CORBA Utilities

Simple-to-use utilities that dramatically simplify the development and deployment of CORBA applications

Ciaran McHale
Principal Consultant
IONA Technologies

`http://www.iona.com/devcenter/corba/utilities.htm`

Contents

Preface

Purpose of the CORBA Utilities

The *CORBA Utilities* package has grown organically from the real-world experiences I have had over the last 8 years in my work as a consultant and trainer for IONA Technologies. This collection of utilities provides both software and practical advice that dramatically simplify the development and deployment of CORBA applications. I have used them to speed up the development of applications that are easy to write/maintain, flexible in how they are deployed, and portable to different CORBA products. A lot of people, both inside IONA and elsewhere, also have found these utilities to be helpful.

Obtaining and Installing the Software

The CORBA Utilities package (including both the software and documentation) is available at: `http://www.iona.com/devcenter/corba/utilities.htm`. New versions are placed there a few times per year. The author usually announces the new release on the `comp.object.corba` newsgroup. If you would like to receive an email notification whenever a new version of the CORBA Utilities package has been released then send an email to the author (`Ciaran.McHale@iona.com`). Your email address will be used *only* for the purposes of notifying you that a new version of the CORBA Utilities package is available; your email address will *not* be shared with other people/organizations, and you will *not* receive any SPAM email from the author.

Instructions for installing the CORBA Utilities package are given in the `README.txt` file in the top-level directory of the distribution.

How to Read this Book

In general, each of the CORBA Utilities can be used independently of the others. This means that you do *not* have to read this book from start to finish. Instead, just read the individual chapter for the utility that is of interest to you. The implementations of some of the C++ classes rely upon the portability header files (Chapter 4). However, that is an implementation detail, and you do *not* need to read the Chapter 4 unless the subject matter is of interest to you.

Technical Support

The CORBA Utilities package is not an official product of IONA Technologies. Instead, it has been developed and is maintained by Ciaran McHale, who is a Principal Consultant at IONA Technologies. Bug reports, requests for enhancements and miscellaneous comments should be sent by email to: `Ciaran.McHale@iona.com`.

Training Courses

Since developing the CORBA Utilities, the author has completely overhauled IONA's CORBA development training courses so that they now embody the CORBA Utilities package. For example, the courses:

- Discuss both the "raw" CORBA APIs for creating POAs and the `PoaUtility` class (Chapter 5). Students usually agree that the `PoaUtility` class is far simpler to use.

- Discuss the "raw" CORBA APIs for importing and exporting object references via files and the Naming Service, plus the `importObjRef()` and `exportObjRef()` functions (Chapter 2). Students can appreciate the ease-of-use and flexibility offered by the latter.

- Source code portability is stressed throughout the training courses. The exercise system of the C++ course makes use of the portability header files (Chapter 4). In fact, the source code of the entire C++ exercise system (about 15,000 lines of code) compiles cleanly with Orbix, Orbacus and TAO. Likewise, the entire source code of the Java exercise system compiles cleanly with Orbix and Orbacus. We may consider enhancing the training courses to cover other open-source CORBA implementations in the future, if there is demand.

If you are impressed with the high quality and practical advice of the CORBA Utilities package then you will probably also find the training courses to be equally impressive. You can find details of these training courses on the IONA web site (`www.iona.com`).

Acknowledgments

Firstly, thank you to: Roland Schnir for contributing Chapter 1 (*Tips for Windows*); Adrian Trenaman for porting the `importObjRef()` and `exportObjRef()` functions (Chapter 2) from C++ to Java; Perry Russell for providing TAO support; and Duncan Grisby for his help that made the omniORB port possible. Thanks also to Oliver Kellogg who has given me information that will help me port the utilities to other CORBA implementations (although, unfortunately, that will have to wait until I have more spare time).

Secondly, thank you to people who have given me feedback on the utilities that has allowed me to mature them: Adrian Trenaman, Brian Kelly, Donal Arundel, Michael McKnerney, Perry Russell, Rebecca Bergersen and Steve Vinoski.

Finally, thank you to others who have helped, in one way or another, with the practical issues of making the CORBA Utilities freely available: Jane Merritt, Klaus Hofmann zur Linden, Neil Kenealy and Stephen Zangerl-Salter.

Chapter 1

Tips for Windows

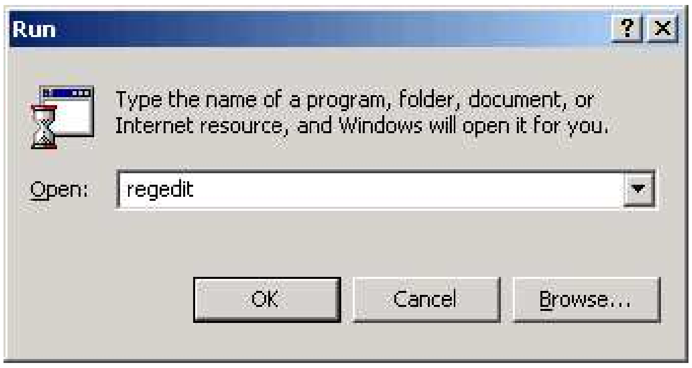
1.1 Enabling Filename Completion

Some UNIX shells perform filename completion whenever you hit the TAB key. This capability can be enabled for Windows command shells.

First, run the `regedit` utility, with the following sequence of mouse clicks:

Start → *Run*

Then type `regedit` in the dialog box and click *OK*.

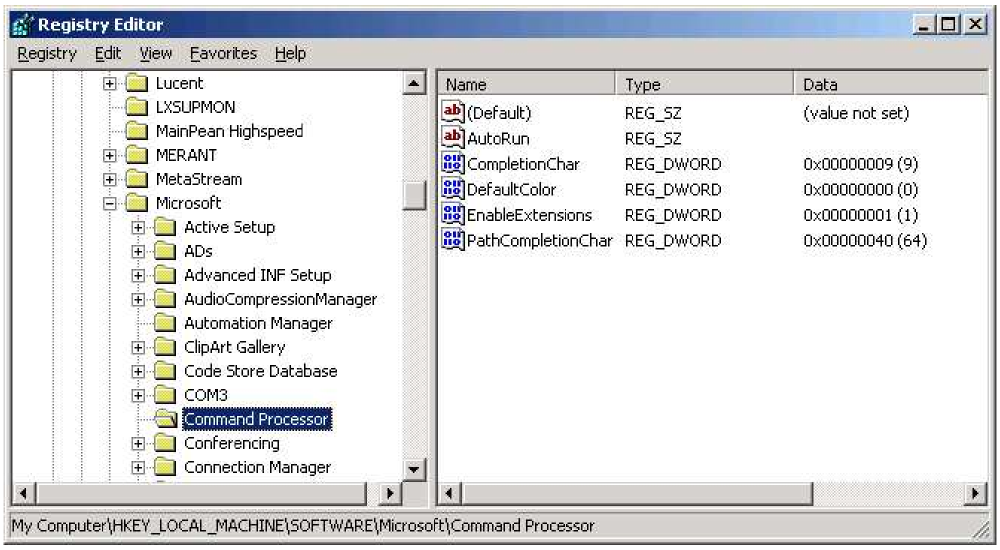

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open: regedit
OK
Cancel
Browse...

Within `regedit`, navigate down to one of the following entries. If you have administrator privileges on your Windows machine then use:

HKEY_LOCAL_MACHINE → *SOFTWARE* → *Microsoft* → *Command Processor*

If you do *not* have administrator privileges then use:

HKEY_CURRENT_USER → *SOFTWARE* → *Microsoft* → *Command Processor*

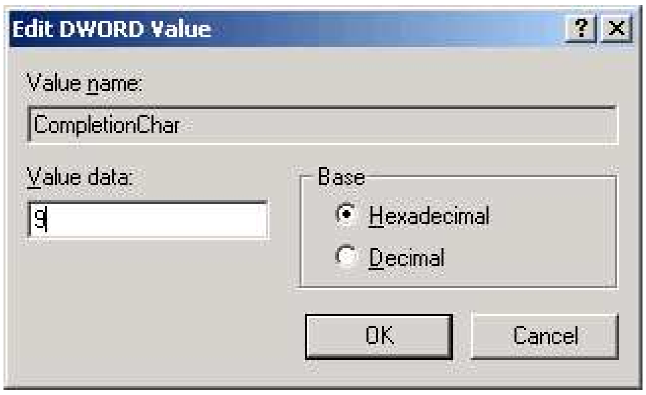


Lucent
LXSUPMON
MainPean Highspeed
MERANT
MetaStream
Microsoft
Active Setup
ADs
Advanced INF Setup
AudioCompressionManager
Automation Manager
ClipArt Gallery
Code Store Database
COM3
Command Processor
Conferencing
Connection Manager

Name	Type	Data
(Default)	REG_SZ	(value not set)
AutoRun	REG_SZ	
CompletionChar	REG_DWORD	0x00000009 (9)
DefaultColor	REG_DWORD	0x00000000 (0)
EnableExtensions	REG_DWORD	0x00000001 (1)
PathCompletionChar	REG_DWORD	0x00000040 (64)

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Command Processor

Double click on `CompletionChar` and enter the value 9 hex, which is the ASCII code for the TAB key.



Value name: CompletionChar

Value data: 9

Base: [✓ Hexadecimal] Decimal

Then click on *OK*, and exit from `regedit`. From now on, whenever you press the TAB key you will activate filename completion. Note that this will work in any *new* command windows that you create. It will not work in any command windows that you had *already* created prior to changing the `CompletionChar` entry in `regedit`.

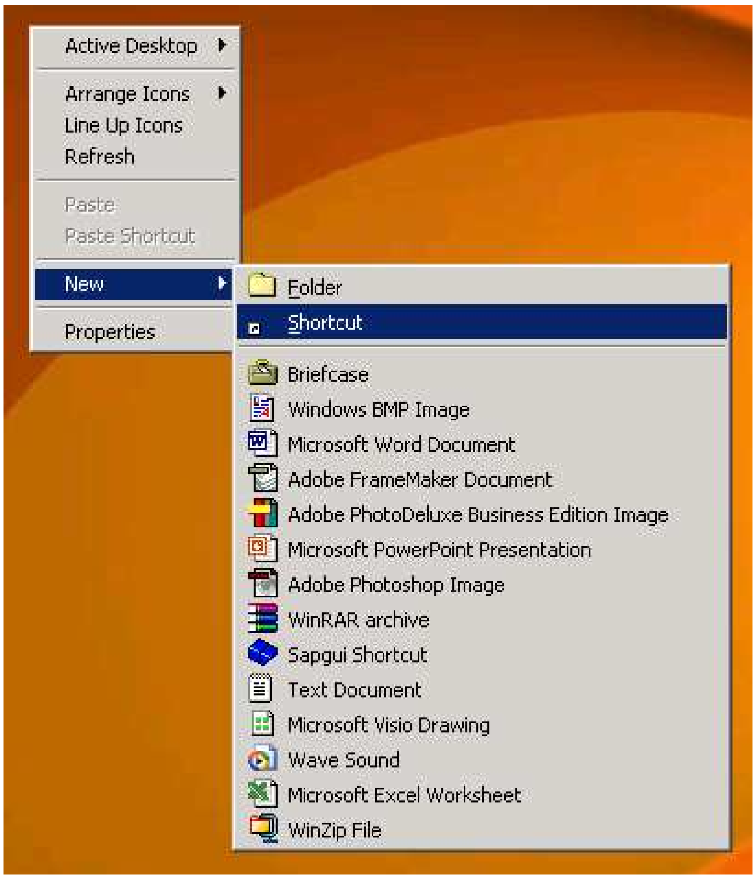
1.2 Creating a Shortcut on the Desktop

When using a command-line based development tool on a Windows machine, you typically have to set some environment variables, such as `PATH`, `CLASSPATH`, `JAVA_HOME`, and also set some environment variables that are specific to the development tool. If you will be using the same development tool *on a daily basis* then it is probably best to set the relevant environment variables in the *Environment Variables* dialog box, which is discussed in Section 1.3 on page 6. However, sometimes you may wish to just experiment with an evaluation version of a development tool, or perhaps you need to switch between different development tools (or different versions of the same tool) on a regular basis. In such cases, it can be more convenient to create a desktop shortcut that opens a new command window and automatically runs a batch file to set up environment variables within that command window. You can create several such shortcuts—one for each development tool.

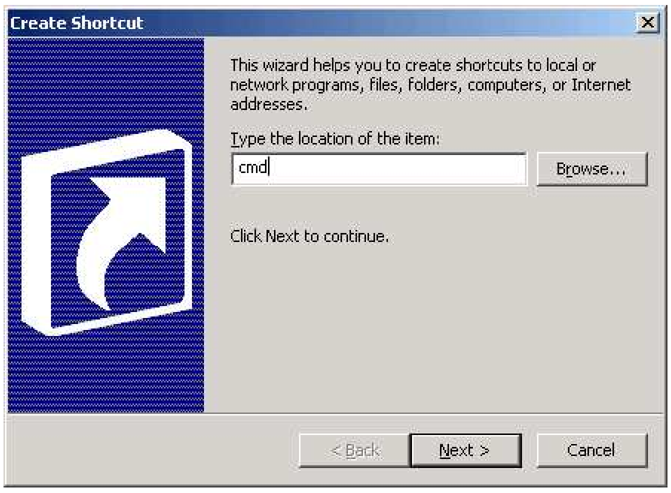
First, you should create a batch file that sets whatever environment variables you want. Use your favorite text editor to create this file. Make sure that it has a `.bat` file extension.

You then create a shortcut on the desktop that, when activated, launches a new command window and automatically runs the batch file. Creating such a shortcut is done with a few mouse clicks.

Right click on the desktop and select *New* → *Shortcut*



This then starts a wizard that guides you through the process. In the first dialog box of the wizard, type `cmd`. This specifies that the shortcut will be for a command window. Then click on *Next*.

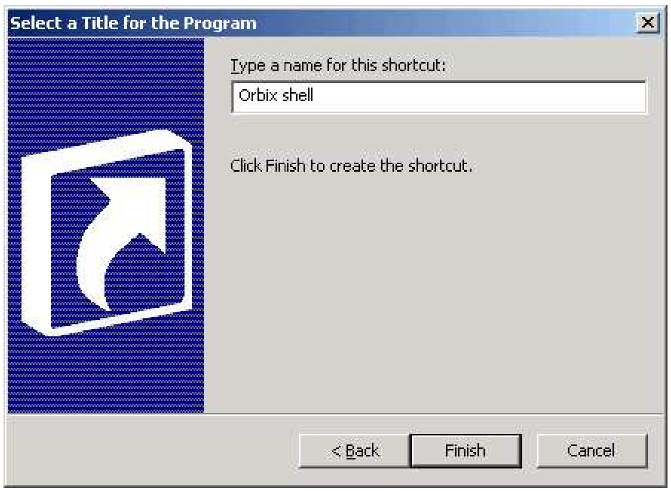

Create Shortcut

This wizard helps you to create shortcuts to local or network programs, files, folders, computers, or Internet addresses.

Type the location of the item: cmd

Click Next to continue.

In the next dialog box, you must specify a name for the shortcut. For example, if you will be using this shortcut for Orbix then you might call it `Orbix shell`.

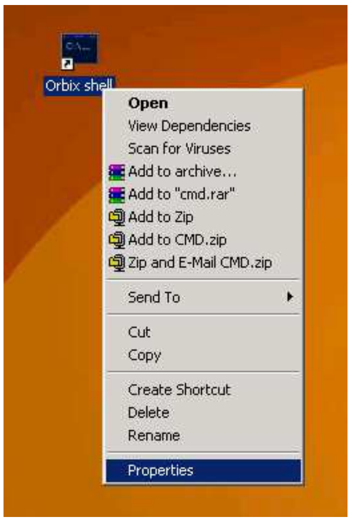

Select a Title for the Program

Type a name for this shortcut: Orbix shell

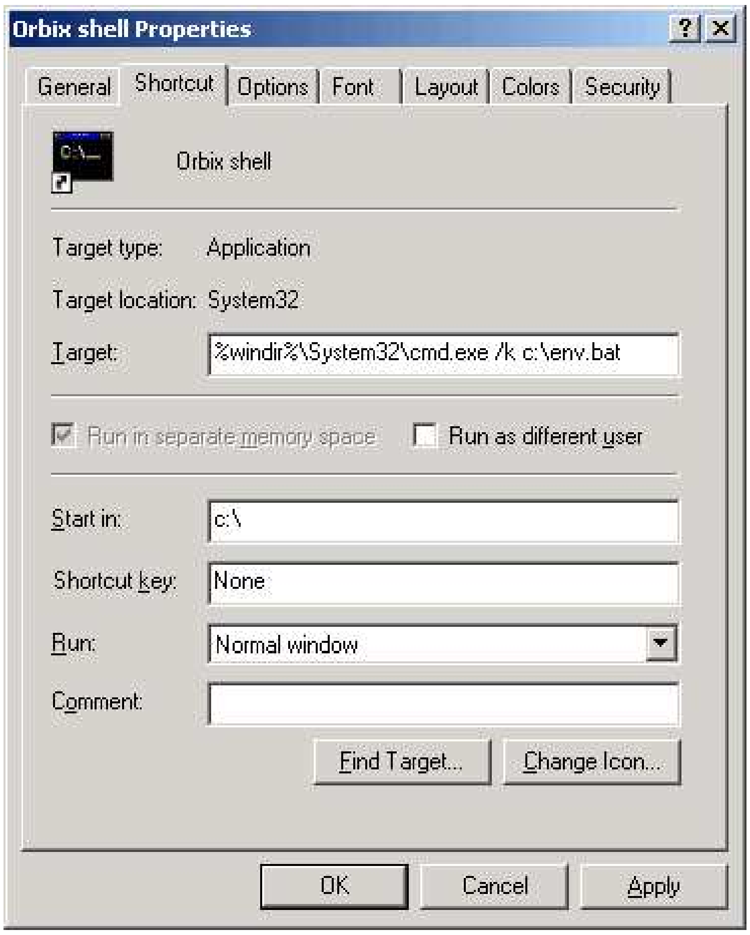
Click Finish to create the shortcut.

Then click on *Finish*. You now have an icon named `Orbix shell` on the desktop.

Right click on the shortcut icon. This allows you to change the *Properties*.



Within the properties dialog box, select the *Shortcut* tab.



Orbix shell

Target type: Application

Target location: System32

Target: %windir%\System32\cmd.exe /k c:\env.bat

[✓] Run in separate memory space
[ ] Run as different user

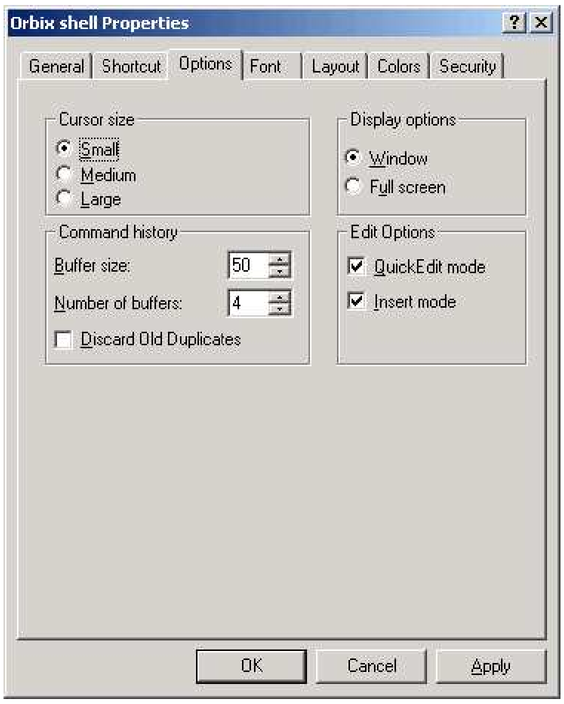
Start in: c:\

Shortcut key: None

Run: Normal window

Comment:

Add `"/k <full-path-to-your-batch file>"` to the *Target* field. Also, use the *Start in* field to specify in which directory you want the newly launched command window to start. You can optionally select some of the other tabs to modify other properties.


Orbix shell Properties

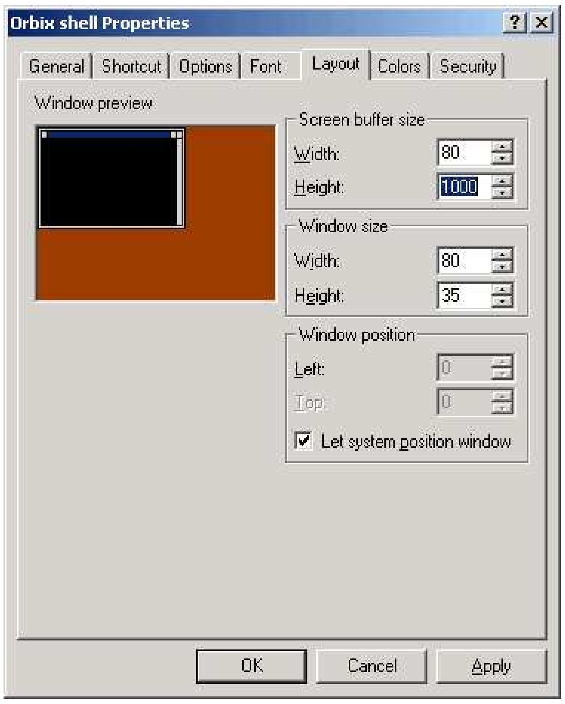
Options tab

Cursor size: [✓ Small] Medium Large

Display options: [✓ Window] Full screen

Command history: Buffer size: 50; Number of buffers: 4; Discard Old Duplicates: ☐

Edit Options: QuickEdit mode ✓; Insert mode ✓


Orbix shell Properties

Layout tab

Window preview

Screen buffer size: Width: 80; Height: 1000

Window size: Width: 80; Height: 35

Window position: Left: 0; Top: 0; Let system position window ✓

When you are happy, click on *Apply* and then *OK*.

1.3 Adding Variables to the Windows Environment

If you will be using the same development tool on a daily basis then it is probably best to set the relevant environment variables in the *Environment Variables* dialog box. In Windows NT, you access this dialog box with the following sequence of mouse clicks:

Start → *Settings* → *Control Panel* → *System* → *Environment*

In Windows 2000, you use a slightly different sequence of mouse clicks:

Start → *Settings* → *Control Panel* → *System* → *Advanced* → *Environment*

Chapter 2

Importing and Exporting Object References

2.1 Introduction

Although a CORBA server may contain many objects, it is typical for just one or two of these objects to be the initial point(s) of contact for client applications. For example, consider a server that contains one `FooFactory` object and many `Foo` objects. When the server starts up, it might *export* (advertise) an object references for its FooFactory object. When a client application starts, it *imports* the reference to the `FooFactory` object and then invokes, say, `lookup()` or `create()` operations on it to get access to some `Foo` objects. The pseudo-code below illustrates the main() function of a typical CORBA server that exports one object reference:

```
1 main(int argc, char ** argv)
2 {
3     orb = CORBA::ORB_init(argc, argv);
4     obj = ...;
5     exportObjRefToNamingService(obj, ...);
6     orb->run();
7 }
```

The above code initializes CORBA (line 3). It then creates one or more objects (line 4) and exports one of these (line 5) to, say, the Naming Service, before going into the event loop (line 6) to receive incoming requests. For the purposes of brevity, some details have been omitted, such as creation of a POA hierarchy and activation of POA managers. The above code contains two common flaws:

1. The server unconditionally (re-)exports the object reference every time the server is run. Doing this might appear to be a harmless practice, but it can cause problems in some kinds of deployment.

2. The server is hard-coded to export the object reference to the Naming Service (line 5). Although the Naming Service is a popular place for exporting object references, the decision

on where to export object references should be left to deployment time rather than being hard-coded at compilation time. For example, some users may prefer to export object references with another technology, such as, say, a Trading Service, a file, a database or FTP. A server that is hard-coded to export object references with one technology can prove to be inconvenient.

The rest of this paper discusses these two problems and explains how they can be overcome in simple, yet very effective ways.

2.2 Servers should Conditionally Export Object References

The pseudo-code below illustrates a good, high-level structure for a server that exports one object reference:

```
 1 main(int argc, char ** argv)
 2 {
 3     orb = CORBA::ORB_init(argc, argv);
 4     parseCmdLineArgs(argc, argv, exportMode, runMode);
 5     obj = new ...;
 6     if (exportMmode) {
 7         exportObjRef(orb, obj, "...");
 8     }
 9     if (runMode) {
10         orb->run();
11     }
12 }
```

The above code initializes CORBA (line 3) and then parses any remaining command-line arguments (line 4). Very importantly, the code checks for the presence of command-line options called, say, `-export` and `-run` and sets boolean variables `exportMode` and `runMode` to true if these options are used (line 4). The server exports its main object reference (line 7) only if the `-export` option was used. Likewise, it goes into the event loop (line 10) only if the `-run` option was used.

Designing a server to support the -export and -run options provides useful flexibility. For example:

1. Some people may wish to re-export object references *every* time a server is restarted. Developers typically find this mode of operation to be convenient. This can be achieved by always specifying *both* `-export` and `-run` command-line options when running the server.

2. Other people may prefer to export an object reference just once, during the initial *installation* of a server. This can be achieved by running the server with just the `-export` option, which causes the server to export its object reference and immediately die (because the `-run` option was not given). Once the server has been installed, thereafter the server can

be launched with just the `-run` option, which will cause the server to go into its event loop without re-exporting its object reference.

This mode of launching a server without re-exporting its object reference is vitally important if the deployment site is running, say, the Orbix Naming Service in replicated mode. In this mode, one replica of the Naming Service is the *master* and it can both read and update its database. All the other replicas of the Naming Service are *slaves* and they have read-only access to the database. If a slave receives a request that involves updating the database then it forwards that request to the master. If the master replica is not running then the Naming Service becomes *read only* until the master is restarted. The main purpose of having a replicated Naming Service is to prevent it from becoming it a single point of failure. This scheme works well if the *only* time a server exports an object reference is when the server is being installed and thereafter the server does *not* attempt to re-export an object reference whenever it is (re-)started. However, if a server insists on always exporting an object reference whenever it is re-started then the server may fail to start up if the master replica of the Naming Service is currently not running, thereby defeating the purpose of having a replicated Naming Service.

The point is that developers should provide a mechanism that allows exporting of object references and running of the server to be performed *independently* of each other. This then allows others to choose the *policy* of how the server should be used in practice. This flexibility is important because developers do not always know how others (such as system administrators) might like to deploy applications. Failing to provide this flexibility can actually cause significant hindrance, as discussed above in the case where a deployment site has a replicated Naming Service.

It may be useful to provide applications with a third option called, say, `-install`. When a server is run with this option, it performs all the steps required to install the server, such as set up configuration files and databases, register the server with the implementation repository and, of course, export an object reference. Alternatively, some people prefer to write a separate install utility to perform these steps, rather than embed this functionality into the executable of a server application. If you do provide an `-install` command-line option for a server then it is *still* useful to provide the `-export` option because a user may want to do a normal install initially and then later re-export the server's object reference to a different location.

2.3 Flexibility in Importing/Exporting Object References

Some CORBA clients and servers are hard-coded to import and export object references through text files. This is especially common in demonstration programs in magazine articles or supplied with CORBA vendor products. This approach is simple but it suffers from a lack of geographic scalability because it requires that a client and server have access to a shared file system. Obviously, this will be the case if the client and server are running on the same computer, and it *may* be the case if they are running on different computers in the same local area network. However, it is rarer for a shared file system to span a wider area network. For this reason, many programmers are told that it is a bad idea to import and export object references through text files, and that use of the Naming Service is better. The result is that many programmers hard-code their clients and servers to import and export object references through the Naming Service.

In reality, it is a bad idea for a programmer to hard-code use of *any* particular import/export technology (such as text files or the Naming Service) in clients or servers. This is because applications have a habit of being used in ways that their developers did not foresee. For example, a developer might think that importing/exporting object references with the Naming Service is a good idea, but an end user might prefer to import/export with a database, or FTP, or even email. Sometimes the developer works in the same organization where the application is going to be deployed and hence s/he *knows* what is the preferred technology for importing/exporting object references. However, even in these situations, there might be a change in the preferred import/export technology in a few months time for any variety of reasons. For example, as the organization's use of CORBA changes, they might prefer to import/export with a Trading Service rather than with a Naming Service. Alternatively, a system administrator might not want to allow a Naming Service to be accessed across a firewall and it might become more convenient to import/export object references through FTP instead. It is time consuming and expensive to have to modify an existing application's hard-coded import/export logic whenever a change occurs in the organization's preference for how object references should be imported or exported.

A better approach is for an application to offer some flexibility in how it imports/exports object references. Ideally, an application should be able to import/export object references with *any* technology (text files, Naming Service, Trading Service, fax, email, FTP, databases or whatever), and the choice of which technology to use should be left to deployment time. Achieving this goal turns out to be surprisingly easy, as we now discuss. Consider the following two utility functions (shown first in C++ and then in Java):

```
// C++ version (defined in "import_export.h")
namespace corbautil {
    CORBA::Object_ptr
    importObjRef(
        CORBA::ORB_ptr          orb,
        const char *            instructions)
            throw(ImportExportException);
    void
    exportObjRef(
        CORBA::ORB_ptr          orb,
        CORBA::Object_ptr       obj,
        const char *            instructions)
            throw(ImportExportException);
};

// Java version
package com.iona.corbautil;
import org.omg.CORBA.*;
public class ImportExport {
    static public org.omg.CORBA.Object
    importObjRef(
        ORB                     orb
        String                  instructions)
            throws ImportExportException;
    static public void
    exportObjRef(
        ORB                     orb
        org.omg.CORBA.Object    obj,
        String                  instructions)
            throws ImportExportException;
}
```

As their names suggest, `importObjRef()` and `exportObjRef()` are used to import and export object references. Both take a reference to an `ORB` as a parameter, so that they can invoke operations upon it—for example, `string_to_object()`, `object_to_string()` or `resolve_initial_references()`—as an aid to importing or exporting an object reference. Both functions also take an `instructions` parameter that specifies *how* the object reference should be imported or exported. This parameter will be discussed in detail shortly. Finally, the `exportObjRef()` function takes another parameter, which is the object reference to be exported, while `importObjRef()` returns the imported object reference.

2.3.1 Instructions Passed to `exportObjRef()`

The `instructions` parameter passed to `exportObjRef()` is a string that can be in any of the following formats:

- `"`**`name_service`**`#path/within/naming/service"`
  This exports an object reference to the Naming Service.
  - Example: `"`**`name_service`**`#foo/bar/acme"`
- `"`**`file`**`#path/to/file"`
  This exports an object reference by stringifying it and writing it to a file.
  - Example: `"`**`file`**`#/tmp/obj_ref.ior"` (full path to file)
  - Example: `"`**`file`**`#obj_ref.ior"` (relative filename)
- `"`**`exec`**`#command with IOR place-holder"`
  This exports an object reference by stringifying it and passing it as a command-line argument to the specified command that is executed. The IOR place-holder in the command is replaced with the stringified object reference before the command is executed.
  - Example: `"`**`exec`**`#/usr/bin/perl some_script.pl IOR"`
  - Example: `"`**`exec`**`#cmd /c echo IOR > /tmp/obj_ref.ior"`
  - Example: `"`**`exec`**`#nsadmin -b foo/bar/acme IOR"`

- "**java_class**#fully.scoped.class.name with optional arguments"
  This variant works only with the Java implementation. It exports an object reference by using Java's reflection APIs to create an instance of the specified class and invoking `exportObjRef()` upon it.
  - Example: "**java_class**#full.package.name.of.class"

The `"name_service#..."` and `"file#..."` variants are provided because users commonly want to export with the Naming Service or a file. The `"exec#..."` variant is provided as a fallback mechanism in case users want to export an object reference with a different technology. Specifically, many technologies (such as FTP, databases, email and so on) can be manipulated by command-line utilities. Exporting an object by such a technology is made possible by an `"exec#..."` instruction.

The `"exec#..."` instruction subsumes the power of both `"name_service#..."` and `"file#..."` because it can execute command-line utilities that will bind (advertise) an object reference into the Naming Service or write a stringified object reference to a file. However, the `"name_service#..."` and `"file#..."` variants are provided both for convenience and for efficiency.

The `"java_class#..."` variant is supported only in the Java implementation. This variant allows developers to implement alternative algorithms for `exportObjRef()` in Java. Details on how to write such algorithms are given in Section 2.4.

2.3.2 Instructions Passed to `importObjRef()`

The instructions parameter passed to `importObjRef()` is a string that can be in any of the following formats:

- "**name_service**#path/within/naming/service"
  This imports an object reference from the Naming Service.
  - Example: "**name_service**#foo/bar/acme"
- "**file**#path/to/file"
  This reads a stringified object reference from a file.
  - Example: "**file**#/tmp/obj_ref.ior" (full path to file)
  - Example: "**file**#obj_ref.ior" (relative filename)
- "**exec**#command"
  This imports an object reference by executing the specified command and interpreting the standard output of that command as a stringified object reference.
  - Example: "**exec**#/usr/bin/perl some_script.pl"
  - Example: "**exec**#cat /tmp/obj_ref.ior"

  - Example: `"`**`exec`**`#nsadmin -r foo/bar/acme"`

- `"`**`java_class`**`#fully.scoped.class.name with optional arguments"`
  This variant works only with the Java implementation. It imports an object reference by using reflection APIs to create an instance of the specified class and then invoking `importObjRef()` upon it.

  - Example: `"`**`java_class`**`#full.package.name.of.class"`

- `"IOR:..."`, `"corbaloc:..."` or `"corbaname:..."`
  Any of these formats import an object reference by calling `string_to_object()`.

The `"name_service#..."`, `"java_class#..."` and `"file#..."` instruction variants have a similar format for both `exportObjRef()` and `importObjRef()`. However, the `"exec#..."` variant is different, depending on whether it is used in `exportObjRef()` or `importObjRef()`. In `exportObjRef()`, `"exec#..."` takes an IOR place-holder, but in `importObjRef()` it does not take an IOR place-holder; instead, the executed command should write the stringified object reference to its standard output.

In addition, `importObjRef()` can accept instructions in any of the URL formats that are specified by the CORBA specification. These include `"IOR:.."`, `"corbaloc:..."` and `"corbaname:..."`. Since the CORBA specification may define new URL formats in the future (or a CORBA vendor may support proprietary URL formats), `importObjRef()` assumes that any string starting with letters and a colon is a URL and passes it as a parameter to `string_to_object()`.

2.3.3 C++ Usage

The **bold** code in the pseudo-code below shows how a C++ server can export an object reference with the `exportObjRef()` function:

```
#include "import_export.h"
main(int argc, char ** argv)
{
    orb = CORBA::ORB_init(argc, argv);
    parseCmdLineArgs(argc,argv, exportMode, runMode, instructions);
    obj = ...;
    if (exportMode) {
        try {
            corbautil::exportObjRef(orb, obj, instructions);
        }
        catch (const corbautil::ImportExportException & ex) {
            cerr << ex << endl;
            orb->destroy();
            exit(1);
        }
16     }
17     if (runMode) {
18         orb->run();
19     }
20     orb->destroy();
21 }
```

The `instructions` that specify where the object reference is exported to should *not* be hard-coded into the application, but rather should be obtained from, say, a command-line argument (line 5) or a runtime configuration file. If `exportObjRef()` fails for any reason then it throws an exception that contains a descriptive error message. For this reason, the call to `exportObjRef()` (line 9) is enclosed in a `try-catch` clause. If an exception is thrown then the exception message is printed out and the application gracefully terminates.

The use of `importObjRef()` is similar, and is shown in **bold** in the pseudo-code below:

```
#include "import_export.h"
main(int argc, char ** argv)
{
    orb = CORBA::ORB_init(argc, argv);
    parse_cmd_line_args(instructions);
    try {
        obj = corbautil::importObjRef(orb, instructions);
    } catch (const corbautil::ImportExportException & ex) {
        cerr << ex << endl;
        orb->destroy();
        exit(1);
    }
    ... // narrow obj and invoke upon it
}
```

2.3.4 Java Usage

The **bold** code in the pseudo-code below shows how a Java server can export an object reference with the `exportObjRef()` function:

```
import com.iona.corbautil.*;
import org.omg.CORBA.*;
...
public static void main(String[] args)
{
    BooleanHolder exportMode = new BooleanHolder();
    BooleanHolder runMode = new BooleanHolder();
    StringHolder instructions = new StringHolder();
    orb = ORB.init(args);
    parseCmdLineArgs(args, exportMode, runMode, instructions);
    obj = ...;
12     if (exportMode.value) {
13         try {
14             ImportExport.exportObjRef(orb, obj,
15                     instructions.value);
16         } catch (ImportExportException ex) {
17             System.out.println(ex.getMessage());
18             orb.destroy();
19             System.exit(1);
20         }
21     }
22     if (runMode.value) {
23         orb.run();
24     }
25 }
```

The `instructions` that specify where the object reference is exported to should *not* be hard-coded into the application, but rather should be obtained from, say, a command-line argument (line 10) or a runtime configuration file. If `exportObjRef()` fails for any reason then it throws an exception that contains a descriptive error message. For this reason, the call to `exportObjRef()` (line 14) is enclosed in a `try-catch` clause. If an exception is thrown then the exception message is printed out and the application gracefully terminates.

The use of `importObjRef()` is similar, and is shown in **bold** in the pseudo-code below:

```
 1  import com.iona.corbautil.*;
 2 import org.omg.CORBA.*;
 3 ...
 4 public static void main(String[] args)
 5 {
 6     StringHolder instructions = new StringHolder();
 7     orb = ORB.init(args);
 8     parseCmdLineArgs(args, instructions);
 9     try {
10         obj = ImportExport.importObjRef(orb,
11                                           instructions.value);
12     } catch (ImportExportException ex) {
13         System.out.println(ex.getMessage());
14         orb.destroy();
15         System.exit(1);
16     }
17     ... // narrow obj and invoke upon it
18 }
```

2.4 Implementing Import/Export Algorithms as Java Classes

The `ImportExportAlgorithm` interface (defined in the `com.iona.corbautil` package) defines the signatures of the `importObjRef()` and `exportObjRef()` methods. If you write a Java class that implements this interface then you can use that class to import/export object references by using the `"java_class#full.package.name.of.class"` variant of instructions.

The code below is from the `ImportExportExampleAlgorithm` class that is provided in the `com.iona.corbautil` package. This code imports/exports object references through standard input/output and provides an example of how to write your own Java classes to import/export object references.

```
package com.iona.corbautil;
import org.omg.CORBA.*;
import java.io.*;
public class ImportExportExampleAlgorithm
implements ImportExportAlgorithm
{
    public void exportObjRef(
        ORB                   orb,
        org.omg.CORBA.Object  obj,
        String                instructions)
                                 throws ImportExportException
    {
        String strIOR = null;
        try {
            strIOR = orb.object_to_string(obj);
        } catch(Exception ex) {
            throw new ImportExportException(
                  "export failed for instructions ‘"
                + instructions
                + "’: object_to_string() failed: " + ex);
        }
        System.out.println("instructions = ‘"
                          + instructions + "’");
        System.out.println("IOR = " + strIOR);
    }
    public org.omg.CORBA.Object importObjRef(
        ORB     orb,
        String  instructions) throws ImportExportException
    {
        System.out.println("instructions: " + instructions);
        System.out.println("Enter a stringified obj ref: ");
        try {
            BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
            String strIOR = stdin.readLine();
            return orb.string_to_object(strIOR);
        } catch (Exception ex) {
            throw new ImportExportException(
                  "import failed for "
                + "instructions ‘" + instructions
                + "’: error importing a stringified "
                + "object reference from the console: "
                + ex);
        }
    }
}
```

2.5 Benefits of `importObjRef()` and `exportObjRef()`

The `importObjRef()` and `exportObjRef()` functions offer several benefits:

- Applications that use `importObjRef()` and `exportObjRef()` do not have to be hard-coded to use one specific import/export technology, but rather can choose which import/export technology to use at runtime. Thus, an important decision is moved from being a compile-time choice made by developers to being a deployment time choice made by users.
- These utility functions can protect developers from vendor lock-in. For example, some CORBA vendors provide proprietary load-balancing enhancements to their implementations of the Naming Service. Applications no longer need to use proprietary APIs to export an object reference to a load-balancing Naming Service. Instead, a runtime configuration file or command-line argument can specify an `"exec#..."` variant of instructions to export an object reference to the load-balancing Naming Service using a vendor-supplied command-line utility. If the CORBA vendor does not supply such a command-line utility then the developer can *write* their own such utility, or implement a Java class that provides this functionality.
- The flexibility provided by `importObjRef()` and `exportObjRef()` helps to *simplify* application code. As the code shown earlier illustrates, using `importObjRef()` and `exportObjRef()` in applications requires remarkably few lines of code: far fewer than a developer would write if using the raw API of, say, the Naming Service. Furthermore, not only do developers have to write fewer lines of code, the developers are also insulated from some of the complexity of the Naming Service API (or some other import/export technology).

Chapter 3

The `corbaloc` and `corbaname` URLs

3.1 Introduction

URLs used on the world wide web (WWW) begin with the name of a protocol, followed by `":"`, for example, `"http:"`, `"ftp:"` or `"file:"`. A stringified object reference begins with `"IOR:"` so this also looks similar to a URL.

In early versions of CORBA, the only kind of string parameter that could be passed to `string_to_object()` was a stringified object reference. CORBA has now matured to allow *other* URL-like strings to be passed as parameters to `string_to_object()`. A CORBA product may optionally support the `"http:"`, `"ftp:"` and `"file:"` formats. The semantics of these is that they provide details of how to download a stringified IOR (or, recursively, download another URL that will eventually provide a stringified IOR).

Although support for `"http:"`, `"ftp:"` and `"file:"` is optional, all CORBA products must support `"corbaloc:"` and `"corbaname:"`, which are two URLs defined by the OMG. The purpose of these is to provide a human readable/editable way to specify a location where an IOR can be obtained.

3.2 The `corbaloc` URL

Some examples of `corbaloc` URLs are shown below:

```
corbaloc:iiop:1.2@host1:3075/NameService
corbaloc:iiop:host1:3075,iiop:host2:3075/NameService
```

The first URL specifies that an IOR can be obtained by using version 1.2 of the IIOP protocol to send a *LocateRequest* message with parameter `"NameService"` to port 3075 on host `host1`.

The second URL is different in two ways. Firstly, by omitting `"1.2@"`, it uses the default version (`1.0`) of the IIOP protocol. Secondly, the URL specifies two `<host>:<port>` addresses rather than one. In general, any number of `<host>:<port>` addresses can be specified, separated by commas. This second form is used to provide fault tolerance: the *LocateRequest* message will be sent to one of the addresses in the list; if that `<host>:<port>` cannot be contacted then another address in the list will be tried, and so on.

Many parts of the `corbaloc` URL have default values:

- The default protocol is `iiop`.
- If the protocol is `iiop` then the default *version* of IIOP that is used is 1.0. It is advisable to specify the most recent version of IIOP that is understood by both the client and server application. This is because more modern versions of IIOP tend to have better capabilities that might make client-server interaction more efficient.
- The default port number is 2809. This is the port that the Internet Assigned Numbers Authority (`www.iana.org`) has assigned for use with `corbaloc`.

The CORBA specification currently specifies two protocols that can be used in `corbaloc` URLs. One protocol is `iiop`, which has already been discussed. The other protocol is called `rir`, which seems like a strange name until you realize that it is an acronym for *resolve initial references*. Unsurprisingly, this protocol specifies that an object reference should be obtained by calling the `resolve_initial_references()` operation, passing the specified name as a parameter. For example, the `corbaloc` URL below specifies that an IOR should be obtained by calling `resolve_initial_references("NameService")`:

```
corbaloc:rir:/NameService
```

One benefit of the `rir` protocol is that allows `string_to_object()` to subsume the functionality of `resolve_initial_references()`. For example, instead of an application being hard-coded to find the Naming Service by passing `"NameService"` to `resolve_initial_references()`, an application can now be hard-coded to find the Naming Service by obtaining a string from a command-line argument or a configuration file and passing this to `string_to_object()`. *If* the string happens to be `"corbaloc:rir:/NameService"` then it is just as if the programmer had used `resolve_initial_references()`, but now there is the flexibility for the string parameter to be a stringified IOR or a `corbaloc` URL that uses the `iiop` protocol. In this way, applications have some extra flexibility in how they find a CORBA Service.

The `rir` protocol is not used often in `corbaloc` URLs. However, it is used more commonly in `corbaname` URLs, which we now discuss.

3.3 The corbaname URL

A `corbaname` URL is a `corbaloc` that specifies how to contact the Naming Service, followed by `"#"` and then a name within the Naming Service. Some examples are shown below:

```
corbaname::foo.bar.com:2809/NameService#x/y
corbaname::host1,:host2,:host3/NameService#x/y
corbaname:rir:/NameService#x/y
```

Passing of the above strings as a parameter to `string_to_object()` causes the Naming Service to be located and `resolve_str()` to be invoked to obtain an IOR from the Naming Service. As the above examples illustrates, a `corbaname` URL can use either the `iiop` or `rir` protocols to locate the Naming Service.

3.4 Architectural Support for `corbaloc`

3.4.1 Client-side Support for `corbaloc`

The `string_to_object()` operation has built-in support for `corbaloc` and `corbaname` URLs:

- If the parameter to `string_to_object()` starts with `"IOR:"` then the operation treats it as a stringified object reference and builds a corresponding proxy.
- If the parameter starts with `"corbaloc:rir"` then the `string_to_object()` operation calls `resolve_initial_references()` and passes the specified name as a parameter.
- If the parameter is a `corbaloc` URL that uses the `iiop` protocol then the operation opens a socket connection to the specified host and port, and sends a *LocateRequest* message, using the specified name as the `object key` in the header of the message. The IOR embedded in the returned *LocateReply* message is used as the return value of `string_to_object()`. An important point to note is that `corbaloc` is built on top of *existing* low-level GIOP messages so the OMG did *not* have to define a new version of GIOP to support `corbaloc` URLs.
- If the parameter to `string_to_object()` is a `corbaname` URL then the embedded `corbaloc` details are use to locate a Naming Service. Then `string_to_object()` invokes `resolve_str()` on the Naming Service, passing it the string after the embedded `"#"` as a parameter. The IOR returned from `resolve_str()` is used as the return value of `string_to_object()`.

3.4.2 Server-side Support for `corbaloc`

CORBA does *not* standardize the server-side support for `corbaloc` URLs, nor even the *terminology* for this server-side support. This means that CORBA products provide proprietary mechanisms, often with proprietary terminology. For example:

- The Orbix implementation repository has built-in, server-side support for `corbaloc` URLs, and this is referred to as *named keys*. A named key is a mapping from the *name* component in a `corbaloc` URL to a stringified IOR. The `named_key` sub-commands of the `itadmin` administration utility are used to `create`, `show`, `list` and `delete` named keys. By default, the Orbix implementation repository listens on port 3075 so `corbaloc` URLs should be formatted as shown below:

```
corbaloc::<host-of-IMR>:3075/<name>
```

When the `itconfigure` utility is used to set up an Orbix domain, named keys are automatically created for whatever CORBA Services are added to the domain. For example, if the domain has a Naming Service then a named key called `NameService` is created.

Orbix does *not* expose APIs for embedding server-side `corbaloc` support in normal server applications.

- Orbacus provides some proprietary APIs (in the `BootManager` interface) that can be used by developers to embed server-side `corbaloc` support in their own server applications. These APIs are used by the Orbacus implementation repository, which looks up *name→stringified-IOR* mappings in a configuration file.
- TAO provides proprietary APIs that have different names, but similar semantics, to those of Orbacus.
- OmniORB server-side support for `corbaloc` URLs relies upon placing objects into a specific, predefined POA. OmniORB also provides a prewritten server application called `omniMapper` that listens on a specified port and looks up *name→stringified-IOR* mappings in a configuration file.

As can be seen, each CORBA product has its own different "look and feel" for server-side support of `corbaloc` URLs. Because of this, there is *no* portable way for a CORBA server to use a `corbaloc` URL to advertise one of its own objects. Developers concerned with writing portable CORBA applications should use `corbaloc` URLs *only for* CORBA Services, for example, the Naming Service, Notification Service, Trading Service and so on.

3.5 Bootstrapping Interoperability Problems

One obvious requirement for interoperability between different CORBA products is that they must be able to speak the same on-the-wire protocol (IIOP). However, that by itself it not sufficient. Another, less obvious requirement for interoperability is for one CORBA product to be able to *find*, say, the Naming Service or the Notification Service of another CORBA product. For example, how can an Orbix client *find* (connect to) the Naming Service of an Orbacus installation. This is often called a bootstrapping problem. The `corbaloc` and `corbaname` URLs were invented to address such bootstrapping issues, as we now discuss.

A CORBA application connects to a CORBA Service—for example, the Naming Service, Notification Service, and so on—by calling `resolve_initial_references()` and passing the name of the desired service as a parameter. The CORBA specification does *not* specify how `resolve_initial_references()` works (that is an implementation detail), but in most CORBA products this operation looks in a configuration file to find a *name-of-CORBA-service→stringified-IOR* mapping[1] and then passes the stringified IOR as a parameter to the

[1] For example, the entry in the Orbacus configuration file is `ooc.orb.service.<service>`. The corresponding entry in the Orbix configuration file is `initial_references:<service>:reference`.

`string_to_object()` operation. These mappings are normally set up during the installation and configuration of a CORBA product. To configure, say, Orbix to use an Orbacus Naming Service is a matter of obtaining a stringified IOR of the Orbacus Naming Service (typically from the Orbacus configuration file) and copying this into the Orbix configuration file. Then the next time an Orbix client calls `resolve_initial_references("NameService")`, the client will be directed towards the Orbacus Naming Service. This technique works fine, but it is a bit cumbersome because stringified IORs are not human readable. However, with the introduction of `corbaloc` URLs, the technique becomes much easier. Now, instead of copying a stringified IOR of the Orbacus Naming Service into the Orbix configuration file, it is sufficient to copy a `corbaloc` URL into the Orbix configuration file. The fact that `corbaloc` URLs are easy to read (and edit) by humans makes it more feasible for an organization to use several different CORBA products.[2]

Sometimes, practical or organizational issues may make it awkward to update a configuration file with a stringified IOR or `corbaloc` URL for, say, the Naming Service of another CORBA product. To work around this, the OMG defined two standard command-line options that all CORBA products must support.[3]

The first command-line option takes the form:

```
-ORBInitRef <name>=<value>
```

An example is shown below:

```
-ORBInitRef NameService=corbaloc::host1:3075/NameService
```

The `<value>` in `<name>=<value>` is a stringified IOR or URL that is used if `resolve_initial_references()` is called with `<name>` passed as a parameter. This command-line argument takes precedence over any corresponding information in the CORBA product's configuration file. You need to specify this command-line option *each* time you run an application, so regular use of it can get somewhat tedious. However, this command-line option is useful if, say, restrictive file permissions prevent them from modifying the configuration file of a CORBA installation. It can be useful also when trouble-shooting a connectivity problems in a network.

The second command-line option takes the form:

```
-ORBDefaultInitRef <URL-up-to-but-not-including-final-"/">
```

Some examples are shown below:

```
-ORBDefaultInitRef corbaloc:iiop:1.2@host1:3075
-ORBDefaultInitRef corbaname::host1/NameService#x/y
```

[2] It is rare for an organization to *deliberately decide* to use several CORBA products. However, several CORBA products may make their way into an organization if different departments or development teams make independent choices about which middleware technology they will use, or if the development of CORBA applications is outsourced to other organizations.

[3] When a CORBA application calls `ORB_init()`, it passes command-line arguments as a parameter to `ORB_init()`. This provides the mechanism by which command-line arguments are communicated to the CORBA runtime system.

A call to `resolve_initial_references("<name>")`, results in `"/<name>"` being appended to the string provided by the command-line argument after `-ORBDefaultInitRef`; the result of this string concatenation is then passed as a parameter to `string_to_object()`.

The intention of the `-ORBDefaultInitRef` command-line option is that a user can set up a centralized store of *name* $\rightarrow$ *IOR* mappings. Once this has been done, applications can be started with a single `-ORBDefaultInitRef` command-line argument that points to this centralized store. This is usually more convenient than starting many applications, each with several `-ORBInitRef` command-line arguments.

You need to specify the `-ORBDefaultInitRef` command-line option *each* time you run an application so, just as with `-ORBInitRef`, regular use of it can get tedious. In general, it is usually more convenient to create/modify a configuration file for a CORBA installation than to use these command-line options every time you run a CORBA-based application.

If both `-ORBInitRef` and `-ORBDefaultInitRef` command-line arguments are used then the `-ORBInitRef` arguments take precedence.

Chapter 4

Portability of C++ CORBA Applications

The Portable Object Adapter (POA) specification defines a comprehensive set of APIs that are provided by CORBA products. This means that a developer should be able to write a CORBA application that can be re-compiled easily with several CORBA vendor products. This goal has been met with the Java mapping. Unfortunately, the C++ mapping has one annoying hindrance to portability: it does *not* specify the names of CORBA-related header files. The practical effect of this is that a developer must change `#include` directives for CORBA-related header files when porting an application to a different CORBA product. At first this may not seem like a big problem. However, such `#include` directives will appear in most source-code files. Porting an application is much easier if non-portable code is concentrated in just a small number of files rather than being spread thinly over many source-code files. This chapter discusses a simple, yet effective, technique that minimizes the porting headaches of non-portable `#include` directives.

4.1 Introduction to the Problem

The program below is quite simple: it creates an ORB (line 13), outputs `"Hello, world"` (line 14) and then destroys the ORB (line 15). A `try-catch` clause (lines 12–20) is used in case any of the CORBA APIs throws an exception.

```
1   #include <omg/orb.hh>
2   #include <it_cal/iostream.h>
3   IT_USING_NAMESPACE_STD
4
5   int
6   main(int argc, char ** argv)
7   {
8       CORBA::ORB_var orb;
9       int exit_status;
10
11      exit_status = 0;
12      try {
```

```
13          orb = CORBA::ORB_init(argc, argv);
14          cout << "Hello, world" << endl;
15          orb->destroy();
16      }
17      catch (const CORBA::Exception & ex) {
18          cout << ex << endl;
19          exit_status = 1;
20      }
21      return exit_status;
22 }
```

The functional code in the application (lines 5–22) is portable across many CORBA products and operating systems. Unfortunately, the `#include` directives in the code (lines 1–3) are not portable: these lines are specific to Orbix. It is these first few lines of code that would have to be changed if porting the application to another CORBA product. In general, there are three different portability problems associated with `#include`'d filenames for CORBA applications. These problems are discussed in the following subsections, and then a simple solution is discussed in Section 4.2.

4.1.1 Portability Problem 1: CORBA Header Files

The IDL-to-C++ mapping does not specify the names of CORBA-related header files. For example, Orbix defines basic CORBA functionality in the file `<omg/orb.hh>`, while Orbacus defines similar functionality in the file `<OB/CORBA.h>` and TAO uses `<tao/corba.h>`. In general, each CORBA product has a different name for this header file.

4.1.2 Portability Problem 2: Stub Code and Skeleton Code Header Files

If you have an IDL file called `foo.idl` then the Orbix stub-code and skeleton-code header files are called `foo.hh` and `fooS.hh`, respectively, while the equivalent header files in Orbacus are called `foo.h` and `foo_skel.h`, and the TAO versions are called `fooC.h` and `fooS.h`. In general, each CORBA product uses different names for the generated stub-code and skeleton-code files.

4.1.3 Portability Problem 3: Old or Standard C++ Header Files

Although, the C++ language dates from the early 1980s, the language was not standardized until the mid-1990s. In pre-standardized C++, many header files provided with compilers had `".h"` extensions, for example `<iostream.h>`. The standardization committee decided to make two important changes to standard header files: (1) the `".h"` extension was dropped, for example, `<iostream.h>` became `<iostream>`; and (2) the types and global variables defined in these standard header files were defined in the `std` namespace rather than in the global scope, for example, `cout` became `std::cout`.

The statement using `"namespace std;"` can be used to used to refer to `std` types and variables without the `std::` prefix. If a developer wants to write a program that can be compiled with the old or the standard header files then this can be done using the somewhat clumsy coding idiom shown below:

```
#ifdef USE_OLD_HEADER_FILES
#include <iostream.h>
#else
#include <iostream>
using namespace std;
#endif
```

The developer must use similar `#ifdef...#else...#endif` constructs for all header files that have *old* and *standard* counterparts.

Many CORBA products have been developed so they can be used with old or standard header files. Developing a CORBA product with such portability in mind means extra work for a CORBA vendor. Typically it means that the CORBA vendor must provide two versions of libraries: one built with old header files and the other built with standard header files. This extra work undertaken by CORBA vendors has resulted in two important benefits. Firstly, it allows a CORBA product to be made available on platforms that have only the old header files. Secondly, even on platforms that have the standard header files, some developers may be forced to use the old header files because they must link with legacy code that uses the old header files, and the CORBA product can be used by such developers.

It is common for CORBA products to provide their own abstraction layer that can be used to easily switch between using old and standard header files. For example, Orbix provides various header files with names such as `<it_cal/iostream.h>`, `<it_cal/fstream.h>` and so on.[1] These files `#include` either the corresponding old or standard header file, depending on whether or not the symbol `IT_CLASSIC_IOSTREAMS` is defined. Also, depending on whether or not that symbol is defined, the macro `IT_USING_NAMESPACE_STD` (line 3 of the example program given in Section 4.1) expands out to be either an empty string or the statement `"using namespace std;"`.

Developers can choose whether or not they want to make use of a CORBA products abstraction layer for old and standard header files. Obviously, the advantage of using this abstraction layer is that it is a pre-written abstraction layer so the developer does not have to re-invent the wheel. However, there are two disadvantages to using such an abstraction layer. Firstly, the abstraction layer is proprietary to that CORBA product so use of it makes source code non-portable to other CORBA products. Secondly, making use of the abstraction layer typically involves `#include`-ing at least one CORBA product-specific header file into every `.cpp` file, and a developer may not wish to do this in a `.cpp` file that is otherwise independent of CORBA.

[1] The `"it_cal"` prefix is derived as follows: `"it"` is an acronym for IONA Technologies, and `"cal"` is an acronym for Compiler Abstraction Layer.

4.2 A Simple Solution

There is an easy way for developers to protect their source code from differences in the names of include files across CORBA vendor products and also, if desired, between old and standard C++ header files. The way to achieve this is for the developer to use his own portability abstraction layer. This is remarkably easy to do and takes very little time. This section discusses such a portability layer that supports Orbix, Orbacus, TAO and omniORB. It should be easy for readers to extend this to support other CORBA products on an as-needed basis. The principle is illustrated with file below.

```
// File: p_orb.h
#ifndef P_ORB_H_
#define P_ORB_H_

#if defined(P_USE_ORBIX)
#include <omg/orb.hh>
#elif defined(P_USE_ORBACUS)
#include <OB/CORBA.h>
#elif defined(P_USE_TAO)
#include <tao/corba.h>
#elif defined(P_USE_OMNIORB)
#include <omniORB4/CORBA.h>

#else
#error "You must #define P_USE_ORBIX, P_USE_ORBACUS, P_USE_TAO or ..."
#endif

#endif /* P_ORB_H_ */
```

The above file, `p_orb.h`, is an abstraction layer for the CORBA product-specific header file that defines basic ORB functionality. The `"p_"` prefix stands for *portability*. The header file `#include`'s the relevant Orbix-specific header file if the symbol `P_USE_ORBIX` is defined, the Orbacus-specific header file if `P_USE_ORBACUS` is defined, the TAO-specific header file if `P_USE_TAO` is defined, or the omniORB-specific header file if `P_USE_OMNIORB` is defined; otherwise it generates an error message. It should be trivial to extend this file to support other CORBA products. For most C++ compilers, the easiest way to define the appropriate `P_USE_<product-name>` symbol is through the `-D<symbol>` command-line option, for example, `-DP_USE_ORBIX`.

Another header file that should be written in the same style is `p_poa.h`, which `#include`'s the CORBA product-specific header file that defines the POA APIs.

Portability header files that `#include` CORBA product-specific stub-code and skeleton-code header files also need to be written. For an IDL file, `foo.idl`, these portability header files might be called, say, `p_foo_stub.h` and `p_foo_skeleton.h`. The two files below illustrate the general form of these portability header files. Occurrences of `foo` and `FOO` are written in **bold** to indicate which parts of the files depend on the name of the IDL file.

```
// File: p_foo_stub.h
#ifndef P_FOO_STUB_H
#define P_FOO_STUB_H
#if defined(P_USE_ORBIX)
#include "foo.hh"
#elif defined(P_USE_ORBACUS)
#include <OB/CORBA.h>
#include "foo.h"
#elif defined(P_USE_TAO)
#include "fooC.h"
#elif defined(P_USE_OMNIORB)
#include "foo.hh"
#else
#error "You must #define P_USE_ORBIX, P_USE_ORBACUS, P_USE_TAO or ..."
#endif
#endif /* P_FOO_STUB_H */

// File: p_foo_skeleton.h
#ifndef P_FOO_SKELETON_H
#define P_FOO_SKELETON_H
#if defined(P_USE_ORBIX)
#include "fooS.hh"
#elif defined(P_USE_ORBACUS)
#include <OB/CORBA.h>
#include "foo_skel.h"
#elif defined(P_USE_TAO)
#include "fooS.h"
#elif defined(P_USE_OMNIORB)
#include "foo.hh"
#else
#error "You must #define P_USE_ORBIX, P_USE_ORBACUS, P_USE_TAO or ..."
#endif
#endif /* P_FOO_SKELETON_H */
```

The file templates shown above support Orbix, Orbacus, TAO and omniORB. It should be trivial to extend them to support other CORBA products. Because these files are so repetitive and they need to be written for each IDL file used in a project, it is best to write a short script (using Tcl, Perl, sed or whatever scripting language you prefer) that can *generate* these portability header files.

Finally, portability header files can be written that `#include` the appropriate old or standard C++ header files. This is illustrated by example below:

```
#ifndef P_IOSTREAM_H_
#define P_IOSTREAM_H_
#if defined(P_USE_OLD_TYPES)
#include <iostream.h>
```

```
#else
#include <iostream>
using namespace std;
#endif
#endif /* P_IOSTREAM_H_ */
```

As the above example illustrates, if the symbol `P_USE_OLD_TYPES` is defined then the appropriate old header file is included; otherwise, the standard header file is included and the statement `"using namespace std;"` is executed so programmers do not have to use the `std::` prefix.

4.3 Issues not Tackled

This chapter has discussed how a simple technique can dramatically reduce problems in porting applications to use different CORBA products. Some other portability issues exist that are outside the scope of this chapter, such as:

- There will be differences in Makefiles when porting an application from one CORBA product to another. For example, flags passed to the IDL compiler and C++ compiler will change. Other changes will occur in the names of libraries that should be linked into the application. Also, the names of the generated stub-code and skeleton-code `.cpp` files will differ.
- A C++ CORBA product is typically intended to be used with a specific brand of C++ compiler. If you switch from one CORBA product to another then you might also have to switch to a different brand of C++ compiler.
- Although the CORBA specification describes the high-level functionality of the Implementation Repository (IMR), the CORBA specification has not standardized the “look and feel” of the IMR. For this reason, details of how to register a server with the IMR are different in different CORBA products.
- CORBA has not standardized upon APIs for logging diagnostic messages or for retrieving runtime configuration information. Many CORBA products provide logging and configuration APIs as proprietary enhancements. If you make use of such proprietary APIs in your application then this will make porting your application to another CORBA product more difficult.

Chapter 5

Creation of POA Hierarchies Made Simple

The CORBA POA specification is considered by many to be powerful but complex. Actually, the POA specification is powerful and *conceptually simple*. Unfortunately, verbose APIs obscure the simple concepts that are at the heart of the POA specification. It is these verbose APIs that are largely to blame for the reputation that the POA specification has for being complex.

This chapter discusses a class called `PoaUtility` that provides a simplification "wrapper" around the POA APIs. The wrapper API contains just three operations that provide all the power and flexibility previously provided by a dozen operations of the "raw" POA API. The `PoaUtility` class does *not* hide the concepts of the POA. In fact, it is just the opposite: by replacing a dozen low-level operations with a smaller number of higher-level operations, the wrapper allows developers to more easily see the underlying simplicity and elegance of the POA.

Currently, the `PoaUtility` class works "out of the box" with five CORBA products: Orbix/C++, Orbix/Java, Orbacus/C++, Orbacus/Java, and TAO. It should be easy to extend support to other CORBA products and/or languages.

5.1 Introduction

Many people have mixed feelings about the POA specification. On the one hand, it provides CORBA with a lot of power and flexibility. On the other hand, the POA specification can *seem* complex, and this puts a lot of developers off CORBA. This is a shame because the POA specification is conceptually simple and is actually quite elegant. It is just that verbose APIs obscure the simple concepts that are at the heart of the POA specification.

This chapter discusses a class called `PoaUtility` that provides a simplification "wrapper" around the POA APIs. The wrapper API contains just three operations that provide all the power and flexibility previously provided by a dozen operations of the "raw" POA API. The `PoaUtility` class does *not* hide the concepts of the POA. In fact, it is just the opposite: by replacing a dozen low-level operations with a smaller number of higher-level operations, the wrapper allows developers to more easily see the underlying simplicity and elegance of the POA.

Some important benefits of the wrapper API are as follows:

- The `PoaUtility` class reduces the learning curve for the POA without sacrificing any of its power and flexibility. This means that developers will require less time to become skilled in the use of CORBA.

- Developer productivity is increased. Realistically, developers will be able to create a server's POA hierarchy in just a few minutes and with just a few lines of code, *without* having to consult any reference documentation. Furthermore, if the creation of a POA fails (for example, a developer might have assigned it incompatible policies) then the `PoaUtility` class throws an exception that contains a self-explanatory message that helps the developer to quickly diagnose the source of the problem.

- It is common for a CORBA product to allow a server to be deployed (1) with or (2) without an Implementation Repository, and with both of these options the server can (3) listen on a fixed port or (4) listen on a port that is chosen by the operating system. Unfortunately, CORBA does not specify the practical details of how to choose a particular deployment model for a server application. Instead, such details are left to proprietary extensions provided by each CORBA product. In some CORBA products, these proprietary extensions take the form of command-line options or entries in a configuration file. In other CORBA products, the proprietary extensions take the form of additional APIs, the use of which must be hard-coded into the source code of a server application. This is unfortunate because it hinders source-code portability of CORBA applications. The `PoaUtility` class encapsulates use of these proprietary APIs. In doing so, the `PoaUtility` class greatly enhances source-code portability of server applications. The encapsulation also has the benefit of making it trivial for the deployment model to be decided at deployment time rather than being hard-coded at development time. In this way, server applications become more flexible.

Currently, the `PoaUtility` class works "out of the box" with six CORBA products: Orbix/C++, Orbix/Java, Orbacus/C++, Orbacus/Java, TAO and omniORB. It should be easy to extend support to other CORBA products and/or languages. Section 5.5 on page 51 offers some advice for readers who are interested in porting the `PoaUtility` class to other CORBA products.

5.2 Building A POA Hierarchy

We introduce the `PoaUtility` class by showing how it is typically used to construct a POA hierarchy. Let us assume that you are writing a CORBA server with the following characteristics:

- The server implements three interfaces: `Foo`, `FooFactory` (a factory for creating `Foo` objects) and `Administration` (used to perform administration-type operations on the server).

- The server has three POAs, one for each IDL interface. By convention, each POA has a name that is the same as the name of the IDL interface associated with it. For example, servants for IDL interface `Foo` are stored in the POA called `Foo`.

- The server requires two POA managers. One POA manager, which we shall call the *core functionality* POA manager, is used to control the dispatching of requests to the Foo and FooFactory POAs. The other POA Manager, which we shall call the *admin functionality* POA manager, is used to control the dispatching of requests to the Administration POA.

The above functionality can be implemented by a class called, say, PoaHierarchy that makes use of the functionality provided by the PoaUtility class. A C++ implementation of this class is presented in Section 5.2.1, and a Java version is presented in Section 5.2.2.

5.2.1 C++ Version

The declaration of the C++ PoaHierarchy class is shown below.

```
#include "PoaUtility.h"
using namespace corbautil;
class PoaHierarchy : PoaUtility
{
public:
  PoaHierarchy(CORBA::ORB_ptr               orb,
               PoaUtility::DeploymentModel  deployModel)
                                  throw(PoaUtilityException)
  //--------
  // Accessors
  //--------
  POAManager_ptr   core_functionality()
                         { return m_core_functionality.mgr(); }
  POAManager_ptr   admin_functionality()
                         { return m_admin_functionality.mgr(); }
  POA_ptr          FooFactory()     { return m_FooFactory; }
  POA_ptr          Foo()            { return m_Foo; }
  POA_ptr          Administration() { return m_Administration; }

private:
  //--------
  // Instance variables
  //--------
  LabelledPOAManager    m_core_functionality;
  LabelledPOAManager    m_admin_functionality;
  POA_var               m_FooFactory;
  POA_var               m_Foo;
  POA_var               m_Administration;

  //--------
  // The following are not implemented
  //--------
33   PoaHierarchy();
34   PoaHierarchy(const PoaHierarchy &);
35   PoaHierarchy & operator=(const PoaHierarchy &);
36 };
```

The following points should be noted:

- The `PoaUtility.h` file (line 1) defines several types in the `corbautil` namespace (line 2).

  The `PoaHierarchy` class inherits from the `PoaUtility` class (line 3).

- The constructor of the class (line 6) takes two parameters: a reference to an ORB and an `enum` value that specifies the server's deployment model. The issue of server deployment models will be discussed in Section 5.3. If anything goes wrong in the constructor then it throws a `corbautil::PoaUtilityException`.

- The created POAs and POA Managers are stored in instance variables (lines 24 to 28). The variable for a POA Manager is of type `PoaUtility::LabelledPOAManager`, which is a class that has two accessor operations:

  ```
  const char *                       label();
  PortableServer::POAManager_ptr    mgr();
  ```

- The `PoaHierarchy` class defines accessor functions (lines 12 to 18) for the corresponding instance variables (lines 24 to 28).

All the interesting functionality of the `PoaHierarchy` class is implemented in its constructor, which is shown below. Comments follow after the code.

```
 1 #include "PoaHierarchy.h"
 2
 3 PoaHierarchy::PoaHierarchy(CORBA::ORB_ptr              orb,
 4                            PoaUtility::DeploymentModel deployModel)
 5                                       throw(PoaUtilityException)
 6   : PoaUtility(orb, deployModel)
 7 {
 8   //--------
 9   // Create the POA Managers
10   //--------
11   m_core_functionality =
12                     createPoaManager("core_functionality");
13   m_admin_functionality =
14                     createPoaManager("admin_functionality");
15
16   //--------
17   // Create the FooFactory POA
```

```
18   //--------
19   m_FooFactory = createPoa("FooFactory",
20              root(), m_core_functionality,
21              "user_id + persistent + use_active_object_map_only");
22
23   //--------
24   // Create the FooFactory/Foo POA
25   //--------
26   m_Foo = createPoa("Foo",
27              m_FooFactory, m_core_functionality,
28              "system_id + transient + unique_id + retain"
29              "+ use_active_object_map_only");
30
31   //--------
32   // Create the Administration POA
33   //--------
34   m_Administration = createPoa("Administration",
35              root(), m_admin_functionality,
36              "single_thread_model + persistent + user_id"
37              "+ use_active_object_map_only");
38 }
```

The following points should be noted:

- The constructor passes its parameters to its parent-class constructor (line 6).
- A POA manager is created by calling the inherited `createPoaManager()` operation (lines 11–14), and passing a parameter that specifies a unique *label* (name) by which the POA manager will be known to the internals of the `PoaUtility` class.
- A POA is created by calling the inherited `createPoa()` operation (lines 19–37). This operation takes four parameters. The first parameter is the name of the POA to be created. The second parameter is a reference to its parent POA. The inherited operation `root()` can be used to specify the root POA (lines 20 and 35). The third parameter is a labelled POA manager that has been previously obtained by calling `createPoaManager()`. The final parameter is a string of the form `"policy + policy + ..."`.[1] This string represents a list of policy values that should be applied when creating the POA. The names of the policy values are lowercase equivalents of the `enum` values used by the raw POA APIs, without the module prefix. For example, the `PortableServer::PERSISTENT` policy value is represented by `"persistent"`. The use of lowercase makes it easier for programmers because most code is written in lowercase.

[1] The policy values string can use any combination of whitespace, plus signs or commas as separators between policy names. Also, leading or trailing separators are ignored. This flexibility was chosen in order to facilitate developers who wish to use policy value strings that, say, are obtained from a runtime configuration file or have been generated by a code-generation tool.

- The C++ language states that a C++ compiler concatenates adjacent string literals. For example: `"good" "bye"` are concatenated to give `"goodbye"`. This can be used to split a long `"policy + policy + ..."` string over several lines (lines 28–29 and 36–37).

- If anything goes wrong then the `createPoaManager()` and `createPoa()` operations throw an exception (in the form of a `corbautil::PoaUtilityException`) that provides details of the CORBA exception thrown and the POA/POA-manager/policy-list for which the exception applies. Because of this, the `PoaHierarchy` class does *not* have a separate `try-catch` clause around each call to `createPoaManager()` and `createPoa()` to diagnose the source of the problem. Instead, the `PoaHierarchy` class lets such exceptions propagate out to the `main()` function of the application, where a single `try-catch` clause suffices. This will be illustrated in Section 5.2.1.1.

- All the useful functionality of the `PoaUtility` class is available through `public` operations. Because of this, developers are not restricted to using `PoaUtility` by sub-classing from it. For example, the constructor of the `PoaHierarchy` class could have declared a local variable of type `PoaUtility` instead.

- The `PoaUtility` class has a `root()` operation that can be called to specify the root POA as a parent of a POA being created with `createPoa()`. However, the root POA should *not* be used for storing servants. This is because a feature of `PoaUtility` is its ability to allow POAs or POA Managers to optionally listen on fixed port numbers (this is discussed in Section 5.3) and, unfortunately, the `PoaUtility` class cannot always configure the port number on which the root POA listens.

5.2.1.1 Using the POA Hierarchy in a Server Application

The code below illustrates the mainline of a server that uses the `PoaHierarchy` class. Comments follow after the code.

```
 1 CORBA::ORB_var            g_orb;
 2 PoaHierarchy *            g_poa_h;
 3 FooFactory_impl *         g_FooFactory_sv;
 4 Administration_impl *     g_Administration_sv;
 5
 6 int main(int argc, char ** argv)
 7 {
 8     PortableServer::ObjectId_var    obj_id;
 9
10     int exitStatus = 0;
11     try {
12         //--------
13         // Initialize CORBA and create the POA hierarchy
14         //--------
15         g_orb = CORBA::ORB_init(argc, argv);
```

```
        PoaUtility::DeploymentModel deployModel = ...;
        g_poa_h = new PoaHierarchy(g_orb, deployModel);

        //--------
        // Create and activate singleton servants
        //--------
        g_FooFactory_sv = new FooFactory_impl();
        obj_id = ...;
        g_poa_h->FooFactory()->activate_object_with_id(
                                 obj_id, g_FooFactory_sv);
        g_FooFactory_sv->_remove_ref();
        ... // similar code for the Administration singleton

        //--------
        // Export singleton object references
        //--------
        ...

        //--------
        // Activate POA managers and go into the event loop
        //--------
        g_poa_h->core_functionality()->activate();
        g_poa_h->admin_functionality()->activate();
        g_orb->run();
    }
    catch (const CORBA::Exception & ex) {
        cout << ex << endl;
        exitStatus = 1;
    }
    catch (corbautil::PoaUtilityException & ex) {
        cout << ex << endl;
        exitStatus = 1;
    }

    //--------
    // Tidy up and terminate
    //--------
    delete g_poa_h;
    if (!CORBA::is_nil(g_orb)) {
        try {
            g_orb->destroy();
        } catch (const CORBA::Exception & ex) {
            cout << "orb->destroy() failed: " << ex << endl;
            exitStatus = 1;
        }
```

```
61      }
62      return exitStatus;
63 }
```

The following points should be noted:

- In a CORBA server, several objects are typically accessed by different parts of application code. Such objects include the ORB, the POA hierarchy and servants for singleton interfaces. The above code has declared these as global objects (lines 1–4), and used the prefix `"g_"` to indicate that they are global variables. In your own applications, you may wish to replace these global variables with whatever is considered to be a politically correct alternative.

- Once the `CORBA::ORB` has been created (line 15), the `PoaHierarchy` object is created (line 17), passing two parameters to its constructor: the ORB and an `enum` value that specifies the server's deployment model. This `enum` value might be determined based on, say, a command-line option or an entry in a configuration file. If creation of the POA hierarchy fails then the constructor of `PoaHierarchy` throws a `corbautil::PoaUtilityException`. This is caught and printed out by a `catch` clause (lines 45–48). Note that the details of creating the POA hierarchy have been encapsulated in the `PoaHierarchy` class, so the `main()` function requires just one line of code to create the POA hierarchy.

- Having created the `PoaHierarchy` object, accessor operations are invoked on it to access a POA (line 24) or POA Managers (lines–37 38). Notice that these accessor operations do not use `_duplicate()` so there is not need for the calling code to call `CORBA::release()` on the returned reference.

- During graceful shutdown of the server, the `PoaHierarchy` object should be deleted (line 53).

5.2.2 Java Version

The Java version of the `PoaHierarchy` class is shown below.

```
 1 import org.omg.CORBA.*;
 2 import org.omg.PortableServer.*;
 3 import com.iona.corbautil.*;
 4 class PoaHierarchy
 5 {
 6    public PoaHierarchy(ORB orb, int deployModel)
 7                  throws PoaUtilityException
 8    {
 9         PoaUtility util = PoaUtility.init(orb, deployModel);
10         m_core_functionality  = util.createPoaManager(
11                                          "core_functionality");
12         m_admin_functionality = util.createPoaManager(
13                                         "admin_functionality");
14
15         m_FooFactory =  util.createPoa("FooFactory",
16                                   util.root(),
17                                   m_core_functionality,
18                                   "user_id + persistent +"
19                                   + "use_active_object_map_only");
20
21         m_Foo =  util.createPoa("Foo", m_FooFactory,
22                                   m_core_functionality,
23                                   "system_id + transient +"
24                                   + "unique_id + retain +"
25                                   + "use_active_object_map_only");
26
27         m_Administration =  util.createPoa("Administration",
28                                   util.root(),
29                                   m_admin_functionality,
30                                   "single_thread_model +"
31                                   + "persistent + user_id +"
32                                   + "use_active_object_map_only");
33       }
34
35    //--------
36    // Accessors
37    //--------
38    public POA FooFactory()       { return m_FooFactory; }
39    public POA Foo()              { return m_Foo; }
40    public POA Administration()   { return m_Administration; }
41    public POAManager core_functionality()
42                          { return m_core_functionality.mgr(); }
43    public POAManager admin_functionality()
44                          { return m_admin_functionality.mgr(); }
45
46     //--------
47     // Instance variables
48     //--------
49     private LabelledPOAManager    m_core_functionality;
50     private LabelledPOAManager    m_admin_functionality;
51     private POA                   m_FooFactory;
52     private POA                   m_Foo;
53     private POA                   m_Administration;
54 }
```

The following points should be noted:

- The `PoaUtility` class is `abstract`. Its static `init()` operation (line 9) uses Java reflection APIs to create an instance of a concrete sub-class that is suitable for use with a specific CORBA product. Further details of this are provided in Section 5.5.2.

- Because of the `abstract` nature of `PoaUtility`, the `PoaHierarchy` class does not inherit from `PoaUtility`. Instead, `PoaHierarchy` declares a variable of type `PoaUtility` (line 9) and initializes it by calling `PoaUtility.init()`. There are two parameters to `init()`: a reference to an ORB and an `int` that specifies the server's deployment model. The issue of server deployment models will be discussed in Section 5.3.

- The operations and constructor of the `PoaUtility` class throw an exception of type `PoaUtilityException`, which contains a descriptive message. Because of this, the constructor of the `PoaHierarchy` class also lists `PoaUtilityException` in its `throws` clause (line 7).

- The created POAs and POA Managers are stored in instance variables (lines 49–53). The variable for a POA Manager is of type `LabelledPOAManager`, which has two public operations:

  ```
  public String        label();
  public POAManager    mgr();
  ```

- The `PoaHierarchy` class defines accessor functions (lines 38–44) that provide read-only access to the corresponding instance variables.

- A POA manager is created by calling the `createPoaManager()` operation (lines 10–13), and passing a parameter that specifies a unique *label* (name) by which the POA manager will be known to the internals of the `PoaUtility` class.

- A POA is created by calling the `createPoa()` operation (lines 15–32). This operation takes four parameters. The first parameter is the name of the POA to be created. The second parameter is a reference to its parent POA. The operation `root()` can be used to specify the root POA (lines 16 and 28). The third parameter is a labelled POA manager that has been previously obtained by calling `createPoaManager()`. The final parameter is a string of the form `"policy + policy + ..."`.[2]

- This string represents a list of policy values that should be applied when creating the POA. The names of the policy values are lowercase equivalents of the `enum` values used by the raw POA APIs, without the module prefix. For example, the `PortableServer::PERSISTENT` policy value is represented by `"persistent"`. The use of lowercase makes it easier for programmers because most code is written in lowercase.

[2] The policy values string can use any combination of whitespace, plus signs or commas as separators between policy names. Also, leading or trailing separators are ignored. This flexibility was chosen in order to facilitate developers who wish to use policy value strings that, say, are obtained from a runtime configuration file or have been generated by a code-generation tool.

- The `PoaUtility` class has a `root()` operation that can be called to specify the root POA as a parent of a POA being created with `createPoa()`. However, the root POA should *not* be used for storing servants. This is because a feature of `PoaUtility` is its ability to allow POAs or POA Managers to optionally listen on fixed port numbers (this is discussed in Section 5.3) and, unfortunately, the `PoaUtility` class cannot always configure the port number on which the root POA listens.

5.2.2.1 Using the POA Hierarchy in a Server Application

The code below illustrates the mainline of a server that uses the `PoaHierarchy` class. Comments follow after the code.

```
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

public class Server
{
    public static ORB                    orb;
    public static PoaHierarchy           poaH;
    public static FooFactoryImpl         fooFactorySv;
    public static AdministrationImpl     administrationSv;

    public static void main(String args[])
    {
        try {
            //--------
            // Initialize the ORB and create the POA Hierarchy
            //--------
            orb = ORB.init(args, null);
            int deployModel = ...;
            poaH  = new PoaHierarchy(orb, deployModel);

            //--------
            // Create and activate singleton servants
            //--------
            fooFactorySv = new FooFactoryImpl();
            poaH.FooFactory().activate_object_with_id(
                         "FooFactory".getBytes(), fooFactorySv);
            ... // similar code for the Administration singleton

            //--------
            // Export singleton object references
            //--------
            ...

            //--------
35            // Activate POA Managers and go into the event loop
36            //--------
37            poaH.core_functionality().activate();
38            poaH.admin_functionality().activate();
39            orb.run();
40
41        } catch (PoaUtilityException ex) {
42            System.out.println(ex.getMessage());
43        } catch (Exception ex) {
44            System.out.println(ex.toString());
45        }
46
47        //--------
48        // Tidy up and terminate
49        //--------
50        if (orb != null) {
51            try {
52                orb.destroy();
53            } catch (Exception e) {
54            }
55        }
56    }
57 }
```

The following points should be noted:

- Once the `CORBA::ORB` has been created (line 17), the `PoaHierarchy` object is created, passing two parameters to its constructor: the ORB and an `int` value that species the server's deployment model (line 19). This deployment model value might be determined based on, say, a command-line option or an entry in a configuration file. If creation of the POA hierarchy fails then the constructor of `PoaHierarchy` throws a `PoaUtilityException` exception. This is caught and printed out by a catch clause (lines 41–42). Note that all the details of creating the POA hierarchy have been encapsulated in the `PoaHierarchy` class, so just one line of code (to create the `PoaHierarchy` object) is all that is required in the `main()` function.

- Having created the `PoaHierarchy` object, accessor operations are invoked on it to access a POA (line 25) or POA Managers (lines 37–38).

5.3 Server Deployment Models

The CORBA specification describes an *implementation repository*. This term is not very intuitive so it deserves an explanation. *Implementation* is the CORBA terminology for "server application", and *repository* means a persistent storage area, such as a database. Thus, *implementation repository* (commonly abbreviated to IMR) is a database that stores information about

CORBA server applications. Most of the functionality of an IMR must be implemented in a platform-specific manner. For this reason, the CORBA specification just specifies the high-level functionality that an IMR must provide, and does *not* specify any of the "look and feel" of an IMR. This means that IMRs differ widely between different CORBA products. For example, the Orbacus IMR is an executable called `imr`, while the Orbix IMR is an pair of executables: one is called `itlocator` (the locator daemon) and this is supported by `itnode_daemon` (the node daemon).

When a server application is registered with an IMR then the IMR can (re-)launch the server. When a server is deployed in this manner, any persistent object references that are exported by the server do *not* contain the server's host and port, but rather they contain the host and port of the server's IMR. When a client tries to invoke upon such an object, the client sends its first invocation to the IMR's host and port. This gives the IMR a chance to (re-)launch the server if it is not currently running and then redirect the client to the server's actual host and port.

The main benefit of using an IMR to launch a server is that the server can be re-launched automatically if it ever dies. Some IMRs offer additional benefits. For example, the Orbix IMR can launch several *replicas* of a server, in order to provide load balancing and fault tolerance. The benefits of having an IMR are desirable in many circumstances. However, there are some reasons why some people prefer to *not* deploy a server through an IMR:

1. Many people learn CORBA once piece at a time. Because of this, it is common for a person to know how to develop a CORBA server and how to run it from the command-line, but not (yet) be familiar with how to deploy a server through the IMR.

2. In many CORBA products the IMR is a single point of failure. Some organizations cannot risk deploying mission-critical applications that have single points of failure. Such organizations often prefer to deploy CORBA systems without using an IMR. An alternative is to develop applications using a CORBA product (such as Orbix) that provides a *replicated* IMR so that the single point of failure is removed.

3. Some organizations use a variety of different CORBA products. For example, perhaps one internal project is built using one CORBA product, while another internal project (perhaps in a different department) is built using a second brand name of CORBA product. Finally, the organization may have bought a pre-built CORBA server from a third-party company, and this server was built using a third brand of CORBA product. The system administrator in such an organization is faced with learning how to perform administration tasks with the IMRs of each of the three CORBA products. However, this learning curve could be reduced if some (or all) of the servers could be deployed *without* an IMR. In this case, the trade-off is to sacrifice the benefits offered by of IMR in order to simplify administration.

An orthogonal issue for server deployment is whether the server will listen on a fixed port or on an arbitrary port that is chosen by the operating system. Such arbitrary ports are often called *random*, *transient* (meaning temporary) or *ephemeral* (meaning short-lived) ports. Use of a fixed port is often desirable if the server is to be accessed by clients across a firewall, or if the server contains `PERSISTENT` POAs and is being deployed *without* an IMR. However, most CORBA products will, by default, have servers listens on random ports; this is acceptable if the server contains only `TRANSIENT` POAs, or if the server contains `PERSISTENT` POAs but is deployed through an IMR.

Unfortunately, CORBA does *not* specify the practical details of how to choose:

- Whether or not a server is deployed through an IMR.
- Whether or not a server listens on a fixed or random port.

Instead, such details are left to proprietary extensions provided by each CORBA product. In some CORBA products, these proprietary extensions take the form of command-line options or entries in a configuration file. In other CORBA products, the proprietary extensions take the form of additional APIs, the use of which must be hard-coded into the source code of a server application. This is unfortunate because it hinders source-code portability of CORBA applications. However, the proprietary APIs are typically called when creating either POAs or POA Managers. This makes it possible for the `PoaUtility` class to encapsulate the use of such proprietary APIs, and so increase source-code portability of server applications, while at the same time deferring deployment decisions until deployment time rather than prematurely deciding them during development.

5.3.1 Specifying a Server Deployment Model with `PoaUtility`

The constructor of `PoaUtility` takes a parameter called `deployModel` that is used to specify how the server is being deployed. The C++ definition of this parameter's legal values are shown below:

```
namespace corbautil {
    class PoaUtility {
    public:
        enum DeploymentModel {
            RANDOM_PORTS_NO_IMR, RANDOM_PORTS_WITH_IMR,
            FIXED_PORTS_NO_IMR, FIXED_PORTS_WITH_IMR
        };
        static DeploymentModel
                    stringToDeploymentModel(const char * model)
                                  throw(PoaUtilityException);
    ...
    };
};
```

Java does not have an `enum` type so the Java deployment models are denoted as integer constants, as shown below:

```
package com.iona.corbautil;
abstract public class PoaUtility {
    public static final int RANDOM_PORTS_NO_IMR = 0;
    public static final int RANDOM_PORTS_WITH_IMR = 1;
    public static final int FIXED_PORTS_NO_IMR = 2;
    public static final int FIXED_PORTS_WITH_IMR = 3;
    public static int stringToDeploymentModel(String model)
                throws PoaUtilityException
    ...
};
```

The four values simply indicate whether or not a server listens on random ports, and whether or not the server is deployed through the IMR. If a CORBA product requires use of proprietary APIs for any of these deployment options then the `PoaUtility` class calls the appropriate APIs. If a CORBA product does not require use of any proprietary APIs then the `PoaUtility` class simply ignores the `deployModel` parameter. In either case, the end user will still have to use the CORBA vendor's proprietary administration commands, command-line options and/or configuration file entries to set up the necessary environmental support for the chosen deployment model.

The `PoaUtility` class provides a `stringToDeploymentModel()` utility method that converts a deployment model string, such as `"RANDOM_PORTS_NO_IMR"` to the corresponding `enum`/`int` value. This utility method performs a case-insensitive string comparison, which means that lower-case strings, such as `"random_ports_no_imr"` are also acceptable. If an invalid deployment model string (for example, `"foo"`) is passed as a parameter then the method throws a `PoaUtilityException` that contains a message of the form:

```
Invalid DeploymentModel "foo"
```

Use of this utility method makes it trivial for server applications to obtain a deployment model from, say, a command-line option or an entry in a runtime configuration file. This then means that a server's deployment model can be decided at deployment time rather than being hard-coded during development.

5.3.2 Orbix Server Deployment

If deploying an Orbix server through the IMR then the `itadmin` utility must be used to register the server with the IMR. Also, whenever starting an IMR-deployable server (either from the command-line or through the IMR) then you must be consistent in specifying the same `-ORBname <name>` command-line arguments to the server.

If deploying an Orbix server so that it listens on fixed ports then you must indicate the port number used by a POA manager through a configuration variable of the form:[3]

```
<label>:iiop:addr_list
```

[3] Orbix allows individual POAs controlled by the same POA Manager to use different ports; but it also allows POAs controlled by the same POA Manager to share the same port. The `PoaUtility` class keeps administration simple by allowing the choice of port numbers to be chosen at the relatively coarse granularity of POA Managers rather than at the finer granularity of individual POAs.

where `<label>` is the *label* parameter passed to `createPoaManager()`. Some examples are shown below:

```
core_functionality:iiop:addr_list  = ["<host>:6000"];
admin_functionality:iiop:addr_list = ["<host>:6001"];
```

The `"<host>"` string should be replaced with the name of the computer on which the server is running.

5.3.3 Orbacus Server Deployment

If deploying an Orbacus server through the IMR then the `imradmin` utility must be used to register the server with the IMR. Also, if you start an IMR-deployable server from the command-line then you must specify `-ORBServerId <name>` as command-line arguments to the server.

If deploying an Orbacus server so that it listens on fixed ports then you must indicate the port number used by a POA manager through a configuration variable of the form:

```
ooc.orb.poamanager.<label>.endpoint
```

where `<label>` is the *label* parameter passed to `createPoaManager()`. Some examples are shown below:

```
ooc.orb.poamanager.core_functionality.endpoint=iiop --port 6000
ooc.orb.poamanager.admin_functionality.endpoint=iiop --port 6001
```

5.3.4 TAO Server Deployment

If deploying a TAO server through the IMR then the `tao_imr` utility must be used to register the server with the IMR. Also, whenever starting an IMR-deployable server (either from the command-line or through the IMR) then you should specify `-ORBUseIMR 1` as command-line arguments to the server.

If deploying a TAO server so that it listens on fixed ports then you must indicate the port number used by the server by specifying `-ORBEndPoint <endpoint-details>` as command-line arguments to the server. An example is shown below:

```
my_server.exe -ORBEndPoint iiop://foo.acme.com:9999
```

5.3.5 omniORB Server Deployment

OmniORB does not provide an IMR. Instead, all servers must be started manually. By default, an omniORB server listens on a random port. If you want an omniORB server to listen on a fixed port then you can indicate the port number by specifying `-ORBendPoint <endpoint-details>` as command-line arguments to the server. An example is shown below:

```
my_server.exe -ORBendPoint giop:tcp:foo.acme.com:5000
```

Alternatively, the omniORB configuration file could contain an entry like that shown below:

```
endPoint = giop:tcp:foo.acme.com:5000
```

5.4 Using Orbix-proprietary Policies

The `PoaUtility` class provides access to the Orbix-proprietary policies. For example, you can specify one of the proprietary `OBJECT_DEACTIVATION_POLICY` values (`"deliver"`, `"discard"` or `"hold"`) in the list of policies passed as a parameter to `createPoa()`.

Orbix also provides proprietary APIs for creating *work queues*,[4] which can then be associated with POAs. The `PoaUtility` class provides APIs to access this functionality. C++ code that illustrates this is presented in Section 5.4.1, and corresponding Java code is presented in Section 5.4.2.

5.4.1 C++ Version

The class declaration below shows how to make use of the Orbix-proprietary work queue mechanism.

```
 1 #include "PoaUtility.h"
 2 using namespace corbautil;
 3 class PoaHierarchy : public PoaUtility
 4 {
 5 public:
 6   PoaHierarchy(CORBA::ORB_ptr               orb,
 7                PoaUtility::DeploymentModel  deployModel)
 8                                throw(PoaUtilityException)
 9   //--------
10   // Accessors
11   //--------
12   WorkQueue_ptr    manual_wq() { return m_manual.wq(); }
13   ... // other accessors
14
15 private:
16   //--------
17   // Instance variables
18   //--------
19   LabelledOrbixWorkQueue        m_auto;
20   LabelledOrbixWorkQueue        m_manual;
21   ... // other instance variables
22 };
```

The following points should be noted:

- The created work queues are stored in instance variables (lines 19–20). The variable for a work queue is of type `corbautil::LabelledOrbixWorkQueue`, which is a class that has two public operations:

[4] The discussion in this section assumes the reader is already familiar with the concepts of work queues. If you are not familiar with work queues then you can find details in the Orbix *Programmer's Guide* and *Programmer's Reference Guide*.

```
    const char *                    label();
    IT_WorkQueue::WorkQueue_ptr     wq();
```

- An accessor for a work queue can be defined as shown in line 12.

The work queues are created and associated with POAs in the body of the constructor, as shown below:

```
 1 #include "PoaHierarchy.h"
 2
 3 PoaHierarchy::PoaHierarchy(CORBA::ORB_ptr              orb
 4                            PoaUtility::DeploymentModel deployModel)
 5                                      throw(PoaUtilityException)
 6     : PoaUtility(orb, deployModel)
 7 {
 8     //--------
 9     // Create the POA Managers
10     //--------
11     ...
12
13     //--------
14     // Create the work queues
15     //--------
16     m_auto   = createAutoWorkQueue("auto",
17                       1000, 10, 10, 10, 128);
18     m_manual = createManualWorkQueue("manual", 1000);
19
20     //--------
21     // Create the FooFactory POA
22     //--------
23     m_FooFactory = createPoa("FooFactory",
24             root(), m_core_functionality,
25             "user_id + persistent + use_active_object_map_only",
26             m_auto);
27     ...
28 }
```

The following points should be noted:

- An automatic work queue is created though the `createAutoWorkQueue()` operation (lines 16–17). The first parameter is a *label* for the work queue. The remaining parameters to this operation are `max_size`, `initial_thread_count`, `high_water_mark`, `low_water_mark` and `thread_stack_size_kb`. Most of these parameters have meanings identical to those of `create_work_queue_with_thread_stack_size()` in the `IT_WorkQueue::AutomaticWorkQueueFactory` interface. The only exception is `thread_stack_size_kb`, which specifies a stack size in kilobytes (the corresponding parameter to `create_work_queue_with_thread_stack_size()` expresses the stack size in bytes rather than kilobytes).

- A manual work queue is created through the `createManualWorkQueue()` operation (line 18). The parameters to this operation are a *label* for the work queue and `max_size`.
- The `createPoa()` operation is overloaded so it can take an extra parameter to denote a work queue that should be associated with the POA (line 26). If you wish, you can associate the same work queue with several POAs.

5.4.2 Java Version

The class declaration below shows how to make use of the Orbix-proprietary work queue mechanism.

```
 1 import org.omg.CORBA.*;
 2 import org.omg.PortableServer.*;
 3 import com.iona.corba.IT_WorkQueue.*;
 4 import com.iona.corbautil.*;
 5 class PoaHierarchy
 6 {
 7   public PoaHierarchy(ORB orb, int deployModel)
 8                   throws PoaUtilityException
 9   {
10       PoaUtilityOrbixImpl util = (PoaUtilityOrbixImpl)
11                             PoaUtility.init(orb, deployModel);
12
13       //--------
14       // Create the POA Managers
15       //--------
16       ...
17
18       //--------
19       // Create the work queues
20       //--------
21       m_auto_wq   = util.createAutoWorkQueue("auto",
22                               1000, 10, 10, 10);
23       m_manual_wq = util.createManualWorkQueue("manual", 1000);
24
25       //--------
26       // Create the "FooFactory" POA
27       //--------
28       m_FooFactory =  util.createPoa("FooFactory",
29                               util.root(),
30                               m_core_functionality,
31                               "user_id + transient + "
32                               + "use_active_object_map_only",
33                               m_auto_wq);
34        ...
35     }
36
37    //--------
38    // Accessors
39    //--------
40    public WorkQueue manual_wq()  { return m_manual_wq.wq(); }
41    ...
42
43    //--------
44    // Instance variables
45    //--------
46    private LabelledOrbixWorkQueue     m_auto_wq;
47    private LabelledOrbixWorkQueue     m_manual_wq;
48    ...
49 }
```

The following points should be noted:

- To access the Orbix-proprietary functionality, you must typecast the value returned from `PoaUtility.init(orb)` to type `PoaUtilityOrbixImpl` (lines 10–11).

  The created work queues are stored in instance variables (lines 46–47). The variable for a work queue is of type `LabelledOrbixWorkQueue`, which has two public operations:

  ```
  public String       label();
  public WorkQueue    wq();
  ```

- An accessor for a work queue can be defined as shown in line 40.

- You create an automatic work queue by calling `createAutoWorkQueue()` (lines 21–22). The first parameter to this operation is a *label* for the work queue. The remaining parameters are `max_size`, `initial_thread_count`, `high_water_mark` and `low_water_mark`. These parameters are similar to those of `create_work_queue()` in `IT_WorkQueue::AutomaticWorkQueueFactory`.

- A manual work queue is created through the `create_manual_work_queue()` operation (line 23). The parameters to this operation are a *label* for the work queue and `max_size`.

- The `createPoa()` operation is overloaded so it can take an extra parameter to denote a work queue that should be associated with the POA (line 33). If you wish, you can associate the same work queue with several POAs.

5.4.3 Configuration Values for Work Queues

If the *label* parameter passed to a work queue creation operation is an empty string then the other parameter values are used directly when creating the work queue. However, if the *label* parameter is not an empty string then the other parameters can be overridden by runtime configuration entries. If a class has an automatic work queue called `"auto"` and a manual work queue called `"manual"`) then the corresponding runtime configuration entries can be expressed as shown below:

```
auto:max_size              = 1000;
auto:initial_thread_count = 10;
auto:high_water_mark      = 10;
auto:low_water_mark       = 10;
auto:thread_stack_size_kb = 512;
manual:max_size            = 1000;
```

5.5 Porting to Other CORBA Products

5.5.1 C++ Version

The IDL-to-C++ mapping has one annoying hindrance to portability: it does not define the names of CORBA-related header files. The practical effect of this is that a developer must change `#include` directives for CORBA-related header files when porting an application to a different CORBA product. To alleviate this problem, the author has developed a collection of simple "wrapper" header files that (depending on which `#define` symbol as been defined) `#include` product-specific header files. This collection of wrapper header files, which currently supports Orbix, Orbacus, TAO and omniORB, is documented in Chapter 4 (*Portability of C++ CORBA Applications*). The `PoaUtility` class uses the portability header files so that it can `#include` CORBA header files in a portable way.

If you wish to port the `PoaUtility` class to another CORBA vendor's product then the first step is to enhance the portability wrapper header files to support that CORBA vendor's product. In practice, this should take only a few minutes of time. Having done that, the `PoaUtility` class should then compile cleanly with the other CORBA vendors' product. However, it will not (yet) take advantage of the CORBA vendor's proprietary APIs for, say, getting a POA Manager to listen on a fixed port number. To add this capability, you will need to examine the code in `PoaUtility.cxx` and add some `#if...#endif` directives as required.

The history of software development has shown that *excessive* use of `#if...#endif` directives can result in software that is difficult to read and maintain [SC]. Currently, use of `#if...#endif` directives is quite localized within the `PoaUtility` class. However, if this class is extended to support many other CORBA products in the future then proliferation of `#if...#endif` directives might prove troublesome for code readability and maintainability.

5.5.2 Java Version

The Java approach to producing product-specific implementations of an API is to define the API as either an `interface` or an `abstract class` and then use Java's reflection APIs to dynamically load an appropriate implementation. The canonical example of this for CORBA programmers is `org.omg.CORBA.ORB`, which is an `abstract class`. The static `init()` operation on this class uses reflection to create an instance of the subclass specified by the `org.omg.CORBA.ORBClass` system property.

The `PoaUtility` class uses a similar approach. It has a static operation called `init()`. This operation uses reflection to create an instance of a class using the following algorithm:

1. If the `com.iona.corbautil.PoaUtilityClass` system property exists then its value specifies the name of the class to be instantiated.

2. Otherwise, the `org.omg.CORBA.ORBClass` system property is examined.

   (a) If this property has the value `"com.iona.corba.art.artImpl.ORBImpl"` then an instance of `com.iona.corbautil.PoaUtilityOrbixImpl` is created. As its name suggests, this class is an implementation for use with Orbix. Internally, it uses Orbix-proprietary APIs that allow POAs to listen on fixed ports. It also defines additional operations (discussed in Section 5.4) that provide access to the Orbix-proprietary work queues.

   (b) If this system property has the value `"com.ooc.CORBA.ORB"` then an instance of `com.iona.corbautil.PoaUtilityOrbacusImpl` is created. As its name suggests, this class is an implementation for use with Orbacus. Internally, it uses Orbacus-proprietary APIs that allows a POA manager to listen on a fixed port.

3. Otherwise, an instance of `com.iona.corbautil.PoaUtilityPortableImpl` is created. This class uses only CORBA-compliant APIs so it is portable to other CORBA vendor products, but it does not have the ability to allow a POA manager to listen on a fixed port.

The above algorithm could be simplified to just steps 1 and 3. However, step 2 provides a better out-of-the-box experience for Orbix and Orbacus developers because the developers do not need to set up a system property in order for a server to have the ability to listen on a fixed port.

If you want to produce a version of `PoaUtility` that can take advantage of the proprietary APIs of another CORBA product then you should write a class that inherits from `com.iona.corbautil.PoaUtilityPortableImpl` and redefines whatever operations it needs to and/or adds new operations.

Chapter 6

Orbix Administration Made Simple

Orbix administration is performed through sub-commands of the `itadmin` utility. Each sub-command performs a small amount of work so you typically need to execute several `itadmin` commands to complete a useful unit of work, such as registering an Orbix server with the Implementation Repository (IMR) or updating configuration variables. However, `itadmin` has a built-in scripting language. This makes it possible to write a script that performs the entire sequence of `itadmin` commands required to carry out a task. This chapter discusses several useful task-based `itadmin` scripts:

`orbix_srv_admin` can be used to perform several of the most commonly required administration tasks associated with Orbix servers, such as registering an Orbix server with the IMR and initializing or updating configuration values for the server.

`orbix_set_config_vars` can initialize and update configuration values for an application but it does not attempt to interact with the IMR. As such, this script contains a subset of the functionality of `orbix_srv_admin`. This script is typically used with client applications or servers that are being deployed without an IMR.

`orbix_notify_service` registers a Notification Service with Orbix. When Orbix is initially configured with the `itconfigure` utility, you can choose to create *one* Notification Service. However, some organizations like to have *additional* instances of the Notification Service to increase throughput. This utility makes it easy to do this.

`orbix_ns_on_fixed_port` reconfigures the Naming Service so that it can listen on a fixed port.

6.1 Introduction

Many CORBA servers are deployed through an Implementation Repository (IMR).[1] Some administration tasks are commonly performed on such Orbix server applications. For example:

[1] Orbix uses the terminology *location domain* instead of *Implementation Repository.* An overview of the IMR/location domain is provided in Section 6.2.

- When a server is first compiled (or a pre-compiled application is installed), it should be *registered* with the IMR before it is run.

- You may wish to set (and later modify) some configuration variables for the server. For example, you can change the size of a thread pool by setting configuration variables. Likewise, you can enable active connection management for the server by setting configuration variables.

- Over time, the load on the server may increase to the point where a single server cannot handle the high load. Because of this, you may decide to re-register the application as a *replicated* server.

- Finally, you may wish to *unregister* the application if you are taking it out of deployment.

Although many CORBA servers are deployed through an IMR, some are deployed without an IMR and, of course, client applications are also deployed without an IMR. There is still some administration that is associated with such IMR-less applications. In particular, you may wish to set configuration variables for such applications.

All Orbix administration tasks are performed with various sub-commands of the `itadmin` utility. However, each individual sub-command of `itadmin` performs just a small amount of work. Because of this, you typically need to execute *several* `itadmin` commands to complete a useful unit of work, such as registering an Orbix server. However, `itadmin` has a built-in scripting language. This built-in scripting language makes it feasible to write a script that performs the entire sequence of `itadmin` commands required to carry out a task.

This chapter discusses several `itadmin` scripts. One, called `orbix_srv_admin`, performs common IMR-related administration tasks associated with servers and also initializes/updates configuration variables for a server. Another script, called `orbix_set_config_vars`, initializes or updates configuration values for an application but it does not attempt to interact with the IMR. As such, this script contains a subset of the functionality of the `orbix_srv_admin`. This script is typically used with client applications or servers that are being deployed without an IMR. The `orbix_notify_service` script makes it easy to register *several* Notification Services with the IMR. The final script, `orbix_ns_on_fixed_port`, automates the steps required to reconfigure the Naming Service so that it listens on a fixed port (which makes it firewall-friendly).

These scripts can be used to perform several of the most commonly required administration tasks associated with Orbix applications. This has the immediate benefit of simplifying administration of Orbix applications. An additional benefit is that these scripts can echo out the `itadmin` command that they execute. This allows users to see the sequence of `itadmin` commands required for various tasks. Because of this, these scripts can be used as self-teaching tools to help people more quickly master the enormous power and flexibility of `itadmin`.

6.2 What is an Implementation Repository (IMR)

The CORBA specification mentions the concept of an *implementation repository* (IMR), but does not discuss it in much detail because much of the functionality of an IMR is platform-dependent, whereas the CORBA specification focuses on platform-independent concepts.

This section explains the central concepts of an IMR. We start by explaining the CORBA concept of an IMR. We then provide a high level overview of the Orbix implementation of an IMR and finally outline the details of what Orbix stores in its IMR database.

6.2.1 The CORBA Concept of an Implementation Repository

The CORBA specification describes an *implementation repository*. This term is not very intuitive so it deserves an explanation. *Implementation* is the CORBA terminology for "server application", and *repository* means a persistent storage area, such as a database. Thus, *implementation repository* (commonly abbreviated to IMR) is a database that stores information about CORBA server applications.

The CORBA specification contains only a *partial definition* of an IMR. In particular, the specification states the high-level functionality that an IMR should provide, but the specification does *not* state how this functionality should be implemented. Neither does the specification state how the IMR should be administered. The need for a partial specification is because much of the functionality of an IMR must be implemented and administered in a platform-specific manner. For example:

- An IMR should be capable of starting and stopping a server process. Different operating systems have different ways of starting and stopping processes.

- An IMR should record details of servers, such as the command used to launch a server, and whether or not the server is currently running. Some IMRs may store this information in a database. Other IMRs might record this information in a textual file. An IMR running on an embedded device might not have access to a file system or a database and hence might record server details in, say, non-volatile RAM.

In essence, an IMR running on a mainframe would not only be *implemented* differently to an IMR running on a PC or on an embedded device, but it would also be *administered* differently too. Put simply, one CORBA vendor's IMR running on one kind of computer might have a very different "look and feel" to another CORBA vendor's IMR running on a different kind of computer. This wide variation in IMRs is the reason why the CORBA specification contains only a high-level discussion about IMRs.

6.2.2 The Orbix Implementation Repository

The Orbix IMR is implemented with the following components:

1. A database is used to record information about server applications.

2. A *locator daemon* (`itlocator`[2]) provides a CORBA server wrapper around the IMR database.

3. The `itadmin` utility is a command-line-driven CORBA client that communicates with the locator daemon in order to query and update the IMR database.

4. Whenever the locator daemon wants to start or stop a server process, it delegates the starting/stopping task to a *node daemon* (`itnode_daemon`). The node daemon also "pings" servers periodically to check if they are still alive. There should be a node daemon running on every machine on which the IMR may want to launch a server application. When a server application is registered with the IMR, one piece of the registration information specifies the node daemon (host) that should be used for running the server.

Placing some functionality of an IMR in the locator daemon and other functionality in the node daemons is a technique that is used by several CORBA vendors (although other CORBA vendors will use terminology other than *locator daemon* and *node daemon*). This separation of functionality offers one important benefit for a CORBA vendor and a different important benefit for their customers:

- The important benefit for the CORBA vendor is that the much of the platform-specific code in an IMR is concerned with starting and stopping server processes. This platform-specific code can be encapsulated in the node daemon, which enhances maintainability of the source code of the IMR.

- The important benefit for the customers is that the registration details for *all* server applications are stored in one centralized location (in the IMR database accessed via the locator daemon), *irrespective* of how many node daemons (hosts) there are in the IMR. This allows for easy centralized administration of a CORBA system.

You create an IMR by running the `itconfigure` utility. Full details of this are given in the Orbix *Administrator's Guide*. You should note that "Implementation Repository" (IMR) is CORBA terminology. It is common for CORBA vendors to use a different name for their own IMR. For example, the Orbix name for an IMR is a *location domain*. A location domain is simply the contents of the IMR database (that is, the details for all registered server applications) plus the locator daemon and its supporting node daemon(s).

Orbix does not place any restriction on how many or how few IMRs you can create and whether different IMRs run on the *same* or *different* computers. Rather, the choice on the number of IMRs installed in an organization is typically due to pragmatic considerations. For example, it is common for each developer to have his/her own "private" IMR for day-to-day development work. Another IMR might be used for system testing and yet another IMR might be used for deployed applications. An organization might find it convenient to have *several* "deployment" IMRs: perhaps a separate one for each branch or department in the organization, or perhaps one IMR for, say, payroll applications and another IMR for stock-control applications.

[2] Many executables supplied with Orbix start with the prefix `"it"`. This prefix is an acronym for *IONA Technologies*, and is used to prevent namespace pollution of executables installed on a computer.

6.2.3 What is Stored in the Implementation Repository?

When you register an Orbix-based server application with an IMR, you have to register three different kinds of entities: a *process*, an *orbname* and the *POAs* in the server's POA hierarchy.

The term *process* means the set of details required for launching a server application. This includes:

- The full path to the server executable.
- Command-line arguments to be passed to the server application.
- The current working directory. If this information is not specified then it defaults to the root directory.
- Environment variables. If none are specified then the launched server inherits the environment variables of the node daemon process that started it. If some environment variables are specified then *only* those specified variables are passed to the launched application, that is, the launched server will *not* inherit *any* environment variables at all from the node daemon.
- Startup mode. This can be either `on_demand` or `disable`. The `on_demand` mode means that the server will be launched whenever a client tries to interact with it. The `disable` mode means that the IMR will never attempt to launch the server; instead, the server must be launched by some other means, such as having a human run it from the command line.
- The node daemon (host) that is used to launch and monitor the health of the server process.
- On a UNIX machine, you can also specify a username, groupname and umask to be used when launching the server application.

A `process` is registered with the IMR through the following command:

```
itadmin process create [options] <process-name>
```

Command-line options are used to specify the details of the `process`, such as the executable name, the command-line arguments, environment variables and so on.

There is an ORB object inside each CORBA-based application, and this ORB can be given a name through the `"-ORBname <orbname>"` command-line argument when starting an Orbix application. In effect, the `orbname` is the application's name that identifies the application to the IMR. Before running an application, you can register its `orbname` with the IMR. This serves the following purposes:

- You can associate the `orbname` with a `process`, so that the IMR knows how to re-launch the application.
- When you later register the POA hierarchy of the server application, you need to associate each persistent POA with an `orbname`.

An orbname is registered with the IMR through the following command:

```
itadmin orbname create [-process <process-name>] <orb-name>
```

When an application is started, the application's `orbname` determines what set of configuration information will be used by the Orbix runtime system. Application-level code can use Orbix-proprietary APIs to obtain application-level configuration information from the same place, or alternatively an application can obtain its runtime configuration information from somewhere else.

A CORBA server application uses a collection of POAs to manage its objects. POAs in an application are arranged in a hierarchy and each POA is either *transient* or *persistent*:

- A transient POA is *not* associated with an `orbname`, and it needs to be registered with the IMR only if it is the parent (or ancestor) of a persistent POA.
- A persistent POA must be registered with the IMR. A persistent POA is associated with either one `orbname` or a *list* of `orbnames` (and a load-balancing strategy). Associating a POA with a *list* of `orbnames` means that the POA is replicated. In Orbix, replication is provided at the granularity of individual POAs, but you typically replicate *all* the POAs in a server so it looks like server replication.

A `poa` is registered with the IMR through the following command:

```
itadmin poa create [options] <hierarchical-poa-name>
```

Command-line options to the command are used to specify the details of the `poa`, such as whether it is transient or persistent, which (list of) `orbname`(s) it is associated with and, for a replicated poa, its load-balancing strategy.

When you compile or install a pre-compiled Orbix-based server application, you should register its `process`, `orbname` and `poa` hierarchy with an IMR. This registration occurs just once, because the IMR stores all the details in its database.

The registration details of a server are said to be *static* because they rarely change. The IMR also records *dynamic* (that is, frequently updated) information such as the host and ports of servers that are currently running. It uses this information when it needs to redirect a client to an appropriate server and also so that it can re-launch a server on an as-needed basis.

6.3 Building Task-based Utilities with `itadmin`

When you register an Orbix-based server application with an IMR (locator daemon), you typically execute a series of `itadmin` commands to:

- Register a `process`.
- Register an `orbname` and associate it with the `process`.
- Register the `poa` hierarchy of the server, and associate each `poa` with the `orbname`.

- Optionally, set up configuration variables for the server (discussed later).

It is not uncommon for a user to execute a series of 10 or more `itadmin` commands in order to register a server application. Executing this many `itadmin` commands is tedious and somewhat error-prone. However, the task can be made much simpler by writing a higher-level utility that encapsulates the lower-level `itadmin` commands. Before discussing such utilities, it is instructive to discuss *how* the utilities were built, because readers may wish to write their own utilities using similar techniques.

The following is a list of three ingredients vital to building task-based utilities on top of `itadmin`:

1. It is important to know that `itadmin` has a built-in interpreter for the Tcl (pronounced "tickle") scripting language. Tcl scripts have a `".tcl"` file extension. If you write a Tcl script called `foo.tcl` then you can execute it with `itadmin` as shown below:

   ```
   itadmin foo.tcl ...
   ```

2. Tcl programmers often employ the following useful technique. Tcl has a `source` command that is used to read and execute the Tcl commands in another file. This means that a programmer does not have to write a configuration-file parser for a Tcl-based application. Instead, the configuration information can be stored as a collection of Tcl assignment statements in one file and this file can then be `source`d into the main Tcl application. The utilities discussed in this chapter use this technique to read a *description* file that (using Tcl syntax) describes an Orbix server application. Section 6.3.1 provides a brief overview of enough Tcl syntax to make people comfortable editing these Tcl-based description files.

3. Once you have written an `itadmin` script called, say, `foo.tcl`, people *could* run the script as:

   ```
   itadmin /full/path/to/foo.tcl ...
   ```

   However, this is somewhat awkward because users need to type the full path to the script they want to run. Instead, it is a good idea to write a simple Windows batch file (or a UNIX shell script) wrapper that, internally, executes `itadmin` with the appropriate command-line arguments. Once this batch file (or shell script) has been put into a `bin` directory, users can then run the utility by simply typing `"foo ..."`.

6.3.1 Overview of Tcl Syntax

Assignments to variables are made with the `set` command. This is illustrated in the examples below:

```
set a "hello, world"
set b hello
set c 42;                # this is a comment
```

Commands can be terminated by a newline character or optionally by a semicolon. Comments start with # and continue to the end of the line. If you have a comment on the same line as another statement then you *must* terminate the statement with a semicolon before starting the comment. If a command is too long to fit on one line then you can use a backslash immediately followed by a newline character in order to continue the command on the next line.

Tcl treats all values as strings. In the above example, the value `42` assigned to variable `c` looks like an integer but it really is a string.[3] Quotes are needed for the `"hello, world"` string because that string contains a space. However, the use of quotes is optional if a string does not contain any spaces. For example, there are no quotes around the strings `hello` or `42` in the above assignment statements.

The value of a variable is obtained by prefixing the variable's name with `$`. For example:

```
set a hello
set b goodbye
set c "$a and $b"
```

The above example also illustrates that the use of `$` works inside quoted strings.

The backslash character is used as an escape character in Tcl. Because of this, you must use two backslashes if you want to embed a backslash in a string. This is often used to express Windows filenames, as shown in the example below:

```
set filename "C:\\temp\\foo.txt"
```

Actually, Tcl is quite happy using forward slashes in Windows filenames, so the above example could be written more conveniently as shown below:

```
set filename "C:/temp/foo.txt"
```

Tcl uses round brackets to indicate that a variable is an "array". Tcl's concept of an array is more like the concept of a *lookup table* or *map* in other programming languages. The `env()` array is used to access environment variables:

```
set foo(a) hello
set foo(b) goodbye
set c "$foo(a) and $foo(b)"
set x $env(PATH)
```

If you want to use the return value of a function call as a parameter to another command then you surround the function call with square brackets, as shown in the following example:

```
set x [factorial 5]
```

Lists (of strings) are enclosed in braces. For example:

[3] Tcl automatically converts strings to numbers on an as-needed basis in order to perform arithmetic operations, so treating all values as strings does not limit Tcl's power. Once the string-to-integer conversion has been performed, the integer value is cached so future accesses of the value are faster.

```
set colors {red green "light blue"}
```

Using `$` to access the value of a variable does not work inside braces. For example, the following does not work as you might hope:

```
set color1 "red"
set color2 "green"
set color3 "light blue"
set colors {$color1 $color2 $color3}
```

Instead, you can use the list command to build a list:

```
set color1 "red"
set color2 "green"
set color3 "light blue"
set colors [list $color1 $color2 $color3]
```

Obviously, there is a lot more to Tcl, such as while-loops, if-then-else statements and procedures. However, the above discussion of Tcl syntax is sufficient for readers to understand how to edit description files that are used with `orbix_srv_admin` and `orbix_set_config_vars`.

6.4 Using `orbix_srv_admin`

The `orbix_srv_admin` utility is a task-based `itadmin` script that makes it easy to perform common administration tasks associated with Orbix servers. For example, `orbix_srv_admin` can be used to register and unregister Orbix servers. It can also be used to start and stop a registered server and to update configuration variables for a registered server. You can get a usage statement by running `orbix_srv_admin` with the `-h` option. The usage statement looks like that shown below:

```
usage: orbix_srv_admin [options] file.des
options are:
    -s          Silent mode
    -n          Do not execute commands. Just show them
    -h          Print this usage statement
    -create     Create a starting-point description file
    -register   Register the application's details
    -unregister Unregister the application's details
    -start      Start the server
    -stop       Stop the server
    -set_vars   Set or update the server's configuration variables
    -launch_cmd Print the server's launch command
```

As can be seen from the usage statement, `orbix_srv_admin` takes a `.des` (description) file as a command-line argument. This file uses Tcl syntax to *describe* an Orbix server application. You do not need to write such a description file by hand, because `orbix_srv_admin` can create a starting-point description file for you. Let us assume that you want to perform administration tasks on an Orbix-based server that is part of a payroll-processing system. You can create a starting-point description file called `payroll.des` by running the following command:

```
orbix_srv_admin -create payroll.des
```

The first part of the generated description file, shown below, specifies how to launch a server application.

```
set orb_name          "acme.uk.payroll"
set process_name      $orb_name
set root_poa_name     $orb_name
set description       ""
set startup_mode      "on_demand"; # on_demand or disable
#--------
# File names on Windows can be expressed as "C:/full/path/to/file"
# or as "C:\full\path\to\file"
#--------
set executable        "/full/path/to/executable"
set cmd_line_args     "x y z"
set working_directory "/full/path/to/current/working/directory"
#--------
# If "env_var_list" is set to an empty list "{}" then the launched
# application inherits all the environment variables from the node
# daemon that launched it. If "env_var_list" is not empty then the
# launched application does not inherit any environment variables
# from the node daemon, and instead has only the environment
# variables listed in "env_var_list".
#--------
set env_var_list      [list \
            "PATH=$env(PATH)" \
            "CLASSPATH=$env(CLASSPATH)" \
            "SYSTEMROOT=$env(SYSTEMROOT)" \
            "IT_CONFIG_DOMAINS_DIR=$env(IT_CONFIG_DOMAINS_DIR)" \
            "IT_DOMAIN_NAME=$env(IT_DOMAIN_NAME)" \
            "IT_LICENSE_FILE=$env(IT_LICENSE_FILE)" \
        ]
#--------
# If "node_daemon_list" contains several entries then the server
# will be registered as a replicated server and "load_balancer"
# specifies the load-balancing policy (random, round_robin or active)
# to be used. If "node_daemon_list" contains just one entry then
# "load_balancer" is ignored.
#--------
set node_daemon_list  {pizza}
set load_balancer     "random"; # random, round_robin or active
```

```
#--------
# UNIX-specific. These entries are ignored on Windows
#--------
set group           "nobody"
set user            "fred"
set umask           "755"
```

As you can see, the description file contains a sequence of Tcl assignment statements (an overview of Tcl syntax is provided in Section 6.3.1). Once you are familiar with the concepts behind the information stored in the IMR (see Section 6.2.3), then most of the variable names and values are quite intuitive. The comments in the generated file explain most of the subtle points.

The most important point to note that is not discussed in the comments is that although `process_name`, `orb_name` and `root_poa_name` can have different values, it is typically a good idea to use the same name for all of them. This is because the `process_name`, `orb_name` and `root_poa_name` are closely related concepts; using the same name for each makes it easy to see their relationship to each other. The name chosen can have embedded periods, such as `"acme.uk.payroll"` shown in the above example. It is a good idea to use an `orb_name` with embedded periods because when the application is launched, Orbix will first search for runtime configuration information in the `acme.uk.payroll` configuration scope. If any configuration values are missing from this scope then Orbix starts searching in surrounding scopes—in the `acme.uk` scope, then the `acme` scope and finally the global scope—to find runtime configuration values. In essence, variables in outer scopes supply *default* configuration values for applications, and these values can be selectively overridden by redefining the variables in inner scopes. This is a very useful feature of Orbix because related applications tend to have similar configuration values. Rather than specifying dozens of configuration values for each and every Orbix application, it is possible to specify the “common” values in an outer scope and then specify *only* the different values in inner scopes specific to each application.

The next entry in the description file gives details of the POA hierarchy in the server application:

```
#--------
# Each line in poa_hierarchy is a pair of the form:
#       lifespan full/path/to/poa-name
# where lifespan can be one of: transient or persistent
#--------
set poa_hierarchy {
    persistent  FooFactory
    persistent  FooFactory/Foo
    transient   FooFactory/Foo/FooIterator
    persistent  Administration
}
```

The next entry specifies *runtime* configuration variables that should be set in the `orb_name` configuration scope.

```
#--------
# Each line in runtime_config_variables is a triplet of the form:
#       type name value
# The type can be one of: long, bool, list, string or double
# list values are comma-separated strings
# bool values can be: true or false
#--------
set runtime_config_variables {
    string  plugins:local_log_stream:filename "server.log"
    list    event_log:filters                 "*=WARN+ERR+FATAL"
    long    thread_pool:high_water_mark       "5"
    long    thread_pool:low_water_mark        "5"
    long    thread_pool:initial_threads       "5"
    long    thread_pool:max_queue_size        "500"
}
```

Section 6.9 discusses useful runtime configuration variables that you might wish to use in a deployed application.

If you are registering the server as a *replicated* server (that is, with multiple node daemons) then you can optionally also specify runtime configuration variables for each replica with Tcl variables called `runtime_config_variables_replica_<number>`. The generated starting-point description file contains sample details for three replicas:

```
set runtime_config_variables_replica_1 {
    string  plugins:local_log_stream:filename "server.replica_1.log"
}
set runtime_config_variables_replica_2 {
    string  plugins:local_log_stream:filename "server.replica_2.log"
}
set runtime_config_variables_replica_3 {
    string  plugins:local_log_stream:filename "server.replica_3.log"
}
```

You should modify the starting-point description file so that it contains details appropriate for the `payroll` server application. Once you have done that you can register the payroll server with the command:

```
orbix_srv_admin -register payroll.des
```

When you run that command, `orbix_srv_admin` executes all the individual commands required to register the server.

You can start the registered server with the command:

```
orbix_srv_admin -start payroll.des
```

If the server was registered as a replicated server then the above command causes *all* the replicas to be started.

You can stop the registered server with the command:

```
orbix_srv_admin -stop payroll.des
```

If the server was registered as a replicated server then the above command stops *all* the replicas.

If you want to modify the runtime configuration variables for the server then you should modify the `runtime_config_variables` entry in `payroll.des` and then run the command:

```
orbix_srv_admin -set_vars payroll.des
```

Note that an already-running server will not notice the updated runtime configuration variables. Instead, you will have to stop and then re-start the server for the new variables to take effect.

The `-unregister` option of `orbix_srv_admin` can be used to unregister the application. This is useful if you have already deployed a server and want to make changes to how it is deployed, for example, you want to change its command-line arguments or turn it from being an un-replicated server to a replicated one. In such cases, you should unregister the server, then make appropriate changes to `payroll.des` and finally re-register the server.

6.5 Using `orbix_set_config_vars`

The `orbix_set_config_vars` utility contains just a small subset of the functionality of `orbix_srv_admin`.

You can get a usage statement by running `orbix_set_config_vars` with the `-h` option. The usage statement looks like that shown below:

```
usage: orbix_set_config_vars [options] file.des
options are:
    -s          Silent mode
    -n          Do not execute commands. Just show them
    -h          Print this usage statement
    -create     Create a starting-point description file
```

As can be seen from the usage statement, `orbix_set_config_vars` takes a `.des` (description) file as a command-line argument. This file uses Tcl syntax to *describe* the configuration variables required for an Orbix application. You do not need to write such a description file by hand, because `orbix_set_config_vars` can create a starting-point description file for you.

Let us assume that you want to manipulate configuration variables for Orbix-based client application that is part of a payroll-processing system. You can create a starting-point description file called `payroll_client.des` by running the following command:

```
orbix_set_config_vars -create payroll_client.des
```

The generated description file is shown below.

```
#--------
# The orb_name specifies the configuration scope where configuration
# variables will be set.
#--------
```

```
set orb_name          "acme.uk.payroll_client"

#--------
# Each line in runtime_config_variables is a triplet of the form:
#        type name value
# The type can be one of: long, bool, list, string or double
# list values are comma-separated strings
# bool values can be: true or false
#--------
set runtime_config_variables {
    string  plugins:local_log_stream:filename   "server.log"
    list    event_log:filters                   "*=WARN+ERR+FATAL"
    long    thread_pool:high_water_mark         "5"
    long    thread_pool:low_water_mark          "5"
    long    thread_pool:initial_threads         "5"
    long    thread_pool:max_queue_size          "500"
}
```

This file contains just two variables: `orb_name` and `runtime_config_variables`. The latter specifies the runtime configuration variables that should be set, while `orb_name` specifies the configuration scope in which the variables should be set. If these variables look familiar then that is because description file used by `orbix_set_config_vars` is a subset of the description file used by `orbix_srv_admin`.

You should modify the starting-point description file so that it contains details appropriate for the `payroll_client` application. Then you can rerun `orbix_set_config_vars` command (without the `-create` option) on the description file:

```
orbix_set_config_vars payroll_client.des
```

Running the above command causes the Orbix configuration domain to be updated with the variables specified in the description file.

6.6 Using `orbix_notify_service`

When you run the `itconfigure` GUI to set up an Orbix environment, you can choose to have *one* instance of a Notification Service running. A single Notification Service is adequate for many organizations. However, some organizations that make heavy use of a Notification Service may find that a single instance limits scalability, and so may wish to have *several* instances of the Notification Service running, with some consumer and supplier applications configured to use one Notification Service, other consumer and supplier applications configured to use another Notification Service, and so on. The `orbix_notify_service` utility is useful in such situations. It is based on—and has a very similar "look and feel" to—`orbix_srv_admin` (Section 6.4).

All the command-line options of the `orbix_notify_service` utility are identical to those of `orbix_srv_admin` (Section 6.4), but have a behavior that is tailored to the requirements of the Notification Service.

You can create a starting-point description file called `foo.des` by running the following command:

```
orbix_notify_service -create foo.des
```

The first part of the generated description file, shown below, contains a few variable setting that you might wish to change:

```
set group             "nobody"
set user              $::tcl_platform(user); # current user
set umask             "755"
set unique_name_part  "[exec hostname]"
set orb_name          "iona_services.notify_$unique_name_part"
set named_key         "NotificationService_$unique_name_part"
```

The following points should be noted:

- The `group`, `user` and `umask` entries are specific to UNIX, and are ignored on Windows. The expression `$::tcl_platform(user)` evaluates to the username of the person running the `orbix_notify_service` utility.
- It is important to ensure that there are no name clashes between Notification Services installed within the same Orbix domain. For this reason, the `unique_name_part` entry must be set to something unique for each Notification Service. This value of this variable is concatenated with various other entries in the `.des` file. The default value of this entry is `[exec hostname]`, which is the Tcl syntax for denoting the result of executing the `hostname` shell command. This is a convenient default if you want to have a separate instance of the Notification Service installed on different machines.
- By convention, both CORBA services (for example, the Naming Service, Transaction Service and Notification Services) and Orbix-proprietary services (for example, the Location Service, node daemons and configuration repository) have `orbname`s within the `iona_services` scope. The default value for the `orb_name` entry respects this convention, but you can change it if you wish.
- Orbix uses the terminology *named key* to mean a *name* $\rightarrow$ *stringified-IOR* mapping that is stored in the `corbaloc` server functionality of the Orbix IMR. The `itadmin` sub-commands to perform administration of named keys is documented in the Orbix *Administrator's Guide*. The `orbix_notify_service` utility creates a named key mapping for the Notification Service instance, and the `named_key` entry specifies the *name* part of the *name* $\rightarrow$ *stringified-IOR* mapping.

The remaining entries in the generated `.des` file should be left alone, as they have been preset to suitable values for a Notification Service.

Once you are happy with the contents of the `.des` file, you can then register the Notification Service by running `orbix_notify_service` with the `-register` option. Doing this causes the server to be registered similarly to the way that `orbix_srv_admin` registers a server, but with the following "added value" steps:

- It runs `itnotify` (the executable for the Notification Service) with the `prepare` flag. This causes the Notification Service to generate a stringified IOR into a file in the `var` sub-directory of your Orbix domain.
- The generated stringified IOR is then registered as a *named key*.

If you run `orbix_notify_service` with the `-unregister` option then it unregisters the Notification Service in a manner similar to how `orbix_srv_admin` unregisters a server, but it also removes the stringified IOR file in the `var` sub-directory of your Orbix domain and deletes the *named key* entry.

The other command-line options (`-start`, `-stop`, `-set_vars` and `-launch_cmd`) work with `orbix_notify_service` in the same way that they work with `orbix_srv_admin`.

6.7 Using `orbix_ns_on_fixed_port`

If you have a Naming Service in an IMR-based Orbix environment then `itconfigure` configures the Naming Service so that it listens on a random port each time it is started.[4] Having the Naming Service listen on a random port is acceptable for a great many deployed CORBA systems. However, the use of a random port can cause communication problems if the Naming Service is to be accessed through a firewall; this is because firewall routers normally allow TCP/IP communication only to a collection of specified, *fixed* ports. There are two possible approaches to making the Naming Service accessible through a firewall.

One approach is to use the Orbix firewall proxy service that is provided with Orbix 6. The firewall proxy service acts as a delegation server: it listens on a fixed port (so it is firewall-friendly) and delegates all received messages to backend servers, such as the Naming Service. Interested readers should look in the Orbix *Administrator's Guide* for details about this service.

Some people may wish to not use the Orbix firewall proxy service, or they may be using an older version of Orbix that does not provide that service (the firewall proxy service was introduced in Orbix 6.0). Thankfully, there is a second approach to making the Naming Service firewall-friendly. This involves reconfiguring the Naming Service so that it listens on a fixed port. A Knowledge Base article[5] available from the IONA web site explains the steps required to reconfigure the Naming Service so that it listens on a fixed port. The `orbix_ns_on_fixed_port` utility automates most of the steps discussed in the Knowledge Base article, thereby allowing you to complete the reconfiguration faster. Note that `orbix_ns_on_fixed_port` is intended to be used with *non*-replicated Naming Services; it has *not* been designed to reconfigure a replicated Naming Service.

Before using `orbix_ns_on_fixed_port`, it is strongly recommended that you make a backup of the Orbix configuration files and the `var` directory of your Orbix environment. The

[4] The IOR of the Naming Service will actually contain the port of the IMR, so a client's first invocation upon the Naming Service will actually be sent to the IMR. The IMR will then redirect the client to Naming Service's actual port.

[5] `http://www.iona.com/support/articles/3757.360.xml`

configuration files are held in the directory indicated by the `IT_CONFIG_DOMAINS_DIR` environment variable, and the `var` directory is indicated by running the following command:

```
itadmin variable show o2k.data.root
```

The reason for performing a backup is that the `orbix_ns_on_fixed_port` utility makes some very important changes to your Orbix environment. This utility is believed to be bug-free, but if it misbehaves then it could have the side effect of leaving the Naming Service in an unworking state. In such a case, the easiest way to undo the damage is to restore the configuration files and `var` directory from their backups. The Orbix *Administrator's Guide* contains information on two different ways to backup individual databases of the Orbix services. However, the simplest way to perform a backup is as follows:

- Stop the Orbix services by running the `stop_<domain>_services` script, where you replace `<domain>` with the name of your Orbix domain. The value of the `IT_DOMAIN_NAME` environment variable indicates the name of your Orbix domain.
- Use the UNIX `tar` utility or, say, `Winzip` on Windows to make copies of the configuration files and the `var` directory.
- Restart the Orbix services by running the `start_<domain>_services` script.

You can get a usage statement of `orbix_ns_on_fixed_port` by running it with the `-h` option. The usage statement looks like that shown below:

```
usage: orbix_ns_on_fixed_port [options]
options are:
    -s           Silent mode
    -n           Do not execute commands. Just show them
    -l <host>    Local host name           (example: foo)
    -f <host>    Fully-qualified host name (example: foo.bar.com)
    -port <port> Fixed port
    -h           Print this usage statement
```

When you run the utility, you *must* specify the `-port`, `-l` and `-f` options. For example:

```
orbix_ns_on_fixed_port -port 5000 -l foo -f foo.bar.com
```

The `-port` option is used to specify the fixed port on which the Naming Service is to listen. The `-l` option specifies the local hostname of the machine. This is used to find the correct configuration scope for the Naming Service.[6] The `-f` option specifies the hostname or IP address that will be embedded in the IOR for the Naming Service. You typically use this option to specify a fully-qualified hostname, such as `foo.bar.com`, but if you want you could specify an IP address or even just the local hostname.

[6] The `iona_services.naming.<local-host-name>` scope stores configuration information for the Naming Service that runs on the specified host.

When you run orbix_ns_on_fixed_port, it performs *most* of the reconfiguration steps required for the Naming Service to listen on the specified fixed port. However, there are a few steps remaining that must be completed manually. The orbix_ns_on_fixed_port utility prints these steps when it terminates:

```
To finish, you have to do the following...
        1. Ensure that "start_<domain>_services" starts itnaming
        2. Run "stop_<domain>_services"
           (ignore any error messages regarding the Naming Service)
        3. Run "start_<domain>_services"
```

To expand on those notes a bit, you should:

1. Edit the start_<domain>_services UNIX shell script (or Windows batch file), where you replace <domain> with the name of your Orbix domain. Depending on what options you choose when creating your Orbix domain, that file may or may not contain a statement to start itnaming, which is the Naming Service executable. If the file does *not* start itnaming then you should add the following statement at the end of the file:

   ```
   itnaming -background run -ORBname iona_services.naming.<host>
   ```

   Replace <host> with the same host name that you used with the -l option when running orbix_ns_on_fixed_port.

2. Then run stop_<domain>_services, where you replace <domain> with the name of your Orbix domain. When you run this script, it will complain that it cannot kill the Naming Service. It is safe to ignore this error, because the Naming Service is already dead.

3. Finally, run start_<domain>_services, where you replace <domain> with the name of your Orbix domain. This script restarts the Orbix services. As part of doing this, the Naming Service will listen on the desired port.

6.8 Using the Utilities to Learn About itadmin

By default, the utilities discussed in this chapter echo all the itadmin commands that they execute. The -s (silent) command-line option instructs these utilities to *not* echo the itadmin commands that they execute.

The default behavior of echoing each itadmin command as it is executed is a very useful feature because it means that the utilities can be used as a self-teaching tool to learn more about itadmin. For example, you can use orbix_srv_admin to register a server and then look at the echoed itadmin commands in order to understand the steps involved. You can then use the itadmin commands discussed below to query the IMR in order to see the relationships between the created process, orbname and poa entities in the IMR database.

If you are going to type a lot of `itadmin` commands interactively then you will quickly tire of repeatedly typing `"itadmin <name-of-command> <arguments>"`. Instead, if you run `itadmin` without any command-line arguments then you will be put into a Tcl shell. Within this shell you can type commands without having to give the `"itadmin"` prefix. Some useful commands to type within this shell are shown below:

```
process list
process list -active
process show <process-name>
orbname list
orbname list -active
orbname show <orbname>
poa list
poa list -active
poa show <poa-name>
scope list [<scope-name>]
scope show <scope-name>
```

In general, the `list` version of commands prints a list of all the processes/orbnames/poas/scopes. The `"list -active"` version lists only those processes/orbnames/poas that are currently active (running). The `show` version of a command displays details of the specified process/orbname/poa/scope. You can find details of all the available `itadmin` commands in the Orbix *Administrator's Guide*.

6.9 Useful Configuration Variables

The Orbix runtime system makes extensive use of runtime configuration information. A complete list of all the runtime configuration variables can be found in an appendix of the Orbix *Administrator's Guide*. Most of the runtime configuration variables are for low-level parts of Orbix that users typically do not care about. However, *some* of the runtime configuration variables are of a higher-level nature and can be of interest when deploying an application. It is those interesting configuration variables that this section focuses on.

If you want to use some of these runtime configuration variables in a *server* application then a simple way to do so is to list the variables in the `runtime_config_variables` entry in an `orbix_srv_admin` description file, for example, `payroll.des`, and then run:

```
orbix_srv_admin -register payroll.des
```

Alternatively, if you want to modify the runtime configuration variables of a server that is already registered then you can run:

```
orbix_srv_admin -set_vars payroll.des
```

If you want to use some of these runtime configuration variables in a *client* application then a simple way to do so is to list the variables in the `runtime_config_variables` entry in an `orbix_set_config_vars` description file., for example, `payroll_client.des`, and then run:

```
orbix_set_config_vars payroll_client.des
```

If an application is currently running when you update its configuration variables then you will have to stop it and re-start it for it to pick up the new runtime configuration values.

6.9.1 Size of the Thread Pool in Multi-threaded Servers

A server application has a pool of threads that are used for servicing incoming requests for servants in `ORB_CTRL_MODEL` (that is, multi-threaded) POAs. The thread pool has an initial size and can grow and shrink, subject to constraints expressed in some configuration variables.

The default configuration values for the thread pool means it initially contains 5 threads and it can to grow infinitely large (under a heavy load). These default values are dangerous because although threads are cheap, they *do* consume some resources and an Orbix server that continuously adds threads to the pool will eventually run out of memory. Instead, you can safeguard the health of your server by specifying an upper bound on the size of the thread pool. You can do this as shown in the example below:

```
set runtime_config_variables {
    long    thread_pool:high_water_mark   "10"
    long    thread_pool:low_water_mark    "10"
    long    thread_pool:initial_threads   "10"
    long    thread_pool:max_queue_size    "500"
}
```

The above example configures the thread pool to be a fixed size, that it, it cannot grow or shrink. In general though, you may want the thread pool to be able to grow slightly under a heavy load and to later shrink when the server is under a lighter load. To do that, you need to understand the meaning of the configuration variables used. The `initial_threads` variable specifies the initial number of threads in the thread pool. The `high_water_mark` specifies the pool's maximum size, and the `low_water_mark` specifies the pool's minimum size. If the all the threads in the pool are busy and the pool cannot grow then an incoming request will be put into a queue where it will wait until one of the threads in the thread pool becomes free. The maximum size of this queue is specified by `max_queue_size`.

The default value of `high_water_mark` is -1, which allows the thread pool to grown infinitely big (or until the server runs out of memory). The default value of `low_water_mark` is -1, which means that the thread pool will never shrink when the load on the server decreases. The default value of `initial_threads` is 5, which means that the thread pool initially contains 5 threads. The default value of `max_queue_size` is -1, which allows the queue to grow infinitely big (or until the server runs out of memory).

6.9.1.1 Warning for Java Users

In the C++ version of Orbix, all the threads in the thread-pool are available for servicing incoming requests. Unfortunately, the Java version of Orbix uses threads in the thread-pool not just to service incoming requests, but also for other Orbix-internal purposes, such as reading incoming messages from socket connections. This means that not all the threads in the thread pool are available for servicing requests. It also means that if you put a limit on the size of the thread pool in an Orbix/Java server then the server may hang under some circumstances. This is because it may use up all the threads in the thread pool for monitoring socket connections and not have any threads left over to service incoming requests or monitor a newly opened socket connection. For this reason, you are strongly advised to *not* impose an upper limit on the size of the thread pool in an Orbix/Java application. IONA plans to fix this mis-feature of Orbix/Java in a future release.

6.9.2 Specifying References for CORBA Services

When an application calls `resolve_initial_references("NameService")`, Orbix finds the Naming Service by retrieving a stringified object reference from a runtime configuration variable and calling `string_to_object()` on this stringified object reference. Orbix uses this technique to locate *any* service requested in a call to `resolve_initial_references()`. Sometimes, there may be *several* Naming Services installed in an organization. If your application is currently pointing to the "wrong" Naming Service (or some other CORBA Service such as, say, the Trading Service) then you can fix this by modifying your application's configuration settings. You do this by obtaining a stringified object reference (or a corbaloc reference) for the "correct" Naming Service and then set a configuration entry as shown below:

```
set runtime_config_variables {
    string  initial_references:NameService:reference     "IOR:..."
    string  initial_references:TradingService:reference  "IOR:..."
}
```

6.9.3 Loading Extra Plugins

Orbix is built using a micro-kernel, plugin architecture, which means that the kernel (core) of Orbix is very small but it knows low to load additional functionality through "plugins". A plugin is implemented as a shared-library/DLL for C++ Orbix, or as a Java class for Java Orbix.

The plugins that are loaded into an application by default are shown below:

```
set runtime_config_variables {
    list orb_plugins "local_log_stream,iiop_profile,giop,iiop"
}
```

The above example shows that commas separate the values in a list. The above example also illustrates that even *basic* functionality, such as the on-the-wire protocol stack (GIOP and IIOP), is implemented via plugins rather than being hard-coded into the core of Orbix. This flexibility is important because it makes it feasible to plugin a *different* on-the-wire protocol, if the need ever arises.

Some plugins that you might need to load for some applications include:

`local_log_stream` This plugin is loaded by default. It allows Orbix-generated messages to be logged to standard error or to a file. This is discussed more in Section 6.9.4.

`system_log_stream` If you load this plugin then Orbix-generated messages will be written to *syslog* on UNIX machines, or to the *event log* on Windows machines. This is discussed more in Section 6.9.4.

`ots` You will need to load this plugin if your application makes use of the Object Transaction Service (OTS) CORBA Service. See the Orbix *CORBA OTS Programmer's Guide* for more details.

`shmiop` If a client and server *both* load this plugin *and* if the client and server are running on the same computer then they can communicate through shared memory rather than through IIOP.

`iiop_tls` TLS is the new name for Secure Sockets Layer (SSL) which is an encryption standard that is popular on the Internet. If you want secure communications between clients and servers then your applications should load this plugin. See the Orbix *SSL/TLS Guide* for more details.

`giop_snoop` This plugin uses an *interceptor* to examine all the on-the-wire messages that an application sends or receives. This plugin can print out diagnostics for each message, which can be a useful debugging aid. See Section 6.9.4 for more details.

Each plugin that is loaded may have its own configuration variables. By convention, a plugin called `1<name>` uses configuration variables with names that have `"plugins:<name>:"` as a prefix. For example, configuration variables related to IIOP use `"plugins:iiop:"` as a prefix on their names. The use of such prefixes avoids name clashes of configuration variables. Orbix uses the terminology *namespace* to refer to a prefix that contains a `":"` character.

You may be curious about how C++ Orbix knows *which* shared-library/DLL to load for a plugin, or how Java Orbix knows *which* Java class to load for a plugin. The answer is that C++ Orbix uses the variable `"plugins:<name>:shlib_name"` to determine the name of the shared library , and that Java Orbix uses the variable `"plugins:<name>:ClassName"` to determine the name of the Java class.

6.9.4 Controlling Diagnostic Messages

6.9.4.1 Controlling the Destination of Diagnostics

The CORBA specification does not standardize how a CORBA vendor product should generate diagnostic messages. Because of this, CORBA products generate diagnostic output in a product-specific way. Whenever Orbix generates a diagnostic message, it logs it by invoking a logging operation on *all* the objects of type `IT_Logging::LogStream` inside the application. This interface is documented in the Orbix *Programmer's Reference* manual. If you want Orbix diagnostic messages to be logged to, say, a database then you could write an implementation of the `IT_Logging::LogStream` interface to do that.

Orbix provides two pre-written implementations of `IT_Logging::LogStream`, both of which are provided as plugins. If these plugins are loaded (see Section 6.9.3) then Orbix uses them for logging diagnostic information. These plugins are *not* mutually exclusive, so you can load neither of them, just one of them or both of them.

One of the logging plugins is called `system_log_stream`. If this plugin is loaded then Orbix diagnostic messages are sent to the system log device. On UNIX, this is called *syslog*, whereas on Windows it is called the *event log*. This plugin is not loaded by default.

The other logging plugin is call `local_log_stream`. If this plugin is loaded then Orbix diagnostic messages, by default, are written to *standard error*. However, you can redirect these messages to a file with the following runtime configuration variable:

```
set runtime_config_variables {
    string  plugins:local_log_stream:filename   "/path/to/server.log"
}
```

If you do this then Orbix appends a date suffix (in the format `".DDMMYYYY"`) onto the end of the name of the log file. Rather than log all messages to this file, Orbix opens a new log file for each day. Thus, after a week of operation, an application would have 7 log files.

6.9.4.2 Log Filters

Another important part of controlling log messages is setting up *filters* that determine *which* messages are logged. The default filter is illustrated below:

```
set runtime_config_variables {
    list    event_log:filters   "*=WARN+ERR+FATAL"
}
```

The Orbix runtime system is composed of various sub-systems. These have names like `IT_CORE`, `IT_GIOP`, `IT_POA` and so on. The `"*"` character is used as a wildcard to means *all* the sub-systems. For each sub-system, you can specify a list of severity levels. These have names like `INFO`, `WARN`, `ERR` and `FATAL`. The elements in the list of severity levels are separated by `"+"` characters. The above example (which is the default filter) causes Orbix to generate diagnostics for warnings, (non-fatal) errors and fatal errors. You can turn on *all* diagnostic messages with the following:

```
set runtime_config_variables {
    list    event_log:filters   "*=*"
}
```

The final example, below, illustrates the syntax for selectively enabling different filters for different sub-systems:

```
set runtime_config_variables {
    list event_log:filters "IT_GIOP=*,IT_CORE=FATAL,*=WARN+ERR+FATAL"
}
```

See the Orbix *Administrator's Guide* for a full list of the names of both sub-systems and logging severity levels.

6.9.4.3 Obtaining Per-request Diagnostics

If you load the `giop_snoop` plugin (See Section 6.9.3) then Orbix generates diagnostic messages for each message (such as requests and replies) that are sent and received by an application. The `giop_snoop_README.txt` file provided in the `doc` sub-directory of an Orbix installation contains full details on how to enable and use the the `giop_snoop` plugin.

6.9.5 Avoiding POA name clashes

Each CORBA server can have POAs, and each POA has a name. The POAs are arranged in a hierarchy, and the fully-qualified name of a POA is expressed in the form:

```
full/path/to/POA-name
```

This syntax is similar in concept to the syntax used to express the `/full/path/to/a/file` used in the UNIX file system, except that there is *not* a leading `"/"` when writing a POA's full name.

Persistent POAs have to be registered with the IMR (locator daemon). When registering a persistent POA, you specify the `orbname` (conceptually, an application) with which the POA is associated. Transient POAs do not have to be registered, unless a transient POA is a parent (or ancestor) of a persistent POA.

Within an IMR, the `full/path/to/POA-name` entries must be unique for all the registered, persistent POAs. However, it is possible that two independently developed applications might have POAs with similar names. To resolve such clashes on the names of POA, Orbix allows a prefix to be applied to the names of POAs within an application. By registering different applications with different prefixes for their POAs, you can avoid POA name clashes. This prefix is obtained from the value of the `plugins:poa:root_name` configuration variable. If you are using the `orbix_srv_admin` utility then there is no need to explicitly set a value for this variable because `orbix_srv_admin` will do it for you automatically. The value that `orbix_srv_admin` assigns to this configuration variable is determined by the `root_poa_name` entry in the `orbix_srv_admin` description file. If you are using the `orbix_set_config_vars` utility then you should explicitly set a value for this variable if you want to avoid POA name clashes.

6.9.6 Recycling Connection Resources

When a client application communicates with objects in a server application, there will be a socket connection between the client and the server applications. Note that there will be one socket connection between a client and server, irrespective of how many *objects* the client is communicating with in the server. CORBA allows the socket connection between a client and a server to be closed if it is *idle*, that is, if there are no requests from the client currently being processed by the server. CORBA does not consider the closing of an idle socket connection to be an error condition. Rather, if an idle connection is closed then the client application will transparently re-open the connection if/when it makes another remote call to an object in a server. The Orbix terminology for the automatic closing of idle socket connections is *Active Connection Management* (ACM).

ACM is disabled by default, that is, the default behavior of Orbix is that it does *not* close idle socket connections. However, you can set some runtime configuration variables to enable ACM. This is illustrated by the example below:

```
set runtime_config_variables {
    long  plugins:iiop:incoming_connections:soft_limit   "500"
    long  plugins:iiop:incoming_connections:hard_limit   "600"

    long  plugins:iiop:outgoing_connections:soft_limit   "-1"
    long  plugins:iiop:outgoing_connections:hard_limit   "-1"
}
```

The `soft_limit` variables specify the number of connections at which the Orbix runtime system (actually the IIOP plugin) should begin closing connections. However, it may take the Orbix runtime system some time before it can find a connection that is *idle*. While one thread in the Orbix runtime system is searching for idle connections that it can close, more connections might be established/accepted by other threads. This is why it is necessary to also specify a `hard_limit`, which determines the maximum number of connections. Above this, new connection attempts will be rejected. You should always ensure that the `hard_limit` is greater than the `soft_limit`. Alternatively, you can set these values to -1, which is the default. Doing this instructs Orbix to not apply any limits.

As the above example illustrates, hard and soft limits can be specified for both *incoming* and *outgoing* connections. The concept of *incoming* and *outgoing* refers to the direction in which requests flow. Thus, a server application can place a limit on the number of connections from clients that it will keep open by specifying `soft_limit` and `hard_limit` values for the incoming connections.

It is common for *many* clients to connect to a single server, so specifying *incoming* connection limits on the server is important. However, it is very rare for one client application to open connections to *many* servers. Because of this, it is rarely necessary to specify limits on the *outgoing* connections.

The reason why the default values for connection limits is set to -1 is because there are no default values that would be suitable for all application loads on all operating systems. Instead, a user (or system administrator) should consult the documentation for their operating system to determine how many socket connections the operating system can support before performance is degraded and then ensure that the Orbix limits are set below this point. Also, if there will be many applications (each of which might use socket connections) running on the same machine then you should scale back the Orbix connection limits even further so that a heavily loaded Orbix server does not cause other applications to become starved of socket connections.

6.9.6.1 Recycling Connections for Other Protocols

The example in Section 6.9.6 showed how to set up ACM for the IIOP protocol. If your server uses the `shmiop` or `iiop_tls` plugins then you can enable ACM for these protocols by replacing `iiop` in the name of the configuration variables with `shmiop` or `iiop_tls`. This is illustrated in the example below:

```
set runtime_config_variables {
    long  plugins:iiop_tls:incoming_connections:soft_limit   "500"
    long  plugins:iiop_tls:incoming_connections:hard_limit   "600"

    long  plugins:iiop_tls:outgoing_connections:soft_limit   "-1"
    long  plugins:iiop_tls:outgoing_connections:hard_limit   "-1"
}
```

Chapter 7

Generic Synchronization Policies in C++

7.1 Introduction

Writing the synchronization code for a multi-threaded application traditionally has been both difficult (due to the predominance of low-level APIs) and non-portable.

The portability problem arises because neither C nor C++ provide a standard class library for synchronization. As a result, many operating systems provide their own, proprietary APIs for synchronization. Many companies and individuals have successfully tackled this problem by writing (and porting) a *portability layer* that hides the proprietary APIs of the underlying operating system. Although these portability-layer libraries solve the portability problem, they also tend to provide a low-level API. As such, they do *not* simplify the writing of synchronization code.

In the author's experience, most uses of synchronization code in multi-threaded applications falls into a small number of "usage patterns", or what is called *Generic Synchronization Policies* (GSPs). This chapter discusses how to write a C++ class library that provides direct support for these commonly-used GSPs. This chapter also illustrates how the use of such GSPs dramatically simplifies the writing of thread-safe classes.

7.1.1 Structure of this Chapter

The rest of this chapter is structured as follows.

Section 7.2 discusses a C++ programming idiom that has many uses. That section illustrates some of the (non-synchronization) uses of this idiom in order to introduce it to readers.

Section 7.3 discusses how this C++ idiom can be used for synchronization.

Section 7.4 introduces a notation that can be used to elegantly express the signature of a Generic Synchronization Policy (GSP). The mapping from this notation to the C++ idiom is also discussed, and is illustrated with the implementation and instantiation of several commonly used GSPs.

Section 7.5 offers a critique of the techniques advocated in this chapter, pointing out their strengths and limitations.

Section 7.6 provides details of where you can find an implementation of some GSPs for several threads packages.

7.2 A Useful C++ Idiom

For most classes, the *constructor* and *destructor* are used only to initialize and destroy an object, while the useful functionality of the class is provided by its public operations. However, it is possible to write some useful classes that have *only* a constructor and destructor, and do not have any operations. This is illustrated by two examples in the following sub-sections.

7.2.1 A Trace Class

Consider the `Trace` class below. Relevant details are indicated in **bold**:

```
class Trace {
public:
    Trace(const char * str) : m_str(str)
    { cout << "start " << m_str << endl; }
    ~Trace()
    { cout << "end " << m_str << endl; }
protected:
    const char * m_str;
};
```

The constructor and destructor of this class just print out a message. This class can be a surprisingly effective (albeit crude) debugging aid. To understand why, consider the (pseudo-code) function below:

```
void foo()
{
    Trace scoped_diagnostics("foo()");
    ...
    if (...) return;                    // line 1
    if (...) throw some_exception;     // line 2
    ...
}                                       // line 3
```

This function declares a `Trace` variable, and passes the name of the function as a parameter to its constructor. The effects of doing this are twofold:

1. When this function is called, the message `"start foo()"` is printed by the constructor of the Trace object.

2. The C++ specification guarantees that the destructor of all local variables (in this case the `Trace` object) are called when a function terminates; this calling of the destructor is guaranteed regardless of *how* the function terminates. Thus, no matter whether the function explicitly returns (line 1), throws an exception (line 2) or implicitly returns at the end of the function (line 3), the destructor of the `Trace` object is called and hence the message `"end foo()"` is printed.

By printing out the `"start..."` and `"end..."` messages, the `Trace` class provides an easy way for a programmer to track the lifetime of function calls.

Strictly speaking, the lifetime of a local variable is not determined by the lifetime of its enclosing *function*, but rather is determined by the lifetime of its enclosing *scope*, that is, a pair of braces. Thus, if we declare a `Trace` variable inside a nested scope then we obtain diagnostics for the nested scope. For example, consider the following (pseudo-code) function:

```
void foo()
{
    Trace scoped_diagnostics("foo()");
    ...
    {
        Trace scoped_diagnostics2("real work");
        ...
    }
    ...
}
```

The diagnostic output obtained from the above function is as follows:

```
start foo()
start real work
end real work
end foo()
```

A `Trace` variable could also be declared as an instance variable inside another class. Doing this would make it possible to trace the lifetimes of objects.

7.2.2 A Timer Class

Another use for a class that contains just a constructor and destructor is to print timing information. For example, assume that the function `get_current_time()` returns the current time of day expressed as a `double`. The `Timer` class below uses this function to print the elapsed time between a `Timer` object's constructor and destructor.

```
class Timer {
public:
    Timer(const char * str)
        : m_str(str)
        , m_start_time(get_current_time())
        { }
    ~Timer()
    { cout << m_str << " took "
           << get_current_time() - m_start_time
           << " seconds" << endl;
    }
protected:
    const char *    m_str;
    double          m_start_time;
};
```

This `Timer` class can be used to determine how long a function takes to execute, as is illustrated in the example below:

```
void foo()
{
    Timer scoped_timer("foo()");
    ...
    if (...) return;
    if (...) throw some_exception;
    ...
}
```

7.3 Applying the C++ Idiom to Synchronization

Much, perhaps most, synchronization code is split up into *pre-* and *post-synchronization* code. For example, consider the (pseudo-code) function below, which executes in mutual exclusion:

```
void foo()
{
    get_lock(a_mutex);
    ... // body of operation
    release_lock(a_mutex);
}
```

The call to `get_lock()` is the pre-synchronization code, and the call to `release_lock()` is the post-synchronization code. This coding style can exploit the C++ idiom discussed in Section 7.2. A class that calls `get_lock()` in its constructor and `release_lock()` in its destructor is shown below:

```
class ScopedMutex {
public:
    ScopedMutex(Mutex & mutex) : m_mutex(mutex)
    { get_lock(m_mutex); }
    ~ScopedMutex()
    { release_lock(m_mutex); }
protected:
    Mutex &    m_mutex;
};
```

We can use this class to synchronize functions just by declaring a local variable of type `ScopedMutex`, as is illustrated in the example below:

```
void foo()
{
    ScopedMutex scoped_lock(a_mutex);
    ...
    if (...) return;                  // line 1
    if (...) throw some_exception;    // line 2
    ...
}                                     // line 3
```

An important benefit of using this class is that it guarantees that the mutex lock will be released no matter how the function terminates: whether by an explicit return (line 1), by throwing an exception (line 2) or by an implicit return at the end of the function (line 3). This guarantee eliminates a major source of bugs in multi-threaded programs.

This technique of using a constructor/destructor class for synchronization is *partially* well-known within the C++ community:

1. The author's experience as a consultant and trainer has given him the opportunity to work with many C++ programmers in many different organization. He has found that about half the programmers he works with are familiar with this technique and view it as being a basic C++ idiom, while the same technique is new to the other half.

2. Among programmers who have used this constructor/destructor technique for synchronization, usage of this technique invariably is confined to mutual exclusion (and *occasionally* the readers-writer policy). However, this technique is applicable to other, more complex synchronization policies too.

Before discussing how to apply this technique to other synchronization policies, it is necessary to take a slight detour. In particular, we have to introduce a new notation: that of *generic* synchronization policies.

7.4 Generic Synchronization Policies

C++ provides support for *template* types. For example, a list class might be denoted as:

```
template<T> class List { ... };
```

Once implemented, this template type could then be instantiated multiple times to obtain, say, a list of `int`, a `list` of `double` and a list of `Widget`, as is shown below:

```
List<int>        my_int_list;
List<double>     my_double_list;
List<Widget>     my_widget_list;
```

The ability to define template types is not unique to C++. Several other languages provide similar functionality, although there are often some differences in terminology and syntax. For example, other languages often use the term *generic* types rather than *template* types, and the type parameters are enclosed within `[` and `]` instead of `<` and `>`.

The concept of genericity is not restricted to just types. It can also be applied to synchronization. For example, using a *pseudo-code* notation, we can denote some well-known synchronization policies as follows:

```
Mutex[Op]
RW[ReadOp, WriteOp]
```

In this notation, the name of the generic synchronization policy is given first, and is then followed by a parameter list enclosed in square brackets. Each parameter denotes a set of operation names. For example, the `Mutex` policy is instantiated upon a set of operations (`Op`), while the `RW` (readers-writer) policy is instantiated upon a set of read-style operations (`ReadOp`) and a set of write-style operations (`WriteOp`).

Consider a class that has two read-style operations called `Op1` and `Op2`, and a write-style operation called `Op3`. We extend the pseudo-code notation slightly with the use of braces to indicate a set of operation names. With this extension, we can denote the instantiation of the `RW` policy upon these operations as follows:

```
RW[{Op1, Op2}, {Op3}]
```

Likewise, an instantiation of the Mutex policy upon these three operations can be denoted as follows:

```
Mutex[{Op1, Op2, Op3}]
```

A synchronization policy commonly found in the academic literature (but rarely used in practice) is the Shortest Job Next (SJN) scheduler. This policy allows only one thread at a time to execute an operation, and it arranges the other, pending threads in a queue. The order of the threads in the pending queue is determined by the estimated *length* of time that it will take each thread to execute the operation. This length is specified as a parameter to the operation. This policy can be denoted as follows:

```
SJN[Op(int length)]
```

Consider the following (pseudo-code) class:

```
class Printer {
public:
    ... // constructor and destructor
    void print(
        const char *    file_name,
        int             file_size,
        ...);
};
```

The `SJN` policy can be instantiated upon the `print()` operation as follows:

```
SJN[ {print(file_size)} ]
```

In this example, the `file_size` parameter is used as the estimated *length* of the time it will take to execute the operation.

The *producer-consumer* policy is useful where a buffer in shared memory is used to transfer information from one thread (the producer) to another thread (the consumer). The producer thread *puts* items into the buffer and then, sometime later, the consumer thread *gets* these items. If the consumer thread tries to get an item from an empty buffer then it will be blocked until the buffer is non-empty. Furthermore, the put-style and get-style operations execute in mutual exclusion; this is to prevent the buffer from becoming corrupted due to concurrent access. This policy can be denoted as follows:

```
ProdCons[PutOp, GetOp, OtherOp]
```

`OtherOp` denotes any other (non put-style and non get-style) operations on the buffer class. For example, perhaps there is an operation on the buffer that returns a count of how many items are currently in the buffer. Such an operation might need to run in mutual exclusion with the put-style and get-style operations to ensure its correct operation. Consider the following (pseudo-code) class:

```
class WidgetBuffer {
public:
    ... // constructor and destructor
    void insert(Widget * item);
    Widget * remove();
};
```

The `ProdCons` policy can be instantiated upon this class as follows:

```
ProdCons[{insert}, {remove}, {}]
```

Notice that the `WidgetBuffer` class has *only* put-style and get-style operations. Because of this, the `OtherOp` parameter of the policy is instantiated upon an *empty* set of operations names.

The subclass below introduces a new operation, called `count()`:

```
class EnhancedWidgetBuffer: pubic WidgetBuffer {
public:
    ... // constructor and destructor
    int count();
};
```

The `ProdCons` policy can be instantiated upon this subclass as follows:

```
ProdCons[{insert}, {remove}, {count}]
```

A common variation of the producer-consumer policy is the *bounded* producer-consumer policy. In this case, the buffer has a fixed size. This prevents the buffer from growing infinitely large if one thread puts items into the buffer faster than the other thread can get them. In this policy, if the producer thread tries to put an item into an already-full buffer then it will be blocked until the buffer is non-full. This policy is denoted as follows:

```
BoundedProdCons(int size)[PutOp, GetOp, OtherOp]
```

Notice that the `size` of the buffer is specified as a parameter to the name of the policy. Such parameters are usually instantiated upon a corresponding parameter to the constructor of the buffer. For example, consider the following (pseudo-code) class:

```
class BoundedWidgetBuffer {
public:
    // constructor and destructor
    BoundedWidgetBuffer(int buf_size);
    ~BoundedWidgetBuffer();
    void insert(Widget * item);
    Widget * remove();
};
```

The `BoundedProdCons` policy can be instantiated upon this class as follows:

```
BoundedProdCons(buf_size)[{insert}, {remove}, {}]
```

7.4.1 C++ API of a Generic Synchronization Policy

A GSP can be translated into the signatures of C++ classes. This task is performed as follows:

1. The name of the policy maps into a C++ class that has a default constructor and a destructor, but does not have any operations. The name of this class should be the same as the name of the policy, except that it is prefixed by `"GSP_"`. This prefix is used to prevent name-space pollution.

2. Each set of operation names maps into a C++ *nested* class that has a constructor and destructor, but does not have any operations. The constructor takes one parameter. This parameter is a reference to a variable of the enclosing class.

3. If the policy or its sets of operation names take parameters then these parameters are represented as additional parameters to the constructor of the corresponding C++ classes.

As an example, the `Mutex[Op]` policy maps into the following C++ classes:

```
class GSP_Mutex {
public:
    GSP_Mutex();
    ~GSP_Mutex();
    class Op {
    public:
        Op(GSP_Mutex &);
        ~Op();
    protected:
        ...                 // implementation details
    }; // class Op
protected:
    ...                     // implementation details
}; // class GSP_Mutex
```

Likewise, the `RW[ReadOp, WriteOp]` policy maps into the following C++ classes:

```
class GSP_RW {
public:
    GSP_RW();
    ~GSP_RW();
    class ReadOp {
    public:
        ReadOp(GSP_RW &);
        ~ReadOp();
    protected:
        ...                 // implementation details
    }; // class ReadOp
    class WriteOp {
    public:
        WriteOp(GSP_RW &);
        ~WriteOp();
    protected:
        ...                 // implementation details
    };
protected:
    ...                   // implementation details
}; // class GSP_RW
```

As a final example, the `SJN[Op(int length)]` policy maps into the following C++ classes:

```
class GSP_SJN {
public:
    GSP_SJN();
    ~GSP_SJN();
    class Op {
    public:
        Op(GSP_SJN &, int length);
        ~Op();
    protected:
        ...               // implementation details
    }; // class Op
protected:
    ...                   // implementation details
}; // class GSP_SJN
```

7.4.2 C++ Implementation of a Generic Synchronization Policy

The primary principle used to implement the C++ classes for a GSP are illustrated in the following example, which is a POSIX Threads implementation of `GSP_Mutex`.[1] Relevant implementation details are indicated in **bold**. Comments follow after the code:

```
class GSP_Mutex {
public:
    inline GSP_Mutex();
    inline ~GSP_Mutex();

    class Op {
    public:
        inline Op(GSP_Mutex &);
        inline ~Op();
    protected:
        GSP_Mutex &    m_data;
    }; // class Op
    protected:
        friend class ::Mutex::Op;
```

[1] In order to keep the code concise, error checks on the return values of the Pthread library calls have been omitted.

```
        pthread_mutex_t    m_mutex;
}; // class GSP_Mutex

inline GSP_Mutex::GSP_Mutex()
{ pthread_mutex_init(m_mutex, 0); }

inline GSP_Mutex::~GSP_Mutex()
{ pthread_mutex_destroy(&m_mutex); }

inline GSP_Mutex::Op::Op(GSP_Mutex & data) : m_data(data)
{ pthread_mutex_lock(&m_data.m_mutex); }

inline GSP_Mutex::Op::~Op()
{ pthread_mutex_unlock(&m_data.m_mutex); }
```

The main points to note about the code are as follows:

- The outer class contains (operating-system specific) instance variables that are used to implement the synchronization policy. In the above example, the only variable required is of type `pthread_mutex_t`.
- The nested class(es) contain one instance variable that is a reference to the outer class. This, combined with a friend declaration, allows the nested class(es) to access the instance variables of the outer class.
- The pre- and post-synchronization code required for the synchronization policy is executed by the constructor and destructor of the nested class(es). If a policy has several nested classes then the constructor and destructor of each nested class contains the pre- and post-synchronization code for its operation type. For example, the constructor and destructor of `BoundedBuffer::PutOp` contains the pre- and post-synchronization code for a put-style operation while `BoundedBuffer::GetOp` contains the pre- and post-synchronization code for a get-style operation.

7.4.3 C++ Instantiation of a Generic Synchronization Policy

Instantiating a GSP upon the operations of a C++ class involves the following three steps:

1. `#include` the header file for the GSP. The name of the header file is the same as name of the GSP class, but written in lowercase letters. For example, the header file for the `GSP_RW` class is `"gsp_rw.h"`.
2. Add an instance variable to the C++ class that is to be synchronized. The instance variable's type is that of the GSP's outer class.
3. Inside the body of each operation that is to be synchronized, declare a local variable, the type of which is that of a nested class of the GSP.

These steps are illustrated in the following (pseudo-code) example, which instantiates the `RW[{Op1, Op2}, {Op3}]` policy upon a class. Relevant details are shown in **bold**.

```
#include "gsp_rw.h"
class foo {
public:
    foo() { ... }
    ~foo() { ... }
    void Op1(...) {
        GSP_RW::ReadOp     scoped_lock(m_sync);
        ... // normal body of operation
    }
    void Op2(...) {
        GSP_RW::ReadOp     scoped_lock(m_sync);
        ... // normal body of operation
    }
    void Op3(...) {
        GSP_RW::WriteOp    scoped_lock(m_sync);
        ... // normal body of operation
    }
protected:
    GSP_RW    m_sync;
    ... // normal instance variables of class
};
```

As a final example, the following is an instantiation of the `BoundedProdCons` policy upon a class:

```
#include "gsp_boundedprodcons.h"
class WidgetBuffer {
public:
    // constructor and destructor
    WidgetBuffer(int buf_size) : m_sync(buf_size) { ... }
    ~WidgetBuffer() { ... }
        void insert(Widget * item) {
        GSP_BoundedProdCons::PutOp scoped_lock(m_sync);
        ... // normal body of operation
    }
    Widget * remove() {
        GSP_BoundedProdCons::GetOp scoped_lock(m_sync);
        ... // normal body of operation
    }
protected:
    GSP_BoundedProdCons    m_sync;
    ... // normal instance variables of class
};
```

7.4.4 Non-C++ Language Support for Generic Synchronization Policies

The implementation of GSPs shown in this chapter relies upon constructors and destructors to automate the execution of pre-synchronization and post-synchronization code. Although object-oriented languages usually provide constructors, not all object-oriented languages provide destructors, especially if the language has a built-in garbage collector. This may lead readers to conclude that GSPs cannot be implemented in existing languages that do not provide destructors. While this may be so, it would be possible for designers of *future* languages to incorporate support for GSPs into their language design. For example, earlier research by the author [McH94] documents how support for GSPs can be incorporated into a compiler for an object-oriented language in a straightforward manner.

7.5 A Critique of Generic Synchronization Policies

The following subsections briefly discuss: (1) the benefits to be gained from the use of GSPs; (2) some commonly voiced, but misplaced, concerns of GSPs; and (3) some issues not addressed by GSPs.

7.5.1 Strengths of GSPs

GSPs provide a good form of *skills reuse*. In particular, it is a lot easier to *use* a GSP than it is to *implement* one. Thus, a programmer skilled in synchronization programming can implement whatever GSPs are needed for a project, and then other, lesser skilled, programmers can use these pre-written GSPs.

GSPs aid code clarity (and hence maintainability) by separating synchronization code from the “normal”, functional code of a class.

As discussed in Section 7.3, the placing of synchronization code in the constructor and destructor of the GSP classes means that locks are released even if an operation terminates via throwing an exception. This eradicates a significant source of bugs in multi-threaded programs.

GSPs provide not only ease of use; they also provide a portability layer around the underlying synchronization primitives. Of course, some companies already have their own, home-brew, portability libraries that hide the differences between synchronization primitives on various platforms, and some other companies make use of third-party portability libraries, for example, RogueWave’s Threads.h++ library. The use of GSPs is compatible with such existing libraries: GSPs can be implemented just as easily on top of Threads.h++ (or some other portability library) as they can be implemented directly on top of operating-system specific synchronization primitives.

Finally, the implementation of a GSP can be inlined. Thus, the use of GSPs need not impose a performance overhead.

7.5.2 Commonly Voiced Criticisms of GSPs

The concept of GSPs is elegant and simple. As such, it is natural to wonder if perhaps they are *too* simple and must have a lurking, fatal flaw. This section discusses two commonly voiced criticisms of GSPs.

The first commonly voiced criticism of GSPs is that: "GSPs are limited; they cannot handle *all* the synchronization needs that I might have." However, in many activities, a disproportionately large amount of results come from a relatively small amount of investment. This is generally known as the 80/20 principle [Koc00]. Examples of this are sayings such as "80% of the work done in an organization is performed by 20% of its people", or "80% of a programs time is spent executing just 20% of its code". Although known as the 80/20 rule, the percentages involved are not always 80/20, and indeed do not necessarily add up to 100%. For example, an analysis of a company's sales might indicate that 95% of sales are to just 10% of customers. Whatever the percentages involved, the principle (that most results come from a relatively small amount of investment) is widely applicable. In the author's experience, this principle applies to the synchronization requirements of applications. As such, a small set of GSPs is likely to suffice for most (probably more than 80%) of the synchronization needs of programmers. So, even if a small set of pre-written GSPs cannot handle *all* the synchronization needs that a programmer will face, the 80/20 principle suggests that the use of GSPs would be useful *often enough* to justify their use.

Of course, people are *not* restricted to just a small set of pre-written GSPs. People can define new GSPs. For example, perhaps a programmer needs to write some project-specific synchronization code. Even if this synchronization code will be used in just one place in an application, it is hardly any additional work to implement this as a GSP and then instantiate it, rather than to implement it "in-line" in the operations of a class. Doing do offers several benefits:

1. Implementing the synchronization code as a GSP is likely to aid readability and maintainability of *both* the synchronization code *and* the sequential code of the project.

2. If the programmer later discovers another place that needs to use the same policy then the GSP can be re-used directly, rather than having to re-use in-lined code via copy-n-paste.

The second commonly voiced criticism of GSPs is that: "GSPs are not new. 'GSP' is just a new name for an existing C++ programming idiom". The claim that GSPs are based on an already-known C++ idiom is entirely true. However, this does not diminish the contribution of GSPs. For example, the `ScopedMutex` class discussed in Section 7.3 is a GSP in all but name. However, as also discussed in Section 7.3, the C++ idiom that underlies GSPs was previously used only for mutex and *occasionally* the readers-writer policies. The prime contribution of GSPs is in significantly extending the usage of this already-known C++ idiom and, in so doing, pointing out that the same technique can be used for most, if not all, synchronization policies.

7.5.3 Issues Not Addressed by GSPs

GSPs illustrate the 80/20 principle: most of the synchronization requirements of programmers can be satisfied by a small collection of GSPs. However, there are some synchronization issues that are not tackled by GSPs. These issues are discussed briefly below, so that readers can be forewarned about when the use of GSPs is not suitable.

7.5.3.1 Thread Cancellation

The POSIX Threads specification provides a mechanism for a thread to be *cancelled*, that is, terminated gracefully. When a thread is cancelled, it is important that the thread has a chance to do some tidying up before it is terminated, for example, the thread may wish to release some locks that it holds. This is achieved by having the programmer register some callback functions that will be invoked in the event of the thread being cancelled. The current implementation of GSPs does not provide support for the thread cancellation capability of POSIX Threads. This is not due to any intrinsic incompatibility between GSPs and thread cancellation. Rather it is simply due to the author never having needed to make use of thread cancellation. Integrating GSPs with thread cancellation is left as an exercise to interested readers.

7.5.3.2 Timeouts

Some thread packages provide a timeout capability on synchronization primitives. By using this, a programmer can specify an upper time limit on how long a thread should wait to, say, get a lock. The current implementation of GSPs does *not* provide a timeout capability. There are two reasons for this.

Firstly, timeouts are rarely needed and hence, by following the 80/20 principle, the author decided to not bother supporting them.

Secondly, the provision of a timeout capability can often be relatively complex with the APIs of some threads packages. For example, a `Mutex` policy *without* a timeout capability can be implemented in POSIX Threads by invoking some trivial APIs upon an underlying mutex type. In contrast, implementing a `Mutex` policy *with* a timeout capability in POSIX Threads necessitates the use of a mutex variable *and* a condition variable; the resulting algorithm is more complex to write/maintain and incurs *at least* twice as much performance overhead as a `Mutex` without a timeout capability. This additional performance overhead suggests that if some programmers decide that they require a `Mutex` policy with a timeout capability then they should implement it as a *new* GSP, say, `TimeoutMutex`, rather than add the timeout capability to the existing `Mutex` policy. In this way, programmers will be able to use the `TimeoutMutex` policy on the few occasions when they need to, and can use the more efficient `Mutex` policy on all other occasions.

7.5.3.3 Lock Hierarchies

GSPs are useful for classes or objects that have self-contained synchronization. However, sometimes the synchronization requirements of *several* classes are closely intertwined, and a programmer will need to acquire locks on, say, two (or more) objects before s/he can carry out some task. The need to acquire locks on several objects at the same time is sometimes referred to as a *lock hierarchy*. Attempting to acquire a lock hierarchy can result in deadlock if done incorrectly.

Algorithms for acquiring lock hierarchies safely are outside the scope of this chapter, but can be found elsewhere [But97]. The point to note here is that algorithms for acquiring lock hierarchies safely require unhindered access to the locking primitives. This is in opposition to GSPs, which completely encapsulate the underlying synchronization primitives.

7.6 GSP Class Library

The author has implemented a library of the following GSP classes: `GSP_Mutex`, `GSP_RW`, `GSP_ProdCons` and `GSP_BoundedProdCons`. These GSPs are implemented for the following threads packages: Solaris threads, DCE Threads, POSIX threads and Windows threads. Dummy implementations of these GSPs for non-threaded systems are also provided; this makes it possible to write a class that can be used in both sequential and multi-threaded applications.

All the GSPs are implemented with inline code in header files. Because of this, to make use of a GSP you need only `#include` the corresponding header file; there is no GSP library to link into your application. The name of the header file for a GSP is the same as name of the GSP class, but written in lowercase letters. For example, the header file for the `GSP_RW` class is `"gsp_rw.h"`.

You should use the `-D<symbol_name>` option on your C++ compiler to define one of the following symbols:

- `P_USE_WIN32_THREADS`
- `P_USE_POSIX_THREADS`
- `P_USE_DCE_THREADS`
- `P_USE_SOLARIS_THREADS`
- `P_USE_NO_THREADS`

The symbol instructs the GSP class which underlying threading package it should use.

7.7 Acknowledgments

The concept of GSPs has its roots in the author's Ph.D. thesis [McH94]. Research for this Ph.D. was partially funded by the Comandos ESPRIT project and was carried out while the author was a member of the Distributed Systems Group (DSG) in the Department of Computer Science, Trinity College, Dublin, Ireland. The author wishes to thank DSG for their support. The author also wishes to thank colleagues in IONA Technologies for their comments on earlier drafts of this chapter.

Bibliography

[But97] David Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Koc00] Richard Koch. *The 80/20 Principle: The Secret of Achieving More With Less*. Nicholas Breealey Publishing Ltd., May 2000. ISBN: 187881680. 312 pages.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994.

[SC] Henry Spencer and Geoff Collyer. #ifdef Considered Harmful, or Portability Experiences With C News. In *Proceedings of Summer 1992 UNENIX*, pages 185–197.