UKC Orbix Survival Guide

I.C.A.Buckner & I.A.Penny
Computing Laboratory, University of Kent at Canterbury

May 6, 1997

1 Introduction

So you need to write an Orbix application but don't know where to start? This document will try to break you in as gently as possible by explaining the concepts you need to understand and then show you how Orbix implements them. If you read it in conjunction with the programmer's guide [Iona, 1995a] then you should get a good idea of what is going on. The concepts and much of the implementation details are generic but there are parts which are specific to UKC, and we will highlight these when we come to them.

2 Concepts

In order to understand the fundamentals of Orbix there are two concepts you must be familiar with: distributed systems and object orientation. There are numerous books on these subjects, so we will limit ourselves to a brief introduction.

2.1 Distributed systems

In a traditional, non-distributed, system, when you wish to solve a problem you write a program which starts, calculates the solution and then stops. It is highly likely that you divide your program into a number of sub-tasks, each implemented by a separate function, which, when executed in turn, perform your required task.

In a distributed system we build a solution to a particular problem by building a number of programs which when they communicate perform the required task. The programs can be distributed across a network of machines to take advantage of the resources of each one; this distribution should be transparent to the user. In theory it should not matter if the machine is in the next room or on the next continent, although there are obvious performance advantages in having the machines contained within a local network.

2.2 Object orientation

We are surrounded by objects in everyday life, some simple, others complex. Often complex objects are created from a collection of other, simpler, objects, thus hiding the complexity of the inner workings from us. For example a watch consists of a number of cogs, hands, and springs; we do not interact with these components we interact with the watch as an object in its own right.

In object oriented computing "the underlying concept is that one should model software systems as collections of cooperating objects" [Booch, 1994]. Server objects provide operations which may be invoked by client objects to elicit some change in state; Booch identifies five distinct operations :

- A **Modifier** alters the state of an object.
- A **Selector** accesses the state of an object, but does not alter the state.
- An **Iterator** permits parts of an object to be accessed in a well-defined order.
- A **Constructor** creates an object and initialises its state.
- A **Destructor** frees the object's state and destroys it.

The implementation of an operation within a server object is transparent to client objects. This division between implementation (definition) and interface (declaration) provides a clear boundary to the object which is associated via the interface.

3 Writing Orbix programs

Orbix is an implementation of the Common Object Request Broker Architecture (CORBA) [OMG, 1991]. It combines distributed systems with object orientation to provide a distributed programming environment in which a system can be constructed from a number of communicating objects. The communication mechanism is provided by the environment; the role of the programmer is to design and implement one or more server objects, and a client which invokes the server.

This section describes, in detail, the steps required to write a client and server using Orbix; it is supplemented by a checklist which can be used in your future projects. We shall examine the process by working through an example system, from interface description to compilation of the code. The code used in this example is available online in the directory `/proj/orbix/doc/survival/src`.

3.1 Problem description

"A retail system comprises a component which maintains a record of stock item serial numbers, prices, and descriptions. The serial number and price are of type long. The description is a character string. Operations are required to register an item, delete an item, find its price, change its price, find its description, and change its description."

3.2 Interface description

Before we actually write the code for the server object we must specify the interface which will be used by the client when it wishes to invoke server operations. To do this you must write an interface description language (IDL) file; this file is processed by Orbix's IDL compiler. The retail system, described previously, will use the following IDL file :

```
// In file retail.idl
interface stock {
        // Attributes
        readonly attribute long item_count;

        // Functions
        void newitem(in long serial, in long cost, in string name);
        void delitem(in long serial);

        long getprice(in long serial);
        void setprice(in long serial, in long cost);

        string getname(in long serial);
        void setname(in long serial, in string name);
};
```

We see that there are six functions, four of which return no result and the others return a long integer and a character string respectively. It is also possible to pass these results back as *out* parameters, but we prefer to return them as the result of the function.

3.3 Compiling idl

To compile the IDL file `retail.idl` you must type the following :

```
kestrel% idl -B -S retail.idl
```

The -B flag states that you wish to use the BOAImpl approach, see section 1.4.1 of the programmer's guide for details. The -S flag asks the idl compiler to drop skeleton code for the declaration and definition of the server object class; more on this later. Assuming all goes well the idl compiler will produce the following files:

- retail.hh : A header file common to both the client and the server.
- retail.ic : Skeleton definition (the implementation) of the object.
- retail.ih : Skeleton declaration of the object.
- retailC.cc : Orbix source code for compilation with the client.
- retailS.cc : Orbix source code for compilation with the server.

You must not edit `retail.hh`, `retailC.cc` or `retailS.cc`, merely include the header file in certain places and compile the source code with your own.

3.4 Object skeleton

Figure 1 shows how the classes generated, by compiling the IDL file, fit into the Orbix hierarchy. The diagram also shows how the classes will be used by client and server applications.

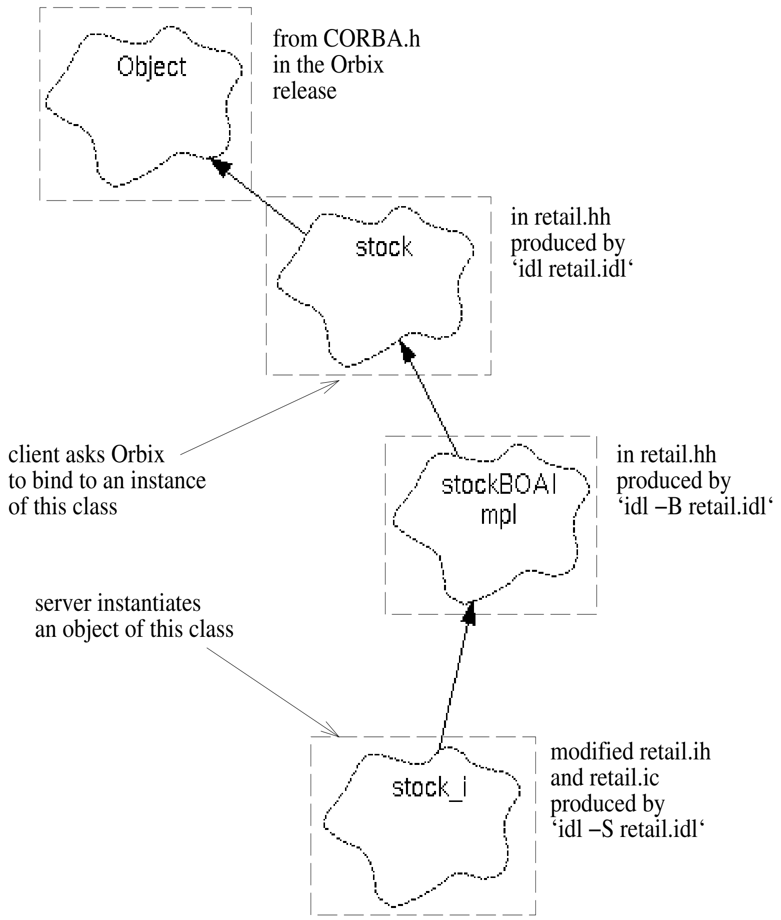

Figure 1: Classes created by IDL compilation

Once you have compiled your IDL file there are several changes you must make before you can compile the generated code. You must modify the object skeletons, declaration and definition, to produce the implementation class.

3.5 Object implementation

3.5.1 Object declaration

Turning the skeleton declaration into an implementation declaration is a simple process. It consists of the following steps:

- Copy the skeleton declaration (`retail.ih`) into a new implementation file `retail_i.h`. We will keep the skeleton version so that we can see, using `vdiff(1)`, the changes we have made.
- Edit the implementation file (`retail_i.h`) and change the name of the class from `stock==` to `stock_i`. The class `stock_i` is the implementation of the base class `stock`; see the inheritance diagram, figure 1, for details.
- Add the inheritance from the Basic Object Adapter class; this allows the server to be automatically started by Orbix if a client invokes a method in the object and currently no server object exists.
- Flesh out the skeleton to make it a complete declaration of the class. This includes declaring a constructor, destructor and the private members which will be used by the implementation. For the retail example we add the following :

```
// Includes deleted for brevity
class stock_i : public stockBOAImpl {
        typedef struct item {   // What we are storing
                long serial;
                long price;
                char *name;
        } item_t;
        item_t** catalogue;
        long count;             // Items in the catalogue
        long max;               // High water mark for allocation

public:
        stock_i(void);          // Constructor
        virtual ~stock_i(void); // Destructor

        // Fn prototypes deleted for brevity
}
```

Note that the idl compiler produces all the function prototypes for you, including those for the attributes. A readonly attribute such as `item_count` in our example IDL has a *selector* but no *modifier*; a read-write attribute has both. Prototypes are produced for attributes but no variable is assigned for them; we declare our own `long count` which corresponds to the `item_count` attribute.

3.5.2 Object definition

Now we have the declaration we can implement the class definition using the skeleton definition as a template.

- Copy the skeleton definition (`retail.ic`) into a new implementation file `retail_i.cc`; again we will keep the skeleton as a reference.
- Change the include statement at the beginning of the file to include your object declaration not the skeleton one.
- Edit the implementation file (`retail_i.cc`) and change all occurrences of `stock==` to `stock_i`.

- Fill in the function bodies. You will need to add the constructor and destructor as they do not appear in the skeleton definition.

The fourth step is the most crucial as it requires you to implement the behaviour of the object. The implementation of the example is excluded to restrict the length of this document; it can be found in the file `/proj/orbix/doc/survival/src/retail_i.cc` on kestrel.

3.6 Server implementation

Once the object has been implemented we must write a server which instantiates the object and declares to Orbix that it is ready to accept client invocations. The following code can be used as a template for your own programs.

```
// In file Srv_Main.cc
#include "retail_i.h"
#include <stream.h>

main()
{
        char *ident = "users/iap/retail";
        retail_i mystock;

        CORBA::Orbix.impl_is_ready(ident);
        cout << ident << " : server terminating" << endl;
}
```

The server program creates a `retail_i` object and then indicates to Orbix that the initialisation is complete by calling `impl_is_ready` on the `CORBA::Orbix` object; this object provides direct communication with Orbix. Note that we indicate which server is terminating; this is important as we do not know who else is using Orbix and we wish to differentiate our server from those of other users.

3.7 Client implementation

The client code will not follow a strict template; hence, we will give an overview of the important features.

- `#include "retail.hh"`

  You should include the common header file (`retail.hh`) and not the implementation declaration.

- `stock *p;`

  You should use a pointer to the base class (`stock`). An object will be instantiated when you bind to the server; you do not need to do it yourself.

- `p = stock::_bind(":users/iap/retail",argv[1],IT_X);`

  This statement binds your client's object reference to the server. It is important to check if any exceptions are raised using the exception handling mechanism:

  ```
  TRY {
          // Statement to execute
  } CATCHANY {
          // What to do if an exception occurs
  } ENDTRY
  ```

- `p->_release();`

  You must *not* call delete to destroy an Orbix object. Orbix provides a `_release` function in the base class to decrement the reference count for the object; if this count reaches zero Orbix will automatically delete the object.

Example client code can be found in the file `/proj/orbix/doc/survival/src/Client.cc` on kestrel.

3.8 Putting it all together

Once you have written your program you will need to compile all the separate modules and link them together to form the client and server. Fortunately, given a suitable Makefile, such as the one in `/proj/orbix/doc/survival/src/Makefile`, `make(1)` will do all the hard work. You will need to copy the Makefile into your development directory and make minor changes to it to reflect your filenames and your login.

Remember there are three places that the server's implementation repository entry should appear.

- In `Srv_Main.cc`, it is used as a parameter to the `impl_is_ready` function call.
- In `Client.cc` it is prefixed by a colon (:) and used as a parameter to the `_bind` function call.
- When you register the server with the implementation repository using the `putit` command.

Orbix will only be able to invoke the server and complete the binding between client and server if these entries exist and agree with one another.

4 The Orbix environment

4.1 Unix requirements

In order to use Orbix you must have the following entries in your environment; they are included in the default environment on kestrel.

Manual pages	$MANPATH	/opt/Orbix/man
Executables	$PATH	/opt/Orbix/bin
Libraries	$LD_LIBRARY_PATH	/opt/Orbix/lib

Table 1: Unix environment requirements

4.2 The log file

Here at UKC the Orbix daemon logs all activity to the file `/proj/orbix/logs/orbixd`. This log is cycled at the beginning of every week; last week's logs can be found in `orbixd.0` and the previous week's in `orbixd.1`. If your server is started automatically, ie. it is in the implementation repository, any output to the standard output or error streams will appear in the logfile. Output to the logfile should be kept to a minimum. If you need to debug your server then start it manually and remove the output statements once you are satisfied it works.

4.3 The implementation repository

We have now written a client and a server program but how does the client know where the service is? The Implementation Repository keeps a register of which services are available in our distributed system; it holds the name of each service, along with the path to the server executable that implements the service and any access permissions associated with it. When a client binds to a service, Orbix uses the Implementation Repository to locate a suitable server and starts it running if necessary (see section 1.3.3 in the Programmers Guide [Iona, 1995a] for a more detailed description).

The Implementation Repository's directory of services is structured much like the Unix filing system. At UKC, each Orbix user has their own *service context* in which they can register services; your home context will be `users/<login>` [1]. The contexts have ownerships associated with them so initially other people can't use your services.

The repository is manipulated by commands which are analogous to a number of Unix file system commands, here we show a common use of each command and explain what it does. There are a number of commands to manipulate services:

[1] If you don't have a home context then contact the Corba administrator by mailing `corba@ukc.ac.uk`

- `kestrel% putit users/iap/retail /home/cur/iap/stock/server`
  Informs the Implementation Repository of the association between a server name and the executable used to launch it.
- `kestrel% rmit users/iap/retail`
  Removes the server entry specified from the repository.
- `kestrel% catit users/iap/retail`
  Shows the current state of the repository entry specified.
- `kestrel% killit users/iap/retail`
  Kills the running server associated with the service specified.

and several commands to manipulate contexts:

- `kestrel% lsit users/iap`
  Shows the contents of the specified context.
- `kestrel% mkdirit users/iap/test`
  Creates a new context. This will only work below your home context.
- `kestrel% rmdirit users/iap/test`
  Removes the specified context.

Each of the above commands will accept the -h option, followed by a host name to manipulate the Implementation Repository on that host (if there is one). You can read more about utility commands in section 9.3 of the Programmers Guide [Iona, 1995a], as well as in the online `man` pages.

4.4 Simplifying the build

We can use a system called `imake(1)` to build our application. The imake system consists of a set of macro function definitions that describe how to create a more complex Makefile; you write an Imakefile which makes calls to the macros, and this is then processed to produce a Makefile. The idea is that using imake helps write portable code - the back end of the macro functions is customised for each platform while the Imakefile remains unchanged.

A template Imakefile can be found in `/proj/orbix/templates/Imakefile` on kestrel. Take a copy of this and modify it for your applications. To write an Imakefile for an Orbix application you need to know which functions are available:

- `ServerProgram(program,objects,libs)` links the object files specified by *objects* with the libraries specified by *libs* and the Orbix server library to produce an executable called *program*. Use this to build an Orbix server or a client which also acts as a server.
- `ClientProgram(program,objects,libs)`. As above, but links with the Orbix client library.
- `IDLDepend(idlname)` creates a set of dependencies for the specified IDL file.
- `IDLCleanList(list)` takes a list of IDL files and makes sure that all files generated by these are removed by `make clean`.

These variables must appear in your Imakefile, even if they are not set:

- `IDLFLAGS`. Any flags to pass to the IDL compiler; you will usually set this to -B to produce BOAImpl classes.

- C++FLAGS. Flags for the C++ compiler. This must be set to at least $(STANDARD_C++FLAGS). A useful flag to add is -g to produce debugging information.
- IDLFILES. A list of all the IDL files used in the build.
- SRCS. A list of all source files used. This is needed when generating dependencies.
- PROGS. A list of the executables to be built.

The following Imakefile is for the stock control example and shows how the functions and variables described above can be used:

```
# flags for idl compiler, use
# -B if you want BOAImpl classes
IDLFLAGS = -B

# flags for C++ compiler
# ie. add -g for debug information
C++FLAGS = $(STANDARD_C++FLAGS)

IDLFILES = retail.idl
CLIENT_OBJS = retailC.o Client.o
SERVER_OBJS = retailS.o retail_i.o Srv_Main.o
SRCS = Client.cc Srv_Main.cc retail_i.cc

# extra libraries to build programs
CLIENT_LIBS =
SERVER_LIBS =

PROGS = stock_clt stock_srv

# compile idl files
all:: $(IDLFILES)

# compile all programs
all:: $(PROGS)

IDLDepend(retail)

ServerProgram(stock_srv,$(SERVER_OBJS),$(SERVER_LIBS))
ClientProgram(stock_clt,$(CLIENT_OBJS),$(CLIENT_LIBS))

IDLCleanList($(IDLFILES))
```

Now we process our Imakefile to get a Makefile:

```
kestrel% orbixmkmf
```

An existing Makefile will be renamed to Makefile.bak just incase you need it. We can now invoke make(1) in several ways:

- kestrel% make

  Compiles all programs specified in $(PROGS) and all IDL files in $(IDLFILES).

- kestrel% make clean

  Removes all files generated by the build.

- `kestrel% make skeletons`

  Compiles all IDL files specified by `$(IDLFILES)` with the -S flag to produce implementation skeletons.

- `kestrel% make depend`

  Adds dependencies to the end of the Makefile for all sources specified in `$(SRCS)`. As this modifies the Makefile you will have to do this after every `orbixmkmf`.

You will find an Imakefile and a Makefile in `/proj/orbix/doc/survival/src/` on kestrel. The Makefile has been written by hand and shows how tedious the process of writing Makefiles can be. We leave it as an exercise for the reader to process the Imakefile and compare the resultant Makefile with the hand written one.

5 Taking it further

For more information about writing Orbix programs see the Programmer's Guide [Iona, 1995a] and the Advanced Programmer's Guide [Iona -Adv, 1995b]; we have copies of both which can be borrowed.

The Networks and Distributed Systems Research Group have been using distributed programming platforms and writing distributed applications for several years and there is a lot of in-house experience. If you encounter problems your first port of call should be the current Orbix administrator who can be contacted by emailing `corba@ukc.ac.uk`.

References

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design*. The Benjamin / Cummings Publishing Company, Inc, 2nd edition.

[Iona, 1995a] Iona Technologies Ltd. (1995a). *Orbix Programmer's Guide*. Iona Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, Version 1.3.

[Iona -Adv, 1995b] Iona Technologies Ltd. (1995b). *Orbix Advanced Programmer's Guide*. Iona Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, Version 1.3.

[OMG, 1991] Object Management Group (1991). *Common Object Request Broker: Architecture and Specification*, omg document 91.12.1 Revision 1.1.

A Checklist

A.1 design

Before you begin coding you should sketch out the design. Of particular importance are the responsibilities of the client and server.

A.2 IDL

- Write IDL file <filename>.idl

```
interface <if_name> {
        // Exceptions
        // Attributes
        // Functions
}
```

- Compile IDL.
  - kestrel% idl -B -S <filename>.idl

A.3 Object declaration

- Copy skeleton declaration to object declaration.
  - kestrel% cp <filename>.ih <filename>_i.h
- Edit <filename>_i.h and change <if_name>== to <if_name>_i.
- Add inheritance from BOAImpl class.
  - class <if_name>_i : public <if_name>BOAImpl
- Flesh out the skeleton.
  - declare constructor and destructor.
  - declare private members

A.4 Object definition

- Copy skeleton definition to object definition.
  - kestrel% cp <filename>.ic <filename>_i.c
- Edit <filename>_i.c and change include file from <filename>.ih to <filename>_i.h.
  - #include "<filename>_i.h"
- Change all occurrences of <if_name>== to <if_name>_i.
- Add constructor and destructor.
  - <if_name>::<if_name> (...)
  - <if_name>::~<if_name> (...)
- Write the function bodies.

A.5 Server implementation

- Instantiate implementation object <if_name>_i.
- Inform Orbix that the server is ready.
  - ORBIX::Orbix.impl_is_ready (...).
- Output terminating message which includes server name.

A.6 Client implementation

- Include common header file, not the implementation declaration.
  - `#include "<filename>.hh"`
- Declare pointer to the server object.
  - `<if_name> *p;`
- Bind the pointer to the server object.
  - `p = <if_name>::_bind(...);`
- Once you have finished with the server release the object.
  - `p->_release();`