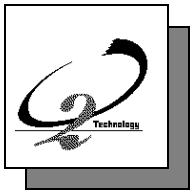


O_2Corba
User Manual

Release 5.0 - May 1998

Information in this document is subject to change without notice and should not be construed as a commitment by O_2 Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other that the purchaser's own use.

O_2 and O_2Engine API, O_2C, O_2Corba, O_2DBAccess, O_2Engine, O_2Graph, O_2Kit, O_2Look, O_2Store, O_2Tools are registered trademarks of O_2 Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Rational Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

Who should read this manual

This manual is for programmers who want to write Object Management Group (OMG) CORBA-compliant applications using the O_2 object database.

The manual introduces O_2Corba technology and describes how to generate interface definition language (IDL) files from O_2 classes. It also explains how to use the Common Object Request Broker Architecture (CORBA) services O_2Corba provides. We assume in this manual that the reader is familiar with CORBA technology and with Object Database Management Group (ODMG) C++ binding.

This manual should be used in conjunction with the ODMG C++ Reference Manual, which contains the full list of O_2 C++ interface options.

Other documents available are outlined, click below.

See O2 Documentation set.

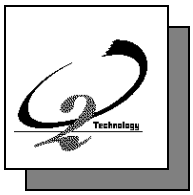


TABLE OF CONTENTS

This manual is divided into the following chapters:

TABLE OF CONTENTS

TABLE OF CONTENTS

1 Introduction

The O_2 database system provides a complete database solution for object developers.

OMG CORBA provides a means to run distributed object applications over a network. Several vendors provide products compliant with this technology. Among these products are Orbix (IONA), Chorus ORB (Sun Microsystems) and Visibroker (Visigenic). These products can be used jointly with O_2 to provide a fully distributed development environment for persistent object applications.

You will find that developing O_2 applications with an object request broker (ORB) has several key benefits: conformance to CORBA and C++ ODMG standards, automatic IDL generation from an O_2 schema, collection management, transaction service and query service via OQL.

This chapter is divided into the following sections:

- System overview
- O2 and CORBA
- Architecture
- Manual overview

Important

We assume that the reader is familiar with the CORBA technology and has experience with an ORB product as well as a knowledge of O_2 .

1 Introduction

1.1 System overview

The system architecture of O_2 is illustrated in Figure 1.1.

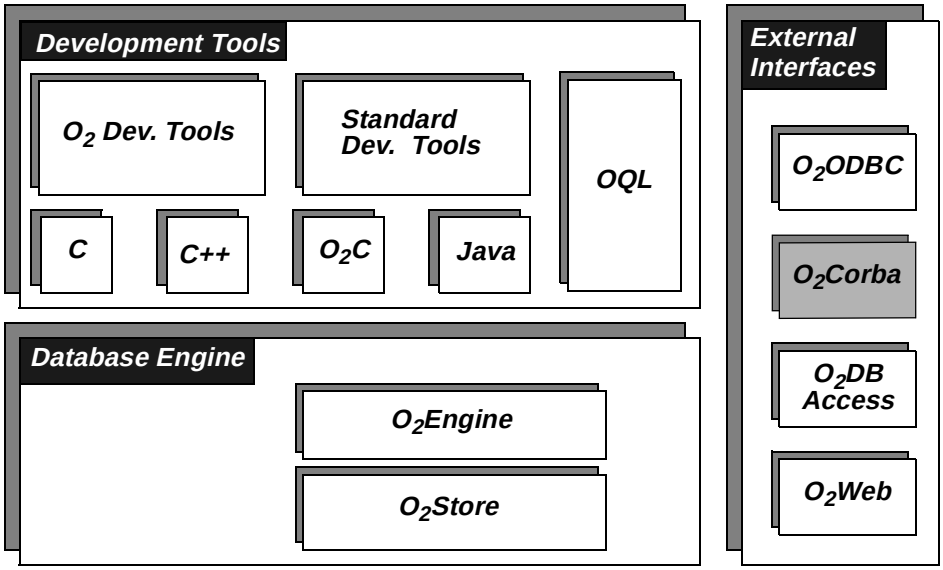


Figure 1.1: O_2 System Architecture

The O_2 system can be viewed as consisting of three components. The *Database Engine* provides all the features of a Database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O_2 development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O_2 is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

O_2Store — The database management system provides low level facilities, through O_2Store API, to access and manage a database: disk volumes, files, records, indices and transactions.

O_2Engine — The object database engine provides direct control of schemas, classes, objects and transactions, through O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial. It includes a Notification manager for informing other clients connected to the same O_2 server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O2 system.

Programming Languages:

O_2 objects may be created and managed using the following programming languages, utilizing all the features available with O_2 (persistence, collection management, transaction management, OQL queries, etc.)

C	O_2 functions can be invoked by C programs.
C++	ODMG compliant C++ binding.
Java	ODMG compliant Java binding.
O_2C	A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
OQL	ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O_2 objects and methods.

O_2 Development Tools:

O_2Graph	Create, modify and edit any type of object graph.
O_2Look	Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
O_2Kit	Library of predefined classes and methods for faster development of user applications.
O2Tools	Complete graphical programming environment to design and develop O2 database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

O_2Corba	Create an O_2/Orbix server to access an O_2 database with CORBA.
O_2DBAccess	Connect O_2 applications to relational databases on remote hosts and invoke SQL statements.
O_2ODBC	Connect remote ODBC client applications to O_2 databases.
O_2Web	Create an O_2 World Wide Web server to access an O_2 database through the internet network.

1.2 O_2 and CORBA

Basic principles

To access an O_2 database “from outside”, IDL is the language which allows you to give an external view of it. In IDL, you define the interface of any objects you want and you select the methods you need for the particular purpose of the view.

An O_2Corba server can be produced as follows:

The interface definition language (IDL) compiler parses the IDL interface and generates a corresponding C++ class. Let us call the IDL interface “K”. The compiler generates a C++ class, called “K”. This class can be used directly by the CORBA client.

The programmer then has to define a C++ *implementation class*, called K_i, which implements the view inside O_2 . This class has to inherit from a generated class, called KBOAImpl (Basic Object Adapter Implementation for K), which implements the CORBA client/server protocol. The class KBOAImpl is a subclass of K. The following figure shows the class hierarchy.

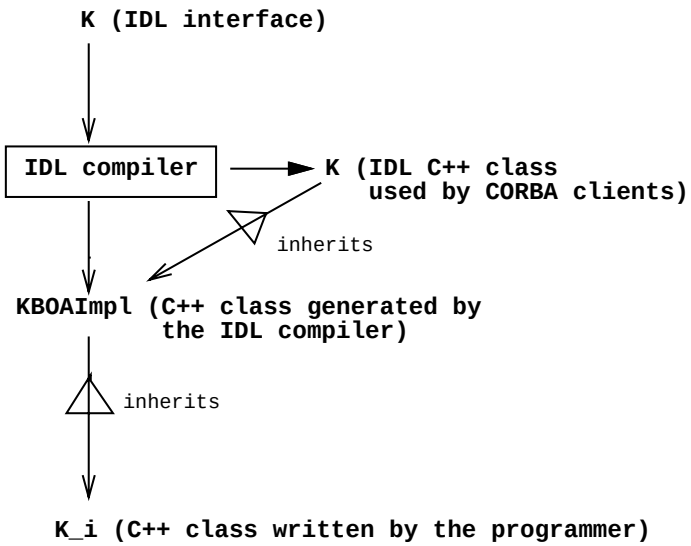


Figure 1.2: The CORBA class hierarchy

K_i can use any features of the C++ binding provided by O_2 . K_i is a pure transient C++ class, it does not need to be known by O_2 . It is linked with an O_2 process which runs a standard O_2 C++ client.

The programmer can define a simple way to bind the K_i class to a persistence capable class, called K_p, which is known by O_2 and which allows you to access and manipulate persistent objects.

The class K_i can be written as follows:

```
class K_i: public virtual KBOAImpl{
private:
   d_Ref<K_p> o2_object;
public:
   // all the methods of the view
};
```

A K_i object is a transient C++ object that refers to an O_2 C++ object of the persistence capable class K_p. It is straightforward to implement the methods of K_i. For instance, assuming that K has an attribute called “att” of type “t”, the methods to read and update the att attribute are defined as follows:

```
class K_i: public virtual KBOAImpl{
private:
   d_Ref<K_p> o2_object;
public:
   t att(){ return o2_object->att;}
   att(t value){ o2_object->att = value;}
   // ...
};
```

Automatic IDL generation

Starting from an existing O_2 schema, O_2 proposes to generate automatically the IDL specification and the C++ *implementation classes*.

The C++ code generation follows the approach described above (i.e. K_i refers to one object of a persistence capable class) and is performed by the **`o2cpp_export`** tool of the O_2 C++ binding development tool (see Chapter 3 for details about IDL generation).

Features and advantages

When using O_2 and CORBA together, the user benefits from:

- ***OMG interface***

The joint O_2-CORBA solution offers a standard OMG compliant interface, with persistence, transaction and query services.

1 Introduction

- ***Fully heterogeneous platforms***

  ORB implementations and O_2 jointly provide a fully heterogeneous platform. For instance, a client from an INTEL PC will be able to access an O_2 database managed on SPARC or RS-6000 platforms.

- ***Multiple servers***

  An O_2 system manages a distributed database, i.e, a set of O_2 schemas and O_2 bases, distributed on a set of disks within a network.

  Because O_2 provides a C++/ODMG-compliant interface and tools to generate IDL files from O_2 schemas, the development of a CORBA server using O_2 is very straightforward.

1.3 Architecture

On the client side, an object of the class `K` generated by the IDL compiler is called a *proxy* object, it implements the support necessary to send requests to the remote object on the server side (object of the class **`K_i`**).

On the O_2Corba server side, an object of the class `K_i` deals with one (or more) database object(s), instance(s) of a persistence capable class. The following figure shows the communication protocol.

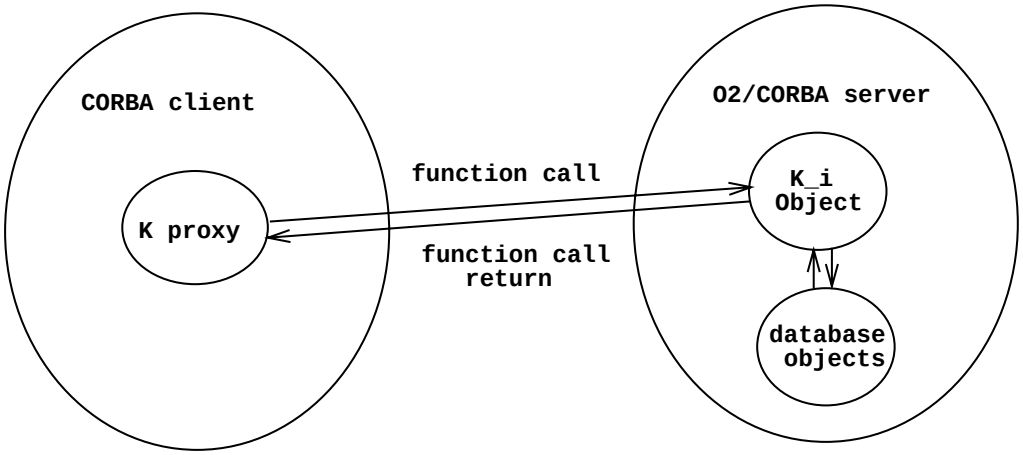


Figure 1.3: The O_2Corba protocol

There can be two possible architectures: (1) one O_2 client for each CORBA client or (2) a single O_2 client for several CORBA clients.

The architecture (1) must be chosen when the CORBA clients run concurrently on the same data in potentially conflicting modes. In this case they need to rely on the O_2 transactional system (one O_2 client for each transaction).

The architecture (2) can be adopted when either the CORBA clients are not conflicting (for instance they all run in read-only mode) or when each request coming from a CORBA client can be implemented as a complete transaction (which is worth only when the implementation of the request involves a significant amount of data manipulation).

1.4 Manual overview

This manual describes how to use an ORB product to access an O_2 database and it is divided into the following chapters:

- ***Introduction***

  Introduces the O_2 and CORBA connection and outlines some of its advantages.

- ***Installation***

  Describes the contents of the Corba connection module and explains how to run the example application in the distribution.

- ***Building an O_2 Corba server***

  Describes how to build a server from O_2 and CORBA using the O_2 ODMG C++ Binding development tools.

- ***Programming***

  Provides guidelines on programming with the CORBA architecture including managing persistent objects, collection manipulation, transaction management, memory management and using OQL.

- ***Example of an O_2 Corba application on top of Orbix***

  Gives a simple example of an O_2Corba server, it implements a two-dimensional grid with Orbix.

- ***Example of an O_2 Corba application on top of Chorus ORB***

  Gives a two-dimensional grid example implemented with Chorus ORB.

2 Installation

This chapter describes the contents of the O_2 Corba connection module and explains how to run the example application in the distribution.

It is divided into the following sections :

- Delivery
- Running the samples

2 Installation

2.1 Delivery

Your <O2-installation-directory> contains two directories related to the O_2 Corba connection module.

- ***<O2-installation-directory>/o2corba***

  This directory contains all the material needed to use the O_2 Corba connection in your applications. According to the platform you are using, there can be several subdirectories, one for each ORB product supported on your platform.

  For each supported ORB version are three directories:

  - include

    The include files needed to use the librairies **`libo2orbs.a`** and **`libo2orbc.a`**.

  - lib

    The `lib` directory contains the libraries from which the O_2 Corba server and the CORBA client must be linked, respectively **`libo2orbs.a`** and **`libo2orbc.a`**.

  - templates

    This directory contains template files which can be instanciated by the programmer (see Chapter 4).

- ***<O2-installation-directory>/samples/o2corba***

  The examples in this directory show how to use the ORB libraries of O_2 on top of the ODMG C++ binding in order to build an O_2 Corba server and client.

2.2 Running the samples

In what follows, the Orb installation directory is denoted as <Orb-installation-directory>.

Running the example program on top of Orbix

- cd <O2-installation-directory>/samples/o2corba
- Edit the file init.env
  - Set the variable **`ORBIX_ROOT`** to <Orb-installation-directory>
  - Set the variable **`O2ORB_VERSION`** to one of the subdirectories available in <O2-installation-directory>/samples/o2corba. It can be, for instance, Orbix_21cMT or Orbix_2.2.
  - Set the O_2 variables O2HOME, O2SYSTEM and O2SERVER.
- Execute the script init.env
  ```
  source init.env
  ```
- Run **`demo.sh`** (shell script which compiles and runs the examples)

Running the example program on top of Chorus ORB

- cd <O2_installation_directory>/samples/o2corba/ChorusOrb
- Edit the file init.env
  - Set the variable **`ORB_ROOT`** to <Orb-installation-directory>
  - Set the variable **`O2ORB_VERSION`** to the COOL directory.
  - Set the O_2 variables O2HOME, O2SYSTEM and O2SERVER.
- Execute the script init.env
  ```
  source init.env
  ```
- Run **`demo.sh`** (shell script which compiles and runs the examples)

2 Installation

3 Building an O_2Corba Server

This chapter is divided into the following sections :

3 Building an O2Corba Server

3.1 Introduction

There are two ways to build a server with O_2 and CORBA:

Starting from an existing O_2 schema, you build an IDL view on it which will be used by the CORBA client. This IDL view, including IDL interfaces and IDL C++ *implementation classes*, can be automatically generated by the **`o2cpp_export`** tool of the O_2 C++ binding programming environment. In this case, an object of the CORBA server will be linked with **one** object of the database (see Chapter 1).

Note

The O_2 schema can be defined using the standard O_2 tools and in particular the **`o2cpp_import`** tool of the O_2 C++ binding programming environment.

Starting from an existing IDL definition, you use the O_2 system and its ODMG C++ interface to implement this IDL interface.

In both cases, an O_2Corba server is an O_2 ODMG C++ application which is linked with the Orbix server library.

The following section describes how to use the **`o2cpp_export`** and **`o2makegen`** commands to produce IDL interfaces from an O_2 schema.

3.2 Export to IDL

The O_2 Export tool allows to export an O_2 class to C++ in order to use the objects and their associated methods in a C++ application[1]. It also provides the generation, for each O_2 exported class, of the corresponding IDL interface and the IDL C++ *implementation class*.

The **o2cpp_export** command provides an option called "**-idl**", this option allows to generate IDL files for a given exported O_2 class.

Example: Let us suppose that we want to export the class K, its public attributes and its method m, and generate IDL files, we invoke **o2cpp_export** as follows:

```
o2cpp_export ... -class "K -idl -type -method m"
```

The generated files are the following:

C++ standard export

1 **K.hxx**
C++ header containing the definition of the C++ class **K**, image of the O_2 exported class.

2 **K_code.cc**
C++ implementation of the function members **K::o2_read**, **K::o2_write**, etc

IDL export

1 **d_K.idl**
Interface header containing the definition of the IDL interface **d_K**, image of the O_2 exported class. Note that the prefix "**d_**" is added.

2 **d_K_i.hxx**
C++ header containing the definition of the C++ IDL *implementation class* **d_K_i.hxx**, subclass of the class **d_KBOAImpl** generated by the IDL compiler (we use the BOAImpl approach).

3 **d_K_i.cc**
C++ Implementation of the function members of the class **d_K_i**.

Note

1 In case the C++ files (i.e. **K.hxx** and **K_code.cc**) were already generated (e.g. by the **o2cpp_import** tool), you can turn off the C++ files generation and only generate the IDL files using the options "**-noheader**" and "**-nocode**" of the **o2cpp_export** tool.

```
o2cpp_export ... -class "K -noheader -nocode -idl -type
-method m"
```

1. Refer to the C++ ODMG Binding Guide and C++ ODMG Binding Reference Manual for details about the **o2cpp_export** command

2 Only the files **d_K.idl**, **d_K_i.hxx** and **d_K_i.cc** will be generated.

All the generated files will be deleted by the `o2unexport` command.

Exporting attributes

The option "**-type**" of the command **o2cpp_export** has to be used to export attributes from O_2 to IDL. **Only public attributes can be exported**.

For each O_2 exported attribute, **o2cpp_export** generates a twin IDL attribute (See the table, "Corresponding O2 and IDL types for export," to learn the type mapping) and two access member functions in the C++ implementation class, one for reading the attribute and one for writing the attribute.

Example: Let K be the class to be exported

```
class K private inherit Object
           type tuple(s: string,
                      o: K2)    /* K2 is an O2 class */
         end;
```

and the command **o2cpp_export**:

```
o2cpp_export ... -class "K -idl -type"
```

The corresponding interface **d_K** will be generated as follows:

```
// forward declarations
          interface d_K2;

          interface d_K {

             attribute string s;
             attribute d_K2 o;
};
```

Note that an interface called **d_K_Factory** will also be generated. This interface will be used to create temporary and persistent objects of the class K, and to access persistent objects of the class K (see Sections 4.5 and 4.4)

```
interface d_K_Factory: o2_Factory {
       d_K create_K();
       d_K create_persistent_K(in string name);
       d_K lookup_K(in string name);
};
```

The class **d_K_i** will be generated with the following access member functions (two functions per attribute):)

```
class d_K_i: public d_KBOAImpl {
         private:
            d_Ref<K> o2_object;
         public:
            ...
         virtual void s (const char* s, CORBA::Environment& env =
                                        CORBA::default_environment);
               virtual char* s (CORBA::Environment& env =
                                        CORBA::default_environment);
               virtual void o (d_K2* o, CORBA::Environment& env =
                                        CORBA::default_environment);
               virtual d_K2* o (CORBA::Environment& env =
                                        CORBA::default_environment);
};
```

Exporting methods

For each O_2 exported method, **o2cpp_export** generates a twin IDL operation (see the table, "Corresponding O_2 and IDL types for export," to learn the type mapping) and a member function in the C++ implementation class.

Example: Let K be the class to be exported

```
class K private inherit Object
      method public get_s:string,
             public set_s(v: string),
             public foo(obj: K2)
      end;
```

and the command **o2cpp_export**:

```
o2cpp_export ... -class "K -idl -method get_s -method set_s -method
```

The corresponding interface **d_K** will be generated as follows:

```
   // forward declarations
          interface d_K2;

          interface d_K {

             string get_s();
             void   set_s(in string v);
             void   foo(in d_K2 obj);
          };
```

The class **d_K_i** will be generated with the following member functions:

```
class d_K_i: public d_KBOAImpl {
  private:
    d_Ref<K> o2_object;
  public:
    ...
  virtual char* get_s (CORBA::Environment& env =
            CORBA::default_environment);
  virtual void  set_s (char* v, CORBA::Environment& env =
            CORBA::default_environment);
  virtual void  foo (d_K2* obj, CORBA::Environment& env =
            CORBA::default_environment);
};
```

O_2 Makegen

ExpIdl directive

The configuration file of **o2makegen** accepts a directive called **ExpIdl** related to export information. This directive indicates whether the IDL generation is performed or not (default is not).

The directive **ExpIdl** has the following syntax (the default is -):

```
+/-[ExpClassName]ExpIdl
```

When this directive is set, the **o2cpp_export** tool will be triggered with the "**-idl**" option and the Makefile generated by **o2makegen** will define rules in order to compile the C++ source files produced by the IDL export mechanism (e.g. **K_i.cc**).

Important

The object files generated from the C++ IDL source files (e.g. **`K_i.o`**) must NOT be in the list of the ProgramObjs directive. These object files will be automatically linked with the final executable.

ExpHeader and ExpCode directives

Two directives **`ExpHeader`** and **`ExpCode`** can be used to indicate to the **`o2cpp_export`** tool whether standard C++ files are generated or not.

The syntax of these two directives is the following (the default is +):

```
+/-[ExpClassName]ExpHeader
+/-[ExpClassName]ExpCode
```

When these directives are set, the **`o2cpp_export`** tool will be triggered with the "**`-noheader`**" and "**`-nocode`**" options and the standard C++ files will not be generated.

In the example below, only the IDL files will be produced for the class `K`:

```
-[K]ExpHeader
-[K]ExpCode
```

ExpType directive

The directive **`ExpType`** must be used to trigger the **`o2cpp_export`** tool with the "**`-type`**" option in order to export public attributes of an O_2 class.

The directive **`ExpType`** has the following syntax (the default is +):

```
+/-[ExpClassName]ExpType
```

Link and preprocessing directives

The link edition of the final application must be performed using the multi-thread option (-mt) and the C++ files must be compiled using the **`_REENTRANT`** option. To set these options, use the directives **`UserLdFlags`** and **`Define`** of the configuration file as follows:

```
UserLdFlags = -mt
Define      = _REENTRANT
```

3 Building an O2Corba Server

The final executable must be linked with the O_2 library **libo2orbs.a** and your ORB library (for example, library **libITsrvmt** for Orbix) using the **ProgramLib** directive:

```
ProgramLib: ITsrvmt  o2orbs
```

Creating a Makefile

If you prefer not to use the **o2makegen** tool, you have to write your own Makefile.

This section outlines the compilation options and libraries you need to generate an O_2Corba server. For a full description of the Makefile creation, refer to the O_2 C++ ODMG Binding Guide.

Compilation

The source files used to build the O_2Corba server must be compiled with the **_REENTRANT** compilation flag.

Link edition

An O_2 Corba server must be linked with the following O_2 libraries:

- **libo2common**
- **libo2util**
- **libo2store**
- **libo2api**
- **libo2runtime**
- **libo2cppruntime**
- **liboql** – if you use the Object Query Language (OQL) service
- **libo2orbs**

and your ORB server library (such as **liborbix.so** for Orbix)

The CORBA client must be linked with the O_2 library **libo2orbc.a**.

Finally, the link edition of the server and the client must be performed using the multithread option **-mt**.

3.3 Building an O_2Corba server with O_2

Building an O_2Corba server with the C++ Interface to O_2 consists in compiling and linking the files produced by the O_2 Export tool (if you use the IDL automatic generation) and the IDL compiler, the C++ applications files (if needed), the CORBA server library and the O_2 runtime libraries (see Figure 3.1).

To produce an O_2Corba server, you must go through the following steps:

- Export the O_2 classes (automatic IDL generation)

  For a class K, the following files are generated: **`K.hxx, K_code.cc`**, **`d_K_i.hxx`**, **`d_K_i.cc`** and **`d_K.idl`**.

- Call the IDL compiler

  The file **`d_K.idl`** generated by **`o2cpp_export`** will be used as input of the IDL compiler. The following files are generated: **`d_K.hh`** and **`d_KS.cc`**.

- Compile and link

  All the generated files plus the application sources must be compiled and linked with the ORB and O_2 libraries.

The **o2makegen** tool takes charge of the whole process.

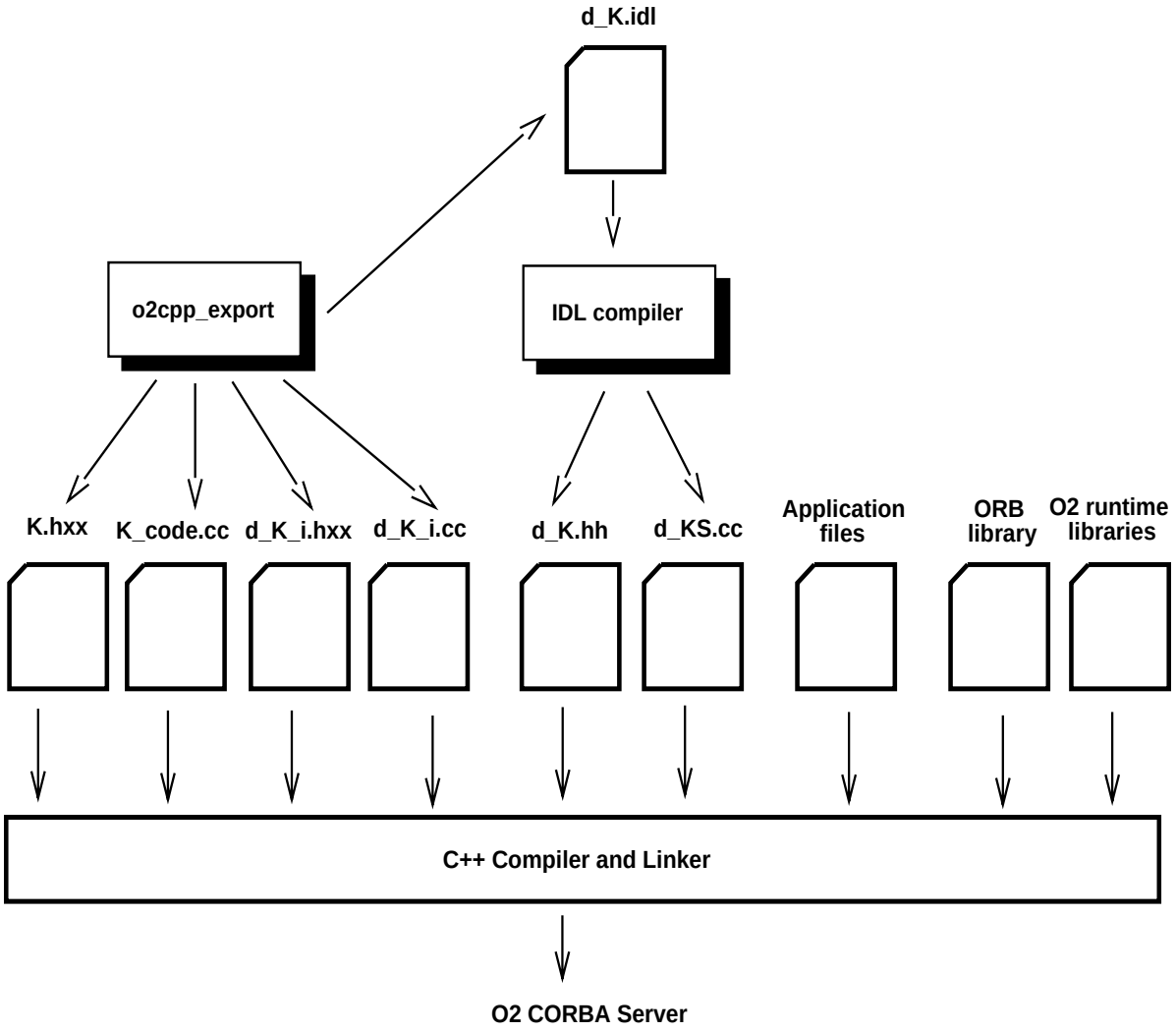


Figure 3.1: Building an O_2Corba server with O_2

3.4 O_2 to IDL mapping

An O_2 exported class is mapped into an IDL interface, the name of the interface is the name of the O_2 class prefixed by "**d_**".

For each exported (public) attribute, a corresponding IDL attribute is generated with the same type.

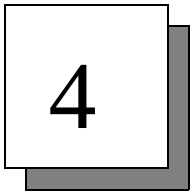
For each exported method, a corresponding IDL operation is generated with the same name and the same signature. All the parameters will be declared using the **in** parameter passing mode, **out** parameters will be managed by return values.

The following table, "Corresponding O2 and IDL types for export," shows the O_2 types that you can use as parameter types and result type for exported O_2 methods and as attribute type for exported attributes. The corresponding IDL types are also given. **Q** is taken as any O_2 type shown, and **P** its corresponding IDL type.

Table 3.1 ***Corresponding O_2 and IDL types for export***

O2 Type	IDL Type
no type	void
integer	long
real	double
Boolean	Boolean
char	char
string	string
K (O_2 class)	d_K (IDL interface)
02_list_Q (O_2 collection class)	d_List_P (IDL collection interface)
02_setQ (O2 collection class)	d_Set_P (IDL collection interface)
O2_bag_Q (O2 collection class)	d_Bag_P (IDL collection interface)

3 Building an O2Corba Server



Programming

This chapter is divided into the following sections :

4 Programming

4.1 Introduction

This chapter explains how to program a CORBA client connected to an O_2Corba server. It presents how to create and access persistent objects, how to define transactions, how to use collections and OQL and how the memory is managed. We use Orbix as an example of an ORB. Its binding mechanism, using markers, may be replaced by the use of a name service on other ORBs.

4.2 Structure of an O_2Corba server

An O_2Corba server is an O_2 ODMG C++ application (an O_2 client) which is linked with the ORB server library. Therefore an object of class **`o2_Database_i`** must be created in order to establish a connection to O_2 and to be able to open a database on the client side. The server then runs under the control of the O_2 transaction manager (**`d_Transaction::begin`**, **`d_Transaction::commit`**, etc).

The following example shows a server program in which an object of class **`d_Grid`** will run. The CORBA server has been registered with the name **`d_Grid`**

```
int main(int argc, char* argv[]) {

    o2_Database_i* database_i = new  o2_Database_i("d_Grid", argc, argv);

    d_Grid_Factory_i* grid_i = new d_Grid_Factory_i("Grid");

    // tell Orbix that we have completed the server's initialisation:
    CORBA::Orbix.impl_is_ready("d_Grid");
    cout << "Server terminating" << endl;

    session.end();

    return 0;
}
```

The declaration of the **`o2_Database_i`** constructor is:

```
o2_Database_i::o2_Database_i( const char* name,
                              int argc,
                              char* argv,
                              CORBA::Environment &env);
```

where **name** is a marker name.

Factories must also be created before connection to the CORBA server.

4.3 Structure of a CORBA client

A client deals with Database and Transaction through the mirror classes **o2_Database** and **o2_Transaction**, images of the ODMG classes **d_Database** and **d_Transaction** (see Transaction service and Database management in this chapter).

Note

All these classes are provided by the O_2 ORB library **libo2orbc.a**

A CORBA client can be written as follows:

```
int main (int argc, char **argv) {
o2_Database* database;
    o2_Transaction* trans;
    d_Grid_Factory* gfact;
    d_Grid* g;
    long v;
    // first bind to the database object.
    // argv[1] is the hostname of the target d_Grid object;
    database = o2_Database::_bind("d_Grid:d_Grid", argv[1]);
    database->open("grid_b", read_write);
    trans = database->create_transaction();
    // bind to the d_Grid_Factory object
    gfact = d_Grid_Factory::_bind("Grid:d_Grid", argv[1]);
    // get the d_Grid object from its name "the_grid"
    g = gfact->lookup_Grid("the_grid");

    v = g->get(1, 2);
    cout << "g2 [1][2] is " << v << endl;
    trans->begin();
    g->set(1, 2, 15);
    trans->validate();
    database->close();
    // release proxies
    g->release_instance();
    trans->_release();
    database->_release();
    gfact->_release();
    g->_release();
    return 0;
}
```

A client will have first to get an **`o2_Database`** object using the Orbix binding mechanism in order to open a database and create transactions.

Then it can get an object using the associated **`Factory`** class and the binding mechanism. For example an object of class **`d_Grid`** will be retrieved using the **`d_Grid_Factory::lookup_Grid`** member function.

Object that are not returned through the Orbix binding mechanism must be released by the **`release_instance`** member function in order not to populate the server memory space.

4.4 Creating persistent objects

A persistent object of the class **d_Grid** can be created using the **d_Grid_Factory::create_persistent_Grid** member function. The first parameter of this function will be the name of the object[1].

The class **d_Grid_Factory** is automatically generated by the O_2 Export tool when the **d_Grid** class is exported (see Chapter 3).

Example:

```
d_Grid* g;
d_Grid_Factory* gfact;
...
gfact = d_Grid_Factory::_bind("Grid:d_Grid", hostname);
g = gfact->create_persistent_Grid("my_grid");
```

4.5 Accessing persistent objects

A persistent object of the class **d_Grid** can be accessed by name using the **d_Grid_Factory::lookup_Grid** member function. The first parameter of this function will be the name of the object.

Example:

```
d_Grid* g;
d_Grid_Factory* gfact;
...
gfact = d_Grid_Factory::_bind("Grid:d_Grid", hostname);
g = gfact->lookup_Grid("the_grid");
```

An object can also be accessed by name through the **o2_Database::lookup_object** member function, in this case the returned object is typed as **any** and must be converted to the right class.

1. The semantics of names corresponds to the ODMG semantic

```
o2_Database* database;
CORBA::any* a;
d_Grid* g;
...
a = database->lookup_object("the_grid", "Grid");
if ((*a >>= g) != 0) {
   cerr << "ref any did not return a grid" << endl;
}
```

4.6 Collection management

Unlike C++, IDL does not provide generic parametric types nor method overloading. This implies that we cannot provide an IDL interface for List<T>. We thus have to define an IDL class for each needed typed collection.

For instance, if we want a List of K, we will have the C++ class **d_List_K**, which implements the same interface as the ODMG class **d_List,** enhanced with some operations allowing to manipulate sequences.

In addition, the **d_List** interface inherits the **Collection** interface of the OMG Query Service.

For example, the interface **d_List_Grid** will be defined as follows:

Collection management :

```
module CosQueryCollection {
   exception ElementInvalid{};
   exception PositionInvalid{};
   exception IteratorInvalid{};

   typedef string  Istring;
   struct NVPair {Istring name; any value;};
   typedef sequence<NVPair> ParameterList;

   interface Iterator {
       boolean more();
       any next() raises (IteratorInvalid, PositionInvalid);
       void reset();
   };

   interface Collection {
       readonly attribute long cardinality;

       void add_element(in any element) raises (ElementInvalid);
       void add_all_elements(in Collection elements)
           raises (ElementInvalid);
       void insert_element_at(in any element, in Iterator where)
           raises (PositionInvalid, IteratorInvalid,  ElementInvalid);
       void replace_element_at(in any element, in Iterator where)
           raises (PositionInvalid, IteratorInvalid,  ElementInvalid);
       void remove_element_at(in Iterator where)
           raises (PositionInvalid, IteratorInvalid);
       void remove_all_elements() raises (ElementInvalid);
       any retrieve_element_at(in Iterator where)
           raises (PositionInvalid, IteratorInvalid);
       Iterator create_iterator();
   };

};
```

```
interface odmg_Collection : CosQueryCollection::Collection {
    void release_instance();
    long is_empty();
    long is_ordered();
    long allows_duplicates();
    void remove_all();
};

interface d_Grid;
interface d_Iter_List_Grid;
interface d_List_Grid;

typedef sequence<d_Grid> d_Sequence_Grid;

interface d_List_Grid : odmg_Collection {
    readonly attribute d_Sequence_Grid collection_value;
    void display();
    long contains_element(in d_Grid element);
    void insert_element(in d_Grid element);
    void remove_element(in d_Grid element);
    d_Grid select_element(in string predicate);
    long  query(out d_List_Grid l, in string predicate);
    long  sequence_query(out d_Sequence_Grid l, in string predicate);
    d_Iter_List_Grid create_d_Iterator();
    d_Iter_List_Grid select(in string predicate);
    d_Grid retrieve_first_element();
    d_Grid  retrieve_last_element();
    void remove_first_element();
    void remove_last_element();
    long find_element(in d_Grid element, inout unsigned long position) ;
    d_Grid retrieve_element_at_pos(in unsigned long position) ;
    void remove_element_at_pos(in unsigned long position);
    void replace_element_at_pos(in d_Grid element, in unsigned long
position);
    void insert_element_first(in d_Grid element);
    void insert_element_last(in d_Grid element);
    void insert_element_after(in d_Grid element, in unsigned long position);
   void insert_element_before(in d_Grid element, in unsigned long position);
    d_List_Grid concat (in  d_List_Grid l);
    d_List_Grid append (in  d_List_Grid l);
    d_Sequence_Grid concat_list (in  d_List_Grid l);
    d_Sequence_Grid append_list (in  d_List_Grid l);
    d_Sequence_Grid concat_sequence (in  d_Sequence_Grid l);
    d_Sequence_Grid append_sequence (in  d_Sequence_Grid l);
};
```

The same technique as persistent object is available to create and access a collection, i.e. through an auxilliary interface **`d_List_K_Factory`**.

The following example shows how to use a list of objects of the class **d_Grid**:

```
d_Grid *g1;
d_Grid *g2;
d_List_Grid* lg1;
d_List_Grid* lg2;
d_List_Grid_Factory *lfact;
...
trans->begin();
lfact = d_List_Grid_Factory::_bind("List_Grid:d_Grid", hostname);
lg1 = lfact->create_persistent_list_Grid("the_list_grid");
lg1->insert_element(g1);
long res =  lg1->contains_element(g1);
cout << "lg1 contains g1 :" << res << endl;

lg1->insert_element(g2);
lg1->find_element(g2, res);
cout << "pos of g2 " << res << endl;

lg2 = lfact->create_list_Grid();
lg1->append(lg2);

trans->validate();

g1->release_instance();
g2->release_instance();
lg1->release_instance();
g1->_release();
g2->_release();
lg1->_release();
lg2->_release();
```

Note

The classes **d_List_Grid** and **d_List_Grid_Factory** are generated by the IDL compiler from the IDL interfaces **d_List_Grid and d_List_Grid_Factory**. You have to instanciate the template file **d_List_Ref.idl** with the class **Grid** in order to produce the file **d_List_Grid.idl**. This file will be used as input to the IDL compiler.

Important

You do not need to instanciate templates for atomic collections (e.g. **d_List_long**, **d_List_string**), they are provided by the library **libo2orbc.a.**

Iterators

For a list of objects of the interface **d_Grid**, an interface **d_Iter_List_Grid** implements the behavior for iteration, with the same interface as the ODMG class **d_Iterator.** In addition the **d_Iter_List** interface inherits the Iterator interface of the OMG query service.

```
#include "CosQueryCollection.idl"

interface d_Grid;

interface d_Iter_List_Grid : CosQueryCollection::Iterator {
    readonly attribute long not_done;
    void release_instance();
    void advance();
    d_Grid get_element();
    long get_next(out d_Grid element);
};
```

Example:

```
d_List_Grid* lg;
d_List_Grid_Factory *lfact;
d_Iter_List_Grid* itg;
long card;
...
lfact = d_List_Grid_Factory::_bind("List_Grid:d_Grid", hostname);
lg = lfact->lookup_list_Grid("the_list_grid");
itg = lg->create_d_Iterator();
card = lg->cardinality();
cout << "lg cardinality " << card << endl;
while(itg->not_done()) {
        g = itg->get_element();
        v = g->get(1, 2);
        cout << "g [1][2] is " << v << endl;
        itg->advance();
    }
    itg->reset();
```

Note

You have to instantiate the template file **d_Iter_List_Ref.idl** with the class Grid in order to produce the file **d_Iter_List_Grid.idl**. This file will be used as input to the IDL compiler.

4.7 ODMG Services

Transaction service

A class **o2_Transaction**, image of the ODMG class **d_Transaction**, is available to manage the transaction service.

The class **o2_Transaction** is defined as follows:

```
class o2_Transaction: public virtual CORBA::Object {
public:
     ...
   virtual void begin (CORBA::Environment &IT_env=
                                CORBA::default_environment);
   virtual void commit (CORBA::Environment &IT_env=
                                CORBA::default_environment);
   virtual void validate (CORBA::Environment &IT_env=
                                CORBA::default_environment);
   virtual void abort (CORBA::Environment &IT_env=
                                CORBA::default_environment);
   virtual void checkpoint (CORBA::Environment &IT_env=
                                CORBA::default_environment);
   virtual void promote (CORBA::Environment &IT_env=
                                CORBA::default_environment);
};
```

An object of the class **o2_Transaction** can be created using the **o2_Database::create_transaction** member function.

Example: :

```
o2_Transaction* trans;
 ...
 database->open("grid_b", read_write);
 trans = database->create_transaction();
 trans->begin();
 ...
 trans->commit();
```

Note

The class **o2_Transaction** is provided by the O_2 ORB library **libo2orbc.a**.

OQL service

To execute a query whose result is a List of **K**, we will use the C++ class **d_Query_List_K** which implements the **oql_execute** function.

For example, the class **d_Query_List_Grid** will be defined as follows:

```
class d_Query_List_Grid: public virtual o2_OQL_Query {
public:
   ...
   virtual d_List_Grid* oql_execute (const char * query,
                                     const parameterList& params,
                                     CORBA::Environment &IT_env=
                                             CORBA::default_environment);
};
```

Like for persistent objects and collection, an object of the class **d_Query_List_Grid** will be created through an auxilliary class **d_Query_List_Grid_Factory**.

```
class d_Query_List_Grid_Factory: public virtual CORBA::Object {
public:
   ...
   virtual void release_instance (CORBA::Environment &IT_env=
                                                 CORBA::default_environment);
   virtual
    d_Query_List_Grid* create_query_evaluator (CORBA::Environment &IT_env=
                                                 CORBA::default_environment);
};
```

Example:

```
d_Query_List_Grid_Factory * qlfact;
d_Query_List_Grid * qlg;
d_List_Grid* lg;
d_Iter_List_Grid* itg;
d_Grid* g;
long v;
parameterList params(2);
CORBA::any a;

    ...
qlfact = d_Query_List_Grid_Factory::_bind("Query_Grid:d_Grid", hostname);
qlg = qlfact->create_query_evaluator();

params.length(2);
v = 1;
a << v;
params[0] = a;
v = 2;
a << v;
params[1] = a;

   lg = qlg->oql_execute("sort x in the_list_grid by x->get($1, $2)", params);
    itg = lg->create_d_Iterator();
    while(itg->not_done()) {
        g = itg->get_element();
        v = g->get(1, 2);
        cout << "g [1][2] is " << v << endl;
        itg->advance();
}
itg->reset();
```

Note

You have to instantiate the template file **d_Query_List_ref.idl** with the class Grid in order to produce the file **d_Query_List_Grid.idl**. This file will be used as input to the IDL compiler.

Database management

A class **o2_Database** provides an interface to the ODMG class **d_Database**.

The class **o2_Database** is defined as follows :

```
class o2_Database: public virtual CORBA::Object {
public:
virtual void release_instance (CORBA::Environment &IT_env=
                                          CORBA::default_environment);
virtual void create (const char * database_name,
                     const char * schema_name,
                    CORBA::Environment &IT_env=CORBA::default_environment);
virtual void create_on_volume (const char * database_name,
                                  const char * schema_name,
                                  const char * volume_name,
                                  long size,
                                  long factor,
                                  CORBA::Environment &IT_env=
                                         CORBA::default_environment);
virtual void destroy (const char * database_name,
                    CORBA::Environment &IT_env=CORBA::default_environment);
virtual void open (const char * database_name,
                  access_rights status,
                    CORBA::Environment &IT_env=CORBA::default_environment);
virtual void close (CORBA::Environment &IT_env=CORBA::default_environment);
virtual void garbage (const char * database_name,
                    CORBA::Environment &IT_env=CORBA::default_environment);
virtual void set_object_name (CORBA::Any* the_object,
                                 const char * the_name,
                                 const char * old_name,
                                 CORBA::Environment &IT_env=
                                         CORBA::default_environment);
virtual char * get_object_name (CORBA::Any* the_object,
                                   CORBA::Environment &IT_env=
                                         CORBA::default_environment);
virtual void rename_object (const char * old_name,
                               const char * new_name,
                               CORBA::Environment &IT_env=
                                         CORBA::default_environment);
virtual o2_Transaction* create_transaction (CORBA::Environment &IT_env=
                                             CORBA::default_environment);
virtual CORBA::Any* lookup_object (const char * the_name,
                                      const char * factory_name,
                                      CORBA::Environment &IT_env=
                                               CORBA::default_environment);
virtual void create_persistent_root (const char * root_name,
                                        const char * class_name,
                                        long mode,
                                         CORBA::Environment &IT_env=
                                               CORBA::default_environment);
virtual void destroy_persistent_root (const char * root_name,
                                         CORBA::Environment &IT_env=
                                                CORBA::default_environment;
```

```
virtual void extend (const char * database_name,
                        const char * volume_name,
                    CORBA::Environment &IT_env=CORBA::default_environment);
virtual void set_current (o2_Database* cur,
                                 CORBA::Environment &IT_env=
                                                    CORBA::default_environment);
virtual void set_default_vol (const char * volume_name,
                                 CORBA::Environment &IT_env=
                                                    CORBA::default_environment);
virtual char * get_default_vol (CORBA::Environment &IT_env=
                                                    CORBA::default_environment);
virtual o2_Database* get_current (CORBA::Environment &IT_env=
                                                    CORBA::default_environment);
virtual void disconnect (CORBA::Environment &IT_env=
                                                    CORBA::default_environment);
};
```

The disconnect operation shuts down the server process. It is only effective in the scope of the Chorus ORB.

Example:

```
o2_Database* database;
CORBA::Any* ref;
o2_Transaction* trans;
...
database = o2_Database::_bind("d_Grid:d_Grid", hostname);
database->open("grid_b", read_write);
trans = database->create_transaction();
ref = database->lookup_object("the_grid", "Grid");
trans = database->create_transaction();
```

Note

The class **`o2_Database`** is provided by the O_2 ORB library **`libo2orbc.a`**.

4.8 Memory management

As explained in Section 4.1, the connection between the CORBA class implementation (**K_i**) and the O_2 class (**K_p**) is performed through the attribute **o2_pointer** of type **d_Ref<K_p>**.

On the server side, when a CORBA implementation method must return a pointer to **K**, a **K_i** object must be created to carry the **o2_pointer**.

As it is likely that there will be many accesses to the **same K** object, it would be prohibitive to create as many proxies **K_i** as the number of accesses to be done. Therefore, we ensure that a single proxy **K_i** is created for a corresponding **K_p**.

When a method must return a **Q***, O_2 Corba checks whether the **Q_i** already exists for the corresponding **d_Ref**. If it exists, it just returns it, otherwise a new **Q_i** is created and fixed in memory.

Releasing proxies

Objects retrieved by the CORBA binding mechanism should not be released because further bind calls would fail. Objects created by factories or returned by method calls should be released when they are no longer used.

As long as a **K_i** object refers to a **K_p** object, the **K_p** object cannot be released by O_2. It is up to the application to free a **K_i** object when the CORBA client (the application) does not need the corresponding **K** object any longer.

To make this possible, we will provide in each **K_i** class (and thus in each IDL class too) a method named *release_instance* that the client can call explicitely to release the CORBA object on the server.

In any case the method release_instance should be called by the application as early as possible in order to minimize memory consumption. As soon as a client does not need an object any longer, it is a good practice then to release both proxies on the client and on the server. To do this the client just writes the following.

```
d_Grid* g;
...
// assuming g has been set, then used and finally g is no more needed:
g->release_instance(); // Release the server proxy
g->_release(); //Release the client proxy
```

Commit and abort

All CORBA objects must be deleted at commit and abort time, since O_2 objects are no more valid in memory.

4 Programming

5 Example of an O_2 Corba Application on top of Orbix

This chapter is divided into the following sections :

5 Example of an O2 Corba Application on top of

5.1 Introduction

The example that we develop in this Section corresponds to the one described in the Orbix documentation. This example is a two-dimensional grid, it is extended by some additional operations in order to illustrate the use of ODMG collections and OQL via Orbix. Refer to the Orbix programmer's guide for more details about this example.

This application is an O_2 ODMG C++ application and it is linked with the Orbix server library in order to provide an O_2Corba server.

A simple client will also be shown.

To run the **`grid`** example you have to go through the following steps which are the remaining sections of this chapter :

- ***Define the class Grid in file Grid.hxx***
- ***Implement the class Grid in file Grid.cc***
- ***Implement the O2Corba server in file srv_main.cc***
- ***Implement the CORBA client in Client.cc***
- ***Compile the O2Corba server***
- ***Compile the client***
- ***Running***

5.2 Define the class Grid in file Grid.hxx

Define the C++ classes **Grid** and **Row** in the file **Grid.hxx.** This file will be used as input to the **o2cpp_import** command.

```
class Row {
          public:
             d_List<int> l;
             Row(){};
};

class Grid {
protected:
          short m_height;  // store the height
          short m_width;   // store the width
          d_List<d_Ref<Row> > m_a;
public:
          // constructors
          Grid();
Grid(const short h, const short w);

// destructor
virtual ~Grid();

// functions corresponding to the IDL operations
virtual int width() const;
virtual int height() const;
virtual void  set(const int n, const int m, const int value);
virtual int get(const int n, const int m) const;
virtual d_List<int> get_row(const int range);
virtual void set_row(d_List<int> l, const int range);
virtual void set_grid(d_Ref<Grid> g);
virtual void initialize (int h, int w);
};
```

5.3 Implement the class Grid in file Grid.cc

The following gives the C++ code which implements the class **Grid**. This is in file **Grid.cc**

```
#include "Grid.hxx"

// ctor
Grid::Grid(const short h, const short w) {
    m_height=h;   // set up height
    m_width=w;    // set up width
    for (int i = 0; i < h; i++ ) {
        d_Ref<Row> r = new Row;
        for (int j = 0; j < w; j++)
            r->l.insert_element_last(j);
        m_a.insert_element_last(r);
    }
}

Grid::Grid() {
    m_height=10;   // set up height
    m_width=10;    // set up width
    for (int i = 0; i < 10; i++ ) {
        d_Ref<Row> r = new Row;
        for (int j = 0; j < 10; j++)
            r->l.insert_element_last(j);
        m_a.insert_element_last(r);
    }
}

// dtor
Grid::~Grid() {
    // then free the overall array:
  //  delete[] m_a;
}

void  Grid::initialize (int h, int w){
    m_height=h;   // set up height
    m_width=w;    // set up width
    m_a.remove_all();
    for (int i = 0; i < h; i++ ) {
        d_Ref<Row> r = new Row;
        m_a.insert_element_last(r);
        for (int j = 0; j < w; j++)
            m_a[i]->l.insert_element_last(j);
    }
}
```

Implement the class Grid in file Grid.cc :

```
// implementation of the function which reads the height attribute
int Grid::height() const {
    return m_height;
}

// implementation of the function which reads the width attribute
int Grid::width() const {
    return m_width;
}

// implementation of the set operation:
void Grid::set(const int n, const int m, const int value) {
    m_a[n]->l.replace_element_at(value, m);
}

// implementation of the get operation:
int Grid::get(const int n, const int m) const {
    return m_a[n]->l[m];
}

// implementation of the get_row operation:
d_List<int> Grid::get_row(const int range) {
    d_List<int> res ("the_row");
    d_List<int> tmp;
    tmp.insert_element_first(m_a[range]->l[0]);
    for (int i =0; i < (m_height-1); i++)
        tmp.insert_element_after(m_a[(int)range]->l[i+1],i);
    res = tmp;
    return (res);
}

void Grid::set_row(d_List<int> l, const int range){
    for (int i =0; i < m_height; i++)
    m_a[(int)range]->l.insert_element_last(l[i]);
 }

void Grid::set_grid(d_Ref<Grid> g) {
    this->m_height = g->height();
    this->m_width = g->width();
}
```

5.4 Implement the O_2Corba server in file srv_main.cc

This section gives the main program which implements the O_2 Corba server. This program connects to the O_2 server by creating an **o2_Database_i** object, creates two *factory* objects and starts the CORBA server.

```
#include <stdlib.h>
#include <iostream.h>
#include "o2lib_CC.hxx"
#include "Grid.hxx"
#include "o2_Database_i.hxx"
#include "o2_Factory_i.hxx"
#include "d_Grid_i.hxx"
#include "d_List_Grid_i.hxx"

int main(int argc, char* argv[]) {
    o2_Database_i * database_i;
    d_Grid_Factory_i* grid_i;
    d_List_Grid_Factory_i * lgrid_i;

    database_i = new o2_Database_i("d_Grid", argc, argv);

    grid_i = new d_Grid_Factory_i("Grid");
    lgrid_i = new  d_List_Grid_Factory_i("List_Grid");

    try {
        // tell Orbix that we have completed the server's initialisation:
        CORBA::Orbix.impl_is_ready("d_Grid");
    }
    catch(const CORBA::SystemException& se) {
       // an error occured calling impl_is_ready() - output the error.
       cerr << "Unexpected exception " << endl << &se;
    }
    cout << "server exiting" << endl;
    return 0;
}
```

5.5 Implement the CORBA client in Client.cc

This section gives the main program which implements a CORBA client.

```
#include <stream.h>
#include <stdlib.h>
#include "d_Grid.hh"
#include "o2_Ref_Any.hh"
#include "o2_Session.hh"
#include "o2_Database.hh"
#include "o2_Transaction.hh"

int main (int argc, char **argv) {
  o2_Database_var database;
  o2_Transaction_var trans;
  d_Grid_var g;
  d_Grid_Factory_var gfact;
  d_List_long_var l;
  CORBA::Short h, w;
  CORBA::Long v;
  if (argc < 2) {
     cout << "usage: " << argv[0] << " <hostname>" << endl;
     exit (-1);
  }
  cout << "CORBA client is running" << endl;

  try {
   database = o2_Database::_bind("d_Grid:d_Grid", argv[1]);
   database->open("grid_b", read_write);
   cout << "Client has opened base grid_b" << endl;
   }
      catch(const CORBA::SystemException& se) {
         // an error occurred while trying to bind to the database object.
         cerr << "open base failed" << endl;
         cerr << "Unexpected exception " << endl << &se;
         exit(1);
   }
   try {
      trans = database->create_transaction();
      trans->begin();
      // First retrieves the grid object.
      gfact = d_Grid_Factory::_bind("Grid:d_Grid", argv[1]);
      g = fact->lookup_Grid("the_grid");
      g->initialize(10,10);
   }
      catch(const CORBA::SystemException& se) {
      // an error occurred while trying to bind to the factory object.

                                                                  //continue...
```

```
  cerr << "Bind to factory or lookup the_grid failed" << endl;
  cerr << "Unexpected exception " << endl << &se;
  exit(1);
}

try {
  // try to read the height and width of the grid:
  h = g->height();
  w = g->width();
}

catch(const CORBA::SystemException& se) {
  //an error occurred while trying to read the height and width:
  cerr << "call to height or width failed" << end1;
  cerr << "Unexpected exception" << end1
    <<&se;
  end(1);
}

  // no problem reading the height and width:
  cout << "height is " << h << endl;
  cout << "width  is " << w << endl;

try {
  // try to set element [2,4] of the grid - to value 123
  g->set(2, 4, 123);
  // then read back what we have just set:
  l = g->get_row(2);
  v = l->retrieve_element_at(4);
}
catch(const CORBA::SystemException& se) {
  // an error occurred while calling set, get or retrieve_element_at:
  cerr << "Call to set, get or retrieve_element_at failed" << endl;
  cerr << "Unexpected exception "  << endl << &se;
  exit(1);
}

// no problem setting and getting the elememt:
cout << "grid[2,4] is " << v << endl;

// make sure we got the value 123 back:
  if ( v != 123 ) {
cerr << "something went seriously wrong" << endl;
exit(1);
}
try {
g->release_instance();
l->release_instance();
trans->commit();                                          //continue...
```

```
  trans->release_instance();
  }
  catch(const CORBA::SystemException & se {
    //an error occurred while releasing instances:
    cerr << "Call to release_instance failed" << end1;
    cerr << "Unexpected exception " << end1 << &se;
    exit (1);
  }
  database->close();
return 0;
}
```

5.6 Compile the O_2Corba server

- ***Through o2dsa_shell, create a schema grid_s:***

```
create schema grid_s;
```

- ***Create a configuration file Grid.cf***

For this application, you have two source files `srv_main.cc` and `Grid.cc`. This application uses a list of Grid and an iterator on it, so you have to add to the Sources list the appropriate files (e.g. **`d_List_Grid_i.cc`**). These files are obtained by instantiation of the delivered template files (e.g. **`d_List_ref_i.cc`**).

You need to import the classes **`Grid`** and **`Row`** and three collections **`list(integer)`**, **`list(Row)`** and **`list(Grid)`**. All the public member functions of the class **`Grid`** are imported.

You also need to generate IDL interfaces for the class **`Grid`** using the **`o2cpp_export`** tool. All the methods which will be used from the CORBA client must be exported.

Finally the final executable must be linked with the Orbix server library **`liborbix.so`** and the O_2 library **`libo2orbs.a`** in order to produce an O_2Corba server (**`grid_server`**).

5 Example of an O2 Corba Application on top of

```
# configuration file: grid.cf
O2Home: $O2HOME
O2System: $O2SYSTEM
O2Server: $O2SERVER
O2Schema: grid_s

ImpFiles: Grid.hxx
[Grid.hxx]ImpClasses:  Row Grid
[Grid.hxx][Grid]ImpAllPublicMemberFunc
[Grid.hxx]ImpOutputDir: out
ImpList: int Row Grid;
ExpClasses = Grid
+[Grid]ExpIdl
-[Grid]ExpHeader
-[Grid]ExpCode
 [Grid]ExpMethods = width height set get get_row set_row set_grid initialize

ProgramName: grid_server
Sources: srv_main.cc Grid.cc  d_List_Grid_i.cc
            d_Iter_List_Grid_i.cc d_List_GridS.cc d_Iter_List_GridS.cc
ProgramObjs: srv_main.o  Grid.o  d_List_Grid_i.o
                d_Iter_List_Grid_i.o  d_List_GridS.o  d_Iter_List_GridS.o
ProgramLibDir:  <ORB_INSTALLATION_PATH>/lib
ProgramLib: C orbix o2orbs

Define:  _REENTRANT
Include:  <ORB_INSTALLATION_PATH>/include
UserLdFlags = -mt
+UseOql
+UseConfirmClasses
```

- ***Generate a Makefile***

```
o2makegen grid.cf
```

Do not forget to set the environment variables O2HOME, O2SERVER and O2SYSTEM and to change the installation path of the CORBA ORB product in the configuration file.

- ***Import the C++ classes***

```
make import
```

Do not forget to run the O_2 server before.

Compile the O2Corba server :

- ***Generate IDL files***

```
make export
```

Do not forget to run the O_2 server before.

- ***Call the IDL compiler on the generated IDL files***

```
idl -B -A d_Grid.idl
```

The options -B and -A must be used.

- ***Compile the O_2Corba server***

```
make
```

The following figure shows how to create the O_2Corba server for the **`Grid`** application.

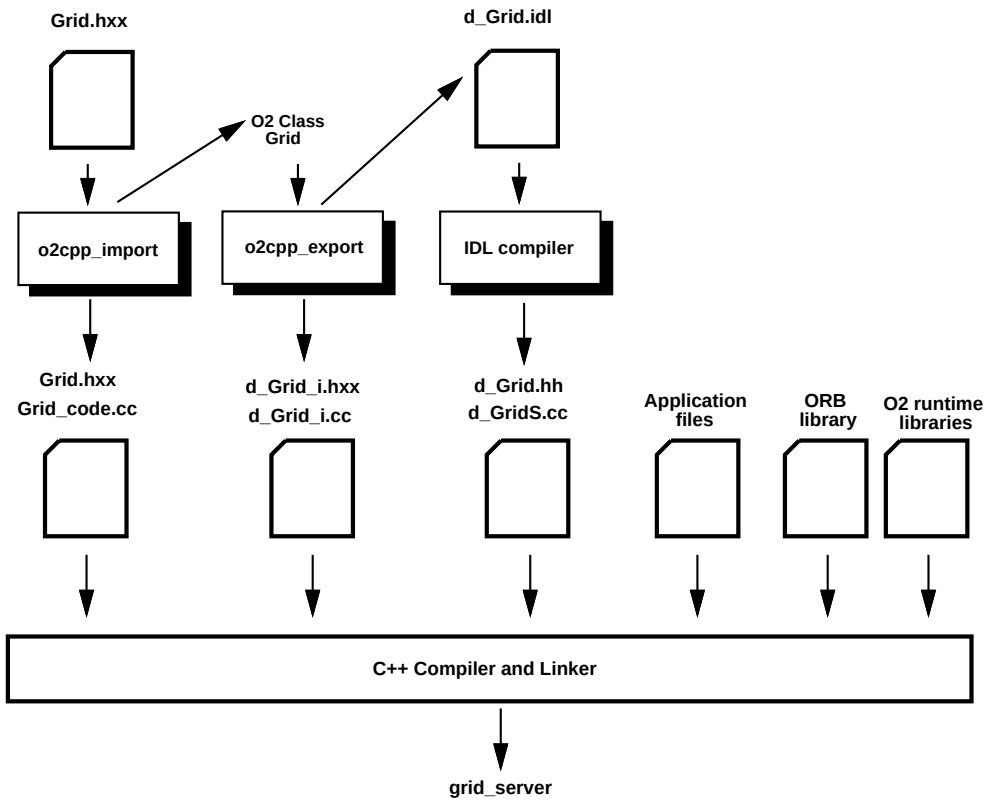


Figure 5.1: Building the Grid O_2Corba server

- ***Through o2dsa_shell, create a base grid_b***

```
create base grid_b;
```

- ***Register the CORBA server called*** `d_Grid` ***using the correct ORB program (such as putit for Orbix)***

```
putit d_Grid -h'<hostname>' <ORBIX_INSTALLATION_PATH>/grid/grid_server
```

5.7 Compile the client

There is no automatic way to generate a client Makefile.

Just follow the Makefile examples delivered with your ORB demos and add :

(1) the include directory of the o2corba distribution to include directives,

(2) the lib directory of the o2corba distribution and the o2orbc library to the link directives.

5.8 Running

- **Start the O_2Corba server**

```
grid_server &
```

- **Start the CORBA client**

```
client hostname
```

6 Example of an O_2 Corba Application on top of Chorus ORB

This chapter is divided into the following sections :

6 Example of an O2 Corba Application on top of

6.1 Introduction

The example that we develop in this section is a two dimensional grid that illustrates the use of ODMG Collections and OQL via Chorus ORB.

This application is an O_2 ODMG C++ application linked with the Chorus ORB Library in order to provide an O_2Corba server.

A simple client will also be shown.

To run the **`grid`** example you have to go through the following steps which are the remaining sections of this chapter :

- ***Define the class Grid in file Grid.hxx using COOL***
- ***Implement the class Grid in file Grid.cc***
- ***Implement the O2Corba server in file srv_main.cc***
- ***Implement the CORBA client in Client.cc***
- ***Compile the O2Corba server***
- ***Compile the client***
- ***Running***

6.2 Define the class Grid in file Grid.hxx using COOL

Define the C++ classes **Grid** and **Row** in the file **Grid.hxx.** This file will be used as input to the **o2cpp_import** command.

```
class Row {
public:
          d_List<int> 1;
          Row(){};
};

class Grid {
protected:
          short m_height;  // store the height
          short m_width;   // store the width
          d_List<d_Ref<Row> > m_a;
public:
          //constructors
          Grid();
          Grid(const short h, const short w);

          //destructor
          virtual ~Grid();

          //functions corresponding to the IDL operations
          virtual    int width() const;
          virtual    int height() const;
          virtual    void set(const int n, const int m, const int value);
          virtual    int get(const int n, const int m) const;
          virtual    d_List<int> get_row(const int range);
          virtual    void set_row(d_List<int> 1, const int range);
          virtual    void set_grid(d_Ref<Grid> g);
          virtual    void initialize (int h, int w);
};
```

6.3 Implement the class Grid in file Grid.cc

The following gives the C++ code which implements the class **Grid**. This is in file **Grid.cc**

```
// Class grid_o2 implements the grid IDL interface;

#include "Grid.hxx"

// ctor
Grid::Grid(const short h, const short w) {
    m_height=h;   // set up height
    m_width=w;    // set up width
    for (int i = 0; i < h; i++ ) {
        d_Ref<Row> r = new Row;
        for (int j = 0; j < w; j++)
            r->l.insert_element_last(j);
        m_a.insert_element_last(r);
    }
}

Grid::Grid() {
    m_height=10;   // set up height
    m_width=10;    // set up width
    for (int i = 0; i < 10; i++ ) {
        d_Ref<Row> r = new Row;
        for (int j = 0; j < 10; j++)
            r->l.insert_element_last(j);
        m_a.insert_element_last(r);
    }
}

// dtor
Grid::~Grid() {
    // then free the overall array:
  //  delete[] m_a;
}

void  Grid::initialize (int h, int w){
    m_height=h;   // set up height
    m_width=w;    // set up width
    m_a.remove_all();
    for (int i = 0; i < h; i++ ) {
        d_Ref<Row> r = new Row;
        m_a.insert_element_last(r);
        for (int j = 0; j < w; j++)
            m_a[i]->l.insert_element_last(j);
    }
}
```

Implement the class Grid in file Grid.cc

```
// implementation of the function which reads the height attribute
int Grid::height() const {
    return m_height;
}

// implementation of the function which reads the width attribute
int Grid::width() const {
    return m_width;
}

// implementation of the set operation:
void Grid::set(const int n, const int m, const int value) {
    m_a[n]->l.replace_element_at(value, m);
}

// implementation of the get operation:
int Grid::get(const int n, const int m) const {
    return m_a[n]->l[m];
}

// implementation of the get_row operation:
d_List<int> Grid::get_row(const int range) {
    d_List<int> res ("the_row");
    d_List<int> tmp;
    tmp.insert_element_first(m_a[range]->l[0]);
    for (int i =0; i < (m_height-1); i++)
        tmp.insert_element_after(m_a[(int)range]->l[i+1],i);
    res = tmp;
    return (res);
}

void Grid::set_row(d_List<int> l, const int range){
    for (int i =0; i < m_height; i++)
    m_a[(int)range]->l.insert_element_last(l[i]);
 }

void Grid::set_grid(d_Ref<Grid> g) {
    this->m_height = g->height();
    this->m_width = g->width();
}
```

6.4 Implement the O_2Corba server in file srv_main.cc

This section gives the main program which implements the O_2Corba server. This program connects to the O_2 server by creating an **o2_Database_i**, creates two *factory* objects and starts the CORBA server.

```
 // The server for the grid example.

#include <stdlib.h>
#include <iostream.h>

#include "o2lib_CC.hxx"
#include <Grid.hxx>
#include "svr_orb_port.hxx"
#include "o2_Session_i.hxx"
#include "o2_Database_i.hxx"
#include "o2_Factory_i.hxx"
#include "d_Grid_i.hxx"
#include "d_List_Grid_i.hxx"
#include "d_Query_List_Grid_i.hxx"

COOL_NamingService_ptr orb_ns;

d_Grid_Factory_i* myGrid;
d_List_Grid_Factory_i * lgrid;
d_Query_List_Grid_Factory_i * qgrid;

int main(int argc, char* argv[]) {
    CORBA(Environment) env;
    o2_Database_i * db;

    CORBA(ORB_ptr) orb = CORBA_ORB(init)(argc, argv, 0, env);

    if (env.exception()) {
    fprintf(stderr, "Impossible to initialize the ORB.\n");
    return 1;
    }
    CORBA(BOA_ptr) boa = orb->OA_init(argc, argv, 0, env);
    if (env.exception()) {
        fprintf(stderr, "Impossible to initialize the object adapter.\n");
        return 1;
    }
    // Get the COOL Naming Service object reference.
    //
    orb_ns = thisCapsule->naming_service(env);
    // instantiate the database object
                                                        // continue...
```

```
    db = new o2_Database_i("grid_b", argc, argv, env);
    myGrid = new d_Grid_Factory_i("Grid");
    lgrid = new  d_List_Grid_Factory_i( "List_Grid");
    qgrid = new  d_Query_List_Grid_Factory_i( "Query_Grid");
    // tell ORB that we have completed the server's initialisation:
    cout << "calling boa->run" << endl;
    boa->run();
    cout << "server exiting" << endl;
    db->release_instance();
    myGrid->release_instance();
    lgrid->release_instance();
    qgrid->release_instance();
    return(0);
}
```

6.5 Implement the CORBA client in Client.cc

This section gives the main program which implements a CORBA client.

```
// Client.cc

// A client for the grid example.
#include <api/api.H>
#include "d_Iter_List_long.H"
#include "d_List_long.H"
#include "d_Grid.H"
#include "o2_Database.H"
#include "o2_Transaction.H"
#include "d_Grid.H"
#include <stream.h>
#include <stdlib.h>
#include "orb_port.hxx"

int main (int argc, char **argv) {
    d_Grid * p;
    o2_Database * db;
    d_List_long * l;
    d_Grid_Factory * fact;
    CORBA_Short h, w;
    CORBA_Long v;
    o2_Transaction * trans;

    if (argc < 2) {
             cout << "usage: " << argv[0] << " <hostname>" << endl;
             exit (-1);
    }
    // Initialize COOL runtime.
    //
    CORBA(Environment) env;
    cout << "CORBA client is running" << endl;
    CORBA_ORB_ptr orb = CORBA_ORB_init(argc, argv, 0, env);
    if (env.exception()) {
    fprintf(stderr, "Impossible to initialize the ORB.\n");
    return 1;
    }
    //
    // Get the COOL Naming Service object reference.
    //
    COOL_NamingService_var naming = thisCapsule->naming_service(env);


                                                        // continue ...
```

```
    // Imports the first interface (must use method import).
    //
    CORBA_Object_ptr bind_obj;
    naming->import("grid_b", bind_obj, env);
    if (env.exception()) {
                        printf("Import failed\n");
                        return 1;
    }
    db = o2_Database::_narrow(bind_obj);
    db->open("grid_b", read_write);
    cout << "Client has opened base grid_b" << endl;
    trans = db->create_transaction();
    trans->begin();
    // First retrieves the grid object.
    CORBA_Object_ptr bind_fact;
    naming->import("Grid", bind_fact, env);
    fact = d_Grid_Factory::_narrow(bind_fact);
    p = fact->lookup_Grid("the_grid");
    p->initialize(10,10);
    // try to read the height and width of the grid:
    h = p->height();
    w = p->width();
    cout << "height is " << h << endl;
    cout << "width  is " << w << endl;
    // set element [2,4] of the grid - to value 123
    p->set(2, 4, 123);
    // then read back what we have just set:
    l = p->get_row(2);
    v = l->retrieve_element_at_pos(4, env);
    // no problem setting and getting the elememt:
    cout << "grid[2,4] is " << v << endl;

    // make sure we got the value 123 back:
    if ( v != 123 ) {
              cerr << "something went seriously wrong" << endl;
              exit(1);
    }
    // p->release_instance();
    l->release_instance();
    db->close();
    trans->commit();
    trans->release_instance();
    CORBA_release(trans);
    CORBA_release(db);
    CORBA_release(fact);
    CORBA_release(l);
    CORBA_release(p);
    return 0;
}
```

6.6 Compile the O_2Corba server

- ***Through o2dsa_shell, create a schema grid_s :***

```
create schema grid_s;
```

- ***Create a configuration file Grid.cf***

For this application, you have two source files **srv_main.cc** and **Grid.cc**. This application uses a list of Grid and an iterator on it, so you have to add to the Sources list the appropriate files (e.g. **d_List_Grid_i.cc**). These files are obtained by instantiation of the delivered template files (e.g. **d_List_ref_i.cc**).

You need to import the classes **Grid** and **Row** and three collections **list(integer)**, **list(Row)** and **list(Grid)**. All the public member functions of the class **Grid** are imported.

You also need to generate IDL interfaces for the class **Grid** using the **o2cpp_export** tool. All the methods which will be used from the CORBA client must be exported.

Finally the final executable must be linked with the COOL server library **libOrb-mt.so** and the O_2 library **libo2orbs.a** in order to produce an O_2Corba server (**grid_server**).

```
#configuration file:  grid.cf
O2Home: $O2HOME
O2System: $O2SYSTEM
O2Server: $O2SERVER
O2Schema: grid_s

ImpFiles: Grid.hxx
[Grid.hxx]ImpClasses:  Row Grid
[Grid.hxx][Grid]ImpAllPublicMemberFunc
[Grid.hxx]ImpOutputDir: out
ImpList: int Row Grid;
ExpClasses = Grid
+[Grid]ExpIdl
-[Grid]ExpHeader
-[Grid]ExpCode
 [Grid]ExpMethods = width height set get get_row set_row set_grid initialize

ProgramName: grid_server
Sources: srv_main.cc Grid.cc  d_List_Grid_i.cc
            d_Iter_List_Grid_i.cc d_List_Grid.cc d_Iter_List_Grid.cc
ProgramObjs: srv_main.o  Grid.o  d_List_Grid_i.o
                d_Iter_List_Grid_i.o  d_List_Grid.o  d_Iter_List_Grid.o
ProgramLibDir:  <ORB_INSTALLATION_PATH>/lib
ProgramLib: C Orb-mt o2orbs

Define:  _REENTRANT COOL_R_4 ORB_O2_TIE
Include:  <ORB_INSTALLATTION_PATH>/include
UserLdFlags = -mt
+UseOql
+UseConfirmClasses
```

- ***Generate a Makefile***

```
o2makegen grid.cf
```

Do not forget to set the environment variables O2HOME, O2SERVER and O2SYSTEM and to change the installation path of the CORBA ORB product in the configuration file.

- ***Import the C++ classes***

```
make import
```

Do not forget to run the O_2 server before.

- ***Generate IDL files***

```
make export
```

Do not forget to run the O_2 server before.

- ***Call the IDL compiler on the generated IDL files***

```
idl -B -A d_Grid.idl
```

The options -B and -A must be used.

- ***Compile the O_2Corba server***

```
make
```

The following figure shows how to create the O_2Corba server for the **`Grid`** application

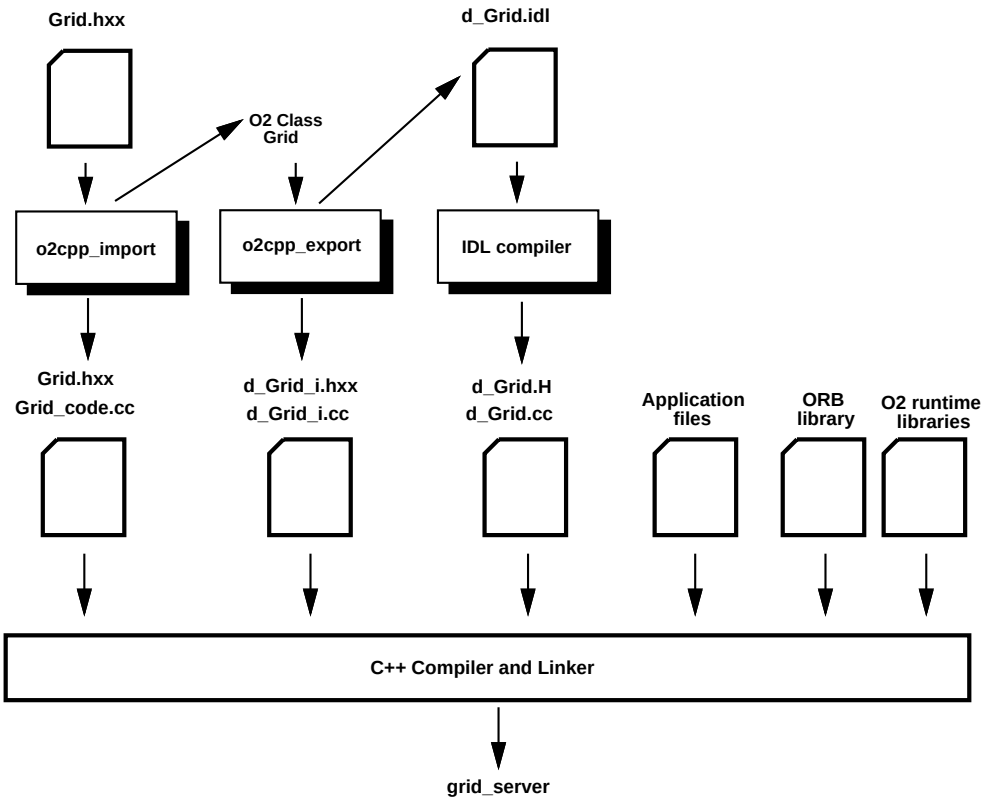


Figure 6.1: Building the Grid O_2Corba server

- ***Through o2dsa_shell, create a base grid_b***

```
create base grid_b;
```

- ***Register the CORBA server called*** **d_Grid** ***using the correct ORB program (such as putit for Orbix)***

```
putit d_Grid -h'<hostname>' <ORB_INSTALLATION_PATH>/grid/grid_server
```

6.7 Compile the client

There is no automatic way to generate a client Makefile.

Just follow the Makefile examples delivered with your ORB demos and add :

(1) the include directory of the o2corba distribution to include directives,

(2) the lib directory of the o2corba distribution and the o2orbc library to the link directives.

6.8 Running

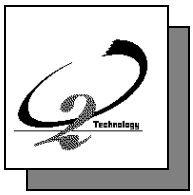
- **Start the O_2Corba server**

```
grid_server &
```

- **Start the CORBA client**

```
client
```

6 Example of an O2 Corba Application on top of



INDEX

Symbols

A

B

C

D

E

F

H

I

J

L

M

O

P

Q

R

S

T

U

V