HxCorba/ServerApp/main.c**.

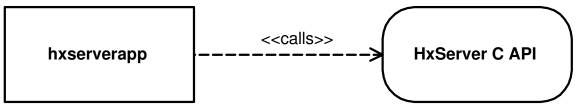

Figure 2.2: Stand-alone Horus server

Command line arguments:

–help show command line options

–interactive run the server (ORB) as a background process and provide a command line to execute some commands (see **showHelp** (p. **??**)) to control the server application.

-OAport port start the server using the given port (default is a random port)

-bind name address:port bind the server (ORB) under the given name in the NameService at the given address:port combination

For example, to start a Horus server on your local machine at port 8010 use:

```
hxserverapp -OAport 8010 --interactive
```

To start a Horus server on your local machine at port 8010 and register it in the Naming Service on machine Mach2 under the name TheServer:

```
hxserverapp -OAport 8010 --interactive -bind TheServer Mach2:8001
```

If you start the server with –interactive you can also do the binding at the command prompt of the already running server:

```
bind TheServer Mach2:8001
```

2.4 C++ client

An example is given in **HxSamples/ConsDemo/clientDemoImageOps.c**.

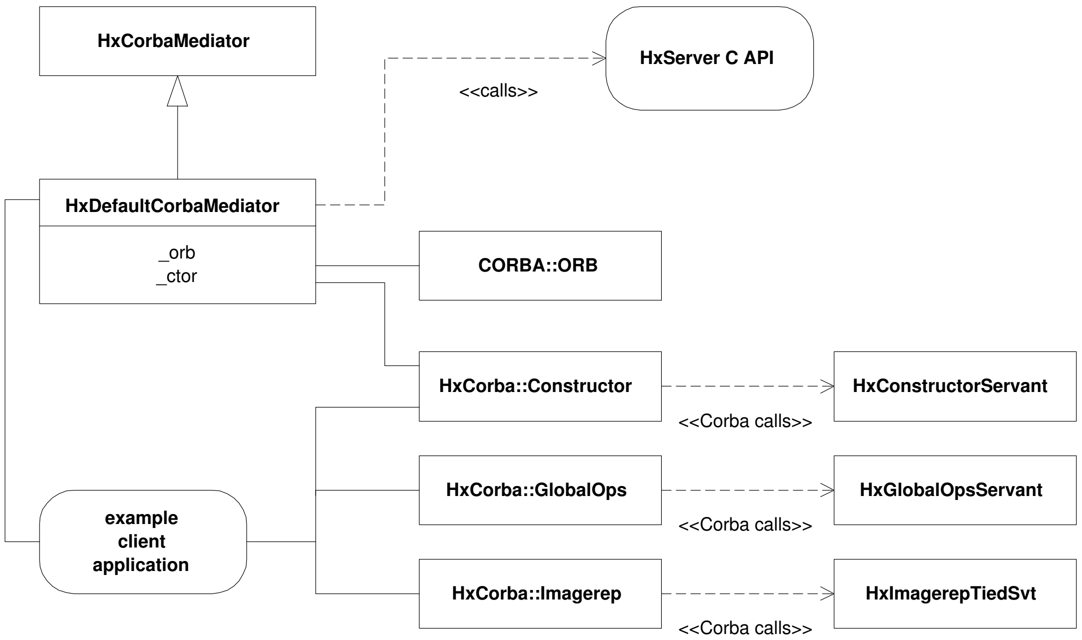

Figure 2.3: C++ client application structure

Basically, a client takes the following steps:

- Instantiate a **HxCorbaMediator** (p. **??**) (usually **HxDefaultCorbaMediator** (p. **??**)) to facilitate the connection to Corba.
- Obtain an initial object (a stub) from the server such as **HxCorba::Constructor** and **Hx-Corba::GlobalOps**.
- Apply functions to Corba objects obtained from calls to the initial objects and previous calls to other Corba objects.

In a normal client-server configuration, any call to a member function of a stub results (in the end) in a call to a servant member function. The <<Corba call>> in the figure is then done via IIOP. However, if the client use the "-createServer" option (see below), the server runs in the same process as the client and the <<Corba call>> is a normal virtual function call.

To indicate the Horus server we want to be a client of there are some command line arguments (handled by **HxDefaultCorbaMediator** (p. **??**) in C++):

-useServer name connect to the server registered in the NameService under "HxCorba/Servers"

-useServerRef ior connect to the server designated by the given ior, e.g. a corbaloc

-createServer start a new server within the client process

For example, to have the client call the Horus server that is running on the machine Mach2 at port 8010 use:

```
clientDemoImageOps -useServerRef corbaloc::Mach2:8010/Constructor
```

To use the Horus server that is registered in the Naming Service under the name TheServer:

```
clientDemoImageOps -useServer TheServer
```

The Corba version of the shot detection program (`clientShotDetection`) is an example of a client that communicates with two Corba servers : a Horus server to do the processing and an OracleServer to store the results in the Oracle database. To use the Horus server running on the localhost at port 8888 as well as the OracleServer running on carol at port 9999 use:

```
clientShotDetection -useServerRef corbaloc::localhost:8888/Constructor \
                    -ORBservice Database corbaloc::carol:9999/Oracle
```

2.5 RGB image display data transfer

Transferring image data from server to client for the purpose of image display is a frequent operation that requires quite a lot of resources. For reasons of efficiency and easy of use, we introduce several ways to transfer image data in RGB format for a Corba object from server to client. The efficiency and feasibility of an approach depend upon the client-server connection.

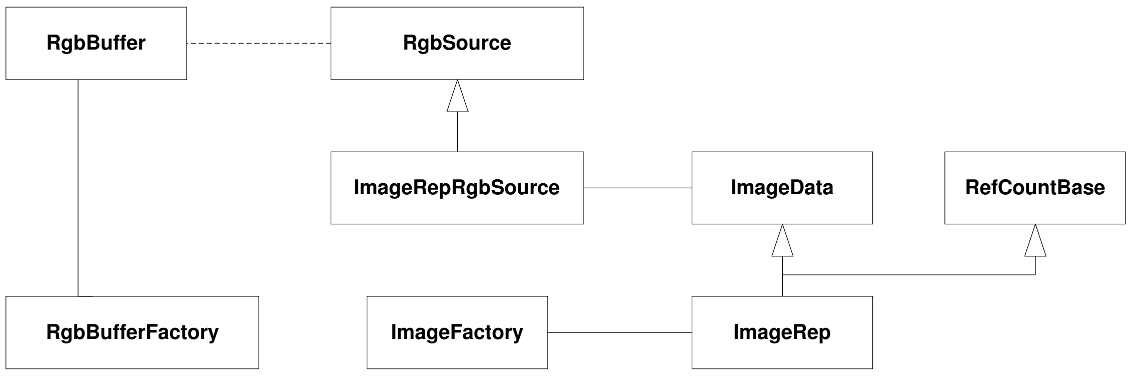

Figure 2.4: IDL definitions related to RGB data transfer

1. Using the Corba (image data) object itself
    (a) Call the getRgb2d member function of the Corba object to obtain an RgbSeq. An RgbSeq is an array of integers with pixel values in RGB format. Currently, the ImageData and ImageSeq IDL definitions support this (see **ad 1.1 - getRgb2d of image object** (p. 16)).
    (b) Call the fillRgb2d member function of the Corba object to have it fill an RgbBuffer. An RgbBuffer is an object that is capable of storing image data in RGB format. In case the RgbBuffer servant was constructed around your own destination buffer the transfer is now completed. Otherwise, you can ask the RgbBuffer to produce an RgbSeq via getRgb. Currently, the ImageData and ImageSeq IDL definitions support filling of an RgbBuffer (see **ad 1.2 - fillRgb2d of image object** (p. 16)).
2. Using an RgbSource obtained from the Corba (image data) object. An RgbSource is used to abstract from the Corba (image data) object, the parameters used in preparing the RGB data, and the actual transfer mechanism used.
    (a) Call the getRgb member function of the RgbSource object to obtain an RgbSeq. Currently, the ImageRepRgbSource IDL definition support this (see **ad 2.1 - getRgb of RgbSource** (p. 17)). **NOTE** that ImageSeq uses an ImageSeqDisplayer that has a function similar to RgbSource but it still has to be rewritten.
    (b) Call the fillRgb member function of the RgbSource object to have it fill an RgbBuffer. Currently, the ImageRepRgbSource IDL definition support this (see **ad 2.2 - fillRgb of RgbSource** (p. 17)). **NOTE** that ImageSeq uses an ImageSeqDisplayer that has a function similar to RgbSource but it still has to be rewritten.

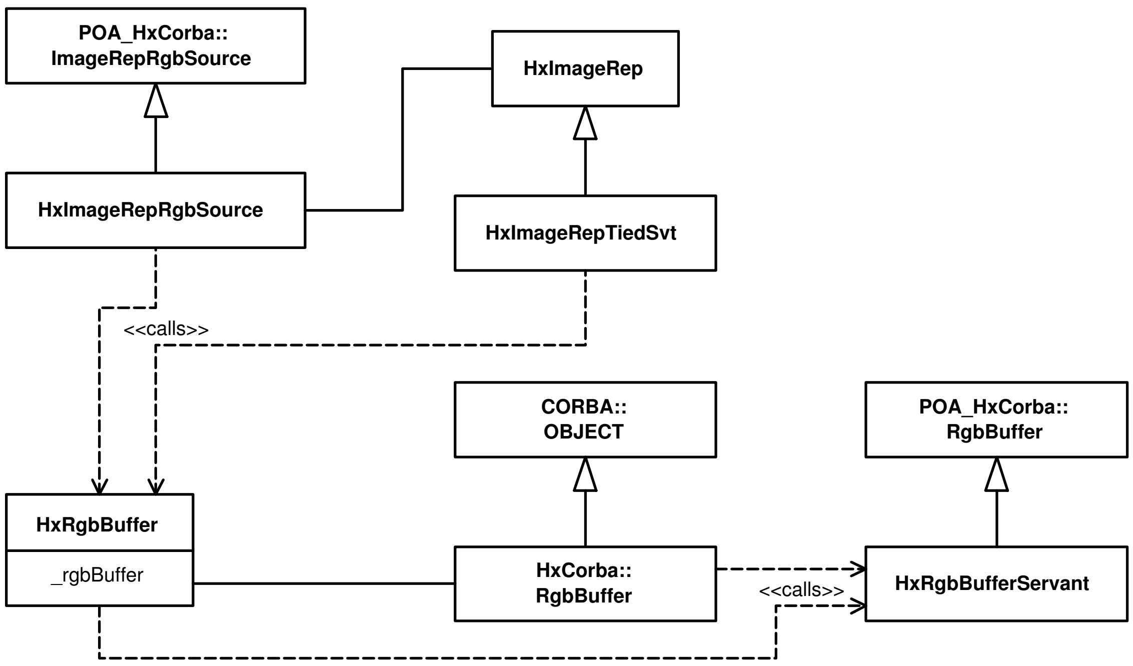

Figure 2.5: Server side classes related to RGB data transfer

Classes in the diagram: **HxImageRepTiedSvt** (p. **??**), **HxImageRepRgbSource** (p. **??**), **HxRgbBuffer** (p. **??**), **HxRgbBufferServant** (p. **??**). A similar structure holds for **HxImageSeqTiedSvt** (p. **??**) and **HxImageSeqDisplayer** (p. **??**).

2.5.1 ad 1.1 - getRgb2d of image object

IDL interface excerpt:

```
interface ImageData
{
    RgbSeq  getRgb2d(in string displayMode);
}

interface ImageSeq
{
    RgbSeq  getRgb2d(in long frameNr, in string displayMode);
}
```

The actual functions called are **HxImageRepTiedSvt::getRgb2d** (p. **??**) and **HxImageSeqTiedSvt::getRgb2d** (p. **??**). They create an **HxRgbBuffer** (p. **??**), ask the corresponding image object to fill the buffer, and ask the **HxRgbBuffer** (p. **??**) to produce an RgbSeq.

2.5.2 ad 1.2 - fillRgb2d of image object

IDL interface excerpt:

```
interface ImageData
```

```
{
    void    fillRgb2d(in RgbBuffer buf, in string displayMode);
}

interface ImageSeq
{
    void    fillRgb2d(in long frameNr, in RgbBuffer buf, in string displayMode);
}
```

The actual functions called are **HxImageRepTiedSvt::fillRgb2d** (p. **??**) and **HxImageSeqTiedSvt::fill-Rgb2d** (p. **??**). They create an **HxRgbBuffer** (p. **??**) and ask the corresponding image object to fill the buffer.

IDL interface excerpt:

```
interface RgbBuffer
{
    RgbSeq  getRgb();
}
```

The actual function called is **HxRgbBufferServant::getRgb** (p. **??**). It creates an RgbSeq from data in its internal buffer.

2.5.3 ad 2.1 - getRgb of RgbSource

IDL interface excerpt:

```
interface RgbSource
{
    RgbSeq  getRgb();
}
```

The actual functions called are **HxImageRepRgbSource::getRgb** (p. **??**) and **HxImageSeq-Displayer::getRgb2d** (p. **??**). They ask the image data object to fill their internal buffer and create an RgbSeq from that buffer. The RgbSeq can be sent in batches or at once.

2.5.4 ad 2.2 - fillRgb of RgbSource

IDL interface excerpt:

```
interface RgbSource
{
    void    fillRgb(in RgbBuffer buffer);
}
```

The actual functions called are **HxImageRepRgbSource::fillRgb** (p. **??**) and **HxImageSeqDisplayer::fill-Rgb2d** (p. **??**). They create an **HxRgbBuffer** (p. **??**) and ask the corresponding image object to fill the buffer. The buffer can be filled in batches or at once.

2.6 Data object management

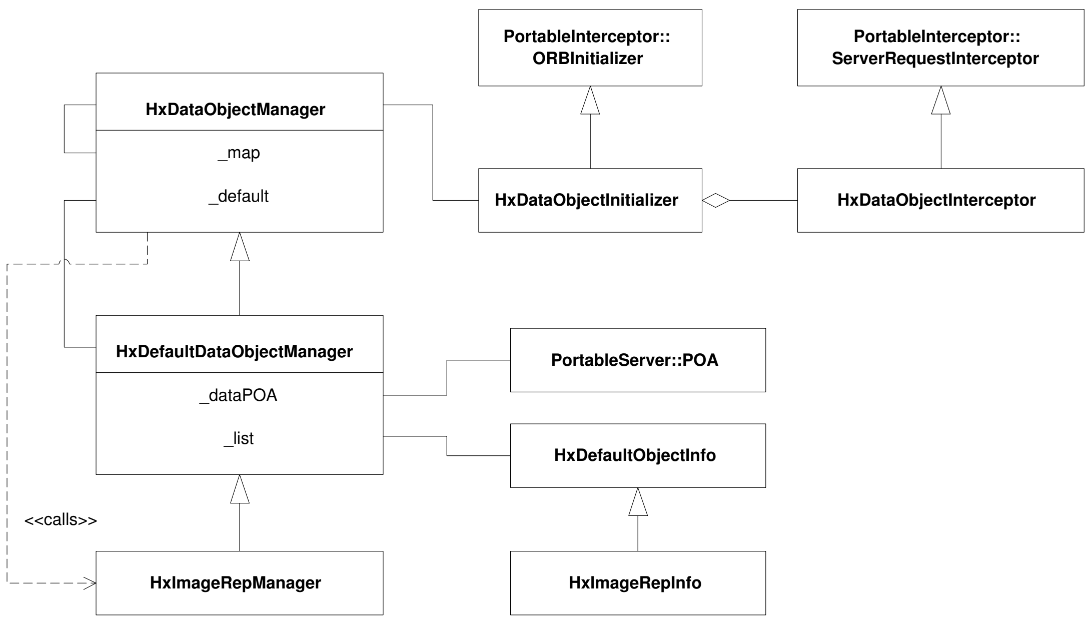

Figure 2.6: Corba data object management

Classes : **HxDataObjectManager** (p. **??**), **HxDefaultDataObjectManager** (p. **??**), **HxImageRepManager** (p. **??**), **HxDataObjectInterceptor** (p. **??**), **HxDefaultDataObjectManager::HxDefaultObjectInfo** (p. **??**), **HxImageRepManager::HxImageRepInfo** (p. **??**)

Chapter 3

Stubs and servants in C++

3.1 Stubs and servants in C++

As an example, we show the stubs and skeletons for the ImageRep IDL definition. Given the IDL interface definitions of RefCountBase, ImageData and ImageRep (where ImageRep inherits from both RefCountBase and ImageData) we use the idl compiler to generate the following C++ binding (note that the same structure holds for any set of IDL definitions):

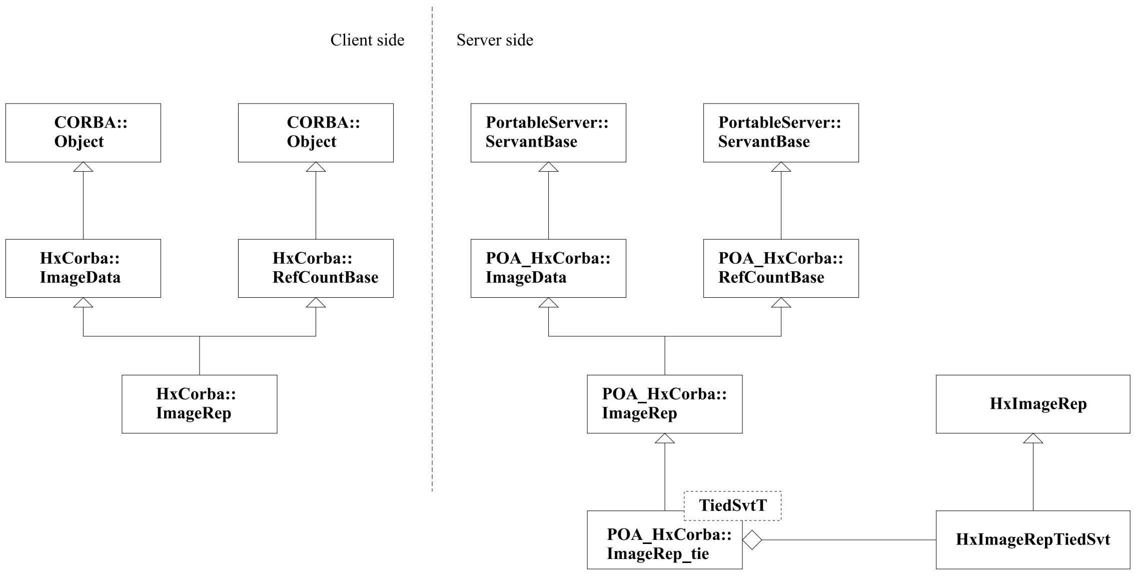

Figure 3.1: IDL binding in C++

The classes CORBA::Object and PortableServer::ServantBase are part of the Corba definition and are implemented by the ORB. The other classes are generated by the IDL compiler. The generated files are located in $HXSRC/HxCorba/idl/stubs.

- Stubs **HxCorba::ImageData** and **HxCorba::ImageRep** are defined in **HxCorbaImageRep.h**
- Skeletons **POA_HxCorba::ImageData** and **POA_HxCorba::ImageRep** are defined in **HxCorbaImageRep_skel.h**

- Tie **POA_HxCorba::ImageRep_tie** is defined in **HxCorbaImageRep_skel_tie.h**

Basically, we employ two ways to implement servants : using ties and using inheritance. The tie approach is used when there is an existing class that matches an IDL definition, e.g. the C++ class HxImage-Rep matches the IDL HxCorba::ImageRep definition. The inheritance approach is used when there is no existing class to implement the IDL definition nor do we want to be able to address the class to be implemented later on without using Corba.

Servants using ties

In the tie approach, the servant object is a combination of a tie (generated by the IDL compiler) and an object that actually implements the functionality. All that the tie does is forward all member function calls to the tied object. In case the argument types and result type of the member functions of the skeleton match the member functions of the ”real” object there is nothing more to do.

However, in the case of e.g. ImageRep, the types do not always match exactly. Consider the following functions from the IDL interface:

```
long     dimensionSize(in long d);
ImageRep unaryPixOp(in string upoName, in TagList tags);
```

The argument and return type of the tie function **POA_HxCorba::ImageRep_tie::dimensionSize** match the **HxImageRep::dimensionSize** function of the real object in that the compiler can find a suited conversion. However, both the argument and the return type of **POA_HxCorba::ImageRep_tie::unaryPixOp** do not match **HxImageRep::unaryPixOp**. So, we have implemented a specialization of **HxImageRep** called **HxImageRepTiedSvt** (p. **??**) with a member function **HxImageRepTiedSvt::unaryPixOp** (p. **??**) that converts the argument, calls **HxImageRep::unaryPixOp**, and converts the return value.

Other classes in $HXSRC/HxCorba/Main/ that employ the tie mechanism are:

HxBlob2dTiedSvt (p. **??**), **HxBSplineCurveTiedSvt** (p. **??**), **HxHistogramTiedSvt** (p. **??**), **HxImage-RepTiedSvt** (p. **??**), **HxImageSeqTiedSvt** (p. **??**), **HxMatrixTiedSvt** (p. **??**), **HxNJetTiedSvt** (p. **??**), **HxPolyline2dTiedSvt** (p. **??**), **HxSampledBSplineCurveTiedSvt** (p. **??**), **HxSFTiedSvt** (p. **??**), **HxTag-ListTiedSvt** (p. **??**), **VxSegmentationTiedSvt** (p. **??**), **VxSegmentTiedSvt** (p. **??**), and **VxStructureTied-Svt** (p. **??**).

A list of these is maintained in **HxTiedServants.h** to ... (check this)

Servants using inheritance

In the inheritance apporach, we make the object that implements the functionality a specialization of the skeleton. That is, in the ImageRep example it would be derived from POA_HxCorba::ImageRep.

Classes in $HXSRC/HxCorba/Main/ that employ the inheritance mechanism are:

HxConfigureServant (p. **??**), **HxConstructorServant** (p. **??**), **HxRegistryServant** (p. **??**), **HxRgb-BufferServant** (p. **??**), and **HxTestServant** (p. **??**).

Taglists

Taglists require special care, because (up till now) **HxCorba::TagList** is the only Corba object that the user can modify. And, only TagList arguments can be nil.

One difference is the constructor. **HxTagList** has an empty constructor. The way a TagList is constructed (Corba TagList, not HxTagList) is that the user creates an empty TagList and calls its operations to add tags. The other Corba objects are created from a C++ object obtained from a C++ operation.

Now we can not instantiate a TagList from an HxTagList. To do that we would have to define a constructor for **HxTagListTiedSvt** (p. **??**) with an HxTagList as argument, with the problem that we would instantiate a servant for a copy of that HxTagList, not that HxTagList itself.

Another difference is the way we obtain the tied object from the servant. This is done with the template function **HxGetTiedObject** (p. **??**). This function has the tied servant as template and requires that the template has the next typedefs:

- HxT: the Horus type (for example HxImageRep)
- CorbaT: the Corba type (for example HxCorba::ImageRep)
- TieT: the TIE class type (for example POA_HxCorba::ImageRep_tie<HxImageRepTiedSvt>)

(we use ImageRep as example, but actually ImageRepTiedSvt doesn't have these typedef's because the conversion functions HxGetTiedObject and HxRegisterTiedServant can not be applied to ImageRep because the ImageRep servants are registered in a different POA :-)

The call to HxGetTiedObject is a little different for TagList because an argument of type TagList can be nil and HxGetTiedObject must return (in case of nil) an empty HxTagList, and this empty HxTagList has to be provided as an argument to HxGetTiedObject.

Why doesn't HxGetTiedObject create itself the empty HxTagList? Because the result to HxGetTiedObject is assigned to a reference of HxTagList, not copied to another HxTagList. And the empty HxTagList will be disposed when it goes out of scope: it doesn't survive the call of HxGetTiedObject.

Why doesn't HxGetTiedObject create the empty HxTagList with "new"? Because the caller of HxGetTiedObject can not do a "delete": this delete will mess up things in the case that HxGetTiedObject returns a tied HxTagList, not an empty one.

And why the result of HxGetTiedObject is assigned to a reference of HxTagList, not just to a HxTagList (using the assignment operator)? The call to HxGetTiedObject for all the other types of tied servants are not assigned to a reference, but HxTagList is different: An HxTagList can be modified via a Corba call and we want to modify the real HxTagList, not a copy.

Chapter 4

Stubs and servants in Java

4.1 Stubs and servants in Java

As an example, we show the stubs and servants for the ImageRep IDL definition. Given the IDL interface definitions of RefCountBase, ImageData and ImageRep (where ImageRep inherits from both RefCount-Base and ImageData) we use the idl compiler to generate the following Java binding (RefCountBase related stuff has been omitted for clarity):

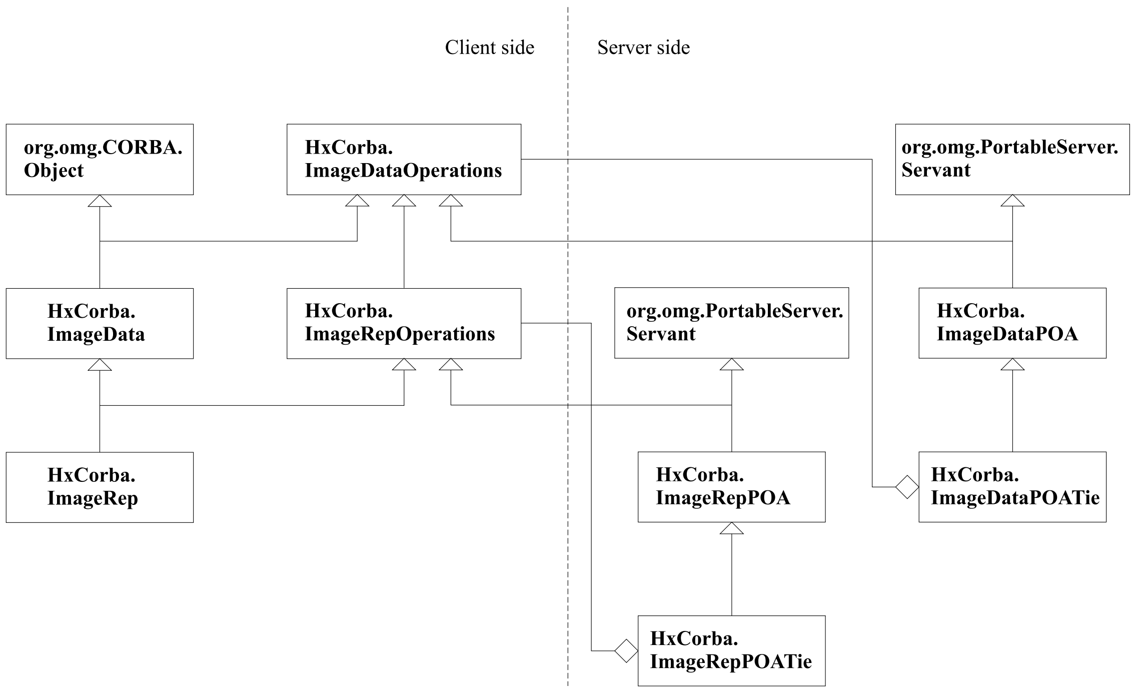

Figure 4.1: IDL Binding in Java

The classes org.omg.CORBA.Object and org.omg.PortableServer.Servant are part of the Corba definition and are implemented by the ORB. The other classes are generated by the IDL compiler. The generated files are located in $HXSRC/HxCorba/idl/HxCorba.

- The common interfaces HxCorba.ImageDataOperations and HxCorba.ImageRepOperations are defined in ImageDataOperations.java and ImageRepOperations.java
- Stubs HxCorba.ImageData and HxCorba.ImageRep are defined in ImageData.java and ImageRep.java
- Skeletons HxCorba.ImageDataPOA and HxCorba.ImageRepPOA are defined in ImageDataPOA.java and ImageRepPOA.java
- Ties HxCorba.ImageDataPOATie and HxCorba.ImageRepPOATie are defined in ImageDataPOATie.java and ImageRepPOATie.java

In the tie approach the tied object is derived from the ”Operations” class and a servant is created by putting the tied object in the ”POATie” class. For example, an ImageData servant is created by instantiating an HxCorba.ImageDataPOATie with a JavaImageData, a class derived from HxCorba.ImageDataOperations.

In the inheritance approach a servant is derived from the ”POA” class. For example, a RgbBufferServant is derived from HxCorba.RgbBufferPOA.

Note that implementation of servants for IDL interfaces that use multiple inheritance is not possible using the inheritance approach since Java does not support multiple inheritance. Such servants have to be implemented using the tie approach.

Chapter 5

Servant generator

5.1 Servant Generator

The ServantGenerator tool generates the source code of the servant for the **HxCorba::GlobalOps CORBA** Object. It should generate member functions that look like:

```
(1) #include "HxAbs.h"

(2) HxCorba::ImageRep_ptr
(3) HxGlobalOpsGenSvt::HxAbs(HxCorba::ImageRep_ptr im)
    {
(4)     HxImageRep _im_arg = HxGetTiedObject<HxImageRepTiedSvt>(_server, im);
(5)     HxImageRep _result = ::HxAbs(_im_arg);
(6)     return HxRegisterTiedServant<HxImageRepTiedSvt>(_server, _result);
    }
```

It also generates the declaration of this function in the class declaration, that is almost the same as lines 2 and 3.

The ServantGenerator (SG for short) obtains from the InterfaceRepository (IR for short) a list of the operations it has to generate. It also obtains the description of the operation (return type, params, etc...).

SG has to be able to generate code like the shown above from the information it gets from the IR. Let's describe a little the code shown and how SG has to generate it:

(1) `#include` **directive:** For each operation of the GlobalOps interface, SG generates a member for the servant. This member funcion calls a global HORUS function that performs the operation. The IDL operation and the HORUS global function should have the same name.

For each member function, the first thing the SG generates is an include directive of a file with the same name as the member function. This file should contain the declaration of a HORUS global function, with the same name, that implements the operation.

Then, the name of the operation obtained from the IR is enough to generate that line.

(2) Return type: Next, SG has to generate the result type of the operation. SG knows (from the IR) the result type of the operation, but the IDL one. IDL types and C++ types are not equal, but there is a mapping between them. For each type we want SG to handle, we have to tell to SG the C++ type that has to be used as result type.

(3) Type of arguments: Now its time to generate the rest of the function header. The class name and function name is easy to generate from the information obtained from IR. But the type of the function arguments is not easy to generate. As for the result type, we have to tell to SG the C++ type that has to be used as argument type. In the example (ImageRep) the type of the argument is the same type as the result type, but this is not true for all types.

(4) Conversion lines: The body of the function has three parts: first, all arguments are converted from **CORBA** values to their HORUS related values. Next the global HORUS function is called and finally the HORUS result is converted to a **CORBA** result and it's returned.

To generate the conversion lines, SG has to know, for each argument, the related HORUS type and the procedure to use for the conversion. These are other information we will have to tell to SG for each type.

Not all types require this conversion. Some types (basic types) can be used both for **CORBA** and HORUS.

(5) Calling HORUS function: To generate the line that calls the HORUS global function, SG has to know the type to use to retrieve the function result. This is related to the type that the IDL operation returns. We will have to tell SG which HORUS type to use to get the result.

(6) Return line: The return line is like the conversion line but the other way around: a HORUS value if converted to a **CORBA** value. We will also have to tell SG how to make that conversion.

As we can see, for each type (IDL type) we want SG to support we have to tell him:

- The **CORBA**/C++ type used as return type.
- The **CORBA**/C++ type used for arguments.
- Its related HORUS type and the procedure to perform the conversion.
- The HORUS type used to retrieve the result from the global HORUS function.
- The way to convert that result to a **CORBA** value.

Example: how to support `HxCorba.ImageRep` interface:

- The **CORBA**/C++ type used as return type is **HxCorba::ImageRep_ptr**.
- The **CORBA**/C++ type used for arguments is also **HxCorba::ImageRep_ptr**.
- **HxImageRep** is its related HORUS type and the conversion is done calling: `HxGetTiedObject<`**HxImageRepTiedSvt** (p. **??**)`>(_server, im)`.
- Global HORUS function returning images will return a **HxImageRep**.
- To convert that result to a **HxCorba::ImageRep_ptr** SG has to call: `HxRegisterTiedServant<`**HxImageRepTiedSvt** (p. **??**)`>(_server, res)`.

This information is given to the SG as a **GenType** (p. 25) object.

To extend SG to a new type, implement a `GenType` for that type and update the `getType` function in `ServantGenerator.c` to include this new GenType.

5.2 GenType Objects

```
class GenType
{
public:
    virtual const char*     paramDecl() = 0;
    virtual const char*     resultDecl() = 0;

    virtual const char*     requiredLine() = 0;
    virtual const char*     conversionLine(const char* argname) = 0;
    virtual const char*     convertedParam(const char* argname) = 0;

    virtual const char*     callAssignment() = 0;
    virtual const char*     returnLine() = 0;
};
```

For each type we want SG to support, we use a GenType object to specify all the information it needs to know about this type:

`const char* paramDecl():` `paramDecl` returns the associated **CORBA**/C++ type used for function parameters. For instace, `paramDecl()` of the `GenType` for `HxCorba.ImageRep` should return **HxCorba::ImageRep_ptr**.

`const char* resultDecl():` `resultDecl` returns the associated **CORBA**/C++ type used as result type. For instace, `resultDecl()` of the `GenType` for `HxCorba.ImageRep` should return also **HxCorba::ImageRep_ptr**.

`const char* requiredLine():` `requiredLine` returns a line of code that has to be added to the implementation file before any function that uses the type. The `GenType` for **enumerators** (p. 27) uses this feature.

`const char* conversionLine(const char* argname):` `conversionLine` returns the line of code used to convert from the **CORBA**/C++ type to the HORUS related type. `argname` is the name of the variable that contains the **CORBA**/C++ value. For instace, `conversionLine("im")` of the `GenType` for `HxCorba.ImageRep` should return **HxImageRep** `_im_arg = HxGetTiedObject<`**HxImageRepTiedSvt** (p. **??**)`>(_server, im);`.

`const char* convertedParam(const char* argname):` `convertedParam` returns the variable name used to retrieve the result of the conversion. Usualy, `convertedParam("N")` returns `"_N_arg"`. If there is no need of conversion, it returns `"N"` itself.

`const char* callAssignment():` `callAssignment` returns how to hold the value returned by the call to the HORUS global function. For instance, `callAssignment()` of the `GenType` for `HxCorba.ImageRep` should return **HxImageRep** `_result = `.

`const char* returnLine():` `returnLine` returns the return statement of the generated function. It contains the conversion, if needed, from the HORUS value to the related **CORBA**/C++ type. For instance, `returnLine()` of the `GenType` for `HxCorba.ImageRep` should return `return HxRegisterTiedServant<`**HxImageRepTiedSvt** (p. **??**)`>(_server, _result);`.

5.2.1 SampleGenType

The class `SampleGenType` can be used to easily create GenTypes. The idea is to build GenType objects just providing a pice of code like:

```
HxCorba::ImageRep_ptr
HxGlobalOpsGenSvt::HxAbs(HxCorba::ImageRep_ptr im)
{
    HxImageRep _im_arg = HxGetTiedObject<HxImageRepTiedSvt>(_server, im);
    HxImageRep _result = ::HxAbs(_im_arg);
    return HxRegisterTiedServant<HxImageRepTiedSvt>(_server, _result);
}
```

SampleGenType extracts all the information it needs from this code.

SampleGenType takes the sample code in a TypeSample struct:

```
struct TypeSample
{
    char*           func_name;
    char*           arg_name;
    char*           header;
    char*           conversionLine;
    char*           callAssignment;
    char*           returnLine;
};
```

Here is an example that shows how to create a GenType for HxCorba.ImageRep using a SampleGenType:

```
char* head = "HxCorba::ImageRep_ptr ImageSample(HxCorba::ImageRep_ptr img)";
char* conv = "HxImageRep _img_arg = HxGetTiedObject<HxImageRepTiedSvt>(_server, img);";
char* call = "HxImageRep _result = ::ImageSample(_img_arg);";
char* retn = "return HxRegisterTiedServant<HxImageRepTiedSvt>(_server, _result);";

char* func = "ImageSample";
char* arg  = "img";

TypeSample sample = {func, arg, head, conv, call, retn};
GenType* typeImageRep = new SampleGenType(sample);
```

- `func_name` and `arg_name` help `SampleGenType` to parse the code provided.
- `paramDecl` and `resultDecl` is implemented using the `header` field.
- `conversionLine` and `convertedParam` is implemented using the `conversionLine` field.
- `callAssignment` is implemented using the `callAssignment` field.
- `returnLine` is implemented using the `returnLine` field.

5.2.2 GenType for enumerators

`EnumGenType` is a class that can be used to implement the GenType for enumerators, but the enumerators should follow these rules:

1. For an enumeration named HxCorba::MyEnum, its HORUS type is HxMyEnum.
2. Both the HORUS enumeration and the **CORBA** enumeration contain the same elements.

In case that the enumeration doesn't follow these rules, EnumGenType should not be used and a GenType has to be defined for that enumeration.

If a GenType is not defined for a given enumeration type, SG instantiates a EnumGenType for it. No need to tell SG explicitly how to support this enum type.