Developing Distributed Systems

Barry Cornelius
Computing Services, University of Oxford
Date: 29th March 1998
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

1 Introduction

Client-server computing has traditionally been done using *socket*s. A socket is a point of connection within a program to the Internet, and a program can start to communicate when it arranges for the socket to bind to a port of the computer that is running the program.

The Java classes `java.net.DatagramSocket`, `java.net.ServerSocket` and `java.net.Socket` provide easy-to-use ways of writing programs that use sockets. However, in Java, client-server computing can be performed without having to work at the level of sockets. This is because Java has several application programming interfaces (APIs) that offer higher levels of abstraction for client-server computing.

In this document, we look at three APIs: Remote Method Invocation, Common Object Request Broker Architecture and Java DataBase Connectivity. The latter is a way in which a Java program can use SQL to access database servers, whereas the other two APIs allow us to rise up above the socket-level details of client-server computing when we want to access objects distributed across different computers.

Although Java applications can use these APIs to attempt to communicate with any computer on the Internet, Java applets are normally restricted to talking to the site from which the applet was downloaded. This hinders the use of applets in client-server computing. This document explains how signed applets can be used to overcome these security restrictions.

In the past, client-server computing has usually involved just two computers talking to one another. Often a lot of the processing is going on in the client, which often means that the client is a big program. More recently, systems have been produced that use an intermediate layer: the client just provides a user interface whereas the intermediate layer contains all the complicated application logic. The final topic of this document introduces the idea of Three Tier Systems.

2 RMI

2.1 What is RMI?

RMI is *Remote Method Invocation*. With RMI, you can access the methods of remote objects. In more detail, on one computer, the *server*, you run:

- the RMI-Registry program;

and:

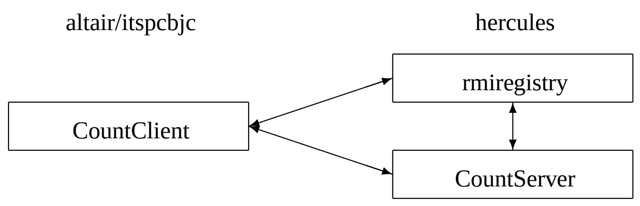
- a Java application that registers one or more Java objects with the RMI-Registry.

And, on the other computer, the *client* computer, you run:

- a Java application that accesses the methods of the objects of the application running on the server.

2.2 How do you use RMI?

Suppose that we want an object that looks after an integral value. We want the ability for any Java application/applet running on any computer to get the latest value, or to set it to some new value:



To use RMI, we need an interface that defines the services that will be available from the object:

```
package RMI.Count;                                           // CountIF.java
public interface CountIF extends java.rmi.Remote {
   int getCount() throws java.rmi.RemoteException;
   void setCount(int vCount) throws java.rmi.RemoteException;
}
```

We now need to define what happens when `getCount` and `setCount` get called. We do this by defining a class that implements the interface:

```
package RMI.Count;                                      // CountServant.java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class CountServant extends UnicastRemoteObject implements CountIF {
   public CountServant() throws RemoteException {
      super();
      System.out.println("CountServant: constructor called");
      iCount = 0;
   }
   public synchronized int getCount() throws RemoteException {
      System.out.println("CountServant: getCount called");
      return iCount;
   }
   public synchronized void setCount(int vCount) throws RemoteException {
      System.out.println("CountServant: setCount called with " + vCount);
      iCount = vCount;
   }
   private int iCount;
}
```

On the server computer, we need a Java application that creates an object that is of the `CountServant` class. The `CountServer` program creates an object called `realCount` and then registers the object with the RMI-Registry. It does this by using the `rebind` method from the class `java.rmi.Naming` passing to it the name `first Count` that will allow any client to refer to the object:

```
package RMI.Count;                                       // CountServer.java
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
public class CountServer {
   public static void main(String[] args) {
      System.setSecurityManager(new RMISecurityManager());
      try {
         CountServant realCount = new CountServant();
         Naming.rebind("first Count", realCount);
      }
      catch(Exception rException) {
         rException.printStackTrace();
      }
   }
}
```

Finally, on the client computer, we want a Java application that accesses the methods of this object:

```
package RMI.Count;                                       // CountClient.java
import java.io.DataInputStream;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
public class CountClient {
   public static void main(String[] args) {
      DataInputStream stdin = new DataInputStream(System.in);
      System.setSecurityManager(new RMISecurityManager());
      try {
         CountIF stubCount =
               (CountIF) Naming.lookup("rmi://" + args[0] + "/first Count");
         System.out.println("before loop");
         while (true) {
            String stdinLine = stdin.readLine();
            if ( stdinLine.equals("quit") ) break;
            if ( stdinLine.equals("show") )
               System.out.println(stubCount.getCount());
            else {
               int intVal = Integer.parseInt(stdinLine);
               stubCount.setCount(intVal);
            }
         }
         System.out.println("after loop");
      }
      catch(Exception rException) {
         rException.printStackTrace();
      }
   }
}
```

The `CountClient` program uses the `lookup` method from the class `java.rmi.Naming` to locate the remote object:

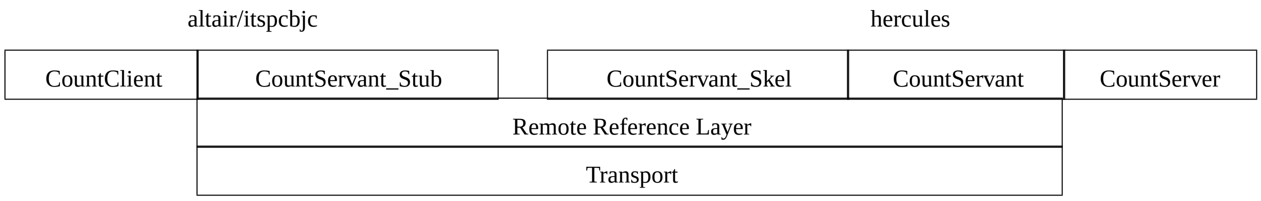
```
         CountIF stubCount =
               (CountIF) Naming.lookup("rmi://" + args[0] + "/first Count");
```

It needs to be passed a string like `"rmi://hercules.dur.ac.uk/first Count"`. This string looks like a URL: it specifies the name of the server and the name of the object. In the `CountClient` program, the name of the server is actually obtained from `args[0]`, the first argument on the command line used to run the program.

The call of `lookup` communicates with the RMI-Registry on the server. It returns a value that can be used as an object of the interface `CountIF`. So, we can use the reference variable `stubCount` to refer to the remote object. In fact, the `CountClient` program contains a call of both `stubCount.setCount` and `stubCount.getCount`.

2.3 How does RMI work?

The client program does not have immediate access to the object on the server. Instead, a lot of magic takes place:



The value returned by the call of `lookup` in the `CountClient` program is a reference to an object that is part of the client program. This object is a *stub* that acts as a proxy for the remote object. When the client program calls a method, say, it calls `stubCount.setCount(42)`, then it is actually calling a method called `setCount` of the stub. This method knows nothing about how to set the remote object's value to 42. Instead, what it does is to transmit the required operation to the server: it does this by arranging for the method and its parameters to be *marshalled* into a stream of bytes. This marshalling makes use of JDK 1.1.x's *object serialization*.

Within the Java application running on the server, there is the real object. However, besides this real object there is also a *skeleton* object. The skeleton has a method that arranges for any incoming requests to be unmarshalled. By this means, it finds out what method is to be called and what arguments are to be supplied to the call. It then calls the appropriate method of the real object with the appropriate arguments.

If the method being called returns some result, then the skeleton's method will get the result, serialize the result, and pass the bytes back to the client computer. The method in the stub will receive these bytes, de-serialize them, and return a value to the caller.

So besides the code that you write for the client and the server, you will also need code for the stub and the skeleton. Fortunately, RMI comes with a tool (called `rmic`) that will automatically generate the stub and skeleton files for you.

2.4 What steps are needed to run the above example?

First, compile the server program with a JDK 1.1.x compiler, e.g.:

```
setenv CLASSPATH /users/dcl0bjc/public_html/Java
javac CountServer.java
```

This `javac` command will compile `CountIF.java`, `CountServant.java` and `CountServer.java` to produce the files `CountIF.class`, `CountServant.class` and `CountServer.class`.

At a Unix/MS-DOS prompt, run RMI's `rmic` command on the file `CountServant.class`:

```
rmic RMI.Count.CountServant
```

This command will create the files: `CountServant_Stub.class` and `CountServant_Skel.class`. These files contain the bytecodes of the stub and skeleton.

Now compile the client program:

```
javac CountClient.java
```

Before running the server, we need to start the RMI-Registry. We can do this using RMI's `rmiregistry` command. On Unix, type:

```
rmiregistry &
```

whereas on Windows 95/NT, type:

```
start rmiregistry
```

The server can then be started using a command like:

```
java -Djava.rmi.server.codebase=http://hercules.dur.ac.uk/~dcl0bjc/Java/ \
   RMI.Count.CountServer &
```

This should output:

```
CountServant: constructor called
```

Finally, run the client program:

```
setenv CLASSPATH /users/dcl0bjc/public_html/Java
java RMI.Count.CountClient hercules.dur.ac.uk
```

The client program continually reads in lines from the keyboard. If the line contains the string `quit`, the client program is terminated; if it contains `show`, the client program displays the current value of the remote object; otherwise, the client program expects the line to contain an integral value and it instructs the server to set the remote object to this value.

2.5 Further information about RMI

The texts of the above programs are at http://www.dur.ac.uk/barry.cornelius/papers/RMI/Count/.

Sun's main RMI page is at http://java.sun.com/products/jdk/rmi/. On that page, you will find links to some useful pages entitled *Getting Started*, *Documentation*, *Examples*, *Specification*, *White Paper* and *FAQ*. You will also find a link which enables you to download RMI for JDK 1.0.2.

Users of RMI can discuss issues and tips with other users via the `rmi-users@javasoft.com` mailing list. You can subscribe to this list by sending an e-mail message containing the single line `subscribe rmi-users` to the address `listserv@javasoft.com`. Sun's main RMI page also contains a link to an archive of the `rmi-users` mailing list.

The changes to RMI that are planned for JDK 1.2 are documented at

http://java.sun.com/products/jdk/1.2/docs/guide/rmi/.

The Cetus Links on RMI are at http://www.parallax.co.uk/cetus/oo_javabeans.html.

3 CORBA

3.1 What is CORBA?

CORBA is the *Common Object Request Broker Architecture*. It is an ongoing effort by the Object Management Group (OMG) to define interfaces for interoperable software. Work on CORBA started several years ago, and there are now over 700 members of OMG including virtually all of the computer manufacturers and most of the major software vendors.

If you are using CORBA, communications between two computers is performed using an *Object Request Broker* (ORB). An object in a program running on some computer (the *client*) can use the ORB to access the public attributes of another object in some other program perhaps on a different computer (the *server*) that is also using an ORB. The client object can:

- *either* hard-code the name of the computer and the name of the object that it wants to contact (*Static Method Invocation*);
- *or* it can use the ORB to discover which objects have the required interface (*Dynamic Invocation Interface*).

In this document, we will only be using Static Method Invocation.

Although an ORB can be running on a single computer, it is more likely to be used to aid communication between one or more client computers and a server computer. If an ORB is unable to satisfy a request from a client, then, if it is connected to other ORBs (elsewhere in the world), the request may be satisfied by an object known to a remote ORB.

One of the key aspects of CORBA is the *Interface Definition Language* (IDL). This is the language that is used to describe the interface that an object has. With RMI, we had to provide the interface in Java: whereas, with CORBA, the interface has to be provided in IDL. It is a language that looks something like C, C++ or Java. For example, it has function declarations and interfaces; it has types such as `boolean`, `char`, `string`, `short`, `long`, `float` and `double`.

Having produced an IDL file that describes the interface that an object has, we also need some code for the server that creates an object that implements that interface, and we will also need codes for clients that wish to access the object. These codes need to be written in *real* programming languages. In order for an interface written in IDL to be usable with existing languages, there needs to be a mapping that says what each IDL feature is in each language. For example, the IDL-to-Java mapping defines that an IDL `long` is an `int` in Java, an IDL `string` is a `String` in Java, ... Currently, there are bindings for Ada, C, C++, Java and Smalltalk.

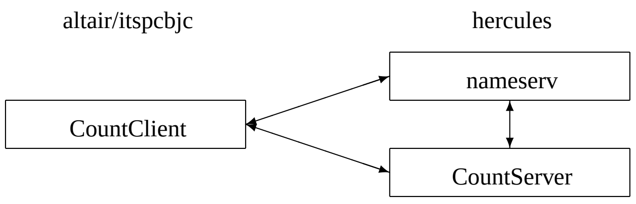
With RMI, both the client and the server have to be Java applications/applets (running in some Java environment). However, with CORBA, the client and server programs may be in different languages. Each program can run on any operating system written in any language so long as you have a compiler to translate the interface from IDL into that language on that operating system.

There are many vendors providing ORBs. In this document, we will look at two products: *JavaIDL* from Sun Microsystems and *OmniBroker* from Object Oriented Concepts. Some other ORBs are listed in the *Further information about some other ORBs* section given later.

3.2 Using Sun's JavaIDL

At the present time, Sun's JavaIDL is not part of the JDK: however, it can be downloaded from Sun's JavaIDL page at http://java.sun.com/products/jdk/idl/ and used with JDK 1.0.2 or JDK 1.1.x. JavaIDL will form part of JDK 1.2, and a beta release of JDK 1.2 is available from http://java.sun.com/products/jdk/1.2/. A version of JavaIDL is also supplied with Sun's Solaris 2.6 operating system.

The use of JavaIDL will be demonstrated by using the same example as was used for RMI:



As before, we will need to provide an interface for our object: this time it is provided in IDL:

```
module CountApp {
   interface Count {
      long getCount();
      void setCount(in long vCount);
   };
};
```

When JavaIDL's `idltojava` command is used to translate this file into Java, it will also produce a class called `_CountImplBase` in a subdirectory called `CountApp`. We can implement the above interface by extending the class `_CountImplBase`:

```
package IDL.Count;                                      // CountServant.java
import IDL.Count.CountApp._CountImplBase;
public class CountServant extends _CountImplBase {
   public CountServant() {
      super();
      System.out.println("CountServant: constructor called");
      iCount = 0;
   }
   public synchronized int getCount() {
      System.out.println("CountServant: getCount called");
      return iCount;
   }
   public synchronized void setCount(int vCount) {
      System.out.println("CountServant: setCount called with " + vCount);
      iCount = vCount;
   }
   private int iCount;
}
```

It is on the server that we have an object that is of the class `CountServant`. Here is a Java application that creates an object of this class and then asks the ORB's *Naming Service* to bind the name `second Count` to the object:

```java
package IDL.Count;                                       // CountServer.java
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CORBA.ORB;
public class CountServer {
   public static void main(String[] args) {
      try{
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);
         // create a CountServant object and register it with the ORB
         CountServant realCount = new CountServant();
         orb.connect(realCount);
         // get the root naming context
         org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
         NamingContext ncRef = NamingContextHelper.narrow(objRef);
         // bind the Object Reference in Naming
         NameComponent nc = new NameComponent("second Count", "");
         NameComponent path[] = {nc};
         ncRef.rebind(path, realCount);
         // wait for invocations from clients
         java.lang.Object sync = new java.lang.Object();
         synchronized (sync) {
            sync.wait();
         }
      }
      catch (Exception rException) {
         rException.printStackTrace();
      }
   }
}
```

Finally, here is a client program. This is the program that accesses the methods of the object on the server:

```java
package IDL.Count;                                       // CountClient.java
import IDL.Count.CountApp.Count;
import IDL.Count.CountApp.CountHelper;
import java.io.DataInputStream;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CORBA.ORB;
public class CountClient {
   public static void main(String[] args) {
      DataInputStream stdin = new DataInputStream(System.in);
      try{
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);
         // get the root naming context
         org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
         NamingContext ncRef = NamingContextHelper.narrow(objRef);
         // resolve the Object Reference in Naming
         NameComponent nc = new NameComponent("second Count", "");
         NameComponent path[] = {nc};
         Count stubCount = CountHelper.narrow(ncRef.resolve(path));
         // enter a loop which processes requests from the keyboard
         System.out.println("before loop");
         while (true) {
            String stdinLine = stdin.readLine();
            if ( stdinLine.equals("quit") ) break;
            if ( stdinLine.equals("show") )
               System.out.println(stubCount.getCount());
            else {
               int intVal = Integer.parseInt(stdinLine);
               stubCount.setCount(intVal);
            }
         }
         System.out.println("after loop");
      }
      catch (Exception rException) {
         rException.printStackTrace();
      }
   }
}
```

```java
package IDL.Count;                                       // CountClient.java
import IDL.Count.CountApp.Count;
import IDL.Count.CountApp.CountHelper;
import java.io.DataInputStream;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CORBA.ORB;

public class CountClient {
   public static void main(String[] args) {
      DataInputStream stdin = new DataInputStream(System.in);
      try{
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);
         // get the root naming context
         org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
         NamingContext ncRef = NamingContextHelper.narrow(objRef);
         // resolve the Object Reference in Naming
         NameComponent nc = new NameComponent("second Count", "");
         NameComponent path[] = {nc};
         Count stubCount = CountHelper.narrow(ncRef.resolve(path));
         // enter a loop which processes requests from the keyboard
         System.out.println("before loop");
         while (true) {
            String stdinLine = stdin.readLine();
            if ( stdinLine.equals("quit") ) break;
            if ( stdinLine.equals("show") )
               System.out.println(stubCount.getCount());
            else {
               int intVal = Integer.parseInt(stdinLine);
               stubCount.setCount(intVal);
            }
         }
         System.out.println("after loop");
      }
      catch (Exception rException) {
         rException.printStackTrace();
      }
   }
```

3.3 What steps are needed to run the above example?

First, you need to compile the interface with JavaIDL's `idltojava` command.

```
setenv JAVAIDL_HOME /users/hplives/JavaIDL-1.1EA
setenv PATH $JAVAIDL_HOME/bin:$PATH
idltojava -fclient -fserver -p IDL.Count -j ../.. Count.idl
```

Because the `Count.idl` file has a module called `CountApp`, this command will create a directory called `CountApp` in which it puts the files that implement the stub and skeleton objects.

Now compile the server program with a JDK 1.0.2 or JDK 1.1.x compiler, e.g.:

```
setenv CLASSPATH $JAVAIDL_HOME/lib/classes.zip:../..
javac CountServer.java
```

This `javac` command will compile `CountServant.java` and `CountServer.java` and some of the files in the `CountApp` directory.

Before running the server, we need to start JavaIDL's Naming Service program. We can do this using JavaIDL's `nameserv` command:

```
nameserv -ORBInitialPort 1050 &
```

This command will produce output like:

```
Initial Naming Context:
IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436
f6e746578743a312e3000000000010000000000000034000100000000000968657263756c657300
00805e0000001cafabcafe000000023448a5050000000000000008000000000000000000
TransientNameServer: setting port for initial object references to: 1050
```

The server can then be started using a command like:

```
java IDL.Count.CountServer -ORBInitialPort 1050 &
```

This should output:

```
CountServant: constructor called
```

Finally, compile the client program with a JDK 1.0.2 or JDK 1.1.x compiler:

```
javac CountClient.java
```

and run it:

```
java IDL.Count.CountClient -ORBInitialHost hercules.dur.ac.uk -ORBInitialPort 1050
```

The client program continually reads in lines from the keyboard. If the line contains the string `quit`, the client program is terminated; if it contains `show`, the client program displays the current value of the remote object; otherwise, the client program expects the line to contain an integral value and it instructs the server to set the remote object to this value.

3.4 Further information about JavaIDL

The texts of the above programs are at http://www.dur.ac.uk/barry.cornelius/papers/IDL/Count/.

Sun's main JavaIDL page is at http://java.sun.com/products/jdk/idl/. On that page, you will find links to some useful pages entitled *Documentation*, *Examples*, *Download/EarlyAccess* and *Other References*. In particular, the *Download/EarlyAccess* link enables you to download the Early Access Release of JavaIDL for use with JDK 1.0.2 or JDK 1.1.x.

The above code is derived from Sun's `Hello` example which is subject to the copyright and licensing information given at http://java.sun.com/products/jdk/idl/docs/examplelicense.html.

The details of the JavaIDL being released with JDK 1.2 are documented at
http://java.sun.com/products/jdk/1.2/docs/guide/idl/.

In many ways, RMI and CORBA are competing technologies. Sun's position about this is explained at http://java.sun.com/features/1997/nov/rmi.html.

3.5 Using an object's IOR instead of the Naming Service

Earlier, when we started `nameserv` (JavaIDL's Naming Service program), we saw that it output a line containing the characters `IOR:` followed by a very long string of hexadecimal characters. (In the output given earlier, the single line containing this string wraps round on to three lines.) This is an example of an *Interoperable Object Reference* (IOR). In this case, it is the IOR of the object that is the Naming Service.

Each object that is known to an ORB has an IOR that is unique to the object. In fact, all you need to communicate with a remote object is to know its IOR. So, if you have got the IORs of all the remote objects you want to access, then there is no need for you to run a Naming Service.

The API has two methods that help us work with IORs: they are called `object_to_string` and `string_to_object`. In the `CountServer` program, we currently have statements that register `realCount` with the Naming Service:

```
          CountServant realCount = new CountServant();
          orb.connect(realCount);
          // get the root naming context
          org.omg.CORBA.Object objRef =
                     orb.resolve_initial_references("NameService");
          NamingContext ncRef = NamingContextHelper.narrow(objRef);
          // bind the Object Reference in Naming
          NameComponent nc = new NameComponent("second Count", "");
          NameComponent path[] = {nc};
          ncRef.rebind(path, realCount);
```

These can be replaced by statements that write the IOR of the `realCount` variable to a file called `Count.ior`:

```
          CountServant realCount = new CountServant();
          orb.connect(realCount);
          String realCountString = orb.object_to_string(realCount);
          PrintStream tPrintStream =
                new PrintStream(new FileOutputStream("Count.ior"));
          tPrintStream.println(realCountString);
          tPrintStream.close();
```

And, in the `CountClient` program, the statements that look up the remote object in the Naming Service:

```
          // get the root naming context
          org.omg.CORBA.Object objRef =
                     orb.resolve_initial_references("NameService");
          NamingContext ncRef = NamingContextHelper.narrow(objRef);
          // resolve the Object Reference in Naming
          NameComponent nc = new NameComponent("second Count", "");
          NameComponent path[] = {nc};
          Count stubCount = CountHelper.narrow(ncRef.resolve(path));
```

can be replaced by statements that read the IOR of the `realCount` variable from the file `Count.ior`:

```
        DataInputStream tDataInputStream =
              new DataInputStream(new FileInputStream("Count.ior"));
        String realCountString = tDataInputStream.readLine();
        Count stubCount =
              CountHelper.narrow(orb.string_to_object(realCountString));
```

The complete texts of the programs that do not use a Naming Service are at

http://www.dur.ac.uk/barry.cornelius/papers/IDL/CountNoNS/.

Using the IOR of a remote object may be a useful technique if you are only communicating with one object. However, it becomes more cumbersome if several objects are involved because you will have to look after several IORs (one for each object).

3.6 Communicating with a Naming Service by using its IOR

It is also possible for the server program and/or the client program to use an IOR to contact the Naming Service rather than contact it through the information given by `ORBInitialHost` and `ORBInitialPort`. If we start the Naming Service by the command:

```
nameserv -ORBInitialPort 1050 | tee nameserv.ior &
```

the lines that `nameserv` outputs will be captured in the file `nameserv.ior`. Then instead of using:

```
java IDL.Count.CountServer -ORBInitialPort 1050 &
java IDL.Count.CountClient -ORBInitialHost hercules.dur.ac.uk -ORBInitialPort 1050
```

to tell `CountServer` and `CountClient` where the Naming Service is, we can pass the IOR of the Naming Service as the first argument of the command line being used to run these programs:

```
java IDL.CountNSIOR.CountServer ‘grep IOR nameserv.ior‘ &
java IDL.CountNSIOR.CountClient ‘grep IOR nameserv.ior‘
```

In order to get at this IOR, the statement of `CountServer` that says:

```
        org.omg.CORBA.Object objRef =
                   orb.resolve_initial_references("NameService");
```

has to be changed to:

```
        org.omg.CORBA.Object objRef = orb.string_to_object(args[0]);
```

A similar change has to be made to the `CountClient` program.

The complete texts of the programs where an IOR is used to contact the Naming Service are at http://www.dur.ac.uk/barry.cornelius/papers/IDL/CountNSIOR/.

3.7 Using OmniBroker instead of JavaIDL

Object Oriented Concepts has an ORB called *OmniBroker*. This product supports both C++ and Java and is free for non-commercial purposes: http://www.ooc.com/ob/.

OmniBroker comes with a manual that includes a *Getting started* chapter. Here the authors introduce the idea of communicating with a remote object through its IOR, i.e., this is the technique explained above in the section entitled *Using an object’s IOR instead of the Naming Service*. Elsewhere in the manual, they explain how to use OmniBroker’s implementation of the Naming Service. It is run using OmniBroker’s `nsserv` program.

In the previous section, we gave programs where remote objects were located using a Naming Service and we communicated with the Naming Service by using its IOR. If we want to use these programs with OmniBroker (instead of JavaIDL), then two changes have to be made to the `CountServer` program. First change:

```
import org.omg.CORBA.ORB;
```

to:

```
import org.omg.CORBA.ORB;
import org.omg.CORBA.BOA;
```

and also change:

```
        ORB orb = ORB.init(args, null);
```

to:

```
          ORB orb = ORB.init(args, null);
          BOA boa = orb.BOA_init(args, null);
```

The complete texts of the programs where OmniBroker is used instead of JavaIDL are at http://www.dur.ac.uk/barry.cornelius/papers/OB/CountNSIOR/.

Here are the steps needed to use these programs with OmniBroker. First, you need to compile the interface with OmniBroker's `jidl` command.

```
setenv PATH /usr/local/utils/OB/current/bin:$PATH
jidl --package OB.CountNSIOR --output-dir ../.. Count.idl
```

Because the `Count.idl` file has a module called `CountApp`, this command will create a directory called `CountApp` in which it puts the files that implement the stub and skeleton objects.

Now compile the server program with a JDK 1.0.2 or JDK 1.1.x compiler, e.g.:

```
setenv CLASSPATH /usr/local/utils/OB/current/lib/OB.jar
setenv CLASSPATH $CLASSPATH:/usr/local/utils/OB/current/lib/CosNaming.jar:../..
javac CountServer.java
```

This `javac` command will compile `CountServant.java` and `CountServer.java` and some of the files in the `CountApp` directory.

Before running the server, we need to start OmniBroker's Naming Service program. We can do this using OmniBroker's `nsserv` command:

```
nsserv --ior | tee nsserv.ior &
```

This command line will output the IOR of the Naming Service both to the screen and to the file `nsserv.ior`.

The server can then be started using a command like:

```
java OB.CountNSIOR.CountServer ‘cat nsserv.ior‘ &
```

This should output:

```
CountServant: constructor called
```

Finally, compile the client program with a JDK 1.0.2 or JDK 1.1.x compiler:

```
javac CountClient.java
```

and run it:

```
java OB.CountNSIOR.CountClient ‘cat nsserv.ior‘
```

3.8 Using a mix of JavaIDL and OmniBroker

It is possible to mix the use of vendors' ORBs, e.g., to use OmniBroker to create the server and JavaIDL to create the client. Suppose OmniBroker's Naming Service and a server created using OmniBroker have been started by the commands:

```
nsserv --ior | tee nsserv.ior &
java OB.CountNSIOR.CountServer ‘cat nsserv.ior‘ &
```

then we can communicate with this server from a client that has been created using JavaIDL:

```
java IDL.CountNSIOR.CountClient ‘cat ../../OB/CountNSIOR/nsserv.ior‘
```

It is also possible to use these the opposite way round, i.e., to use JavaIDL to create the server and OmniBroker to create the client. Suppose JavaIDL's Naming Service and a server created using JavaIDL have been started by the commands:

```
nameserv -ORBInitialPort 1050 | tee nameserv.ior &
java IDL.CountNSIOR.CountServer ‘grep IOR nameserv.ior‘ &
```

then we can communicate with this server from a client that has been created using OmniBroker:

```
java OB.CountNSIOR.CountClient ‘grep IOR ../../IDL/CountNSIOR/nameserv.ior‘
```

3.9 Using OmniBroker for a client written in C++

Because OmniBroker supports C++ as well as Java, the client and/or the server can be written in C++ instead of Java. For example, here is the client program that we have been using written in C++ instead of Java:

```
#include <iostream.h>
#include <stdlib.h>
#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include <Count.h>
int main(int argc, char* argv[], char*[]) {
   try {
      CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
      CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
      CORBA_Object_var objRef = orb -> string_to_object(argv[1]);
      CosNaming_NamingContext_var ncRef =
            CosNaming_NamingContext::_narrow(objRef);
      CosNaming_Name path;
      path.length(1);
      path[0].id = CORBA_string_dup("second Count");
      path[0].kind = CORBA_string_dup("");
      CountApp_Count_var stubCount =
            CountApp_Count::_narrow(ncRef -> resolve(path));
      cout << "before loop" << endl;
      while ( true ) {
         char stdinLine[100];
         cin >> stdinLine;
         if ( stdinLine[0] == ’q’ ) break;
         if ( stdinLine[0] == ’s’ )
            cout << stubCount -> getCount() << endl;
         else {
            int intVal = atoi(stdinLine);
            stubCount -> setCount(intVal);
         }
      }
      cout << "after loop" << endl;
   }
   catch ( ... ) {
      cout << "whoops" << endl;
   }
}
```

To support C++, OmniBroker comes with a program called `idl` that can be used to process the interface in the file `Count.idl`:

```
setenv PATH /usr/local/utils/OB/current/bin:$PATH
idl Count.idl
```

This will create the files `Count_skel.h`, `Count_skel.cpp`, `Count.h` and `Count.cpp` in the current directory. The client program can now be compiled using a C++ compiler, such as the GNU C++ compiler:

```
g++ -c --no-implicit-templates -fhandle-exceptions \
   -I. -I/usr/local/utils/OB/current/include CountClient.cpp
g++ -c --no-implicit-templates -fhandle-exceptions \
   -I. -I/usr/local/utils/OB/current/include Count.cpp
```

This will produce the files `CountClient.o` and `Count.o`. These can be linked to produce a binary in the file `CountClient` by the command:

```
g++ -o CountClient CountClient.o Count.o -L/usr/local/utils/OB/current/lib \
   -lOB -lCosNaming -lIDL -lsocket -lnsl
```

Finally, the C++ version of the client program can be run, e.g., by using the command:

```
CountClient ‘grep IOR ../../IDL/CountNSIOR/nameserv.ior‘
```

The text of the above program is at http://www.dur.ac.uk/barry.cornelius/papers/OB/CountNSIORPP/.

3.10 Further information about some other ORBs

Here are a few details about some other ORBs:

- Gerald Brose (of Freie Universitaet Berlin) has developed an ORB for Java called *JacORB*. It is written in Java and is free under the GNU General Public Licence:

  http://www.inf.fu-berlin.de/ brose/jacorb/.

- IONA Technologies has an ORB called *Orbix* that has bindings for C++, Java, Smalltalk and Ada95. For C++ see http://www.iona.com/Products/Orbix/ and for Java see

  http://www.iona.com/Products/Orbix/OrbixWeb/. The latter has free evaluation.

- The Oracle and Olivetti Research Laboratory (ORL) has an ORB called *OmniORB2* which is free under the GNU General Public Licence but which currently only supports C++: http://www.orl.co.uk/omniORB/.

- Visigenic has an ORB called *VisiBroker* which supports both C++ and Java and can be downloaded free for evaluation: http://www.visigenic.com/.

The Cetus Links on ORBs are at http://www.parallax.co.uk/cetus/oo_object_request_brokers.html.

3.11 Further information about CORBA

OMG has a lot of information about CORBA at http://www.omg.org/.

The CORBA FAQ is at http://www.cerfnet.com/ mpcline/Corba-FAQ/.

The Advanced Computing Laboratory at the Los Alamos National Laboratory has a page of resources about CORBA and the OMG at http://www.acl.lanl.gov/CORBA.

The Cetus Links on CORBA are at http://www.parallax.co.uk/cetus/oo_corba.html.

Netscape has a CORBA White Paper http://developer.netscape.com/library/wpapers/corba/, and other CORBA documents at http://developer.netscape.com/library/documentation/corba.html.

4 JDBC

4.1 What is JDBC?

The *Structured Query Language* (SQL) is a language for use with relational databases. One of the most used SQL statements is the `SELECT` statement: it is used to inspect a table of information. The statement:

```
SELECT last, first FROM ages WHERE age >= 21
```

would look at a table called `ages` and produce a new table containing the surnames and first names of the people having an age of at least 21 years.

Although SQL has been evolving since the early 1980's, it was in 1990 that the SQL Access Group defined the *Call Level Interface* (CLI) as a standard for accessing databases. To implement it, you need a *driver* that can translate a CLI call into the language used to access a particular database. Microsoft's Open Database Connectivity (ODBC), an API for Microsoft Windows that implements an extended version of the CLI, was first released in 1992. Most database vendors (including CA/Ingres, IBM, Informix, Oracle and Sybase as well as Microsoft) now support the ODBC API.

ODBC is an API written for the programming language C and although the Java Native Interface (JNI) allows a Java program to call a C function, there are many disadvantages with this approach:

- a Java program calling C code is one which is not so secure, robust and portable;
- calling C code is not usually possible from a Java applet because of security restrictions.

So, instead Java has its own API for submitting SQL statements to a database server: it is JDBC (sometimes called the *Java DataBase Connectivity*).

The JDBC API forms part of JDK 1.1.x: it is in the `java.sql` package. It was not part of JDK 1.0.2: however, if you have a JDK 1.0.2 program, a version of the JDBC API can be downloaded for use with JDK 1.0.2.

The JDBC API is implemented via a driver manager that can support multiple drivers connecting to different databases. A JDBC driver can either be written entirely in Java so that it can be downloaded as part of an applet or it can be implemented using native methods to bridge to existing database access libraries.

At the WWW page http://java.sun.com/products/jdbc/jdbc.drivers.html, Sun gives details of the drivers that are available from a number of vendors. On this page, they distinguish between four types of JDBC drivers:

1. 'The *JDBC-ODBC bridge* provides JDBC access via most ODBC drivers. Note that some ODBC binary code and in many cases database client code must be loaded on each client machine that uses this driver, so this kind of driver is most appropriate on a corporate network, or for application server code written in Java in a 3-tier architecture.'

2. 'A *native-API partly-Java driver* converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.'

3. ‘A *net-protocol all-Java driver* translates JDBC calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect all its Java clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC alternative. ...’

4. ‘A *native-protocol all-Java driver* converts JDBC calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver. Several database vendors have these in progress.’

4.2 A Type 4 driver for mSQL

Hughes Technologies has produced *mSQL* (or Mini SQL). This is a ‘light-weight relational database management system that has been designed to provide rapid access to data sets with as little system overhead as possible’. The system is comprised of a database server that will run on most Unix platforms together with various tools that allow a user or a client application to communicate with the server. Use of mSQL in any commercial environment requires the purchase of a license from Hughes Technologies. However, free licenses are provided to organisations such as Universities, schools and registered charities. For more details about mSQL, see http://www.Hughes.com.au.

George Reese, a software engineer from Minneapolis, provides a free JDBC driver for mSQL. This can be downloaded from the WWW page http://www.imaginary.com/Java/.

4.3 A Type 1 driver for Microsoft Access

An ODBC driver for use with Microsoft Windows 95/NT products, such as Access and SQL Server, can be obtained from http://support.microsoft.com/support/kb/articles/Q159/6/74.asp. Once you have an ODBC driver, then the easiest method to use is the JDBC-ODBC bridge. This is provided as part of JDK 1.1.x.

4.4 How do you use JDBC?

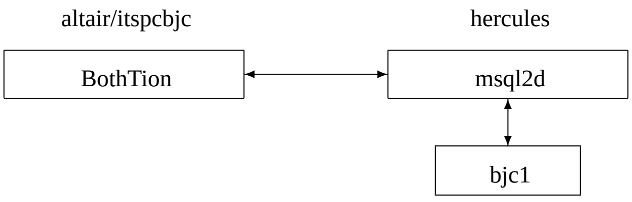
A Java program that wants to use JDBC to access a database server will normally consists of the following steps:

1. Use the `getConnection` method (of the class `java.sql.DriverManager`) to establish a connection to a database.
2. Use the `createStatement`, `prepareStatement` or `prepareCall` method.
3. For the `prepareStatement` and `prepareCall` methods, use calls of `setXXX` methods to establish any parameters that are needed by the statement.
4. Use `execute`, `executeQuery` or `executeUpdate` to execute the statement.
5. If the statement is a query, process the result set that is returned.
6. Close the statement.
7. Do steps 2 to 6 for each statement you wish to process.
8. Close the connection that was established by `getConnection`.

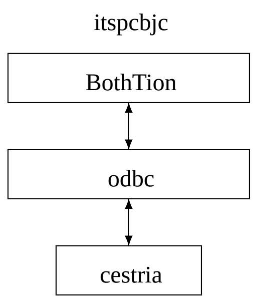
By default, all the SQL statements will automatically be *committed*: this is called *autocommit* mode. However, you can disable this and execute calls of `commit` or `rollback` yourself.

4.5 A Java application that uses JDBC

Here is a program that is divided into the above steps. The program is written so that it accesses either a database called `bjc1` that is attached to an mSQL database server running on a computer called `hercules`:



or it uses the JDBC-ODBC bridge to access a database called `cestria`:



The choice depends on a value passed on the command line used to start the program:

```
import java.sql.Connection;                                 // BothTion.java
import java.io.DataInputStream;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Statement;
public class BothTion {
   public static void main(String[] args) {
      try {
         DataInputStream stdin = new DataInputStream(System.in);
         Connection tConnection = null;
         String tURLString;
         if ( args[0].equals("bjc1") ) {
            Class.forName("COM.imaginary.sql.msql.MsqlDriver");
            tURLString = "jdbc:msql://hercules.dur.ac.uk:4333/bjc1";
         }
         else {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            tURLString = "jdbc:odbc:cestria";
         }
         boolean tStillTrying = true;
         int tNumberOfTries = 0;
         while (tStillTrying) {
            try {
               tConnection = DriverManager.getConnection(tURLString, "", "");
               tStillTrying = false;
            }
            catch (Exception rException) {
               tNumberOfTries++;
               tStillTrying = (tNumberOfTries > 20);
            }
         }
         if ( tConnection == null )  System.out.println("tConnection is null");
         check(tConnection.getWarnings());
         Statement tStatement = tConnection.createStatement();
         boolean tMoreSQLRequests = true;
         while (tMoreSQLRequests) {
            System.out.print("> ");
            String tSQLString = stdin.readLine();
            try {
               if ( tSQLString.startsWith("select") ) {
                  ResultSet tResultSet = tStatement.executeQuery(tSQLString);
                  showResultSet(tResultSet);
                  tResultSet.close();
               }
               else if ( tSQLString.startsWith("quit") ) {
                  tMoreSQLRequests = false;
               }
               else {
                  // assume line contains a SQL insert, update or delete
                  int tNumberOfRows = tStatement.executeUpdate(tSQLString);
                  }
               }
            catch (Exception rException) {
               outputException(rException);
            }
         }
         tStatement.close();
         tConnection.close();
      }
      catch (Exception rException) {
         outputException(rException);
      }
   }
   private static void showResultSet(ResultSet rResultSet) throws SQLException {
      // Get the ResultSetMetaData.  This will be used for
      // the column headings
      ResultSetMetaData tResultSetMetaData = rResultSet.getMetaData();
      // Get the number of columns in the result set
      int tNumCols = tResultSetMetaData.getColumnCount();
      for (int tColNum=1; tColNum<=tNumCols; tColNum++) {
         if (tColNum>1) System.out.print(", ");
         System.out.print(tResultSetMetaData.getColumnLabel(tColNum));
      }
      System.out.println();
      // Display data, fetching until end of the result set
      boolean tMoreRows = rResultSet.next();
      while (tMoreRows) {
         // Loop through each column, getting the
         // column data and displaying
         for (int tColNum=1; tColNum<=tNumCols; tColNum++) {
            if (tColNum>1) System.out.print(", ");
            System.out.print(rResultSet.getString(tColNum));
         }
         System.out.println();
         // Fetch the next result set row
         tMoreRows = rResultSet.next();
      }
   }
   private static boolean check(SQLWarning rSQLWarning) throws SQLException {
      boolean tReturnCode = false;
      if (rSQLWarning != null) {
         System.out.println("\n *** Warning ***");
         tReturnCode = true;
         while (rSQLWarning != null) {
            System.out.println("SQLState: " + rSQLWarning.getSQLState());
            System.out.println("Message:  " + rSQLWarning.getMessage());
            System.out.println("Vendor:   " + rSQLWarning.getErrorCode());
            System.out.println();
            rSQLWarning = rSQLWarning.getNextWarning();
         }
      }
      return tReturnCode;
   }
   private static void outputException(Exception rException) {
      if ( rException instanceof SQLException ) {
         SQLException tSQLException = (SQLException) rException;
         System.out.println("\n*** SQLException caught ***");
         while (tSQLException != null) {
            System.out.println("SQLState: " + tSQLException.getSQLState());
            System.out.println("Message:  " + tSQLException.getMessage());
            System.out.println("Vendor:   " + tSQLException.getErrorCode());
            tSQLException = tSQLException.getNextException();
            System.out.println();
         }
      }
      else {
         rException.printStackTrace();
      }
   }
}
```

4.6 What steps are needed to run the above example?

First, compile the program with a JDK 1.1.x compiler:

```
javac BothTion.java
```

At runtime, the program either loads the class `COM.imaginary.sql.msql.MsqlDriver` or it loads the class `sun.jdbc.odbc.JdbcOdbcDriver`. The latter (which is the JDBC-ODBC Bridge) forms part of the JDK 1.1.x distribution whereas the class files for George Reese's mSQL JDBC driver have been put in the directory `/users/dcl0bjc/public_html/Java/COM/imaginary/sql/msql`. So at runtime the `CLASSPATH` variable needs to be set to ensure that the Java interpreter finds the `MsqlDriver` class:

```
setenv CLASSPATH /users/dcl0bjc/public_html/Java:.
```

When we run the `BothTion` program, an argument on the command line is used to indicate which database we want the program to access:

```
java BothTion bjc1
```

4.7 Further information about JDBC

The texts of the above programs are at http://www.dur.ac.uk/barry.cornelius/papers/JDBC/BothTion/.

Sun has a JDBC page at http://java.sun.com/products/jdk/1.1/docs/guide/jdbc/. A useful guide to JDBC is at http://java.sun.com/products/jdk/1.1/docs/guide/jdbc/getstart/introTOC.doc.html.

Sun's main JDBC page at http://java.sun.com/products/jdbc/ includes details about vendors who have JDBC drivers, an FAQ, and a link to download a JDBC for use with JDK 1.0.2.

Another JDBC FAQ is at http://javanese.yoyoweb.com/JDBC/FAQ.txt.

As mentioned above, details about mSQL can be obtained from http://www.Hughes.com.au and details about the mSQL-JDBC driver can be obtained from http://www.imaginary.com/Java/. This latter page also contains a link to some documentation for the driver, details about the `msql-jdbc` mailing list, and some details about George Reese's book *Database Programming with JDBC and Java*. This book is published by O'Reilly; its ISBN is ISBN 1-56592-270-0.

Netscape has a technical note about writing database applications in Java at

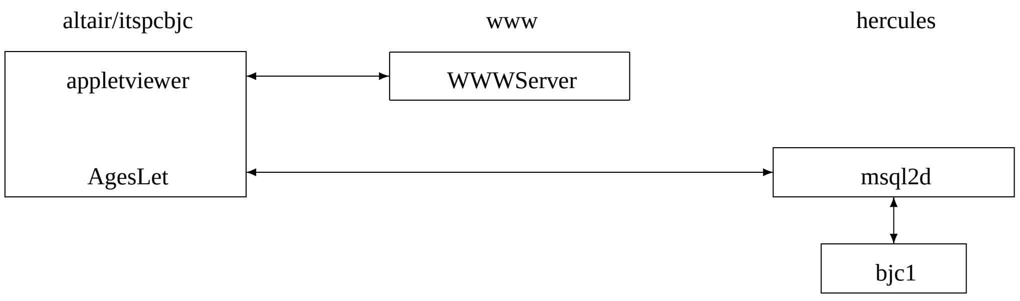
http://developer.netscape.com/library/technote/database/web/.

5 Signed Applets

5.1 Overview

So far we have seen three different techniques (RMI, CORBA and JDBC): in each case, a Java *application* running on one computer has contacted another computer. When a WWW user wishes to download a Java *applet* from a WWW server, there is no problem if that applet wants to access a program running on the same computer as the WWW server. For example, it is reasonably easy to provide a WWW page that runs a Java applet which uses JDBC to connect to a database server (e.g., the MSQL server) if the database server is on the same computer as the WWW server that delivers the WWW page.

However, if the WWW server and the database server are on different computers:



Java's security restrictions are such that the applet is not allowed access to the database server unless the applet is a *trusted applet*. An applet is *trusted* if the owner of the applet has added a *digital signature* to the applet and the visitor to the WWW page has signified to their WWW browser that they trust the person described in the digital signature.

In order for you to add a digital signature to an applet you need to know about JDK's `javakey` command in order to generate keys, certificates and signed applets. Furthermore, in order to use `javakey`, your applet's class must be stored in a *Java Archive*, and so you will also have to know about JDK's `jar` command (in order to create a Java Archive). Although the `jar` and `javakey` commands of JDK 1.1.x enable you to generate signed applets, these signed applets are not understood by Netscape Navigator 4.03. So in order for a visitor to your WWW page to run your signed applet, he/she will have to use the appletviewer that comes with JDK 1.1.x. In a later version of the JDK (maybe as late as JDK 1.2), the security aspects of the JDK will allow those used by Netscape Navigator 4.03 and so hopefully the two will interoperate.

5.2 Creating a Java Archive containing a Java Applet

We will use the following Java applet `AgesLet.java` to communicate with an mSQL database server using the JDBC API:

```
import java.applet.Applet;                                  // AgesLet.java
import java.awt.Button;
import java.sql.Connection;
import java.sql.DriverManager;
import java.awt.Event;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Statement;
import java.awt.TextArea;
import java.awt.TextField;
public class AgesLet extends Applet {
   public void init() {
      add(iQueryTextField);
      add(iQueryButton);
      add(iResultsTextArea);
      try {
         Class.forName("COM.imaginary.sql.msql.MsqlDriver");
         String tURLString = "jdbc:msql://hercules.dur.ac.uk:4333/bjc1";
         boolean tStillTrying = true;
         int tNumberOfTries = 0;
         while (tStillTrying) {
            try {
               iConnection = DriverManager.getConnection(tURLString, "", "");
               tStillTrying = false;
            }
            catch (Exception rException) {
               tNumberOfTries++;
               tStillTrying = (tNumberOfTries > 20);
            }
         }
         if ( iConnection == null)  iResultsTextArea.append("is null\n");
         check(iConnection.getWarnings());
      }
      catch (Exception rException) {
         outputException(rException);
      }
   }
   public void destroy() {
      try {
         iConnection.close();
      }
      catch (Exception rException) {
         outputException(rException);
      }
   }
   public boolean handleEvent(Event rEvent) {
      if (rEvent.target.equals(iQueryButton) && rEvent.id==Event.ACTION_EVENT) {
         try {
            String tSQLString = iQueryTextField.getText();
            Statement tStatement = iConnection.createStatement();
            if ( tSQLString.startsWith("select") ) {
               ResultSet tResultSet = tStatement.executeQuery(tSQLString);
               displayResultSet(tResultSet);
               tResultSet.close();
            }
            else {
               int tNumberOfRows = tStatement.executeUpdate(tSQLString);
            }
            tStatement.close();
         }
         catch (Exception rException) {
            outputException(rException);
         }
         iResultsTextArea.append("--------------------------------\n");
         return true;
      }
      return false;
   }
   private TextField iQueryTextField = new TextField(80);
   private Button iQueryButton = new Button("click here");
   private TextArea iResultsTextArea = new TextArea(20, 80);
   private Connection iConnection = null;
   private void displayResultSet(ResultSet rResultSet) throws SQLException {
      ResultSetMetaData tResultSetMetaData = rResultSet.getMetaData();
      int tNumCols = tResultSetMetaData.getColumnCount();
      boolean tMoreRows = rResultSet.next();
      while (tMoreRows) {
         for (int tColNum=1; tColNum<=tNumCols; tColNum++) {
            iResultsTextArea.append(rResultSet.getString(tColNum));
            iResultsTextArea.append(" ");
         }
         iResultsTextArea.append("\n");
         tMoreRows = rResultSet.next();
      }
   }
   private static boolean check(SQLWarning rSQLWarning) throws SQLException  {
```

... same as the `check` method of `BothTion.java` (see Section 4.5)

```
   }
   private static void outputException(Exception rException) {
```

... same as the `outputException` method of `BothTion.java` (see Section 4.5)

```
   }
}
```

This file can be compiled using the command:

```
javac AgesLet.java
```

The compiler produces a file called `AgesLet.class`.

You will need to put the `classes/COM` directory of the mSQL-JDBC 1.0a4 distribution into the current directory. On a Unix system, you can do this by a command like:

```
cp -pr /usr/local/utils/mSQL/JDBC/1.0a4/classes/COM .
```

This creates a subdirectory called `COM` in the current directory.

You now need to produce a *Java Archive*. This is Java's way of packaging up a lot of files to produce one file. It is done using JDK 1.1.x's `jar` command at the Unix/MS-DOS prompt:

```
jar cf AgesLet.jar AgesLet.class COM
```

This command will produce a *jar file* in the file `AgesLet.jar`.

5.3 Adding a Digital Signature to a Java Archive

You now need to digitally sign the jar file. This is complicated. First, you need to use JDK 1.1.x's `javakey` command to generate an *identity*. For example, suppose you want to use the identity `Barry`. You can do this by typing the Unix/MS-DOS command:

```
javakey -cs "Barry" true
```

On a Unix system, this command creates a file called `identitydb.obj` in your home directory, whereas on Microsoft's Windows 95/NT it will create the file in the top directory of your JDK's directory tree. Whenever you use the `javakey` to make a change, you can check what has happened using:

```
javakey -ld
```

You can now create a pair of *key*s (a *private key* and a *public key*) to be used by this identity. This can be done using the command:

```
javakey -gk "Barry" DSA 1024
```

You now use a text editor to create a *certificate directive file*. Suppose the file `Barry.cdirective` contains the lines:

```
issuer.name=Barry
issuer.cert=1
subject.name=Barry
subject.real.name=Barry Cornelius
subject.org.unit=IT Service
subject.org=University of Durham
subject.country=UK
start.date=1 Nov 1997
end.date=30 Nov 1997
serial.number=1
out.file=Barry.x509
```

You can then issue the command:

```
javakey -gc Barry.cdirective
```

This alters the file `identitydb.obj` and also creates the file `Barry.x509`.

You are now (at long last) in a position to sign your jar file. First, you need to use a text editor to create a *signature directive file*. Suppose the file `Barry.sdirective` contains the lines:

```
signer=Barry
cert=1
chain=0
signature.file=Barry
```

Then type the command:

```
javakey -gs Barry.sdirective AgesLet.jar
```

This creates a signed version of your jar file in a file called `AgesLet.jar.sig`.

5.4 Creating a WWW page that loads the Java Archive

You can then produce an appropriate HTML file in a file called `AgesLet.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML><HEAD><TITLE> The AgesLet Program </TITLE></HEAD>
<BODY>
<APPLET ARCHIVE="AgesLet.jar.sig" CODE="AgesLet.class" WIDTH=700 HEIGHT=700>
</APPLET>
</BODY></HTML>
```

Note that the `APPLET` in this HTML file has both an `ARCHIVE` and a `CODE` section. When a WWW browser accesses this page, it will download all the classes in the Java Archive `AgesLet.jar.sig` and then it will start interpreting the class that came from the file `AgesLet.class`.

5.5 Accessing your WWW page from a WWW browser

The author of this applet can now run it using the appletviewer that comes with JDK 1.1.x, e.g., by typing the Unix/MS-DOS command:

```
appletviewer http://www.dur.ac.uk/~dcl0bjc/Java/Signed/Ages/AgesLet.html
```

If another user wants to execute the above appletviewer command perhaps somewhere else in the world, then first they will have to have a copy of your x509 certificate file (that is in the file `Barry.x509`) and then they will have to execute the Unix/MS-DOS command:

```
javakey -c "Barry" true
```

to introduce Barry as an identity and then execute:

```
javakey -ic "Barry" Barry.x509
```

to add Barry's certificate to their `identitydb.obj` file. Having done that, appletviewer will execute any applet that have been signed by `Barry`.

5.6 Further information about Signed Applets

The texts of the above programs are at http://www.dur.ac.uk/barry.cornelius/papers/Signed/Ages/.

JDK 1.1.x's appletviewer must be used instead of a browser such as Netscape Navigator. This is because JDK 1.1.x's certificates are X509v1 certificates for DSA whereas Netscape's Navigator uses X509v3. However, Sun has said that the next release of the JDK will be able to read X509v3 certificates, and hopefully 'it will be a straightforward matter for the Java-enabled browsers to use the JDK certificate APIs to import and parse the JDK signing certificates'. In the next release, Sun will replace the `javakey` tool by `keytool`, `jarsigner` and `policytool`, and the security database used by these new tools will be incompatible with the database in the `identitydb.obj` file. Although these changes may take place in JDK 1.1.6 (possibly due in December 1997), they are more likely to occur in JDK 1.2 (possibly due in Q2 1998).

Sun has one page about security at http://java.sun.com/products/jdk/1.1/docs/guide/security/. However, their main security page is at http://java.sun.com/security/. There is a guide to using `javakey` at http://java.sun.com/security/usingJavakey.html and recommendations on its use at http://java.sun.com/security/policy.html. The details about the replacement of `javakey` by `keytool`, `jarsigner` and `policytool` is given at http://java.sun.com/products/jdk/1.2/docs/guide/security/.

There is a section about keys, certificates and digital signatures in Sun's *Tutorial* on Java: http://java.sun.com/docs/books/tutorial/security1.1/.

Netscape's WWW page for signed objects is at

http://developer.netscape.com/library/documentation/signedobj/.

6 Three Tier Systems

6.1 Two Tier and Three Tier Systems

In the past, many distributed systems have been built that are *two tier*. This is a system that has a client program communicating with a server program. The examples of RMI, JavaIDL and JDBC given earlier are simple two-tier systems. Typically, the user interface is in the client and the server is there to process data, e.g., to manipulate a database. The complicated application logic (or business logic) is often put in the client (or in the server). More recently, *three tier* systems have come into favour. This is where an additional tier is interposed: it is there to handle the application logic leaving the client just to handle the user interface.

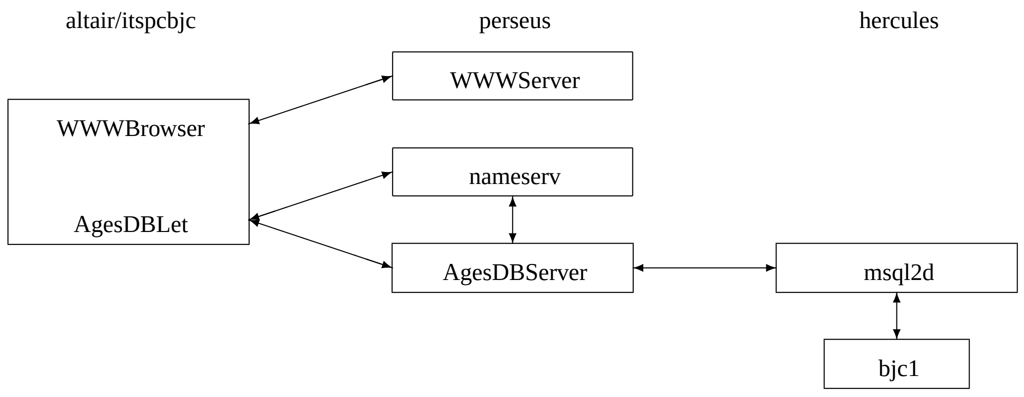
There are a number of advantages to this approach. For example:

- The code running on the client is small.
- When different client programs are needed to handle the differences between different client platforms, the amount of code that has to be provided is small because it only has to handle the user interface.
- It is possible to write the code running on the middle tier so that it handles requests from more than one client. In this way, requests can be more efficiently handled.

6.2 An example of a Three Tier System

We will now look at an unsophisticated three tier system. It will use a WWW browser on a computer called `altair` that downloads a Java applet from a WWW server running on `perseus`. The Java applet will display some buttons and textfields. A click on a button will cause the Java applet to send a request (using JavaIDL) to a

Java application that is also running on `perseus`. This Java application will send (using JDBC) an appropriate SQL statement to a database server running on a computer called `hercules`:



6.3 The code for this Three Tier System

When the following WWW page is visited, the WWW browser will download all the classes in the Java Archive `AgesDBLet.jar` and then it will start interpreting the class that came from the file `AgesDBLet.class` passing to it the two parameters mentioned in the `PARAM` tags:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML><HEAD><TITLE> The AgesDBLet Program </TITLE></HEAD>
<BODY>
<APPLET ARCHIVE=AgesDBLet.jar CODE="AgesDBLet.class" WIDTH=700 HEIGHT=700>
<PARAM NAME=org.omg.CORBA.ORBInitialHost VALUE="perseus.dur.ac.uk">
<PARAM NAME=org.omg.CORBA.ORBInitialPort VALUE=1050>
</APPLET>
</BODY></HTML>
```

The Java applet is going to communicate using JavaIDL with an object that is part of a Java application running on `perseus`. Suppose the object has the following interface (given in IDL):

```
module AgesDBApp {
   interface AgesDB {
      void getPerson(in string rLast, out string rFirst, out string rAge);
      void addPerson(in string rLast, in string rFirst, in string rAge);
      void removePerson(in string rLast);
   };
};
```

Here is the text of the Java applet `AgesDBLet.java`:

```
import AgesDBApp.AgesDB;                                          // AgesDBLet.java
import AgesDBApp.AgesDBHelper;
import java.applet.Applet;
import java.awt.Button;
import java.awt.Event;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CORBA.ORB;
import org.omg.CORBA.StringHolder;
import java.awt.TextField;
public class AgesDBLet extends Applet {
   public void init() {
      // establish connection to the remote object through the iAgesDB variable
      try {
         ORB orb = ORB.init(this, null);
         org.omg.CORBA.Object objRef =
               orb.resolve_initial_references("NameService");
         NamingContext ncRef = NamingContextHelper.narrow(objRef);
         NameComponent nc = new NameComponent("AgesDB", "");
         NameComponent path[] = {nc};
         iAgesDB = AgesDBHelper.narrow(ncRef.resolve(path));
      }
      catch (Exception rException) {
         rException.printStackTrace();
      }
      // add the three buttons to the applet
      add(iGetButton);
      add(iAddButton);
      add(iRemoveButton);
      // add the three textfields to the applet
      add(iLastTextField);
      add(iFirstTextField);
      add(iAgeTextField);
   }
   public boolean handleEvent(Event rEvent) {
      try {
         if (rEvent.id == Event.ACTION_EVENT) {
            if (rEvent.target.equals(iGetButton)) {
               // In the interface, getPerson has two 'out' parameters. However,
               // Java only has 'in'.  The class org.omg.CORBA.StringHolder
               // can be used to provide a wrapper object for a string value.
               StringHolder tFirstHolder = new StringHolder();
               StringHolder tAgeHolder = new StringHolder();
               iAgesDB.getPerson(iLastTextField.getText(),
                     tFirstHolder, tAgeHolder);
               iFirstTextField.setText(tFirstHolder.value);
               iAgeTextField.setText(tAgeHolder.value);
               return true;
            }
            if (rEvent.target.equals(iAddButton)) {
               iAgesDB.addPerson(iLastTextField.getText(),
                     iFirstTextField.getText(), iAgeTextField.getText());
               return true;
            }
            if (rEvent.target.equals(iRemoveButton)) {
               iAgesDB.removePerson(iLastTextField.getText());
               return true;
            }
         }
         return false;
      }
      catch (Exception rException) {
         rException.printStackTrace();
         return false;
      }
   }
   private AgesDB iAgesDB = null;
   private Button iGetButton = new Button("Get");
   private Button iAddButton = new Button("Add");
   private Button iRemoveButton = new Button("Remove");
   private TextField iLastTextField = new TextField(30);
   private TextField iFirstTextField = new TextField(30);
   private TextField iAgeTextField = new TextField(4);
}
```

In the Java application on `perseus`, an object of the following class implements the interface given earlier (in IDL) by extending the class `_AgesDBImplBase`:

```
import AgesDBApp._AgesDBImplBase;                     // AgesDBServant.java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Statement;
import org.omg.CORBA.StringHolder;
public class AgesDBServant extends _AgesDBImplBase {
   public AgesDBServant() {
      super();
      System.out.println("AgesDBServant: constructor start");
      try {
         Class.forName("COM.imaginary.sql.msql.MsqlDriver");
         String tURLString = "jdbc:msql://hercules.dur.ac.uk:4333/bjc1";
         boolean tStillTrying = true;
         int tNumberOfTries = 0;
         while (tStillTrying) {
            try {
               iConnection = DriverManager.getConnection(tURLString, "", "");
               tStillTrying = false;
            }
            catch (Exception rException) {
               tNumberOfTries++;
               tStillTrying = (tNumberOfTries > 20);
            }
         }
         if ( iConnection == null )  System.out.println("iConnection is null");
         check(iConnection.getWarnings());
      }
      catch (Exception rException) {
         outputException(rException);
      }
      System.out.println("AgesDBServant: constructor finish");
   }
   public synchronized void getPerson(String rLast,
                                    StringHolder rFirst, StringHolder rAge) {
      System.out.println("AgesDBServant: getPerson start");
      rFirst.value = "";
      rAge.value = "";
      try {
         Statement tStatement = iConnection.createStatement();
         String tSQLString = "select * from ages where last = ’" + rLast + "’";
         System.out.println(tSQLString);
         ResultSet tResultSet = tStatement.executeQuery(tSQLString);
         if ( tResultSet.next() ) {
            rFirst.value = tResultSet.getString(1);
            rAge.value = tResultSet.getString(3);
         }
         tResultSet.close();
         tStatement.close();
      }
      catch (Exception rException) {
         outputException(rException);
      }
      System.out.println("AgesDBServant: getPerson finish");
   }
   public synchronized void addPerson(String rLast, String rFirst, String rAge){
      System.out.println("AgesDBServant: addPerson start");
      try {
         Statement tStatement = iConnection.createStatement();
         String tSQLString = "insert into ages values(’"
                              + rFirst + "’, ’" + rLast + "’, " + rAge + ")";
         System.out.println(tSQLString);
         int tNumberOfRows = tStatement.executeUpdate(tSQLString);
         tStatement.close();
      }
      catch (Exception rException) {
         outputException(rException);
      }
      System.out.println("AgesDBServant: addPerson finish");
   }

   public synchronized void removePerson(String rLast) {
      System.out.println("AgesDBServant: removePerson start");
      try {
         Statement tStatement = iConnection.createStatement();
         String tSQLString = "delete from ages where last = ’" + rLast + "’";
         System.out.println(tSQLString);
         int tNumberOfRows = tStatement.executeUpdate(tSQLString);
         tStatement.close();
      }
      catch (Exception rException) {
         outputException(rException);
      }
      System.out.println("AgesDBServant: removePerson finish");
   }
   private Connection iConnection = null;
   private static boolean check(SQLWarning rSQLWarning) throws SQLException {
```

... same as the `check` method of `BothTion.java` (see Section 4.5)

```
   }
   private static void outputException(Exception rException) {
```

... same as the `outputException` method of `BothTion.java` (see Section 4.5)

```
   }
}
```

The Java application running on `perseus` creates an object of the above class and registers it:

```
import org.omg.CosNaming.NameComponent;                 // AgesDBServer.java
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CORBA.ORB;
public class AgesDBServer {
   public static void main(String[] args) {
      try{
         ORB orb = ORB.init(args, null);
         AgesDBServant AgesDBRef = new AgesDBServant();
         orb.connect(AgesDBRef);
         org.omg.CORBA.Object objRef =
               orb.resolve_initial_references("NameService");
         NamingContext ncRef = NamingContextHelper.narrow(objRef);
         NameComponent nc = new NameComponent("AgesDB", "");
         NameComponent path[] = {nc};
         ncRef.rebind(path, AgesDBRef);
         java.lang.Object sync = new java.lang.Object();
         synchronized (sync) {
            sync.wait();
         }
      }
      catch (Exception rException) {
         rException.printStackTrace();
      }
   }
}
```

The above Java application uses JDBC to talk to a database server running on `hercules`.

6.4 What steps are needed to run this example?

First, compile the interface in the file `AgesDB.idl` by executing JavaIDL's `idltojava` command:

```
setenv JAVAIDL_HOME /users/hplives/JavaIDL-1.1EA
setenv PATH $JAVAIDL_HOME/bin:$PATH
idltojava -fclient -fserver AgesDB.idl
```

This command creates a directory called `AgesDBApp` for the stub and skeleton classes.

We need to compile `AgesDBServer.java`, the Java application of the middle tier. In order to compile this file, we will need access to the bytecodes of the classes of the JavaIDL API. Earlier, this was done by setting the `CLASSPATH` variable to include the file `$JAVAIDL_HOME/lib/classes.zip`. However, for a technical reason (to do with JavaStations), this file will be unzipped, to create two directories in the current directory called `com` and `org`. Having done that, we can compile the Java application of the middle tier with a JDK 1.1.x compiler:

```
javac AgesDBServer.java
```

Before running the server, we need to start `nameserv` (JavaIDL's Naming Service program):

```
nameserv -ORBInitialPort 1050 &
```

The server can then be started using the commands:

```
setenv CLASSPATH /users/dcl0bjc/public_html/Java:.
java AgesDBServer -ORBInitialPort 1050 &
```

Finally, compile the client program with a JDK 1.0.2 or JDK 1.1.x compiler:

```
javac AgesDBLet.java
```

This compilation will use the JavaIDL classes in the `com` and `org` directories that were created earlier. In order to simplify the loading of these classes whenever the Java applet is downloaded, we will use JDK's 1.1.x's `jar` command to build a Java Archive that contains all of these files:

```
jar cf AgesDBLet.jar AgesDBLet.class AgesDBApp org com
```

This Java applet can be run by anyone having a WWW browser that can handle Java Archives. Examples are Netscape's Navigator 3.01 (or later) and JDK 1.1.x's appletviewer:

```
appletviewer http://perseus.dur.ac.uk/~dcl0bjc/Java/ThreeTier/Ages/AgesDBLet.html
```

Because of an incompatibility, such applets do not work with Netscape's Navigator 4.0x. I have also used this applet as the applet behind one of the buttons on the desktop when using *HotJava Views* on a *JavaStation*.

6.5 Using a C++ program as a client of this Three Tier System

Here is a C++ program that can be used as the client of this Three Tier System:

```
#include <iostream.h>
#include <stdlib.h>
#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include <AgesDB.h>
int main(int argc, char* argv[], char*[]) {
   try {
      CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
      CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
      CORBA_Object_var objRef = orb -> string_to_object(argv[1]);
      CosNaming_NamingContext_var ncRef =
            CosNaming_NamingContext::_narrow(objRef);
      CosNaming_Name path;
      path.length(1);
      path[0].id = CORBA_string_dup("AgesDB");
      path[0].kind = CORBA_string_dup("");
      AgesDBApp_AgesDB_var stubAgesDB =
            AgesDBApp_AgesDB::_narrow(ncRef -> resolve(path));
      char stdinLine[100];
      char in_last[10];
      char in_first[10];
      char in_age[10];
      char *out_firstptr = 0;
      char *out_ageptr = 0;
      bool stillMoreRequests = true;
      while ( stillMoreRequests ) {
         cout << "agqr: ";  cin >> stdinLine;
         switch ( stdinLine[0] ) {
            case 'a':
               cout << "last: ";  cin >> in_last;
               cout << "first: "; cin >> in_first;
               cout << "age: ";   cin >> in_age;
               stubAgesDB -> addPerson(in_last, in_first, in_age);
               break;
            case 'g':
               cout << "last: ";  cin >> in_last;
               stubAgesDB -> getPerson(in_last, out_firstptr, out_ageptr);
               cout << out_firstptr << " " << out_ageptr << endl;
               break;
            case 'q':
               stillMoreRequests = false;
               break;
            case 'r':
               cout << "last: ";  cin >> in_last;
               stubAgesDB -> removePerson(in_last);
               break;
            default:
               break;
         }
      }
   }
   catch ( ... ) {
      cout << "whoops" << endl;
   }
}
```

With the help of OmniBroker, this C++ program can be compiled and executed in a similar way to the C++ program given earlier.

6.6 Further information about Three Tier Systems

The texts of the above programs are at http://www.dur.ac.uk/barry.cornelius/papers/ThreeTier/Ages/ and at http://www.dur.ac.uk/barry.cornelius/papers/OB/ThreeTierPP/.

Sun has a WWW page describing Three Tier Systems at

http://java.sun.com/products/jdk/1.2/docs/guide/idl/jidlDistApp.html.

For more about JavaStations and HotJava Views, see http://www.sun.com/javastation/ and http://java.sun.com/products/hotjavaviews/.

7 Other Resources

One good book is *Client/Server Programming with Java and CORBA* (1997) by Robert Orfali and Dan Harkey published by Wiley (0-471-16351-1): http://www.corbajava.engr.sjsu.edu/Progrmg.htm. It contains extensive sections on both CORBA and JDBC, and the use of CORBA and JDBC to build multi-tier systems. They also compare CORBA with other methods: sockets, HTTP/CGI and RMI. The second edition of this book (0-471-24578-X) 'with 250 pages of new material' is due out in February 1998. Working with Jeri Edwards, Orfali and Harkey have three other books in this area: *Essential Client/Server Survival Guide* (2nd edition 1996), *Essential Distributed Objects Survival Guide* (1996) and *Instant CORBA* (1997). See http://www.corbajava.engr.sjsu.edu/mbooks.htm.

Prashant Sridharan's book *Advanced Java Networking* explores the topics covered in this document. Its ISBN is 0-13-749136-0 and it is published by Prentice-Hall: http://www.prenhall.com/.

Sams.net Publishing have several overlapping books on Java, and their *Unleashed* book keeps moving into new editions. Many of the chapters of *Java 1.1 Unleashed Third Edition* describe various aspects of client-server computing. Its ISBN is 1-57521-298-6. For more details, see: http://merchant.superlibrary.com:8000/catalog/hg/PRODUCT/PAGE/15752/bud/1575212986.html.

The text of this document is updated from time to time. The latest version is available at:

http://www.dur.ac.uk/ dcl0bjc/Java/. This WWW page also contains other documents on Java including a tutorial on Java, an overview of the material of the tutorial and some notes comparing Java and C++.