ORBit Beginners Documentation V1.2

Ewan Birney

Michael Lausch

Todd Lewis

Stéphane Genaud

Frank Rehberger

ORBit Beginners Documentation V1.2

by Ewan Birney, Michael Lausch, Todd Lewis, Stéphane Genaud, and Frank Rehberger

Published 2003

Table of Contents

Chapter 1. Introduction

ORBit is an efficient, free C-based ORB, compliant to CORBA version 2.2. CORBA stands for Common Object Request Broker Architecture. CORBA in many ways is a successor to the Remote Procedure Call (RPC) system common on UNIX and other systems: the best oneline description of CORBA is that it is "Object Orientated RPC".

The key point about CORBA is that it provides a way for two programs to communicate information. The CORBA mechanism allows these two programs to be running on *different machines* and written in *different programming* languages while safely (and portably) exchanging data. They also could be running in the same program, on the same machine, in which case the process of communication is much quickeras ORBit recognises that it does not need to open any communication channel.

ORBit has some important features which distinguish it from other CORBA implementations. Firstly it is primarily a C-based implementation (using the standard C mapping), which many other CORBA implementations do not cater to. Secondly it is a fast implementation, meaning that it moves information on and off the communication channel into the programs quickly. Thirdly it is efficient in terms of its memory usage, meaning it can be run on small machines. Fourthly it works, and works in real life problems. Finally it is free software, that is free in terms of an open license to change the source code. This means that a number of different people (both academics and commerical people) can contribute to the source code and that bugs and problems can be fixed quickly. ORBit unsurprisingly is therefore licensed under the GNU Public License (GPL). More information about how the terms of this license affect you can be found in the FAQ, and you're encouraged to read the actual GPL text that comes in the distribution

The CORBA mechanism is ideal for classic client/server applications, say a graphical client GUI communicating to a database server, or much simpler applications, such as one program asking for a mail service from the CORBA system. ORBit was made principly for the latter case: that is communicating information between different programs running in a desktop enviroment (the Gnome enviroment). ORBit is a fully blown generic ORB and can be used in many other applications outside of the gnome desktop - there is absolutely no requirement to be running Gnome for ORBit.

This documentation is aimed principly at getting people with C programming experience started with ORBit and some discussion about the challenges of writing ORB aware software. If you are a looking for a more fluffy description about what CORBA is I would start at the Resource section below. If you want to know the technical nitty-gritty about ORBit, go to the source code, or hang out on the ORBit mailing list.

It is also hoped that this documentation will be a starting point for a more technical orientated documentation about how ORBit is structured and the internal functions.

About this documentation

This documentation is currently maintained by Frank Rehberger frehberg@gnome.org [1]. For the definitive guide about what is going on, always go to the source code (with any luck, it will have documentation) or the C mapping document at the OMG. If you think there is mistake with this documentation, please email me. If you would like to contribute an example or another chapter, please also get in contact, or post to the orbit mailing list.

This documentation was written with major contributions from Ewan Birney, Michael Launsch and Todd Lewis, edited by Stéphane Genaud, ORBit update by Frank Rehberger. It is written in docbook XML. You can get :

- The documentation and examples at http://www.gnome.org/projects/ORBit2/orbit-docs.tar.gz [2]

- A PostScript version of this document at http://www.gnome.org/projects/ORBit2/orbit-docs/orbit.ps [3]

Notes

1. mailto:frehberg@gnome.org
2. http://www.gnome.org/projects/ORBit2/orbit-docs.tar.gz
3. http://www.gnome.org/projects/ORBit2/orbit-docs/orbit.ps

Chapter 2. Frequently Asked Questions

This is to answer your questions about ORBit and Corba

General Questions

What is CORBA?

CORBA stands for Common Object Request Broker Architecture, standardized by the Object Management Group (OMG). It consists of an Interface Definition Language (IDL), language bindings and protocols to allow interoperation between applications

- written in different programming languages
- running in different processes
- developed for different operating systems

A more detailed answer to this question is available at http://www.omg.org/gettingstarted [1]

What is ORBit?

ORBit is an implementation of CORBA, which is used (amongst other things) in the GNOME project. Active development is done for ORBit2, being a clean re-implementation from scratch. The old ORBit code is no longer maintained.

Why aren't you using ILU/MICO/TAO?

Most of GNOME is written in C, both for performance reasons, and because C++ still isn't as portable and widely available. So integration with CORBA should use the IDL C mapping, which rules out TAO and MICO. ILU was ruled out because the license appeared as too restrictive at the time for free software.

Is ORBit stable?

ORBit is used in pretty much any GNOME application, which gives quite some testing for the features used. Other features have seen little or no testing.

Does ORBit work with other ORBs?

Yes. This is one of the primary features of CORBA, and available through the Internet Inter-ORB Protocol (IIOP). There is one pitfall: ORBit has a proprietary security mechanism to authenticate clients. If interoperability is desired then you might have to figure out a work-around for this.

What license does ORBit get distributed with and how does it affect me?

The idl compiler is *GNU Public licensed* (GPL). The libaries are *Library GPL* (L-GPL) licensed. The code generated from the idl compiler have no restriction. Bascially this means you can use ORBit in your own applications (including proprietary applications), the only restriction being that people whom you distribute your working code to must be able to update the ORBit libraries at will. If you wish to modify and distribute the ORBit idl compiler you have to distribute that modification under the GPL.

Why doesn’t ORBit compile with my cc compiler ?

ORBit uses some of the extensions that gcc allows, including assignment to casts and void * pointer arithmetic. More current versions of ORBit will remove these and eventually it is likely the ORBit will compile with all ANSI C compilers.

You can always install gcc and use that. This will always work.

What is a language binding?

A language binding defines how to use the IDL operations in a programming language. There is a client-side mapping which explains how to call operations, and a server-side mapping which defines how to implement them. The client side mapping is largely implementation-independent.

On the server side, different ’object adapters’ are provided by ORB vendors; only the ’Portable Object Adapter’ (POA) implemented by ORBit allows portability between CORBA implementations.

What language bindings does ORBit provide?

At the moment only the C binding. Python and Perl bindings are being seriously talked about. Bindings (in various degrees of completeness) are also available for C++, Lisp, Pascal, Python, Ruby, and TCL; others are in-progress.

Does ORBit have DII and DSI?

Yes. Though beware that using DII and DSI in some cases requires the use of dynAny, which is not supported yet.

Does ORBit provide a naming service?

Yes. In addition, it is possible to use a third-party naming service, e.g. using the --ORBNamingIOR command line option. The ’Interoperable Naming’ specification is not yet supported.

Can you encrypt a CORBA connection?

Yes, ORBit supports IIOP over SSL. the implementation is based on OpenSSL and support client and server side authentication. Frank O’Dwyer did initial work on this for ORBit.

Common Coding Problems

LD_LIBRARY_PATH not defined

ORBit uses dynamically loadable libraries and these need to be on the LD_LIBRARY_PATH for things to work.

IOR strings with newlines

When you get an object using a stringified IOR, make sure you trim the trailing newline from the string before passing it on.

Activating POAManager before servants can handle requests

You must activate the POAManager before incoming requests can be processed

```
PortableServer_POAManager_activate(PortableServer_POA__get_the_POAManager(poa,&ev),&ev)
```

Notes

1. http://www.omg.org/gettingstarted/

Chapter 3. Installing ORBit

Installing from RPMS

The RPMs can be found on many web sites. The whole ORBit package is broken in two parts (as for most RPMs) : the `ORBit2` package contains the libraries to use it, and the `ORBit2-devel` package contains the necessary files to develop with ORBit, i.e. the idl compiler and the headers files.

I recommend the RPM repositories like http://www.rpmfind.org to find the RPMS that best suit your distribution. Though ORBit is probably installed if you already have GNOME installed, you must download the `ORBit-devel` to develop with ORBit.

Installation is quite simple : once the packages are downloaded, go:

```
bash $ rpm -Uvh ORBit2* ORBit2-devel*
```

Note that this will upgrade any previous rpm-based installation of ORBit.

Installing from Source

Installing from ORBit2 source code is very easy. You need to have glib version 2.0.0 or later series installed. To see if you have glib installed (and what version) go **glib-config --version**

You can obtain the latest version of glib from http://www.gtk.org/

Now just download the latest stable tar ball of ORBit. Start by going to the gnome web site (http://www.gnome.org/), choose download and find your closest mirror site.

ORBit installs as a standard gnu application, so going **`./configure`** followed by **`make`** and then getting root permissions and **`make install`** will do everything as expected.

What to do if you don't have root permissions Basically you want to run all of the above but with **`./configure --prefix=<installation location>`** which will install under *prefix*/bin /libs etc.

If ORBit2 2.7 does not compile with some commercial C compilers, this is because it uses a number of gcc extensions which are not supported elsewhere. *you can always download gcc and install that* for it to work.

Getting the examples to run

Once you have successfully installed orbit, getting the examples to run should be fine. The examples are available by http from http//www.gnome.org/projects/ORBit2/orbit-docs.tar.gz [2]. Gunzip and untar the package, perhaps like:

```
bash $gunzip -c orbit-docs.tar.gz | tar -xvf -
```

This should build a documentation directory that contains this documentation in both sgml (docbook) and html, and an examples directory. Move to the examples directory, and go

```
bash $./configure
bash $cd echo
bash $make
```

Proceed similarly for the calculator example.

To actually run the echo example, open two windows up. Run `echo-server` in one and `echo-client` in the other (echo-server writes out the file `echo.ior` which echo-client reads). If you want to have fun now, compile orbit on a different machine, make echo-client on the different machine, move `echo.ior` from the first machine to the second and run echo-client: eh voila - client/server over tcp/ip with the same code that ran using unix sockets on a single machine.

> **Note:** From ORBit 0.5.3, IIOP communications over IP sockets are disabled by default for security reasons. You should create an `/etc/orbitrc` (for a system-wide settings), or an `~/.orbitrc` (for one user's setting) containing something like :
>
> ```
> ORBIIOPUSock=1
> ORBIIOPIPv4=1
> ORBIIOPIPv6=0
> ```
>
> Alternatively, you can also use command line parameters.

Known platforms where ORBit works

ORBit in theory is portable to any UNIX system (using the GNU configure system to define the system). Try it out - it will probably work. However, you may want to see if your platform has definitely been used before

- GNU/Linux, i386 Redhat/SuSE/Debian distributions
- GNU/Linux, alpha
- GNU/Linux, sparc
- Solaris 2.6 (UltraSparc)
- Digital UNIX 4.0
- FreeBSD 2.2.7-RELEASE (Intel)
- IRIX 6.2 (you can get it from http://freeware.sgi.com)

If you manage to compile it on any other platforms, let me know.

Notes

1. http://www.rpmfind.org
2. http//www.gnome.org/projects/ORBit2/orbit-docs.tar.gz
3. http://freeware.sgi.com

Chapter 4. CORBA concepts

Please skip this chapter and head straight to the first examples if you are the sort of person who understands better just by example. The examples chapter can be read without reading this chapter.

This chapter is to help you get an understanding of all the different terms that are bandied about ORBs and CORBA.

CORBA is a collection of standards defined by the Object Management Group (OMG - the sole purpose of the OMG is to provide a framework for people to agree on standards for CORBA). The OMG does not provide any working implementations of the CORBA standards, and so can be impartial. Most people who use CORBA only really keep one or two of these standards in their heads, but all the standards are free to be read from the OMG web site http://www.omg.org/ [1]. However it is good to have an idea about how all these standards fit together to provide CORBA.

Quick tour of a working client/server

The basic mechanism of CORBA is to run a method (that is a function) of an object from a separate program (in fact the program need not be separate, but don't worry about that yet). Going from one end of this to the other, this is what needs to be defined.

Definition of the object - the IDL

A definition of what methods can be called on what objects need to be provided for both the caller and callee. This is done using the *Interface Definition Language* or IDL of the object. The only information which is required to make a caller able to call an object on a remote server is the IDL of the object. Everything else is handled by the standards defined by the OMG.

Identifying the Object

The calling program needs to identify the (possibly remote) object somehow. This is done by something called an *IOR*, which can be represented as a string, and therefore used as an argument in the program. Decoding the IOR the client can tell the CORBA protocol and character encoding the server supports, beside much more details.

Calling the method

The calling program needs to actually call the method on the object somehow. This is done in the programming language of the caller, and the way the IDL of the maps to the programming language is standardised by the OMG. This is called the *Language Mapping*, and all the different ORBs provide at least one mapping. ORBit provides at the moment a C mapping.

Of course, C is a compiled language, so to write a C program which is going to make an ORBit call, one needs to compile the IDL of the object one wants to access using the orbit-idl compiler, and then write a C program which call functions that the orbit-idl compiler defines, compile this user written code along with the code generated for the client by the orbit compiler, and link it to the orbit and network libraries.

However for more dynamic languages (eg, Python), a more natural way of using CORBA is to dynamically load in the IDL definition and use it without any separate compile phase. The beauty of CORBA is that it supports both mechanisms

Moving the call across the network

The function call on the client now gets handled by the *Object Request Broker* or *ORB* and gets translated to the actual TCP/IP request to the server *ORB* which is handling the object to be called. The two ORBs communicate via a protocol called the *Internet InterOrb Protocol* or *IIOP*, which is standard. This means that the ORBs could come from different suppliers, and never had been tested against each other, and yet still work.

Returning the answer

On the server side, a mirror of the client process happens, translating the request from TCP/IP packets to an actual call in the programming language used to write the server. The servant object, written by whoever wrote the server then provides the return values, and the results are sent back down the ORBs to the client in reply message.

Continue

This may seem like a lot of work for a rather simple thing, but the idea about having so many standards is that some of the standards can change and adapt without everything in CORBA shifting. Learning about CORBA is usually much easier by example, so I would dip into the next chapter and come back to this one later.

Notes

1. http://www.omg.org/

Chapter 5. First CORBA Programs

Examples introduction

Three different, working programs will be presented in this chapter. They are all very simple and worth looking over and running (you will find the source code for them in the examples directory that comes with orbit-docs.tar.gz)

I have tested out the first two examples, echo and calculator and they work from the code in this document. The actual code with makefiles can be found at http://www.gnome.org/projects/ORBit2/orbit-docs.tar.gz [1]. However, it is likely that the code is not is well written as we would like, and I am sure it could get better. Please drop me a note if you would like to contribute something to the examples, preferably as patches to the code in examples directory and the orbit.xml document.

Files at a glance

Before we present these examples, we should sketch what files are generated or written by the programmer in a typical ORBit application. The load of files generated from the IDL specification may seem very complex at a first glance, and the following table tries to make clearer which files are really important. The first file the programmer must write in an ORBit application, is the IDL file. For a `f.idl` IDL specification, running

```
bash $orbit-idl-2 --skeleton-impl f.idl
```

will produce most of the needed files to start writing the application (you can see it as a framework). They are listed in the following table:

File	Usage
f-stubs.c	do not edit (translation of your IDL)
f-skels.c	do not edit (translation of your IDL)
f-common.c	do not edit (generated from IDL)
f.h	do not edit
f-skelimpl.c	add the code of the server methods

Files remaining to write :

f-client.c	write the client code
f-server.c	write the generic code for servant creation

The first three files deal with communication implementation and are automatically generated from your IDL file. The `f-skelimpl.c` file contains the framework of the servant functions: you should find near the end of the file, function prototypes corresponding to the one you have declared in your IDL. You have to fill in these function with the real code. Of course, you should keep this file only on the server side since it is of no use for the client.

The `f-client.c` (you can choose another name) is to be written by the programmer. This is where the programmer initializes the ORB and binds the client to the server using the IOR.

The `f-server.c` (you can choose another name) must also be written but this part is rather generic and is usually copied from an existing application and hardly ever modified during the application developpement. It always deals with the ORB initialization and the POA creation so that servants are spawned to execute the server code defined in `f-skelimpl.c` .

Echo client and server

The aim for this example is to run a client which accepts typed input and this is printed out on the server on stdout

The echo functionality is represented in the following very short IDL.

Example 5-1. Echo IDL file

```
// MyFirstOrbit program - The Echo object
//
// All this does is pass a string from the
// client to the server.

interface Echo {
 void echoString(in string input);
};
```

The *interface* definition is the key part of the definition. Each interface defines an object which can have methods on it. In this case there is one method, which takes a string argument and returns nothing. The *in* declaration before the argument indicates that this arguments is only passed into the method. Generally all arguments are in arguments, for the first case.

This idl is found in the file `echo.idl`. To compile the idl one does the following step: `$ `**`orbit-idl-2 --skeleton-impl echo.idl`** which will produce most of the needed files to start writing the *echo* application (you can see it as a framework). They are listed in the following table:

File	Usage for Client	Usage for Server
echo.h	readonly	readonly
echo-common.c	readonly	readonly
echo-stubs.c	readonly	-
echo-skels.c	-	readonly
echo-skelimpl.c	-	template for user code

Files remaining to write are listed in following table, starting with *echo-client.c* in following chapter.

echo-client.c	write the client code
echo-server.c	write the generic code for servant creation

Echo client

The client code is shown here

Example 5-2. echo-client.c

```
/*
 * Echo client program.. Hacked by Ewan Birney <birney@sanger.ac.uk>
 * from echo test suite, update for ORBit2 by Frank Rehberger
 * <F.Rehberger@xtradyne.de>
 *
 * Client reads object reference (IOR) from local file 'echo.ior' and
 * forwards console input to echo-server. A dot . as single character
 * in input terminates the client.
 */

#include <stdio.h>
#include <orbit/orbit.h>

/*
 * This header file was generated from the idl
 */

#include "echo.h"

/**
 * test for exception
 */
static
gboolean
raised_exception(CORBA_Environment *ev)
{
 return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process
 */
static
void
abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
 if (raised_exception (ev)) {
  g_error ("%s %s", mesg, CORBA_exception_id (ev));
  CORBA_exception_free (ev);
  abort();
 }
}

/*
 * main
 */
int
main (int argc, char *argv[])
{
 FILE * ifp;
 char * ior;
 char filebuffer[1024];

 CORBA_Environment ev[1];
 CORBA_ORB orb;             /* ORB */
 Echo echo_client;          /* the service */

  /*
   * Standard initalisation of the orb. Notice that
   * ORB_init 'eats' stuff off the command line
```

```
 */

CORBA_exception_init(ev);
orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
abort_if_exception(ev, "init ORB failed");

/*
 * Get the IOR (object reference). It should be written out
 * by the echo-server into the file echo.ior. So - if you
 * are running the server in the same place as the client,
 * this should be fine!
 */

ifp = fopen("echo.ior","r");
if( ifp == NULL ) {
 g_error("can not open \"echo.ior\"");
 abort ();
}

fgets(filebuffer,1023,ifp);
ior = g_strdup(filebuffer);

fclose(ifp);

/*
 * Actually get the object. So easy!
 */

echo_client = CORBA_ORB_string_to_object(orb, ior, ev);
abort_if_exception(ev, "bind failed");

/*
 * Ok. Now we use the echo object...
 */

g_print("Type messages to the server\n"
 "a single dot in line will terminate input\n");

while( fgets(filebuffer,1024,stdin) ) {
 if( filebuffer[0] == '.' && filebuffer[1] == '\n' )
  break;

 /* chop the newline off */
 filebuffer[strlen(filebuffer)-1] = '\0';

 /* using the echoString method in the Echo object
  * this is defined in the echo.h header, compiled from
  * echo.idl */

 Echo_echoString(echo_client,filebuffer,ev);
 abort_if_exception(ev, "service not reachable");
}

/* Clean up */
CORBA_Object_release(echo_client, ev);
abort_if_exception(ev, "releasing service failed");

CORBA_ORB_destroy (orb, ev);
abort_if_exception (ev, "cleanup failed");

/* successfull termination */
exit (0);
}
```

The client can be broken down into three distinct sections.

- Initialising the ORB
- Getting the Object
- Using the Object

The key part of the client is when it calls the echoString method on the server. The idl definition

```
void echoString(in string input);
```

Ends up becoming the following definition in the echo.h header file generated from the idl

```
extern void Echo_echoString(Echo                obj,
                            CORBA_char         *astring,
                            CORBA_Environment *ev);
```

This follows the accepted rules for Object based programming in C, that is

- The Object is passed in as the first argument to the function
- The method name is qualified by the object name beforehand
- Exceptions are handled by a structure that is passed in as the last argument to the function

Of course, you don't have to follow this in your own code, but this is how the CORBA C mapping works, and it is not a bad solution.

Echo Server

The server is basically more complicated than the client, but has some commonality with the client. The server has to at the end of the day go into a main loop where it listens to connections. Before that it has to create the ORB and bind its own implementations of the objects to the ORB.

In real life servers, this gets much more complicated, but as this is an example, it is pretty simple once you get through the ORB initialisation process.

Example 5-3. echo-server.c source code

```
/*
 * echo-server program. Hacked from Echo test suite by
 * %lt;birney@sanger.ac.uk>, ORBit2 udpate by Frank Rehberger
 * <F.Rehberger@xtradyne.de>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <orbit/orbit.h>

#include "echo.h"
#include "echo-skelimpl.c"

/**
 * test for exception */
static
gboolean
raised_exception(CORBA_Environment *ev) {
 return ((ev)->_major != CORBA_NO_EXCEPTION);
}
```

```
/**
 * in case of any exception this macro will abort the process  */
static
void
abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
 if (raised_exception (ev)) {
  g_error ("%s %s", mesg, CORBA_exception_id (ev));
  CORBA_exception_free (ev);
  abort();
 }
}

static CORBA_ORB  global_orb = CORBA_OBJECT_NIL; /* global orb */

/* Is called in case of process signals. it invokes CORBA_ORB_shutdown()
 * function, which will terminate the processes main loop.
 */
static
void
echo_server_shutdown (int sig)
{
 CORBA_Environment  local_ev[1];
 CORBA_exception_init(local_ev);

        if (global_orb != CORBA_OBJECT_NIL)
        {
                CORBA_ORB_shutdown (global_orb, FALSE, local_ev);
                abort_if_exception (local_ev, "caught exception");
        }
}

/* Inits ORB @orb using @argv arguments for configuration. For each
 * ORBit options consumed from vector @argv the counter of @argc_ptr
 * will be decremented. Signal handler is set to call
 * echo_server_shutdown function in case of SIGINT and SIGTERM
 * signals.  If error occures @ev points to exception object on
 * return.
 */static
void
echo_server_init (int               *argc_ptr,
    char             *argv[],
    CORBA_ORB         *orb,
    CORBA_Environment *ev)
{
 /* init signal handling */

 signal(SIGINT,  echo_server_shutdown);
 signal(SIGTERM, echo_server_shutdown);

 /* create Object Request Broker (ORB) */

        (*orb) = CORBA_ORB_init(argc_ptr, argv, "orbit-local-orb", ev);
 if (raised_exception(ev)) return;
}

/* Creates servant and registers in context of ORB @orb. The ORB will
 * delegate incoming requests to specific servant object.  @return
 * object reference. If error occures @ev points to exception object
 * on return.
 */
static
Echo
echo_server_activate_service (CORBA_ORB         orb,
         CORBA_Environment *ev)
```

```
{
 Echo                        servant     = CORBA_OBJECT_NIL;
 PortableServer_POA          poa         = CORBA_OBJECT_NIL;
 PortableServer_POAManager  poa_manager = CORBA_OBJECT_NIL;

        /* get Portable Object Adaptor (POA) */

        poa =
  (PortableServer_POA) CORBA_ORB_resolve_initial_references(orb,
           "RootPOA",
           ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

       /* create servant in context of poa container */

 servant = impl_Echo__create (poa, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

        /* activate POA Manager */

        poa_manager = PortableServer_POA__get_the_POAManager(poa, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

 PortableServer_POAManager_activate(poa_manager, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

 return servant;
}

/* Writes stringified object reference of @servant to file
 * @filename. If error occures @ev points to exception object on
 * return.
 */
static
void
echo_server_export_service_to_file (CORBA_ORB          orb,
        Echo               servant,
        char              *filename,
        CORBA_Environment *ev)
{
        CORBA_char *objref = NULL;
 FILE       *file   = NULL;

 /* write objref to file */

        objref = CORBA_ORB_object_to_string (orb, servant, ev);
 if (raised_exception(ev)) return;

 if ((file=fopen(filename, "w"))==NULL)
  g_error ("could not open %s\n", filename);

        /* print ior to terminal */
 fprintf (file, "%s\n", objref);
 fflush (file);
 fclose (file);

        CORBA_free (objref);
}

/* Entering main loop @orb handles incoming request and delegates to
 * servants. If error occures @ev points to exception object on
 * return.
 */
static
void
echo_server_run (CORBA_ORB          orb,
```

```
   CORBA_Environment *ev)
{
        /* enter main loop until SIGINT or SIGTERM */

        CORBA_ORB_run(orb, ev);
 if (raised_exception(ev)) return;

        /* user pressed SIGINT or SIGTERM and in signal handler
  * CORBA_ORB_shutdown(.) has been called */
}

/* Releases @servant object and finally destroys @orb. If error
 * occures @ev points to exception object on return.
 */
static
void echo_server_cleanup (CORBA_ORB          orb,
      Echo               servant,
      CORBA_Environment *ev)
{
 /* releasing managed object */
        CORBA_Object_release(servant, ev);
 if (raised_exception(ev)) return;

        /* tear down the ORB */
        if (orb != CORBA_OBJECT_NIL)
        {
                /* going to destroy orb.. */
                CORBA_ORB_destroy(orb, ev);
  if (raised_exception(ev)) return;
        }
}

/*
 * main
 */

int
main (int argc, char *argv[])
{
 Echo servant = CORBA_OBJECT_NIL;

 CORBA_Environment  ev[1];
 CORBA_exception_init(ev);

 echo_server_init (&argc, argv, &global_orb, ev);
 abort_if_exception(ev, "init failed");

 servant = echo_server_activate_service (global_orb, ev);
 abort_if_exception(ev, "activating service failed");

 echo_server_export_service_to_file (global_orb,
         servant,
         "echo.ior",
         ev);
 abort_if_exception(ev, "exporting IOR failed");

 echo_server_run (global_orb, ev);
 abort_if_exception(ev, "entering main loop failed");

 echo_server_cleanup (global_orb, servant, ev);
 abort_if_exception(ev, "cleanup failed");

 exit (0);
}
```

The key part of the server is when it calls **servant = impl_Echo__create (poa, ev);**. This is a function defined in file `echo-skelimpl.c` being included at top of echo-server.c. For each object method of echo object interface file `echo-skelimpl.c` contains a predefined implementation that must be extended by user (specific regions are marked by comments); incoming requests are delegated by object manager to specific method implementation. - For echo server application only a single line for method **echoString(..)** must be inserted, this line will print the echo-string to console. Let's have a look at `echo-skelimpl.c` that has been generated by **orbit-idl-2** tool as template for user. Therefor only a single line has been added by user **g_print ("%s\n", input);** at very end of file in function body *impl_Echo_echoString(..)*.

> **Note:** Constructor (create) and Destructor (destroy) are defined, too. How to extend those functions defining lifecycle of objects will be subject to next chapters.

Example 5-4. echo-skelimpl.c

```
#include "echo.h"

/*** App-specific servant structures ***/

typedef struct
{
   POA_Echo servant;
   PortableServer_POA poa;

   /* ------ add private attributes here ------ */
   /* ------ ---------- end ------------ ------ */
}
impl_POA_Echo;

/*** Implementation stub prototypes ***/

static void impl_Echo__destroy(impl_POA_Echo * servant,
          CORBA_Environment * ev);
static void
impl_Echo_echoString(impl_POA_Echo * servant,
       const CORBA_char * input, CORBA_Environment * ev);

/*** epv structures ***/

static PortableServer_ServantBase__epv impl_Echo_base_epv = {
   NULL,   /* _private data */
   (gpointer) & impl_Echo__destroy, /* finalize routine */
   NULL,   /* default_POA routine */
};
static POA_Echo__epv impl_Echo_epv = {
   NULL,   /* _private */
   (gpointer) & impl_Echo_echoString,

};

/*** vepv structures ***/

static POA_Echo__vepv impl_Echo_vepv = {
   &impl_Echo_base_epv,
   &impl_Echo_epv,
};

/*** Stub implementations ***/

static Echo
impl_Echo__create(PortableServer_POA poa, CORBA_Environment * ev)
```

```
{
   Echo retval;
   impl_POA_Echo *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_Echo, 1);
   newservant->servant.vepv = &impl_Echo_vepv;
   newservant->poa =
      (PortableServer_POA) CORBA_Object_duplicate((CORBA_Object) poa, ev);
   POA_Echo__init((PortableServer_Servant) newservant, ev);
   /* Before servant is going to be activated all
    * private attributes must be initialized.  */

   /* ------ init private attributes here ------ */
   /* ------ ---------- end ------------- ------ */

   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_Echo__destroy(impl_POA_Echo * servant, CORBA_Environment * ev)
{
   CORBA_Object_release((CORBA_Object) servant->poa, ev);

   /* No further remote method calls are delegated to
    * servant and you may free your private attributes. */
   /* ------ free private attributes here ------ */
   /* ------ ---------- end ------------- ------ */

   POA_Echo__fini((PortableServer_Servant) servant, ev);
}

static void
impl_Echo_echoString(impl_POA_Echo * servant,
       const CORBA_char * input, CORBA_Environment * ev)
{
   /* ------   insert method code here   ------ */
   g_print ("%s\n", input);
   /* ------ ---------- end ------------ ------ */
}
```

Compiling and Running the Server and the Client

The following makefile can be used to compile both, the client and the server. Be aware of the location of ORBit : on my system it has been installed under /usr but it could be /usr/local if you have built it from the sources, and hence the path for ORBIT variables below may vary. If using ORBit binary packages shipped with your Linux or BSD/Unix distribution the makefile below will do.

Example 5-5. Makefile

```
PREFIX=/usr/local
CC = gcc
TARGETS=echo-client echo-server
ORBIT_IDL=orbit-idl-2
CFLAGS=-DORBIT2=1 -D_REENTRANT -I$(PREFIX)/include/orbit-2.0 \
       -I$(PREFIX)/include/linc-1.0 -I$(PREFIX)/include/glib-2.0 \
       -I$(PREFIX)/lib/glib-2.0/include
LDFLAGS= -Wl,--export-dynamic -L$(PREFIX)/lib -lORBit-2 -llinc -lgmodule-2.0 \
```

```
            -ldl -lgobject-2.0 -lgthread-2.0 -lpthread -lglib-2.0 -lm
IDLOUT=echo-common.c echo-stubs.c echo-skels.c echo.h

all: $(IDLOUT) echo-client echo-server

echo-client : echo-client.o echo-common.o echo-stubs.o
echo-server : echo-server.o echo-common.o echo-skels.o

$(IDLOUT): echo.idl
        $(ORBIT_IDL) echo.idl

clean:
        rm -rf *.o *~ $(IDLOUT)

distclean: clean
        rm -rf echo-client echo-server
```

Example 5-6. Invoking make

```
[frehberg@papaya echo]$ make
orbit-idl-2 echo.idl
orbit-idl-2 2.4.1 compiling
 small mode, show preprocessor errors, passes: stubs skels common headers
 skel_impl imodule

gcc -DORBIT2=1 -D_REENTRANT -I/usr/include/orbit-2.0 -I/usr/include/linc-1.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include   -c -o echo-client.o
echo-client.c
gcc -DORBIT2=1 -D_REENTRANT -I/usr/include/orbit-2.0 -I/usr/include/linc-1.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include   -c -o echo-common.o
echo-common.c
gcc -DORBIT2=1 -D_REENTRANT -I/usr/include/orbit-2.0 -I/usr/include/linc-1.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include   -c -o echo-stubs.o
echo-stubs.c
gcc -Wl,--export-dynamic -lORBit-2 -llinc -lgmodule-2.0 -ldl -lgobject-2.0
-lgthread-2.0 -lpthread -lglib-2.0 -lm  echo-client.o echo-common.o
echo-stubs.o   -o echo-client
gcc -DORBIT2=1 -D_REENTRANT -I/usr/include/orbit-2.0 -I/usr/include/linc-1.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include   -c -o echo-server.o
echo-server.c
gcc -DORBIT2=1 -D_REENTRANT -I/usr/include/orbit-2.0 -I/usr/include/linc-1.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include   -c -o echo-skels.o
echo-skels.c
gcc -Wl,--export-dynamic -lORBit-2 -llinc -lgmodule-2.0 -ldl -lgobject-2.0
-lgthread-2.0 -lpthread -lglib-2.0 -lm  echo-server.o echo-common.o
echo-skels.o   -o echo-server
```

After calling **make** in terminal window all sources have been compiled and you should open a second terminal window. In the first window we will start the server with the command: **./echo-server**. The server should print a very long string into the file `echo.ior`, starting with the 4 character sequence `IOR:` In the second window we will print content of `echo.ior` to console and start the client with the command **./echo-client**. You should not try to type the IOR string, instead use the cut and paste functionality of your terminal.

Example 5-7. Terminal 1 - Starting Echo Server

```
[frehberg@papaya echo]$ ./echo-server
```

Example 5-8. Terminal 2 - Starting Echo Client

```
[frehberg@papaya echo]$ cat echo.ior
IOR:010000000d00000049444c3a4563686f3a312e300000000003000000054424f540000000101
020005000000554e4958000000000070000000706170617961000002e0000002f746d702f6f72626974
2d66726568626572672f6c696e632d323230662d302d32353235666332353730643034000000000
0000caaedfba58000000010102002e0000002f746d702f6f726269742d667265686265726672f6c69
6e632d323230662d302d3235323566633235373064303400000001c00000000000000a6361450d7ea
e8a8dc29282828282828010000008af91bdf01000000480000000100000002000000050000001c00
000000000000a6361450d7eae8a8dc29282828282828010000008af91bdf0100000014000000100
00000100010500000000090101000000000
[frehberg@papaya echo]$ ./echo-client
```

```
Type messages to the server
a single dot in line will terminate input:
```

Calculator Client Server

This example will take you further into understanding how to actually make a working server, using the support that comes from ORBit. It will demonstrate handling of replies from server.

The system will not be doing very much. The server just provides two functions, one to add two numbers and one to subtract two numbers. The first thing you have to do is to write the IDL files for the server. In our example it is very simple.

Example 5-9. calculator.idl

```
interface Calculator
{
      double add(in double number1, in double number2);
      double sub(in double number1, in double number2);
};
```

Then you have to generate the skeleton and stub files. In addition to these two files the ORBit IDL compiler also generates a common file and a header file. The common file implements the memory management functions and other things, useful in the client as well as in the server. The sequence to generate the C source files is rather simple. `$ ` **`orbit-idl-2 --skeleton-impl calculator.idl`** geenrates all the files we will use in this example.

File	Usage for Client	Usage for Server
calculator.h	readonly	readonly
calculator-common.c	readonly	readonly
calculator-stubs.c	readonly	-
calculator-skels.c	-	readonly
calculator-skelimpl.c	-	template for user code

Files remaining to write are listed in following table, starting with *calculator-client.c* in following chapter.

calculator-client.c	write the client code
calculator-server.c	write the generic code for servant creation

Calculator Client

The next thing you have to do is to write the server and client programs. We start with the client, because it's easier and not very complicated.

A simple implementation of the client might look like this

Example 5-10. calculator-client.c

```
/* calculator-client.c hacked by Frank Rehberger
 * <F.Rehberger@xtradyne.de>.  */

#include <assert.h>
#include <stdio.h>
#include <orbit/orbit.h>

#include "calculator.h"

/**
 * test for exception
 */
static
gboolean
raised_exception(CORBA_Environment *ev)
{
 return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process
 */
static
void
abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
 if (raised_exception (ev)) {
  g_error ("%s %s", mesg, CORBA_exception_id (ev));
  CORBA_exception_free (ev);
  abort();
 }
}

/*
 * main
 */
int
main(int argc, char* argv[])
{
 char*             ior;

        CORBA_ORB         orb;
        CORBA_Object      server;

        CORBA_double      result=0.0;

        CORBA_Environment ev[1];
        CORBA_exception_init(ev);

 /* init - ORB might 'eat' arguments from command line */
        orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
 abort_if_exception(ev, "init ORB failed");

 /* make sure servant's IOR is given as command argument */
 if (argc<2)
  g_error ("usage: %s <ior>", argv[0]);
 ior=argv[1];

 /* establish servant connection */
        server = CORBA_ORB_string_to_object(orb, ior, ev);
 abort_if_exception(ev, "bind failed");

 /*
  * use calculator server
```

```
 */
       result = Calculator_add(server, 1.0, 2.0, ev);
abort_if_exception(ev, "service not reachable");

/* prints results to console */
       g_print("Result: 1.0 + 2.0 = %2.0f\n", result);

/* tear down object reference and ORB */
       CORBA_Object_release(server,ev);
abort_if_exception(ev, "releasing service failed");

CORBA_ORB_destroy (orb, ev);
abort_if_exception(ev, "cleanup failed");

/* successfull termination */
exit(0);
}
```

Rather simple, but full of unexplained stuff. Let's take a close look to the defined variables.

env

This varaible is used to hold information about exceptions which might have occurred during a function call. How to use this variable to detect errors in function will be explained in a later example.

orb

This is the ORB itself.

server

This is the object reference to the server.

The example above is a full functional client. The magic in this example is the usage of the function *CORBA_ORB_string_to_object* with the parameter *argv[1]*. The explantion is that the program is supposed to be called with the string representation of the Calculator server as the first parameter. How to obtain this string, will be shown in the next example, where I describe how the server.

Calculator Server

To implement the server, the IDL compiler does a great deal of work for you. It can emit all the stuff necessary to set up the data structures and function calls for the server implementation. All you have to write is the setup stuff in your main function and the actual implementation of the server functions. First I'll present the functions and data structures the IDL compiler generates and then I'll show what's necessary to set up the environment for these functions to work properly.

Calculator Implementation Skeleton

To ease the task of implementing the calculator the ORBit IDL compiler can output an implementation skeleton of the server. This is enabled with the --skeleton-impl switch to the IDL compiler. The output of **orbit-idl-2 --skeleton-impl**

calculator.idl looks like this (the default name for the generated source file is `calculator-skelimpl.c`:

Example 5-11. calculator-skelimpl.c

```
#include "calculator.h"

/*** App-specific servant structures ***/

typedef struct
{
   POA_Calculator servant;
   PortableServer_POA poa;

   /* ------ add private attributes here ------ */
   /* ------ ---------- end ------------ ------ */
}
impl_POA_Calculator;

/*** Implementation stub prototypes ***/

static void impl_Calculator__destroy(impl_POA_Calculator * servant,
         CORBA_Environment * ev);
static CORBA_double
impl_Calculator_add(impl_POA_Calculator * servant,
      const CORBA_double number1,
      const CORBA_double number2,
      CORBA_Environment * ev);

static CORBA_double
impl_Calculator_sub(impl_POA_Calculator * servant,
      const CORBA_double number1,
      const CORBA_double number2,
      CORBA_Environment * ev);

/*** epv structures ***/

static PortableServer_ServantBase__epv impl_Calculator_base_epv = {
   NULL,   /* _private data */
   (gpointer) & impl_Calculator__destroy, /* finalize routine */
   NULL,   /* default_POA routine */
};
static POA_Calculator__epv impl_Calculator_epv = {
   NULL,   /* _private */
   (gpointer) & impl_Calculator_add,

   (gpointer) & impl_Calculator_sub,

};

/*** vepv structures ***/

static POA_Calculator__vepv impl_Calculator_vepv = {
   &impl_Calculator_base_epv,
   &impl_Calculator_epv,
};

/*** Stub implementations ***/

static Calculator
impl_Calculator__create(PortableServer_POA poa,
                        CORBA_Environment * ev)
{
   Calculator retval;
   impl_POA_Calculator *newservant;
```

```
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_Calculator, 1);
   newservant->servant.vepv = &impl_Calculator_vepv;
   newservant->poa = poa;
   POA_Calculator__init((PortableServer_Servant) newservant, ev);
   /* Before servant is going to be activated all
    * private attributes must be initialized.  */

   /* ------ init private attributes here ------ */
   /* ------ ---------- end ------------- ------ */

   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}

static void
impl_Calculator__destroy(impl_POA_Calculator * servant,
    CORBA_Environment * ev)
{
   CORBA_Object_release((CORBA_Object) servant->poa, ev);

   /* No further remote method calls are delegated to
    * servant and you may free your private attributes. */
   /* ------ free private attributes here ------ */
   /* ------ ---------- end ------------- ------ */

   POA_Calculator__fini((PortableServer_Servant) servant, ev);
}

static CORBA_double
impl_Calculator_add(impl_POA_Calculator * servant,
      const CORBA_double number1,
      const CORBA_double number2,
      CORBA_Environment * ev)
{
   CORBA_double retval;

   /* ------   insert method code here   ------ */
   retval = number1 + number2;
   /* ------ ---------- end ------------ ------ */

   return retval;
}

static CORBA_double
impl_Calculator_sub(impl_POA_Calculator * servant,
                    const CORBA_double number1,
                    const CORBA_double number2,
      CORBA_Environment * ev)
{
   CORBA_double retval;

   /* ------   insert method code here   ------ */
   retval = number1 - number2;
   /* ------ ---------- end ------------ ------ */

   return retval;
}
```

This source file provides you with most of the magic of a server. Note that we generate this file (with the **--skeleton-impl** switch) only once, and then the makefile invokes **orbit-idl-2** with no switch. If you call **orbit-idl-2 --skeleton-impl** from the makefile, the previous file will be overwritten and your implementation code lost. Once the implementation code is written, just include the source file at the beginning of the `calculator-server.c` file.

For this first example, I won't explain all the bits and pieces of the generated source file. This will be done later. We'll just concentrate on getting the server running.

As you see there are two functions:

```
CORBA_double impl_Calculator_add(impl_POA_Calculator* servant,
CORBA_double number1, CORBA_double number2, CORBA_Environment* ev);
```

and

```
CORBA_double impl_Calculator_sub(impl_POA_Calculator* servant,
CORBA_double number1, CORBA_double number2, CORBA_Environment* ev);
```

These two functions are implementing the function defined in the IDL file. Because the IDL compiler doesn't provide you with a real implementation (it doesn't know what the function should do), you have to extend this skeleton yourself where marked.

The `impl_Calculator_add()` should add it's two parameters and return the result so this function should be changed into:

Example 5-12. calculator-skelimpl.c fragment

```
static CORBA_double
impl_Calculator_add(impl_POA_Calculator * servant,
                    const CORBA_double number1,
                    const CORBA_double number2,
      CORBA_Environment * ev)
{
   CORBA_double retval;

   /* ------   insert method code here   ------ */
   retval = number1 + number2;
   /* ------ ---------- end ------------ ------ */

   return retval;
}
```

Calculator Server Implementation

The things you need in your minimal main function to make things work can be implemented in the following way, note analogy to `echo-server.c` of previous example.

Example 5-13. calculator-server.c

```
/*
 * calculator-server program. Hacked from Frank Rehberger
 * <F.Rehberger@xtradyne.de>.
 */

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <signal.h>
#include <orbit/orbit.h>

#include "calculator.h"
#include "calculator-skelimpl.c"

/**
 * test for exception */
static
gboolean
raised_exception(CORBA_Environment *ev) {
 return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process  */
static
void
abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
 if (raised_exception (ev)) {
  g_error ("%s %s", mesg, CORBA_exception_id (ev));
  CORBA_exception_free (ev);
  abort();
 }
}

static CORBA_ORB  global_orb = CORBA_OBJECT_NIL; /* global orb */

/* Is called in case of process signals. it invokes CORBA_ORB_shutdown()
 * function, which will terminate the processes main loop.
 */
static
void
calculator_server_shutdown (int sig)
{
 CORBA_Environment  local_ev[1];
 CORBA_exception_init(local_ev);

        if (global_orb != CORBA_OBJECT_NIL)
        {
                CORBA_ORB_shutdown (global_orb, FALSE, local_ev);
                 abort_if_exception (local_ev, "ORB shutdown failed");
        }
}

/* Inits ORB @orb using @argv arguments for configuration. For each
 * consumed option from vector @argv the counter of @argc_ptr
 * will be decremented. Signal handler is set to call
 * calculator_server_shutdown function in case of SIGINT and SIGTERM
 * signals.  If error occures @ev points to exception object on
 * return.
 */static
void
calculator_server_init (int               *argc_ptr,
   char              *argv[],
   CORBA_ORB         *orb,
   CORBA_Environment *ev)
{
 /* init signal handling */

 signal(SIGINT,  calculator_server_shutdown);
 signal(SIGTERM, calculator_server_shutdown);

 /* create Object Request Broker (ORB) */
```

```
        (*orb) = CORBA_ORB_init(argc_ptr, argv, "orbit-local-orb", ev);
 if (raised_exception(ev)) return;
}

/* Creates servant and registers in context of ORB @orb. The ORB will
 * delegate incoming requests to specific servant object.  @return
 * object reference. If error occures @ev points to exception object
 * on return.
 */
static
Calculator
calculator_server_activate_service (CORBA_ORB         orb,
        CORBA_Environment *ev)
{
 Calculator                 servant     = CORBA_OBJECT_NIL;
 PortableServer_POA         poa         = CORBA_OBJECT_NIL;
 PortableServer_POAManager  poa_manager = CORBA_OBJECT_NIL;

        /* get Portable Object Adaptor (POA) */

        poa =
  (PortableServer_POA) CORBA_ORB_resolve_initial_references(orb,
           "RootPOA",
           ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

       /* create servant in context of poa container */

 servant = impl_Calculator__create (poa, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

        /* activate POA Manager */

        poa_manager = PortableServer_POA__get_the_POAManager(poa, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

 PortableServer_POAManager_activate(poa_manager, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

 return servant;
}

/* Writes stringified object reference of @servant to file-stream
 * @stream. If error occures @ev points to exception object on
 * return.
 */
static
void
calculator_server_export_service_to_stream (CORBA_ORB          orb,
         Calculator         servant,
         FILE              *stream,
         CORBA_Environment *ev)
{
        CORBA_char *objref = NULL;

 /* write objref to file */

        objref = CORBA_ORB_object_to_string (orb, servant, ev);
 if (raised_exception(ev)) return;

        /* print ior to terminal */
 fprintf (stream, "%s\n", objref);
 fflush (stream);

        CORBA_free (objref);
```

```
}

/* Entering main loop @orb handles incoming request and delegates to
 * servants. If error occures @ev points to exception object on
 * return.
 */
static
void
calculator_server_run (CORBA_ORB          orb,
         CORBA_Environment *ev)
{
        /* enter main loop until SIGINT or SIGTERM */

        CORBA_ORB_run(orb, ev);
 if (raised_exception(ev)) return;

        /* user pressed SIGINT or SIGTERM and in signal handler
  * CORBA_ORB_shutdown(.) has been called */
}

/* Releases @servant object and finally destroys @orb. If error
 * occures @ev points to exception object on return.
 */
static
void calculator_server_cleanup (CORBA_ORB          orb,
    Calculator         servant,
    CORBA_Environment *ev)
{
 /* releasing managed object */
        CORBA_Object_release(servant, ev);
 if (raised_exception(ev)) return;

        /* tear down the ORB */
        if (orb != CORBA_OBJECT_NIL)
        {
                /* going to destroy orb.. */
                CORBA_ORB_destroy(orb, ev);
  if (raised_exception(ev)) return;
        }
}

/*
 * main
 */

int
main (int argc, char *argv[])
{
 Calculator servant = CORBA_OBJECT_NIL;

 CORBA_Environment  ev[1];
 CORBA_exception_init(ev);

 calculator_server_init (&argc, argv, &global_orb, ev);
 abort_if_exception(ev, "init failed");

 servant = calculator_server_activate_service (global_orb, ev);
 abort_if_exception(ev, "activating service failed");

 calculator_server_export_service_to_stream (global_orb, /* ORB    */
          servant,    /* object */
          stdout,     /* stream */
          ev);
 abort_if_exception(ev, "exporting IOR failed");

 calculator_server_run (global_orb, ev);
```

```
 abort_if_exception(ev, "entering main loop failed");

 calculator_server_cleanup (global_orb, servant, ev);
 abort_if_exception(ev, "cleanup failed");

 exit (0);
}
```

I'm not going to explain every line of this example yet, because we want the server up and make our first calls to it. Though one line deserves some explanation and this is the `fprintf(stream, "%s\n",objref)` call. The purpose of this call is to print the string representation of the the object reference. This string, which always starts with the magic sequence "IOR:", is the argument to the client program. It identifies a specific object, the server process which hosts it, the location of the server, and the identity of the object in this specific server, because it's possible that one server hosts many objects. How to get such strings or object references without cutting the output of one program (the server) and pasting it into the commandline of another program (the client) will be explained later.

Compiling and Running the Server and the Client

The following makefile can be used to compile both, the client and the server. Be aware of the location of ORBit : on my system it has been installed under /usr but it could be /usr/local if you have built it from the sources, and hence the path for ORBIT variables below may vary. If using ORBit binary packages shipped with Linux or BSD/Unix the simple makefile below will do.

Example 5-14. makefile

```
PREFIX=/usr/local
CC = gcc
TARGETS=calculator-client calculator-server
ORBIT_IDL=orbit-idl-2
CFLAGS=-DORBIT2=1 -D_REENTRANT -I$(PREFIX)/include/orbit-2.0 \
       -I$(PREFIX)/include/linc-1.0 -I$(PREFIX)/include/glib-2.0 \
       -I$(PREFIX)/lib/glib-2.0/include
LDFLAGS= -Wl,--export-dynamic -L$(PREFIX)/lib -lORBit-2 -llinc -lgmodule-2.0 \
             -ldl -lgobject-2.0 -lgthread-2.0 -lpthread -lglib-2.0 -lm
IDLOUT=calculator-common.c calculator-stubs.c calculator-skels.c calculator.h

all: $(IDLOUT) calculator-client calculator-server

calculator-client : calculator-client.o calculator-common.o calculator-stubs.o
calculator-server : calculator-server.o calculator-common.o calculator-skels.o

$(IDLOUT): calculator.idl
	$(ORBIT_IDL) calculator.idl

clean:
	rm -rf *.o *~ $(IDLOUT)

distclean: clean
	rm -rf calculator-client calculator-server
```

After calling **make** in terminal window all sources have been compiled and you should open a second terminal window. In the first window we will start the server with the command: **calculator-server > calculator.ior**. The server should print a very long string into the file `calculator.ior`, starting with the 4 character sequence `IOR:`

In the second window we start the client with the command **calculator-client ‘cat calculator.ior‘ IOR-string**. You should not try to type the IOR string, instead use the cut and paste functionality of your xterm or whatever you are using.

If everything works, you should get the following output: `Result: 1.0 + 2.0 = 3`.

Account Client and Server

In this third simple example, we will see how we can set up a client that can modify the value of a variable stored on the server. It is basically the way an account manager works. The stored variable here is balance. The idl definition (`account.idl`) for our account is :

Example 5-15. account.idl

```
interface Account {
   void deposit (in unsigned long amount);
   void withdraw (in unsigned long amount);
   readonly attribute long balance;
};
```

Then you have to generate the skeleton and stub files. In addition to these two files the ORBit IDL compiler also generates a common file and a header file. The common file implements the memory management functions and other things, useful in the client as well as in the server. The sequence to generate the C source files is rather simple. `$ orbit-idl-2 --skeleton-impl calculator.idl` geenrates all the files we will use in this example.

File	Usage for Client	Usage for Server
account.h	readonly	readonly
account-common.c	readonly	readonly
account-stubs.c	readonly	-
account-skels.c	-	readonly
account-skelimpl.c	-	template for user code

Files remaining to write are listed in following table, starting with *account-client.c* in following chapter.

account-client.c	write the client code
account-server.c	write the generic code for servant creation

Account Client

There is no difficulty in setting the client (at least no more than in the previous examples). Only one thing has been added : we test for the availabilty of the server (`if (!acc_client)` ...) before invoking calls to the server.

Example 5-16. account-client.c

```
/* account-client.c hacked by Frank Rehberger
 * <F.Rehberger@xtradyne.de>.  */

#include <assert.h>
#include <stdio.h>
#include <orbit/orbit.h>

#include "account.h"

/**
 * test for exception
 */
static
gboolean
raised_exception(CORBA_Environment *ev)
{
 return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process
 */
static
void
abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
 if (raised_exception (ev)) {
  g_error ("%s %s", mesg, CORBA_exception_id (ev));
  CORBA_exception_free (ev);
  abort();
 }
}

/*
 * main
 */
int
main(int argc, char* argv[])
{
 char*             ior=NULL;
 CORBA_long        val=0;
        CORBA_long        balance=0;

        CORBA_ORB         orb;
        Account           server;

        CORBA_Environment ev[1];
        CORBA_exception_init(ev);

 /* init - ORB might 'eat' arguments from command line */
        orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
 abort_if_exception(ev, "init failed");

 /* make sure command lines contains two arguments; IOR and
  * integer value  */
 if (argc<3)
  g_error ("usage: %s <ior> <int>", argv[0]);
 ior = argv[1];
 val = atoi(argv[2]);

 /* establish servant connection */
        server = (Account) CORBA_ORB_string_to_object(orb, ior, ev);
 abort_if_exception(ev, "bind failed");
```

```
 /*
  * use calculator server
  */

        balance = Account__get_balance (server, ev);
 abort_if_exception(ev, "service not reachable");

 g_print ("balance %5d, ", balance);

 if (val > 0)
 {
  Account_deposit (server, val, ev);
  abort_if_exception(ev, "service not reachable");
 }
 else
 {
  Account_withdraw (server, abs(val), ev);
  abort_if_exception(ev, "service not reachable");
 }

        balance = Account__get_balance (server, ev);
 abort_if_exception(ev, "service not reachable");

 g_print ("new balance %5d\n", balance);

 /* tear down object reference and ORB */
        CORBA_Object_release(server,ev);
 abort_if_exception(ev, "releasing service failed");

 CORBA_ORB_destroy (orb, ev);
 abort_if_exception(ev, "cleanup failed");

 /* successfull termination */
 exit(0);
}
```

Account Server

Account Server Skeleton Implementation

For the server, like in the previous example, we first generate the source file `account-skelimpl.c` that will receive the implementation code for the methods. This is done once again with **`orbit-idl-2 --skeleton-impl account.idl`**.

Now, let us edit `account-skelimpl.c`. We search for the the `balance` attribute that was declared in the IDL file. At the beginning of the file, we can spot the way it has been translated into C by the idl compiler:

Example 5-17. account-skelimpl.c fragment - object declaration

```
typedef struct
{
   POA_Account servant;
   PortableServer_POA poa;

   CORBA_long attr_balance;

   /* ------ add private attributes here ------ */
   CORBA_long attr_balance;
   /* ------ ---------- end ------------ ------ */
}
```

```
impl_POA_Account;
```

So, the server methods (withdraw and deposit) will have to manage the balance of the account through the `servant->attr_balance` (the servant variable is passed as parameter to each method).

Now, let us get to the end of the file and find the methods stubs. We find the `impl_Account_*` functions, to which we add the implementation code. This could be:

Example 5-18. account-skelimpl.c fragment - method definition

```
static void
impl_Account_deposit(impl_POA_Account * servant,
       const CORBA_unsigned_long amount, CORBA_Environment * ev)
{
   /* ------   insert method code here   ------ */
   servant->attr_balance += amount;
   /* ------ ---------- end ------------ ------ */
}

static void
impl_Account_withdraw(impl_POA_Account * servant,
        const CORBA_unsigned_long amount,
        CORBA_Environment * ev)
{
   /* ------   insert method code here   ------ */
   servant->attr_balance -= amount;
   /* ------ ---------- end ------------ ------ */
}

static CORBA_long
impl_Account__get_balance(impl_POA_Account * servant, CORBA_Environment * ev)
{
   CORBA_long retval;

   /* ------   insert method code here   ------ */
   retval = servant->attr_balance;
   /* ------ ---------- end ------------ ------ */

   return retval;
}
```

The missing key stone is the constructor that establishes initial, consistent state for object on creation.

Example 5-19. account-skelimpl.c fragment - constructor

```
...
static Account
impl_Account__create(PortableServer_POA poa, CORBA_Environment * ev)
{
   Account retval;
   impl_POA_Account *newservant;
   PortableServer_ObjectId *objid;

   newservant = g_new0(impl_POA_Account, 1);
   newservant->servant.vepv = &impl_Account_vepv;
   newservant->poa =
      (PortableServer_POA) CORBA_Object_duplicate((CORBA_Object) poa, ev);
   POA_Account__init((PortableServer_Servant) newservant, ev);
   /* Before servant is going to be activated all
    * private attributes must be initialized.  */
```

```
   /* ------ init private attributes here ------ */
   newservant->attr_balance = 0;
   /* ------ ---------- end ------------- ------ */

   objid = PortableServer_POA_activate_object(poa, newservant, ev);
   CORBA_free(objid);
   retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

   return retval;
}
..
```

Account Server Implementation

Lastly, we have to write a rather generic code to set up the server. We call it `account-server.c`. It is roughly the same code as in the calculator and echo examples. The code just initializes the ORB and publishes an IOR for the server object.

Example 5-20. account-server.c

```
/*
 * account-server program. Hacked from Frank Rehberger
 * <F.Rehberger@xtradyne.de>.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <orbit/orbit.h>

#include "account.h"
#include "account-skelimpl.c"

/**
 * test for exception */
static
gboolean
raised_exception(CORBA_Environment *ev) {
 return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process  */
static
void
abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
 if (raised_exception (ev)) {
  g_error ("%s %s", mesg, CORBA_exception_id (ev));
  CORBA_exception_free (ev);
  abort();
 }
}

static CORBA_ORB  global_orb = CORBA_OBJECT_NIL; /* global orb */

/* Is called in case of process signals. it invokes CORBA_ORB_shutdown()
 * function, which will terminate the processes main loop.
 */
static
void
```

```
account_server_shutdown (int sig)
{
 CORBA_Environment  local_ev[1];
 CORBA_exception_init(local_ev);

        if (global_orb != CORBA_OBJECT_NIL)
        {
                CORBA_ORB_shutdown (global_orb, FALSE, local_ev);
                abort_if_exception (local_ev, "ORB shutdown failed");
        }
}

/* Inits ORB @orb using @argv arguments for configuration. For each
 * consumed option from vector @argv the counter of @argc_ptr
 * will be decremented. Signal handler is set to call
 * account_server_shutdown function in case of SIGINT and SIGTERM
 * signals.  If error occures @ev points to exception object on
 * return.
 */static
void
account_server_init (int               *argc_ptr,
       char              *argv[],
       CORBA_ORB         *orb,
       CORBA_Environment *ev)
{
 /* init signal handling */

 signal(SIGINT,  account_server_shutdown);
 signal(SIGTERM, account_server_shutdown);

 /* create Object Request Broker (ORB) */

        (*orb) = CORBA_ORB_init(argc_ptr, argv, "orbit-local-orb", ev);
 if (raised_exception(ev)) return;
}

/* Creates servant and registers in context of ORB @orb. The ORB will
 * delegate incoming requests to specific servant object.  @return
 * object reference. If error occures @ev points to exception object
 * on return.
 */
static
Account
account_server_activate_service (CORBA_ORB         orb,
     CORBA_Environment *ev)
{
 Account                    servant     = CORBA_OBJECT_NIL;
 PortableServer_POA         poa         = CORBA_OBJECT_NIL;
 PortableServer_POAManager  poa_manager = CORBA_OBJECT_NIL;

        /* get Portable Object Adaptor (POA) */

        poa =
  (PortableServer_POA) CORBA_ORB_resolve_initial_references(orb,
           "RootPOA",
           ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

       /* create servant in context of poa container */

 servant = impl_Account__create (poa, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

        /* activate POA Manager */

        poa_manager = PortableServer_POA__get_the_POAManager(poa, ev);
```

```
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

 PortableServer_POAManager_activate(poa_manager, ev);
 if (raised_exception(ev)) return CORBA_OBJECT_NIL;

 return servant;
}

/* Writes stringified object reference of @servant to file-stream
 * @stream. If error occures @ev points to exception object on
 * return.
 */
static
void
account_server_export_service_to_stream (CORBA_ORB          orb,
      Account            servant,
      FILE              *stream,
      CORBA_Environment *ev)
{
        CORBA_char *objref = NULL;

 /* write objref to file */

        objref = CORBA_ORB_object_to_string (orb, servant, ev);
 if (raised_exception(ev)) return;

        /* print ior to terminal */
 fprintf (stream, "%s\n", objref);
 fflush (stream);

        CORBA_free (objref);
}

/* Entering main loop @orb handles incoming request and delegates to
 * servants. If error occures @ev points to exception object on
 * return.
 */
static
void
account_server_run (CORBA_ORB          orb,
      CORBA_Environment *ev)
{
        /* enter main loop until SIGINT or SIGTERM */

        CORBA_ORB_run(orb, ev);
 if (raised_exception(ev)) return;

        /* user pressed SIGINT or SIGTERM and in signal handler
  * CORBA_ORB_shutdown(.) has been called */
}

/* Releases @servant object and finally destroys @orb. If error
 * occures @ev points to exception object on return.
 */
static
void account_server_cleanup (CORBA_ORB          orb,
        Account            servant,
        CORBA_Environment *ev)
{
 /* releasing managed object */
        CORBA_Object_release(servant, ev);
 if (raised_exception(ev)) return;

        /* tear down the ORB */
        if (orb != CORBA_OBJECT_NIL)
        {
```

```
                /* going to destroy orb.. */
                CORBA_ORB_destroy(orb, ev);
  if (raised_exception(ev)) return;
        }
}

/*
 * main
 */

int
main (int argc, char *argv[])
{
 Account servant = CORBA_OBJECT_NIL;

 CORBA_Environment  ev[1];
 CORBA_exception_init(ev);

 account_server_init (&argc, argv, &global_orb, ev);
 abort_if_exception(ev, "init failed");

 servant = account_server_activate_service (global_orb, ev);
 abort_if_exception(ev, "activating service failed");

 account_server_export_service_to_stream (global_orb, /* ORB    */
       servant,    /* object */
       stdout,     /* stream */
       ev);
 abort_if_exception(ev, "exporting IOR failed");

 account_server_run (global_orb, ev);
 abort_if_exception(ev, "entering main loop failed");

 account_server_cleanup (global_orb, servant, ev);
 abort_if_exception(ev, "cleanup failed");

 exit (0);
}
```

Compiling the Server and the Client

The Makefile is the roughly the same as the one in the Calculator example. By now the schema should be clear and you should be able to reuse this Makefile for numerous small projects.

Example 5-21. Makefile for the Account example

```
PREFIX=/usr/local
CC = gcc
TARGETS=account-client account-server
ORBIT_IDL=orbit-idl-2
CFLAGS=-DORBIT2=1 -D_REENTRANT -I$(PREFIX)/include/orbit-2.0 \
       -I$(PREFIX)/include/linc-1.0 -I$(PREFIX)/include/glib-2.0 \
       -I$(PREFIX)/lib/glib-2.0/include
LDFLAGS= -Wl,--export-dynamic -L$(PREFIX)/lib -lORBit-2 -llinc -lgmodule-2.0 \
             -ldl -lgobject-2.0 -lgthread-2.0 -lpthread -lglib-2.0 -lm
IDLOUT=account-common.c account-stubs.c account-skels.c account.h

all: $(IDLOUT) account-client account-server

account-client : account-client.o account-common.o account-stubs.o
account-server : account-server.o account-common.o account-skels.o
```

```
$(IDLOUT): account.idl
        $(ORBIT_IDL) account.idl

clean:
        rm -rf *.o *~ $(IDLOUT)

distclean: clean
        rm -rf account-client account-server
```

Notes

1. http://www.gnome.org/projects/ORBit2/orbit-docs.tar.gz

Chapter 6. How to do garbage collection under CORBA

I wanted to do garbage collection under CORBA, and so I asked some questions on the ORBit list. Here is a distillation of the answers which I received, mostly verbatim and not necessarily in order.

Editor's note - this chapter came from Todd Lewis

The question

I asked the following question on the ORBit list:

"As a client, when I no longer need an object, how do I properly release it? I see a couple of different candidate functions, but I'm not sure which one is proper. If there's a piece of example code which I've overlooked, then a simple pointer thereto would suffice. My copy of the Henning & Vinoski book is on the way, and so hopefully I won't be so clueless here soon... "

Difference between the client and the server

Elliot Lee, ever the prescient one, asked:

"Object or object reference? "

And then observed:

"I think Todd is confused as to what the difference between object references and objects is. Think of an object reference and an object as being analogous to a pointer and pointed-to memory, respectively. You can stop using a pointer, but until you get the server side to actually free() the memory (object) being pointed to (referenced), it is still there. "

Indeed I was confused. There are actually two processes which occur within CORBA:

- the client drops all references to an object, after which the ORB local to the client then can clean up all of its data structures related to that object, and
- the server can decide that the object needs to be destroyed, and it then does the job of deactivating the object (so that the POA can know not to answer any more queries to that object) and freeing associated resources, etc.

Since object references can be generated in one program, passed to a second program, the first program can forget about it, and the second can continue, these are separate processes. It was understanding this fact which was the biggest impediment to my figuring this matter out.

Sascha Brawer <brawer@coli.uni-sb.de> described these two potential interpretations of what I wanted to do, reflecting a nuance which I did not appreciate when I asked the question:

- You want the client to forget about the object reference, but the server-side object should continue its life: Call CORBA_free on the client side. The purpose of CORBA_free is to release the memory in the client. Therefore, the server will not get any notification about this. If other clients had references to the same object, they will continue to work.
- You want to destroy the object on the server: define a method in the object's interface, e.g. "destroy" (though any name is fine). First, call that method on the client side. The server will receive the call, and your server implementation will destroy

> the object. Second, call CORBA_free on the client side, to free the client-side memory for the object reference which has now become stale. If other clients had references to the same object, they will receive an exception as soon they call a method on that object. However, they won't be notified in advance, because the server does not keep track on issued references.

Which Elliot Lee then corrected with:

CORBA_Object_release() on object references, CORBA_free() on data structures.

Dick Porter added:

It is _only_ the client side. These pseudo-object methods do not cause requests to be made to the server.

Server-side deactivation

Ok, so CORBA_Object_release takes care of the client side. This isn't enough, because, as Elliot Lee pointed out, to state the point slightly differently:

```
On Tue, 20 Apr 1999, Svanberg Liss wrote:
> Btw, what does CORBA_Object_duplicate & CORBA_Object_release do
> server?

Nothing..
```

So, what do you do on the server side? Elliot answered this, too:

```
> and... hmm, what kind of call does destroy the object in the server
 when
> release doesn't?

You PortableServer_POA_deactivate_object(poa, objid) to tell the POA not
to take any more requests for the specified objid.
```

Let's take a look at what this means in terms of actual code. If you run **`orbit-idl --skeleton_impl foo.idl`** on your idl file, you will get a file foo-impl.c. Inside of that file, you will see functions like the following:

```
/* You shouldn't call this routine directly without first deactivating
 the servant... */

static void

impl_CosTransactions_Control__destroy(

     impl_POA_CosTransactions_Control * servant,

     CORBA_Environment * ev)

{

   POA_CosTransactions_Control__fini((PortableServer_Servant) servant,
 ev);

   g_free(servant);

}
```

Where it says "You shouldn't call this routine directly without first deactivating the servant...", it means that you should call PortableServer_POA_deactivate_object() on the servant first.

FIXME: I don't understand what POA_CosTransactions_Control__fini does here; how is it different from PortableServer_POA_deactivate_object? Anyway, this is the final step you take after your object is deactivated; you can then free the POA servant struct which you created in your factory (or wherever.)

Why is this an issue?

Sascha Brawer suggests:

Maybe the confusion arises because there do exist distributed object systems that perform this sort of reference counting, allowing the server to know when no client has a reference left. DCOM has such a mechanism, and ILU has something similar, AFAIK.

Why doesn't CORBA follow these other systems? My personal impression is that this is another case of CORBA doing the right thing instead of the easy thing. Sascha continues:

However, I've read somewhere (don't remeber the exact location, sorry...) that the problem with distributed reference counting would be that first, it consumes network bandwith, and second, it does not work well with unreliable network connections. It was told in that article that these drawbacks would have been the rationale for designing CORBA without refcounting.

This is a problem. If your client disappears, then you've got unneeded objects lying around. All it takes for clients to disappear is for them to be coded by bad programmers, and we all know how unlikely that is, right? So how does one deal with this? There are several ways. The easiest is to have a timeout. This is the approach that I am using in the GNU Transaction Server; when the timeout expires, I deactivate all of my servant objects and free up their resources. Fortunately, the CosTransactions spec supports timeouts on transactions. Unfortunately, I don't really see how to do timeouts with the normal CORBA event model, so I'm going to have to use the ORBit-specific hack of having a glib event loop which supports both CORBA events and timeouts. Another way, I am told, is something called an Evictor pattern. I should understand this soon once I get my fancy CORBA book in the mail. The final way is just to leak memory like a sieve leaks goldfish; this is the easiest thing to do.