Programming
In Java

BOOK 5

Client-Server

CONTENTS

1 SERVLETS

“The Power behind the Server”… so say Sun and here’s also what they say,

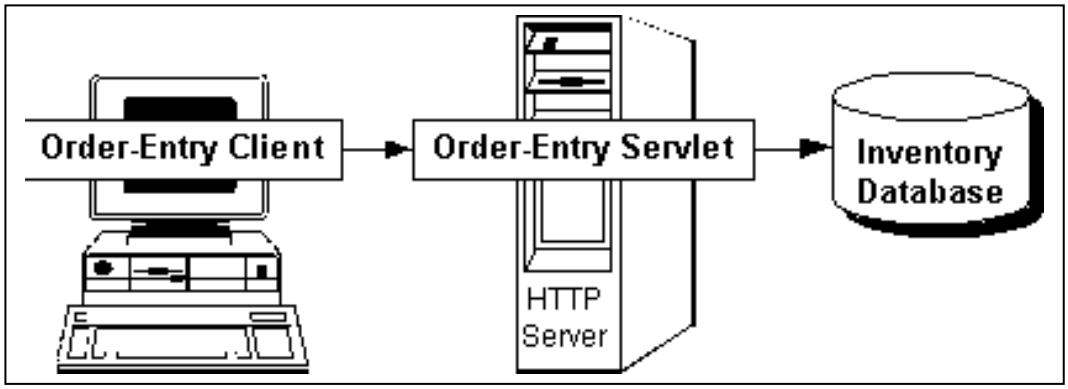
JavaTM Servlet technology provides web developers with a simple, consistent mechanism for extending the functionality of a web server and for accessing existing business systems. A servlet can almost be thought of as an applet that runs on the server side -- without a face. Java servlets have made many web applications possible.

Servlets are the Java platform technology of choice for extending and enhancing web servers. Servlets provide a component-based, platform-independent method for building web-based applications, without the performance limitations of CGI programs. And unlike proprietary server extension mechanisms (such as the Netscape Server API or Apache modules), servlets are server- and platform-independent. This leaves you free to select a "best of breed" strategy for your servers, platforms, and tools.

Servlets have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls and receive all the benefits of the mature Java language, including portability, performance, reusability, and crash protection.

Today, servlets are a popular choice for building interactive web applications. Third-party servlet containers are available for Apache Web Server, iPlanet Web Server, Microsoft IIS, and others. Servlet containers can also be integrated with web-enabled application servers, such as BEA WebLogic Application Server, IBM WebSphere, iPlanet Application Server, and others.”

Servlets are Java code modules that extend request/response-oriented servers, such as Java-enabled web servers. For example, a servlet might be responsible for taking data in an HTML order-entry form and applying the business logic used to update a company's order database.



Servlets are to servers what applets are to browsers. Unlike applets, however, servlets have no graphical user interface.

Servlets can be embedded in many different servers because the servlet API, which you use to write servlets, assumes nothing about the server's environment or protocol. Servlets have become most widely used within HTTP servers; many web servers support Java Servlet technology.

1.1 Use Servlets instead of CGI Scripts!

Servlets are an effective replacement for CGI scripts. They provide a way to generate dynamic documents that is both easier to write and faster to run. Servlets also address the problem of doing server-side programming with platform-specific APIs: they are developed with the Java Servlet API, a standard Java extension.

You can use servlets to handle HTTP client requests. For example, have servlets process data POSTed over HTTPs using an HTML form, including purchase order or credit card data. A servlet like this could be part of an order-entry and processing system, working with product and inventory databases, and perhaps an on-line payment system.

1.1.1 Other Uses for Servlets

Here are a few more of the many applications for servlets:

- Allowing collaboration between people. A servlet can handle multiple requests concurrently, and can synchronize requests. This allows servlets to support systems such as on-line conferencing.

- Forwarding requests. Servlets can forward requests to other servers and servlets. Thus servlets can be used to balance load among several servers that mirror the same content, and to partition a single logical service over several servers, according to task type or organizational boundaries.

Servlets are pieces of Java source code that add functionality to a web server in a manner similar to the way applets add functionality to a browser. Servlets are designed to support a request/response computing model that is commonly used in web servers. In a request/response model, a client sends a request message to a server and the server responds by sending back a reply message. The 'message' of course could be data that is sent to the server, and vice-versa.

From the Java Servlet Development Kit (JSDK), you use the Java Servlet API to create servlets for responding to requests from clients. These servlets can many tasks, like processing HTML forms with a custom servlet or manage middle-tier processing to connect to existing data sources behind a corporate firewall. In addition, servlets can maintain services, like database sessions, between requests to manage resources better than Common Gateway Interface (CGI) technologies.

The Java Servlet API is based on several Java interfaces that are provided in standard Java extension (javax) packages.

1.2 The Servlet Add on Library – Javax

Before you can use the servlet classes in your programs you will need to install the sun add on library javax. This library must be available at the time you compile your servlet code. This means it will have to be on the compilation classpath.

1.3 Using the Tomcat Web Server

You will also need a servlet enabled HTTP Web server. I used the popular Tomcat Web Server. There are other Web Servers that can also handle servlets such as the one provided by sun, JavaWebServer. In the examples that follow it is assumed that the Tomcat web server is installed as

```
c:\tomcat
```

1.3.1 The Servlets Directory on the Tomcat Web Server

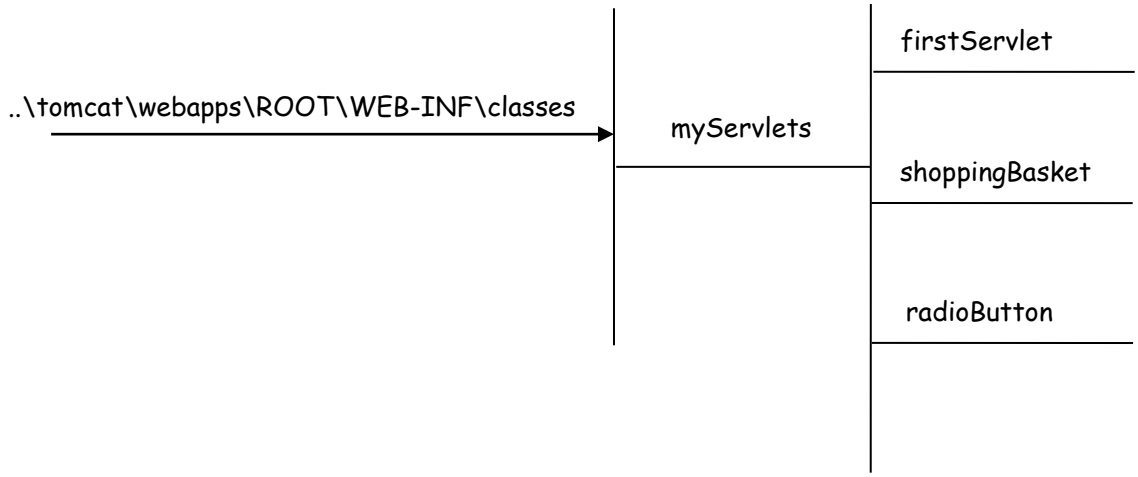
Servlets are installed in

```
c:\tomcat\webapps\ROOT\WEB-INF\classes
```

directory. The Tomcat server detects when servlets have been added to this directory.

1.3.2 The Importance of the Servlets Package

The idea of a servlet belonging to a package package is vitally important. The directory structure to hold the servlets used in these notes was organised as follows,



This structure means that all servlets in my servlet library will have to be headed with the line package myServlets.actualPackage;

1.4 A First Servlet Example

I will now demonstrate how to create, install and access a servlet.

This was the source code for MyFirstServlet.java. When invoked all it does is to return a very simple web page to the client browser.

```
package myServlets.firstServlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class MyFirstServlet extends HttpServlet {
    public void service(HttpServletRequest rq,HttpServletResponse rp)
    throws ServletException, IOException
    {
        rp.setContentType("text/html");
        PrintWriter browserOut = rp.getWriter();
        browserOut.println("<HTML>");
        browserOut.println("<HEAD><TITLE> The First Servlet</TITLE></HEAD>");
        browserOut.println("<BODY>");
        browserOut.println("<H3>Hello there... from the First Servlet and from Jeff...</H3>");
        browserOut.println("<H3>This is a very simple Servlet...</H3>");
        browserOut.println("</BODY></HTML>");
    }
}
```

1.4.1 Developing, Compiling and Installing The Servlet

To develop the servlet I worked in the sub-directory called firstServlet. Development and compilation took place in this directory although in a real situation it would be better to have a development directory somewhere else in your system and just copy the completed class files to the various directories in the tomcat hierarchy.

1.4.2 Running The Servlet

Ok, now that I have written, compiled and installed my first servlet, lets now see how I can run the servlet. There are several ways to do this. The first one is impractical in a real situation but will serve as an introduction and demonstration. We assume the Tomcat Web Server is running.

1. Direct from the browser

You can run a servlet directly from your browser. Just point your browser to

http://127.0.0.1:8080/servlet/myServlets.firstServlet.MyFirstServlet

and the servlet should run.

Note : the URL begins with the 'keyword' servlet (singular...), also note how the package is reflected as myServlets.firstServlet.MyFirstServlet

2. As a link from within a web document (html file)

You can run a servlet by invoking it directly from within a web document by linking to it's URL. The web document is loaded into the browser and the following text is made into a link...

http://localHost:8080/servlet/myServlets.firstServlet.MyFirstServlet

When the link is executed the servlet MyFirstServlet is executed at the Web Server. As before all it does it to return a very simple web page to the client browser. It's really the same as 1 above but the URL of the servlet has been transferred into the link in the web page.

1.5 A Second Servlet Example - Maintaining a Shopping Basket

Web servers are what are known as stateless servers. What this means is that in their pure form they keep no memory of what has previously happened to them between requests. For example when a request is processed by a Web server for a page the Web Server has no direct knowledge about whether the page request was made by the same browser that asked for a previous page to be returned or is a completely new request from another browser.

While this was not serious when Web servers were being mainly used for dispensing documentation (their original use) it is a serious problem in e-commerce. One example of this is the shopping cart, or as it is known in the United Kingdom, the shopping trolley. When you visit an e-tailer and purchase goods you interact with a simulation of a shopping cart which keeps details of the goods that you have purchased. At the end of your interaction a Web page, often known as a checkout page, will display the contents of the shopping cart and present you with the monetary total of your purchases. Web servers as originally envisaged are unable to do this, as they have no knowledge of any previous visit: they would not be able to remember the previous purchase.

In the comparatively early days of the Web this was seen to be a problem and a form of programming known as Common Gateway Interface programming was developed which enabled a Web server to have a memory. There are a number of other, more recent technologies, which have been developed to cope with this problem. The first is cookies; these are chunks of data which are stored on the computer running the Web browser and which can be accessed by the browser throughout their interaction with a particular Web site. Such cookies can, for example, store the data associated with a shopping cart.

1.5.1 A Browser Session – Maintaining State Through a Session

Normally, the state of the client/server connection cannot be maintained across different request/response pairs. However, session information is maintainable with servlets as this next example will demonstrate.

A session is a continuous connection from the same browser over a fixed period of time. Servlets allow you to maintain session information by using the HttpSession class. The HttpServletRequest provides the current session with the getSession(boolean) method. If the boolean parameter is true, a new session will be created when a new session is detected. This is, normally, the desired behavior. In the event the parameter is false, then the method returns null if a new session is detected.

Once you have access to an HttpSession, you can maintain a collection of key-value-paired information, for storing any sort of session-specific data. To store session-specific information, you use the putValue(key,value) method. To retrieve the information, you ask the session object with getValue(key). The following example demonstrates this, by continually summing the current basket cost of an on-line shopper.

```
// getAttribute() and setAttribute() have replaced
// getValue() and putValue() but were not available on either
// IE5 or Netscape 4.72 at the time of developing this servlet
package myServlets.shoppingBasket;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class ShoppingBasket extends HttpServlet {
    private Vector anOrderVector;

    public void doGet(HttpServletRequest rq,HttpServletResponse rp)
    throws ServletException, IOException
    {
        rp.setContentType("text/html");
        PrintWriter browserOut = rp.getWriter();
        HttpSession basket = rq.getSession(true);
        String numberReq = new String(rq.getParameter("numberReq"));
        String unitPrice = new String(rq.getParameter("unitPrice"));
        // get the customer id from the incoming document ????
        String costSoFar = this.processBasket("customerID",basket,numberReq,unitPrice);
        String headLine = "<H2>Thank you for your order.</H2>";
        browserOut.println("<HTML>");
        browserOut.println("<BODY>");
        browserOut.println(headLine);
        browserOut.println("<H3>The total cost of your basket so far is : " + costSoFar + "</H3>");

        browserOut.println("<p>");
        browserOut.println("<a href=/MyWebSite/ServletDemos/shoppingBasket.html>Continue shopping or
Go to checkout</a>");
        browserOut.println("<p>");
        browserOut.println("</BODY></HTML>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
    doGet(request, response);
   }

    private String processBasket(String customerID, HttpSession s, String numReq, String unitPrice) {
        // Does the customer already have an order... ?
        //anOrderVector = (Vector)s.getAttribute(customerID);
        anOrderVector = (Vector)s.getValue(customerID);
        if (anOrderVector == null) // must be a new customer...
            anOrderVector = new Vector();
        String cost = new Integer((Integer.parseInt(numReq) * Integer.parseInt(unitPrice))).toString();
        anOrderVector.addElement(cost);
        //s.setAttribute("customerID",anOrderVector);
        s.putValue(customerID,anOrderVector);
        // How much so far ?
        Enumeration e = anOrderVector.elements();
        int nextCost, totalValue = 0;
        while (e.hasMoreElements()) {
            nextCost = new Integer((String)e.nextElement()).intValue();
            totalValue += nextCost;
        }
        return new Integer(totalValue).toString();
    }
}
```

1.6 The Servlet API

The Java Servlet API defines the interface between servlets and servers. This API is packaged as a standard extension to the JDK under javax:

Package javax.servlet
Package javax.servlet.http

The API provides support in four categories:

Servlet life cycle management
Access to servlet context
Utility classes
HTTP-specific support classes

1.7 The Servlet Life Cycle

Servlets run on the web server platform as part of the same process as the web server itself. The web server is responsible for initializing, invoking, and destroying each servlet instance.

A web server communicates with a servlet through a simple interface, javax.servlet.Servlet. This interface consists of three main methods:

init()
service()
destroy()

and two ancillary methods:

getServletConfig()
getServletInfo()

You may notice a similarity between this interface and that of Java applets. This is by design! Servlets are to web servers what applets are to web browsers.An applet runs in a web browser, performing actions it requests through a specific interface. A servlet does the same, running in the web server.

1.7.1 The init() Method

When a servlet is first loaded, its init() method is invoked. This allows the servlet to perform any setup processing such as opening files or establishing connections to their servers. If a servlet has been permanently installed in a server, it loads when the server starts to run. Otherwise, the server activates a servlet when it receives the first client request for the services provided by the servlet.

The init() method is guaranteed to finish before any other calls are made to the servlet--such as a call to the service() method. Note that init() will only be called once; it will not be called again unless the servlet has been unloaded and then reloaded by the server.

The init() method takes one argument, a reference to a ServletConfig object which provides initialization arguments for the servlet. This object has a method getServletContext() that returns a ServletContext object containing information about the servlet's environment (see the discussion on Servlet Initialization Context below).

1.7.2 The service() Method

The service() method is the heart of the servlet. Each request message from a client results in a single call to the servlet's service() method. The service() method reads the request and produces the response message from its two parameters:
A ServletRequest object with data from the client. The data consists of name/value pairs of parameters and an InputStream. Several methods are provided that return the client's parameter information. The InputStream from the client can be obtained via the getInputStream() method. This method returns a ServletInputStream, which can be used to get additional data from the client. If you are interested in processing character-level data instead of byte-level data, you can get a BufferedReader instead with getReader().

A ServletResponse represents the servlet's reply back to the client. When preparing a response, the method setContentType() is called first to set the MIME type of the reply. Next, the method getOutputStream() or getWriter() can be used to obtain a ServletOutputStream or PrintWriter, respectively, to send data back to the client.

As you can see, there are two ways for a client to send information to a servlet. The first is to send parameter values and the second is to send information via the InputStream (or Reader). Parameter values can be embedded into a URL. How this is done is discussed below. How the parameter values are read by the servlet is discussed later.

The service() method's job is conceptually simple--it creates a response for each client request sent to it from the host server. However, it is important to realize that there can be multiple service requests being processed at once. If your service method requires any outside resources, such as files, databases, or some external data, you must ensure that resource access is thread-safe. Making your servlets thread-safe is discussed in a later section of this course.

1.7.3 The destroy() Method

The destroy() method is called to allow your servlet to clean up any resources (such as open files or database connections) before the servlet is unloaded. If you do not require any clean-up operations, this can be an empty method.

The server waits to call the destroy() method until either all service calls are complete, or a certain amount of time has passed. This means that the destroy() method can be called while some longer-running service() methods are still running. It is important that you write your destroy() method to avoid closing any necessary resources until all service() calls have completed.

1.8 HTTP Support

Servlets that use the HTTP protocol are very common. It should not be a surprise that there is specific help for servlet developers who write them. Support for handling the HTTP protocol is provided in the package javax.servlet.http. Before looking at this package, take a look at the HTTP protocol itself.

HTTP stands for the HyperText Transfer Protocol. It defines a protocol used by web browsers and servers to communicate with each other. The protocol defines a set of text-based request messages called HTTP methods. (Note: The HTTP specification calls these HTTP methods; do not confuse this term with Java methods. Think of HTTP methods as messages requesting a certain type of response). The HTTP methods include:

GET
HEAD
POST
PUT
DELETE
TRACE
CONNECT
OPTIONS

1.8.1 The HTTP GET Method

The HTTP GET method requests information from a web server. This information could be a file, output from a device on the server, or output from a program (such as a servlet or CGI script).

An HTTP GET request takes the form:

GET URL <http version>
Host: <target host>

in addition to several other lines of information.

For example, the following HTTP GET message is requesting the home page from the MageLang web site:

```
GET / HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (
 compatible;
 MSIE 4.01;
 Windows NT)
Host: www.magelang.com
Accept: image/gif, image/x-xbitmap,
 image/jpeg, image/pjpeg
```

On most web servers, servlets are accessed via URLs that start with /servlet/. The following HTTP GET method is requesting the servlet MyServlet on the host www.magelang.com:

```
GET /servlet/MyServlet?name=Scott&
  company=MageLang%20Institute HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (
 compatible;
 MSIE 4.01;
 Windows NT)
Host: www.magelang.com
Accept: image/gif, image/x-xbitmap,
 image/jpeg, image/pjpeg
```

The URL in this GET request invokes the servlet called MyServlet and contains two parameters, name and company. Each parameter is a name/value pair following the format name=value. The parameters are specified by following the servlet name with a question mark ('?'), with each parameter separated by an ampersand ('&').

Note the use of %20 in the company's value. A space would signal the end of the URL in the GET request line, so it must be "URL encoded", or replaced with %20 instead. As you will see later, servlet developers do not need to worry about this encoding as it will be automatically decoded by the HttpServletRequest class.

HTTP GET requests have an important limitation. Most web servers limit how much data can be passed as part of the URL name (usually a few hundred bytes.) If more data must be passed between the client and the server, the HTTP POST method should be used instead.

It is important to note that the server's handling of a GET method is expected to be safe and idempotent. This means that a GET method will not cause any side effects and that it can be executed repeatedly.

When a server replies to an HTTP GET request, it sends an HTTP response message back. The header of an HTTP response looks like the following:

```
HTTP/1.1 200 Document follows
Date: Tue, 14 Apr 1997 09:25:19 PST
Server: JWS/1.1
Last-modified: Mon, 17 Jun 1996 21:53:08 GMT
Content-type: text/html
Content-length: 4435

<4435 bytes worth of data -- the document body>
```

The HEAD Method

The HTTP HEAD method is very similar to the HTTP GET method. The request looks exactly the same as the GET request (except the word HEAD is used instead of GET), but the server only returns the header information.

HEAD is often used to check the following:

The last-modified date of a document on the server for caching purposes
The size of a document before downloading (so the browser can present progress information)
The server type, allowing the client to customize requests for that server
The type of the requested document, so the client can be sure it supports it

Note that HEAD, like GET, is expected to be safe and idempotent.

1.8.2 The POST Method

An HTTP POST request allows a client to send data to the server. This can be used for several purposes, such as

Posting information to a newsgroup
Adding entries to a web site's guest book
Passing more information than a GET request allows

Pay special attention to the third bullet above. The HTTP GET request passes all its arguments as part of the URL. Many web servers have a limit to how much data they can accept as part of the URL. The POST method passes all of its parameter data in an input stream, removing this limit.

A typical POST request might be as follows:

```
POST /servlet/MyServlet HTTP/1.1
User-Agent: Mozilla/4.0 (
 compatible;
 MSIE 4.01;
 Windows NT)
Host: www.magelang.com
Accept: image/gif, image/x-xbitmap,
 image/jpeg, image/pjpeg, */
Content-type: application/x-www-form-urlencoded
Content-length: 39

name=Scott&company=MageLang%20Institute
```

Note the blank line--this signals the end of the POST request header and the beginning of the extended information.

Unlike the GET method, POST is not expected to be safe nor idempotent; it can perform modifications to data, and it is not required to be repeatable.

HTTP Support Classes

Now that you have been introduced to the HTTP protocol, consider how the javax.servlet.http package helps you write HTTP servlets. The abstract class javax.servlet.http.HttpServlet provides an implementation of the javax.servlet.Servlet interface and includes a lot of helpful default functionality. The easiest way to write an HTTP servlet is to extend HttpServlet and add your own custom processing.

The class HttpServlet provides an implementation of the service() method that dispatches the HTTP messages to one of several special methods. These methods are:

```
doGet()
doHead()
doDelete()
doOptions()
doPost()
doTrace()
```

and correspond directly with the HTTP protocol methods.

As shown in the following diagram, the service() method interprets each HTTP method and determines if it is an HTTP GET, HTTP POST, HTTP HEAD, or other HTTP protocol method:

The class HttpServlet is actually rather intelligent. Not only does it dispatch HTTP requests, it detects which methods are overridden in a subclass and can report back to a client on the capabilities of the server. (Simply by overriding the doGet() method causes the class to respond to an HTTP OPTIONS method with information that GET, HEAD, TRACE, and OPTIONS are all supported. These capabilities are in fact all supported by the class's code).

In another example of the support HttpServlet provides, if the doGet() method is overridden, there is an automatic response generated for the HTTP HEAD method. (Since the response to an HTTP HEAD method is identical to an HTTP GET method--minus the body of the message--the HttpServlet class can generate an appropriate response to an HTTP HEAD request from the reply sent back from the doGet() method). As you might expect, if you need more precise control, you can always override the doHead() method and provide a custom response.

Using the HTTP Support Classes

When using the HTTP support classes, you generally create a new servlet that extends HttpServlet and overrides either doGet() or doPost(), or possibly both. Other methods can be overridden to get more fine-grained control.

The HTTP processing methods are passed two parameters, an HttpServletRequest object and an HttpServletResponse object. The HttpServletRequest class has several convenience methods to help parse the request, or you can parse it yourself by simply reading the text of the request.

A servlet's doGet() method should

Read request data, such as input parameters
Set response headers (length, type, and encoding)
Write the response data

It is important to note that the handling of a GET method is expected to be safe and idempotent.

Handing is considered safe if it does not have any side effects for which users are held responsible, such as charging them for the access or storing data. Handling is considered idempotent if it can safely be repeated. This allows a client to repeat a GET request without penalty.

Think of it this way: GET should be "looking without touching." If you require processing that has side effects, you should use another HTTP method, such as POST.

A servlet's doPost() method should be overridden when you need to process an HTML form posting or to handle a large amount of data being sent by a client. HTTP POST method handling is discussed in detail later.

HEAD requests are processed by using the doGet() method of an HttpServlet. You could simply implement doGet() and be done with it; any document data that you write to the response output stream will not be returned to the client. A more efficient implementation, however, would check to see if the request was a GET or HEAD request, and if a HEAD request, not write the data to the response output stream.

1.9 Applets and Servlets

1.9.1 A Direct Call to a Servlet

This example shows how a Servlet can be invoked directly from an applet. When the button is pressed the applet builds the URL of the Servlet as follows

```
URL theUrl = this.getDocumentBase();
```

```
String theHost = theUrl.getHost();
String thePort = new Integer(theUrl.getPort()).toString();
String theServletStr = "http://" + theHost + ":" + thePort + "/servlet/MyFirstServlet";
```

and then the Servlet is executed/invoked with

```
this.getAppletContext().showDocument(theServletUrl)
```

MyFirstServlet is the name of the servlet stored in the servlets directory of the host Web Server.

2 JAVA SERVER PAGES

2.1 Introduction

While there are numerous technologies for building web applications that serve dynamic content, the one that has really caught the attention of the development community is JavaServer Pages. And not without ample reason either. JSP not only enjoys cross-platform and cross-Web-server support, but effectively melds the power of server-side Java technology with the WYSIWYG features of static HTML pages.

JSP pages typically comprise of:
- Static HTML/XML components.
- Special JSP tags
- Optionally, snippets of code written in the Java programming language called "scriptlets."

Consequently, you can create and maintain JSP pages by conventional HTML/XML tools.

It is important to note that the JSP specification is a standard extension defined on top of the Servlet API. Thus, it leverages all of your experience with servlets. There are however, significant differences between JSP and servlet technology. Unlike servlets, which is a programmatic technology requiring significant developer expertise, JSP appeals to a much wider audience. It can be used not only by developers, but also by page designers, who can now play a more direct role in the development life cycle.

Another advantage of JSP is the inherent separation of presentation from content facilitated by the technology, due its reliance upon reusable component technologies like the JavaBeans component architecture and Enterprise JavaBeans technology. This course provides you with an in-depth introduction to this versatile technology, and uses the Tomcat JSP 1.1 Reference Implementation from the Apache group for running the example programs.

2.1.1 JSP Advantages

Separation of static from dynamic content: With servlets, the logic for generation of the dynamic content is an intrinsic part of the servlet itself, and is closely tied to the static presentation templates responsible for the user interface. Thus, even minor changes made to the UI typically result in the recompilation of the servlet. This tight coupling of presentation and content results in brittle, inflexible applications. However, with JSP, the logic to generate the dynamic content is kept separate from the static presentation templates by encapsulating it within external JavaBeans components. These are then created and used by the JSP page using special tags and scriptlets. When a page designer makes any changes to the presentation template, the JSP page is automatically recompiled and reloaded into the web server by the JSP engine.

Write Once Run Anywhere: JSP technology brings the "Write Once, Run Anywhere" paradigm to interactive Web pages. JSP pages can be moved easily across platforms, and across web servers, without any changes.

Dynamic content can be served in a variety of formats: There is nothing that mandates the static template data within a JSP page to be of a certain format. Consequently, JSP can service a diverse clientele ranging from conventional browsers using HTML/DHTML, to handheld wireless devices like mobile phones and PDAs using WML, to other B2B applications using XML. Recommended Web access layer for n-tier architecture: Sun's J2EE. Blueprints, which offers guidelines for developing large-scale applications using the enterprise Java APIs, categorically recommends JSP over servlets for serving dynamic content.

Completely leverages the Servlet API: If you are a servlet developer, there is very little that you have to "unlearn" to move over to JSP. In fact, servlet developers are at a distinct advantage because JSP is nothing but a high-level abstraction of servlets. You can do almost anything that can be done with servlets using JSP--but more easily!

2.1.2 Comparing JSP with ASP

Although the features offered by JSP may seem similar to that offered by Microsoft's Active Server Pages (ASP), they are fundamentally different technologies, as shown by the following table:

JavaServer Pages	Active Server Pages
Web Server Support	
Most popular web servers including Apache, Netscape, and Microsoft IIS can be easily enabled with JSP.	Native support only within Microsoft IIS or Personal Web Server. Support for select servers using third-party products.
Platform Support	
Platform independent. Runs on all Java-enabled platforms.	Is fully supported under Windows. Deployment on other platforms is cumbersome due to reliance on the Win32-based component model.
Component Model	
Relies on reusable, cross-platform components like JavaBeans, Enterprise JavaBeans, and custom tag libraries.	Uses the Win32-based COM component model.
Scripting	
Can use the Java programming language or JavaScript.	Supports VBScript and JScript for scripting.
Security	
Works with the Java security model.	Can work with the Windows NT security architecture.
Database Access	
Uses JDBC for data access.	Uses Active Data Objects for data access.
Customizable Tags	
JSP is extensible with custom tag libraries.	Cannot use custom tag libraries and is not extensible.

2.1.3 JSP or Servlets?

It is true that both servlets and JSP pages have many features in common, and can be used for serving up dynamic web content. Naturally, this may cause some confusion as to when to opt for one of the technologies over the other. Luckily, Sun's J2EE Blueprints offers some guidelines towards this.

According to the Blueprints, use servlets strictly as a web server extension technology. This could include the implementation of specialized controller components offering services like authentication, database validation, and so forth. It is interesting to note that what is commonly known as the "JSP engine" itself is a specialized servlet running under the control of the servlet engine. Since JSP only deals with textual data, you will have to continue to use servlets when communicating with Java applets and applications.

Use JSP to develop typical web applications that rely upon dynamic content. JSP should also be used in place of proprietary web server extensions like server-side includes as it offers excellent features or handling repetitive content.

3 JDBC AND DATABASES

The JDBC is a powerful tool for distributed systems. By providing database connectivity to applications and applets, it enables distributed software to tap into distributed databases. This results in information-rich Java applications that can be used to replace and enhance current legacy information systems.

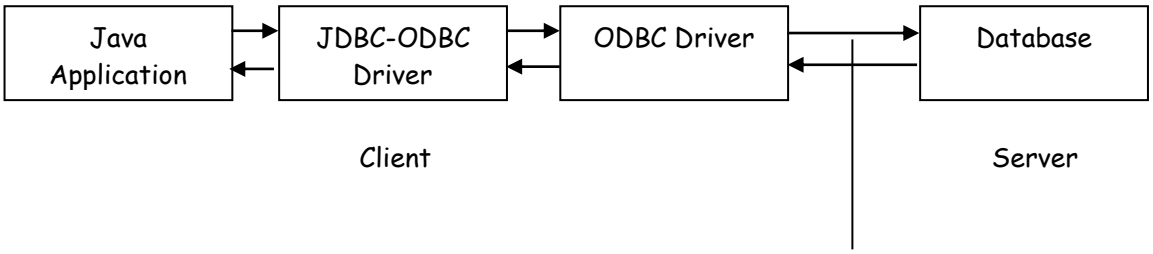
The first part of the chapter will focus on how we make connections to the various databases and will be restricted to a single machine so that we can concentrate on the understanding of the variety of components involved in the connection process. Once we have understood how to connect to the database from our java application the second part will introduce the mechanisms for setting up distributed systems. In particular we will look at the three tier architectures used in many distributed environments. In the third part we will concentrate on the SQL features that allow us to query and update the databases.

3.1 Types of Drivers

A JDBC driver is a class or set of classes that implement the interfaces for a particular database system. In the java.sql package most of its important members such as Connection, Statement and ResultSet are interfaces instead of classes. Writers/developers of drivers have to implement these interfaces for specific database systems.
Different database systems require different JDBC drivers. JDBC drivers have been broken into four different categories. These categories define the methods by which the driver communicates with the database server. Here is a summary of the types and their definitions as related to JDBC drivers:

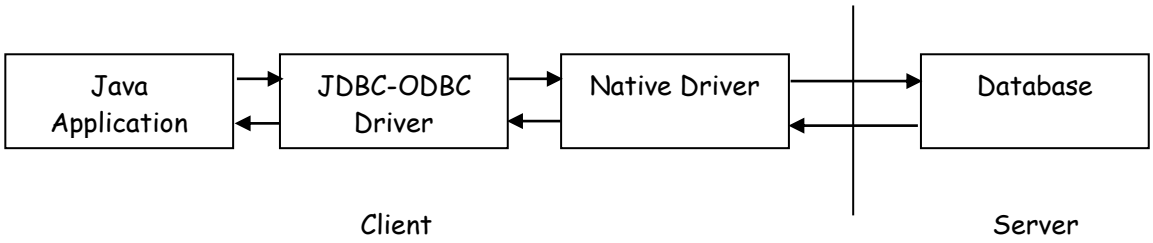
3.1.1 Type 1 Driver

Type 1 drivers are a JDBC-ODBC bridge that utilizes most ODBC drivers to connect to the database. The driver is not written in Java, which means a .dll or .os file will be required on each client. Currently, major browsers do not allow the use of binary libraries, which restricts applets from using this style of driver. Type 1 drivers tend to be less efficient due to the required conversion from JDBC to ODBC and finally to the database API. This driver is appropriate mostly for an application server written in Java that is part of a three-tier architecture. The following figure shows a diagram of a type 1 driver.



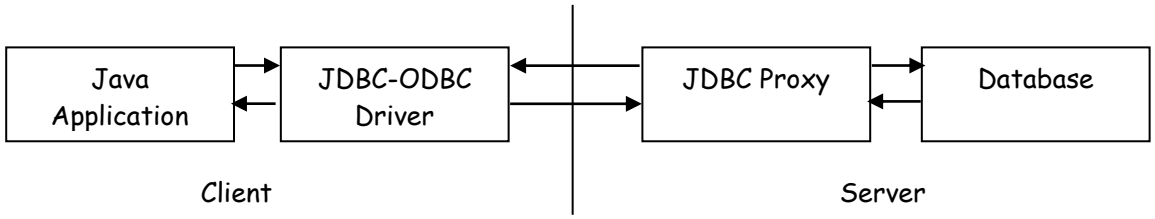
3.1.2 Type 2 Driver

Type 2 drivers are developed for a specific DBMS such as Oracle, Sybase, or Informix, and convert JDBC calls to the proper API calls for the database. The driver is partly Java, but still requires a binary driver on the part of the client. Drivers of this nature tend to be more efficient than type 1 drivers since they need to be converted only from JDBC to the database API.



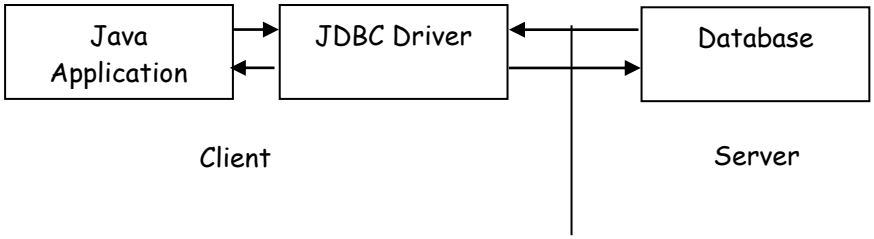
3.1.3 Type 3 Driver

Type 3 drivers are written fully in Java and translate JDBC calls into an independent Internet protocol, which is then translated into the proper DBMS protocol by the server. This method usually requires a server-side proxy or interpreter for the Internet protocol. The proxy may be specifically designed or purchased from a third-party vendor. Security is usually not a priority for vendor solutions and may require extra measures for Internet uses. Type 3 drivers use a proxy to allow applets/applications to connect to a database.



3.1.4 Type 4 Driver

Type 4 drivers are written purely by a DBMS manufacturer and are designed fully in Java. They translate JDBC calls into a DBMS-specific library and are communicated directly to the database's native library. Drivers in this category are generally used to connect a small number of users efficiently or large numbers of users to a large capacity database. Type 4 drivers are currently very rare but are expected to emerge soon.



3.2 Making a Connection to a Database

These are three main steps.

3.2.1 Identify the Database - Create a JDBC URL

The JDBC utilizes the same URL specification system to enable an application to specify a particular database source and a JDBC driver. The JDBC format for a database URL is modeled after the Internet URL format, but must be constructed specifically for the particular driver being used. It was chosen due to the varying needs of the drivers. The URL format is standardized, making it easy to parse and determine if it should be accepted or rejected.

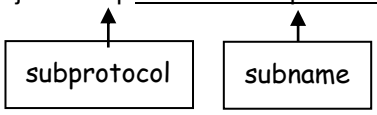
To create a general JDBC URL, the following format is used:

jdbc:<subprotocol>:<subname>

The subprotocol is driver-specific and will likely be the DBMS name and possibly a version number.

For example, when using mSQL, a SQL-compliant database distributed as shareware, a format similar tothis would be used,

jdbc:msql:// hostname: port/database



3.2.2 Name The Driver to be Used and Register it with the DriverManager

The DriverManager class is the main entity on which the JDBC depends. It is responsible for keeping track of the different drivers used in an application and also manages the loading and unloading of the database drivers. Although the functions are generally hidden to the program text, it is the key to how connections are established and maintained.

Drivers are registered with the DriverManager in several ways.

- One way is to use a command line argument to inform the DriverManager that drivers are to be registered. The format is:

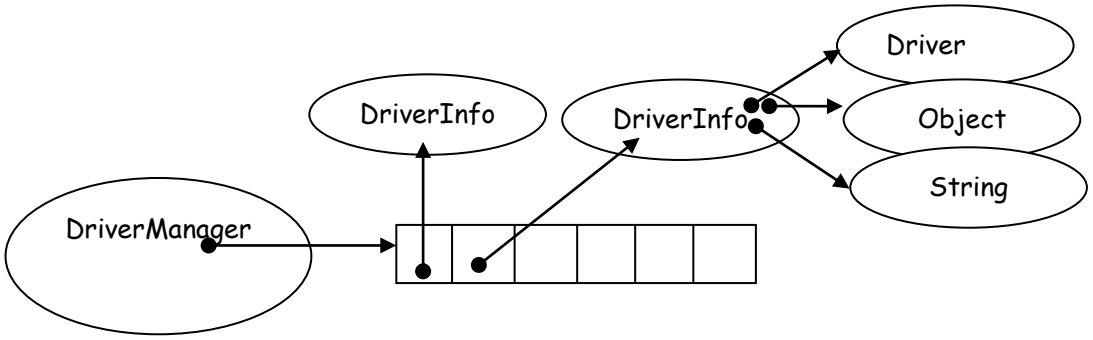
  java -Djdbc.drivers=another.new.Driver:newone.sql.Driver:yet.another.Driver myApplication

- Also, the system property may be set with the key of jdbc.drivers and a value of another.new.Driver:newone.sql.Driver:yet.another.Driver

- Finally, you may instantiate a new driver object using the Class.forName() method. This will explicitly load and register the driver with the DriverManager. This is the approach I use in the examples. *If you use this method make sure that the driver can be found on your system classpath.*

3.2.3 Create a Connection Object

The DriverManager determines which driver to use based on the JDBC URL sent to the getConnection() method. The DriverManager uses the subprotocol of the JDBC URL to select a suitable driver from the list of drivers that have been registered. All known drivers are registered to the DriverManager in a vector. The URL is then compared to each registered driver and chosen accordingly. If the driver is found the getConnection() method returns a connection object primed with all the information to make the connection to the target database.



(See UEJ page 37)

3.3 Some Type 1 Connections : JDBC-ODBC Bridge

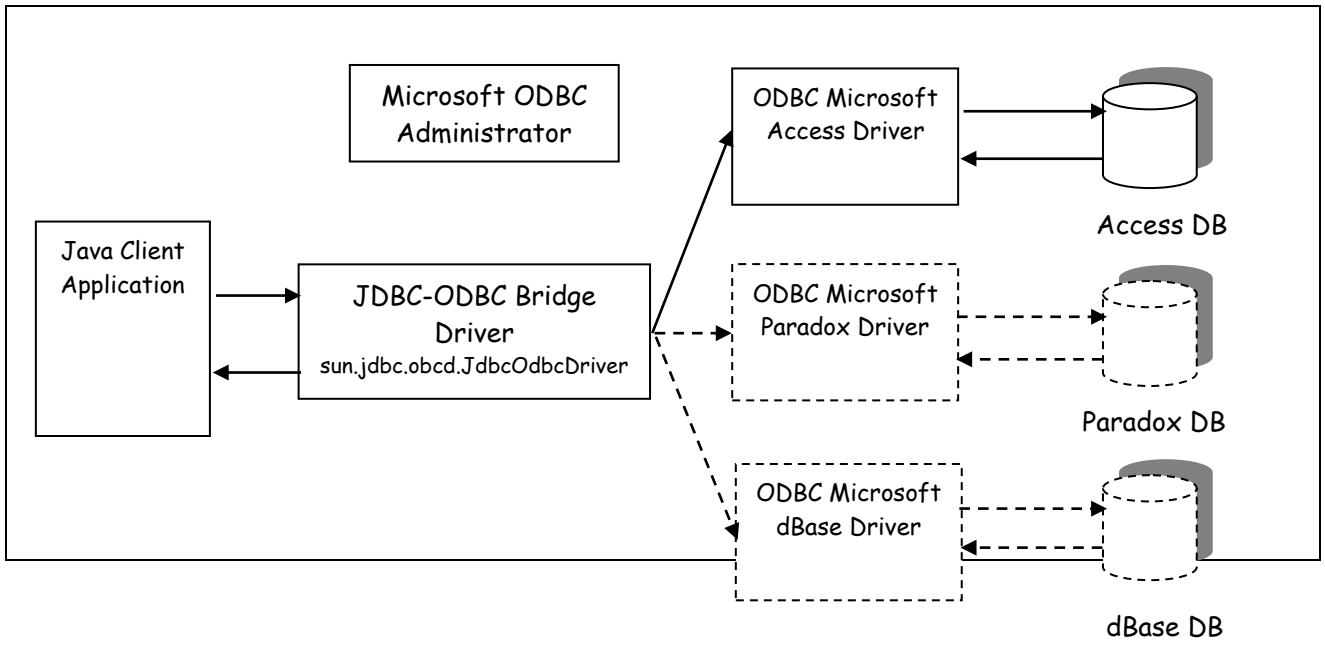
3.3.1 Type 1 - Example 1 (All Software Components on the same machine)

This first example involves a simple Java application making a connection to a Microsoft Access database. All participating software is located on the same machine. The software components involved are

- the Java application
- the JDBC-ODBC bridge driver – sun.jdbc.odbc.JdbcOdbcDriver – a class from the JRE
- the Microsoft ODBC Administrator
- the Microsoft ODBC Access Driver
- the target database – a MS Access database (OrderSys.mdb)

OrderSys.mdb is registered with the Microsoft ODBC Administrator with the DSN of OrderSystem

Note that sun.jdbc.odbc is a package in the Java Runtime Environment. It is not included in the JDK classes. The actual driver is JdbcOdbcDriver.class.



If we were to use another type of database eg a Paradox database then the Microsoft Paradox driver would be used. Each different database type has a corresponding ODBC Microsoft Driver.

```
// Type 1 Connection - Example 1
// The database is located on the on the same machine as the java application.
// The software involved is,
// - the java application
// - the ODBC-JDBC bridge Driver classes in the java API
// - the ODBC Administrator
// - the Microsoft ODBC Access Driver
// - the target database – a MS Access database OrderSys.mdb
// OrderSys.mdb is registered with the Microsoft ODBC Administrator with the
// DSN of OrderSystem
import java.sql.*;
public class Type1DB {
    private String subprotocol;
    private String subname;
    private String theJdbcUrl;
    private String theDriver;
    private Connection theConnection;

    public Type1DB() {
        try {
            // The DSN of OrderSys.mdb as registered with the Microsoft ODBC Administrator
            // is OrderSystem
            subname = "OrderSystem";
            // set up the subprotocol
            subprotocol = "odbc";
            // step 1 - form the JDBC URL - jdbc:<subprotocol>:<subname>
            theJdbcUrl = "jdbc" +":" + subprotocol + ":" + subname;
            // step 2 - name the driver (a class) to be used.. sun.jdbc.odbc is a package from the JRE
            theDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
            // load it and register it with the DriverManager
            Class.forName(theDriver);
            // step 3 - create the connection object...
            theConnection = DriverManager.getConnection(theJdbcUrl);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void readDatabase() {
        // Identify a table in the OrderSys.mdb database
        String theTableName = "tbl_customer";
        try {
            // create a sqlStatement object from theConnection...
            Statement sqlStatement = theConnection.createStatement();
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theTableName);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            // and the column names...
        System.out.println("Number of columns = " + nCols);
```

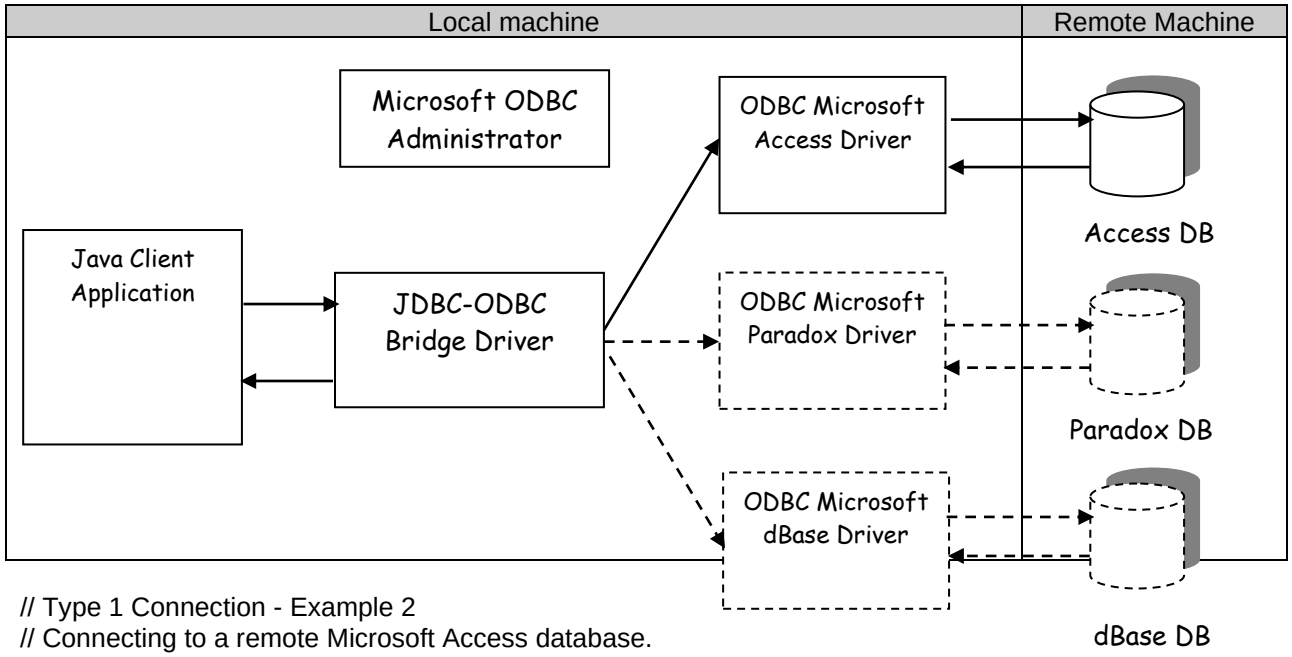
```
            for (int col = 1; col <= nCols; ++col)
                System.out.println(rmd.getColumnName(col));
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        Type1DB t = new Type1DB();
        t.readDatabase();
    }
}
```

3.3.2 Type 1 - Example 2 (Connecting to a remote database)

An alternative architecture is shown below. This time 2 networked machines are shown. The distribution of the various components is evident. The 'remote' database is registered with the MS ODBC Administrator on the local machine. This registration is carried out from the local machine using the ODBC Administrator.
The software involved is,

- On the local machine...
  - the java application
  - the ODBC-JDBC bridge Driver classes in the java API
  - the ODBC Administrator
  - the Microsoft ODBC Access Driver
- On the remote machine…
  - the target database - BookSale.mdb - is located on the remote machine but is registered with the ODBC Administrator on the same machine as the java application.

The database is registered with the local ODBC Administrator with the DSN of RemoteDatabase.




```
// Type 1 Connection - Example 2
// Connecting to a remote Microsoft Access database.
// The software involved is,
// On the local machine...
// - the java application
// - the ODBC-JDBC bridge Driver classes in the java API
// - the ODBC Administrator
// - the Microsoft Access Driver
// On the remote machine
```

```
// - the target database – the MS Access database OrderSys.mdb
// The database - BookSale.mdb - is located on the remote machine but
// is registered with the ODBC Administrator on the same machine as
// the java application.
// The database is registered with the DSN of RemoteDatabase.
import java.sql.*;
public class Type1DB {
    private String subprotocol;
    private String subname;
    private String theJdbcUrl;
    private String theDriver;
    private Connection theConnection;
    public Type1DB() {
        try {
            // The DSN of BookSale.mdb as registered with the local Microsoft ODBC Administrator
            // is RemoteDatabase - BookSale.mdb is located on the remote machine
            subname = "RemoteDatabase";
            // set up the subprotocol
            subprotocol = "odbc";
            // step 1 - form the JDBC URL - jdbc:<subprotocol>:<subname>
            theJdbcUrl = "jdbc" +":" + subprotocol + ":" + subname;
            // step 2 - name the driver (a class) to be used.. sun.jdbc.odbc is a package from the JRE
            theDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
            // load it and register it with the DriverManager
            Class.forName(theDriver);
            // step 3 - create the connection object...
            theConnection = DriverManager.getConnection(theJdbcUrl);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void readDatabase() {
        // Identify a table in the BookSale.mdb database
        String theTableName = "Titles" ;
        try {
            // create a sqlStatement object from theConnection...
            Statement sqlStatement = theConnection.createStatement();
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theTableName);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            // and what are the column names...
        System.out.println("Number of columns = " + nCols);
            for (int col = 1; col <= nCols; ++col)
                System.out.println(rmd.getColumnName(col));
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void main(String args[]) {
        Type1DB t = new Type1DB();
        t.readDatabase();
    }
}
```

3.3.3 Type 1 - Example 3 (A dBase database)

This example does not add anything really new except the target database is now a dBase file. For this, a different Microsoft ODBC driver has to be used. Instead of using the Microsoft Access driver we now use the Microsoft dBase driver. However there is one subtle difference to be aware of and that is a Microsoft Access database consists of a number of tables and therefore the java application has to identify not only the database but also the particular table in the database. The following code from example 1 shows this,

```
// This is the name registered with the Microsoft ODBC Administrator
subname = "OrderSystem";
String theTableName = "tbl_customer";
```

tbl_customer is the name of a table within the database OrderSys.mdb. OrderSystem is the DSN as registered with the ODBC Administrator.

With the dBase database used in example 3 the database consists only of one table so that the stored name of the database Supplier.dbf is now used in place of the table.

```
// This is the DSN registered with the Microsoft ODBC Administrator
subname = "dBaseFiles";
theDatabaseFile = "Supplier.DBF";
```

Special note : When the DSN is being created with the Microsoft ODBC Administrator all that is necessary is to register the directory in which the database is located. So the directory d:\dbFiles\dbf is selected as the directory from the dialog box of the ODBC Administrator and it is this directory that is effectively identified as “dBaseFiles”. The registration process then consists of identifying the directory where the dBase files is stored and the java application combines this DSN with the name of the dBase file (client.dbf) to fully determine the database.

By using the directory as the DSN we can identify any of the database files within that directory from within our application. We simply identfy the required database as follows,

```
theDSN = "dBaseFiles";
// Identify the required database...
//String theDatabaseFile = "CLIENTS.DBF";
//String theDatabaseFile = "EMPLOYEES.DBF";
//String theDatabaseFile = "HOLDINGS.DBF";
//String theDatabaseFile = "MASTER.DBF";
String theDatabaseFile = "SUPPLIER.DBF";
```

We can choose any one of the 6 databases stored in the MDB directory.

```
// Type 1 Connection - Example 3
// Very similar to Example 1 except this time we use a dBase
// database. The difference is that we now have to refer to
// the database file by name rather than refer to a table as we
// did when using an Access database.
// The database is located on the on the same machine as the java application.
// The software involved is,
// - the java application
// - the ODBC-JDBC bridge Driver classes in the java API
// - the ODBC Administrator
// - the Microsoft dBase Driver
// - the dBase database Supplier.dbf
// The directory containing the dbase files is registered with the local
// Microsoft ODBC Administrator with the DSN of dBaseFiles
import java.sql.*;
public class Type1DB {
    private String subprotocol;
```

```
    private String subname;
    private String theJdbcUrl;
    private String theDriver;
    private Connection theConnection;
    public Type1DB() {
        try {
            // The DSN of the dBase files as registered with the Microsoft ODBC Administrator
            // is dBaseFiles
            subname = "dBaseFiles";
            // set up the subprotocol
            subprotocol = "odbc";
            // step 1 - form the JDBC URL - jdbc:<subprotocol>:<subname>
            theJdbcUrl = "jdbc" +":" + subprotocol + ":" + subname;
            // step 2 - name the driver (a class) to be used.. sun.jdbc.odbc is a package from the JRE
            theDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
            // loadit and register it with the DriverManager
            Class.forName(theDriver);
            // step 3 - create the connection object...
            theConnection = DriverManager.getConnection(theJdbcUrl);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void readDatabase() {
        // Identify the required database...there are a number to choose from
        //String theDatabaseFile = "CLIENTS.DBF";
        //String theDatabaseFile = "EMPLOYEES.DBF";
        //String theDatabaseFile = "HOLDINGS.DBF";
        //String theDatabaseFile = "MASTER.DBF";
        String theDatabaseFile = "SUPPLIER.DBF";
        try {
            // create a sqlStatement object from theConnection...
            Statement sqlStatement = theConnection.createStatement();
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theDatabaseFile);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            // and what are the column names...
        System.out.println("Number of columns = " + nCols);
            for (int col = 1; col <= nCols; ++col)
                System.out.println(rmd.getColumnName(col));
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        Type1DB t = new Type1DB();
        t.readDatabase();
```

```
    }
}
```

3.3.4 Serious Restriction of Type 1 Architecture

A serious restriction with type 1 architecture, is that the drivers must be on the same machine as the client software. This restriction has major drawbacks when we come to consider distributed software systems across the Internet. If the client software is a Java applet then this means that when the applet is downloaded from some web server the drivers also have to be downloaded. Both drivers, the JDBC-ODBC bridge and the ODCB, must be installed on the client's machine.

3.4 Type 3 Connections : Third Party Drivers and Servers

3.4.1 What is The IDS Server?

The IDS Server is a Type 3 Internet database access server. It enables both HTML and Java developers to create and deploy database interactive Web pages, Java applets and Java programs. IDS Server supports access to all ODBC compliant database systems, as well as native Oracle, Sybase and Informix databases through their respective client application programming interface (API). These client APIs are Oracle Call Interface, Sybase CT-Lib and Informix Client-SDK. Platform support of IDS Server include Windows 98/95/NT/2000 on Intel x86, Windows NT on Alpha, Solaris on Sun Sparc, and Linux on Intel x86.

IDS Server comes with IDS JDBC Driver. IDS JDBC Driver is a high performance and yet extremely compact and efficient Type-3 JDBC driver. JDBC compliant Java applications can access any databases connected to IDS Server. This JDBC driver is written in 100% platform independent pure-Java and is suitable for both Java applets and Java programs. It works with all Java enabled browsers, all versions of Java Development Kit (JDK™), Microsoft SDK for Java, Visual Cafe, JBuilder, VisualAge for Java and other popular Java development tools.

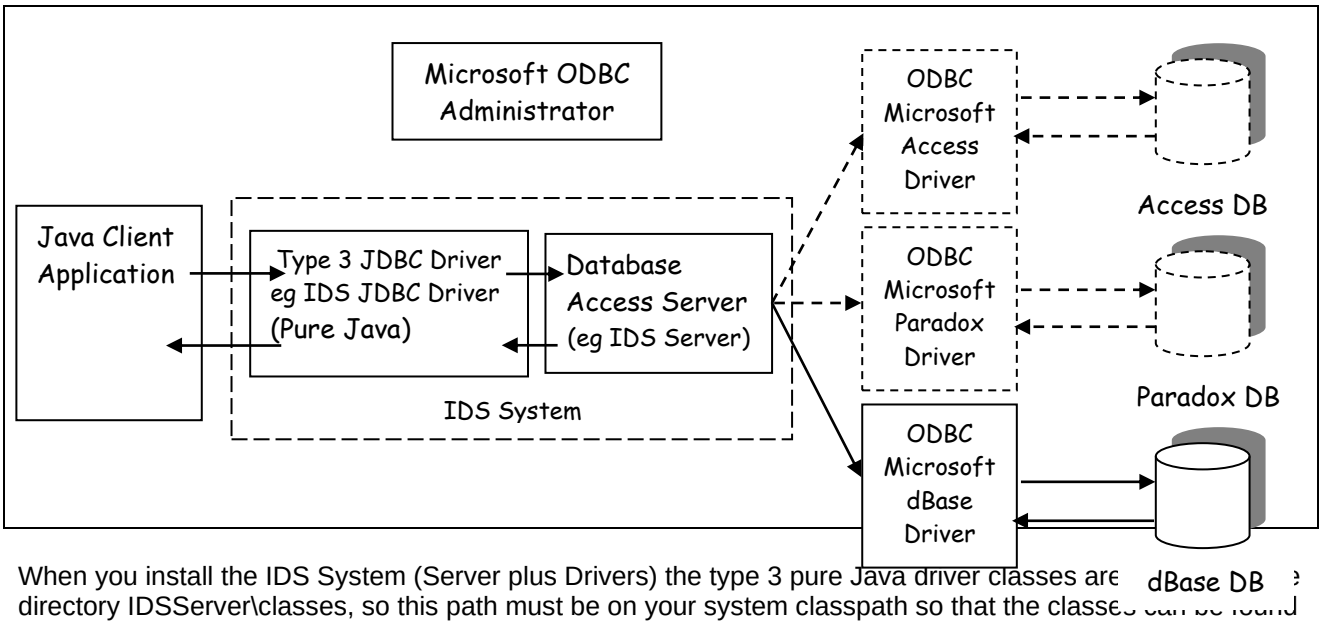
IDS Server and IDS JDBC Driver also provides a state of the art JDBC security feature called Secure JDBC. Secure JDBC uses Secure Socket Layer (SSL) protocol to protect the data exchanged between IDS JDBC Driver and IDS Server against eavesdroppers and malicious attacks. Cryptography technologies used in SSL includes public-key cryptography, 128-bit Blowfish cipher, Triple-DES cipher, RSA and discrete logarithm digital signature, etc.

Aside from being a high performance database access server, an IDS Server can also serve as a simple Web server. Therefore, an IDS Server alone is capable of hosting and deploying an entire database enabled Internet application system.

3.4.2 Type 3 - Example 1 (All Software Components on the same machine)

Architecture is restricted to one machine...

- the java application
- the IDS Driver classes (in the d:\IDSServer\classes\ids directory)
- the IDS Access Server
- the Microsoft ODBC Administrator
- the Microsoft dBase Driver
- the target database... Clients.dbf, Employee.dbf, etc, in d:\dbFiles\DBF and given the collective DSN of localDBFiles.



When you install the IDS System (Server plus Drivers) the type 3 pure Java driver classes are [illegible] directory IDSServer\classes, so this path must be on your system classpath so that the classes can be found for loading by the JVM.

```
// Type 3 Connection - Example 1
// Architecture is restricted to one machine...
// - the java application
// - the IDS Driver classes (in the d:\IDSServer\classes\ids folder)
// - the IDS Access Server
// - the Microsoft ODBC Administrator
// - the Microsoft dBase Driver
// - the database... Clients.dbf, Employee.dbf, etc, in d:\dbFiles\DBF
//                                  with a DNS of localDBFiles
import java.sql.*;
public class Type3DB {
    private String subprotocol;
    private String subname;
    private String theJdbcUrl;
    private String theDriver;
    private Connection theConnection;
    public Type3DB() {
        try {
            // The DSN of the dBase files as registered with the Microsoft ODBC Administrator
            // is localDBFiles
            subname = "//127.0.0.1:12/conn?dbtype=odbc&dsn=localDBFiles";
            // set up the subprotocol
            subprotocol = "ids";
            // step 1 - form the JDBC URL - jdbc:<subprotocol>:<subname>
            theJdbcUrl = "jdbc" +":" + subprotocol + ":" + subname;
            // step 2 - name the driver (a class) to be used.. ids.sql.IDSDriver
            // can be found under the IDSServer directory...
            theDriver = "ids.sql.IDSDriver";
            // load it and register it with the DriverManager
            Class.forName(theDriver);
            // step 3 - create the connection object...
            theConnection = DriverManager.getConnection(theJdbcUrl);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
```

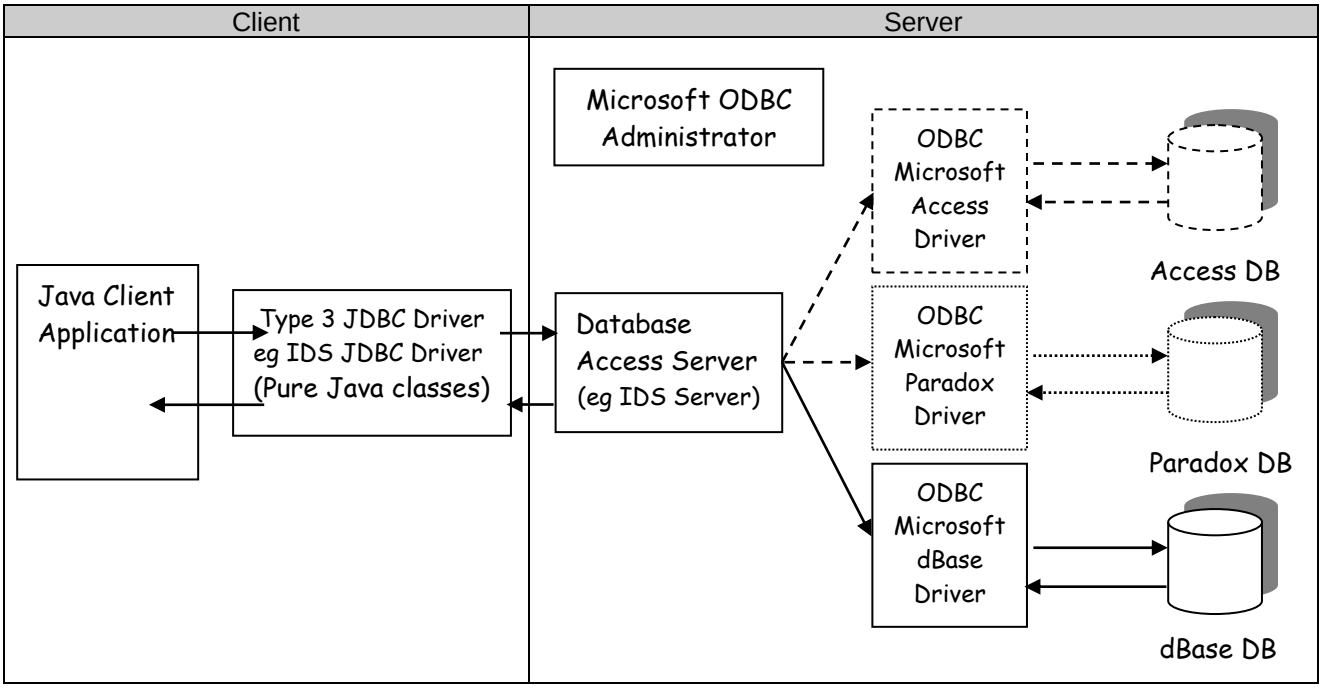
```
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void readDatabase() {
        // For DBF databases all we have to do is to register the folder
        // containing the files with the ODBC Administrator. Hence as far as
        // the adminisator is concerned the DSN is the folder containing
        // the files. The specific database required is then identified
        // by its file name as shown below...
        // String theDatabaseFile = "CLIENTS.DBF";
        // String theDatabaseFile = "EMPLOYEE.DBF";
        // String theDatabaseFile = "MASTER.DBF";
        String theDatabaseFile = "SUPPLIER.DBF";
        //String theDatabaseFile = "HOLDINGS.DBF";
        try {
            // create a sqlStatement object from theConnection...
            Statement sqlStatement = theConnection.createStatement();
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theDatabaseFile);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            // and what are the column names...
        System.out.println("Number of columns = " + nCols);
            for (int col = 1; col <= nCols; ++col)
                System.out.println(rmd.getColumnName(col));
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        Type3DB t = new Type3DB();
        t.readDatabase();
    }
}
```

3.4.3 Type 3 - Example 2 (A Distributed System – or nearly…)

This next example illustrates a more realistic remote connection. AcmeDatabase is a remote dBase database. The database is located on the server machine and is registered with the ODBC Administrator on the server machine with the DSN of AcmeDatabase.

- On the client machine...
  - the java application
  - the IDS Driver classes
- On the server machine...
  - the IDS Server
  - the Microsoft ODBC Administrator
  - the Microsoft dBase Driver
  - the database (AcmeDatabase)

This architecture is much closer to the type of distributed systems we shall study in the next part of this chapter. Their architectures are those used in Internet Distributed Systems. Notice now that the client machine only requires the java application and the drivers to be present.

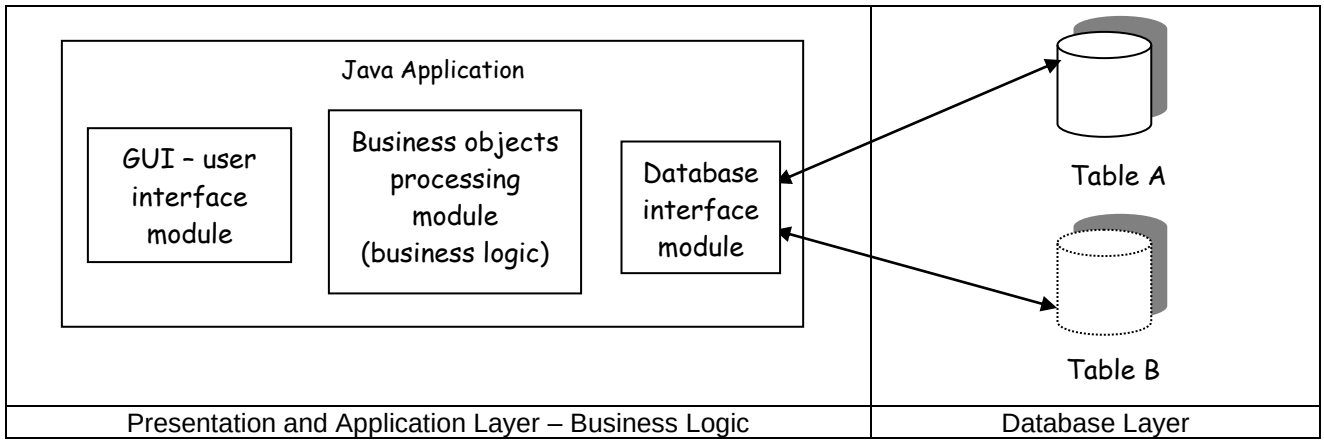



```
// Type 3 Connection - Example 2
// Architecture is now distributed
// On the client
// - the java application
// - the IDS Driver classes
// On the server
// - the IDS Access Server
// - the Microsoft ODBC Administrator
// - the Microsoft dBase Driver
// - the database... AcmeDatabase - a group of dBase files

import java.sql.*;
public class Type3DB {
    private String subprotocol;
    private String subname;
    private String theJdbcUrl;
    private String theDriver;
    private Connection theConnection;

    public Type3DB() {
        try {
            // The DSN of the dBase files as registered with the Microsoft ODBC Administrator
            // is AcmeDatabase
            subname = "//192.168.0.1:12/conn?dbtype=odbc&dsn=AcmeDatabase";
            // set up the subprotocol
            subprotocol = "ids";
            // step 1 - form the JDBC URL - jdbc:<subprotocol>:<subname>
            theJdbcUrl = "jdbc" +":" + subprotocol + ":" + subname;
            // step 2 - name the driver (a class) to be used.. this is shipped with the app
            theDriver = "ids.sql.IDSDriver";
            // load it and register it with the DriverManager
            Class.forName(theDriver);
            // step 3 - create the connection object...
```

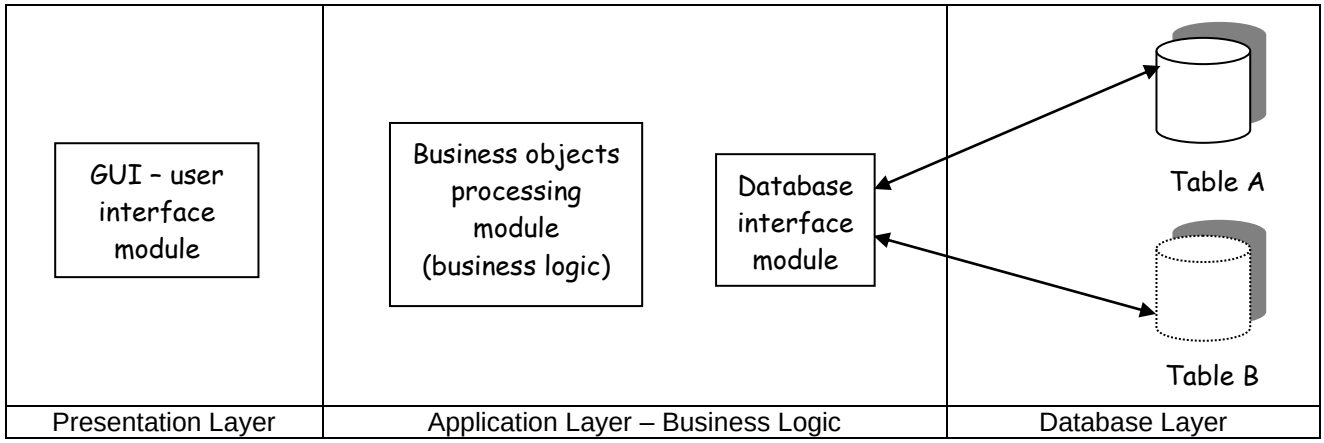
```
            theConnection = DriverManager.getConnection(theJdbcUrl);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private void readDatabase() {
    // For DBF databases all we have to do is to register the folder
    // containing the files with the ODBC Administrator. Hence as far as
    // the administrator is concerned the DSN is the folder containing
    // the files. The specific database required is then identified
    // by its file name as shown below...
        // String theDatabaseFile = "Clients.dbf";
        String theDatabaseFile = "Holdings.dbf";
        //String theDatabaseFile = "Master.dbf";
        try {
            // create a sqlStatement object from theConnection...
            Statement sqlStatement = theConnection.createStatement();
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theDatabaseFile);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            // and what are the column names...
        System.out.println("Number of columns = " + nCols);
            for (int col = 1; col <= nCols; ++col)
                System.out.println(rmd.getColumnName(col));
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        Type3DB t = new Type3DB();
        t.readDatabase();
    }
}
```

3.5 Three Tier Architecture

In a standard two-tier application model, the application contains the business logic, the user interface, and the interface to the database.

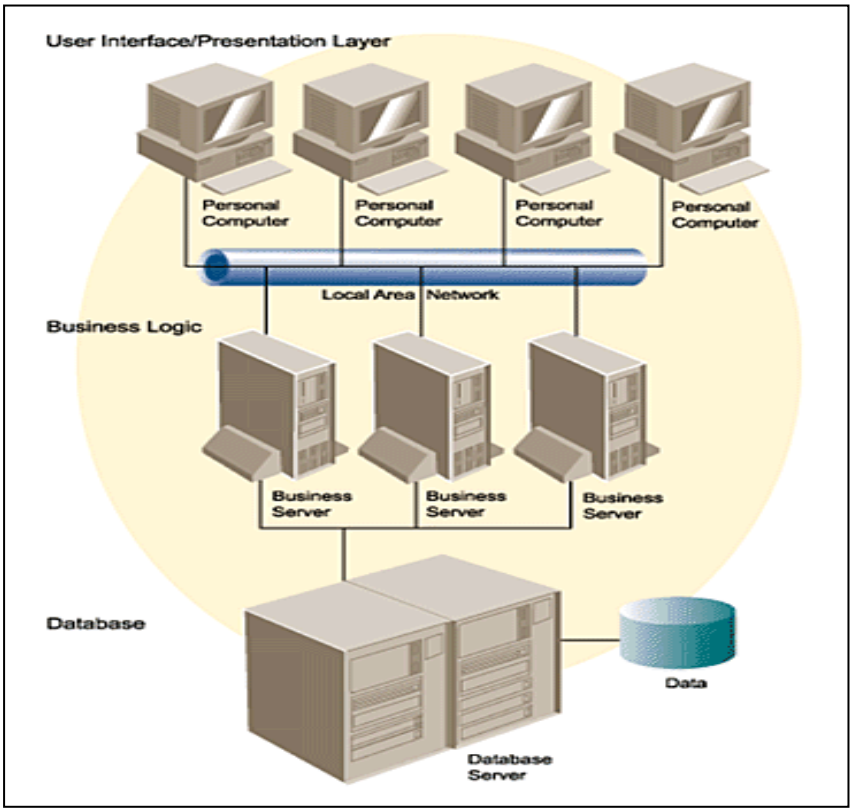


As businesses start to change the way they function, the application must be altered to reflect these changes. To prevent large applications from being rewritten, the three-tier application model was developed. In this model, the user interface becomes its own module, and the business logic becomes the middle piece for the application. In addition, multiple applications will use the same business logic module. The business logic becomes responsible for retrieving the information from the database and organizing it for all of the end-user applications. When a business must change its strategy or functions, only the business logic module requires updating. All applications that depend on the same module escape the need for rewriting, thus saving time and expense in code revisions.



The basic idea of the three-tier approach is that the presentation layer deals with the presentation of information to the user; the business layer incorporates the business logic and the data layer deals with the actual physical storage. Like all layered systems, communication only occurs between adjacent layers. The middle tier is the part of the system where the main business logic is executed (not where it is stored). It is normally separate from the user interface code.

Business Objects are usually software components, which encapsulate all the processing logic, and data access for a specific business entity. Basically they are programs, or groups of programs but structured around a data entity. Typical Business Objects will be such entities like Customer, Order or StockItem. They usually corresponding to Java objects.



3.6 Stored Procedures

Here's an example stored-procedure for Microsoft's SQL Server:

```
CREATE PROC pub_info2
  @pubname varchar(40) = 'Algodata Infosystems'
AS
SELECT au_lname, au_fname, pub_name
FROM authors a INNER JOIN titleauthor ta
ON a.au_id = ta.au_id
   JOIN titles t ON ta.title_id = t.title_id
   JOIN publishers p ON t.pub_id = p.pub_id
WHERE @pubname = p.pub_name
```

This would then be called using just this:

```
EXECUTE pub_info2
```

Stored procedures can be far more complex than this, possibly hundreds of lines long. Note that this stored procedure is written in SQL - don't think stored procedures are instead of SQL queries. One reason stored-procedures are more efficient is because the SQL has already been compiled and optimized, so when the procedure is executed there is no need to compile and optimize the query as that stage has already been done. Another reason the course text mentions is that we only send a small amount of data across the network e.g. just 'EXECUTE pub_info2' rather than sending possibly hundreds of lines of SQL.

The example procedure I've given is written in SQL, but other languages such as Java can be used to write stored-procedures, although that depends on whether a specific DBMS supports that.

For more Microsoft examples see
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/createdb/cm_8_des_07_786r.asp

3.7 Database MetaData

All relational databases designed for use in a commercial environment have the functionality to record and report **metadata** about the contents of a given database instance. Metadata is "data about data" it describes how the data in a database is collected and how it is formatted. Relational databases provide metadata for a large number of features of a database including full details about a database itself (e.g. the product name and version of the database), and about the tables in a database and the columns they contain (e.g. names of tables and columns and the types and sizes of data held within columns).

If a system administrator wishes to determine information about the tables in a database via a program this is done in a manner similar to that used by programs to obtain actual data from a database. Although most modern programming languages provide facilities to access relational databases (good point...) the following examples all use Java.

Provided the administrator knows the locations on the network (and a suitable database driver) of the files holding the tables, or the URL of the appropriate database server, then metadata about these tables may be obtained. The following code extracts will load a driver and connect to a pre-registered ODBC database called M897test using the JDBC:ODBC bridge

```
// Load the required packages
import java.sql.*;
// Load the database driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// Establish a connection to the database
Connection conn =
  DriverManager.getConnection("jdbc:odbc:M879test, "Neil", "");
```

And this code extract connects to a database server using a JDBC driver

```
import java.sql.*;
import org.gjt.mm.*;
Class.forName("org.gjt.mm.mysql.Driver");
Connection conn =
   DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/M879test");
```

Having connected to the database in either of the ways shown above the program can now obtain the metadata for the database's tables. To do this Java uses a class in the java.sql.* package DatabaseMetaData. The code below shows the creation of an instance of this class for the database:

```
DatabaseMetaData dbmd = conn.getMetaData();
```

The program can now use the many methods of this class to retrieve information about the tables in the database. The two most useful methods for this purpose are the getTables and getColumns methods. Evaluating the expressions:

```
ResultSet rsTab =
  dbmd.getTables(null, null, null, new String[] {"TABLE"});
```

And:

```
ResultSet rsCol = dbmd.getColumns(null, "%", "%", null);
```

Produces two result sets (ResultSet is another class in the java.sql.* package that contains the rows returned by a database query) the first containing details of all the tables in the database and the second set containing details of all the columns in all the tables in the database. Iterating over each element in these results sets will return the details for each column or table in the database. In the example below the name of each table in the database is displayed along with some additional (self explanatory) details:

```
    while (rsTab.next()) {
        System.out.println ("Table: " + rsTab.getString("TABLE_NAME") +
                            ", Type: " + rsTab.getString("TABLE_TYPE") +
                            ", Remarks: " + rsTab.getString("REMARKS"));
    }
```

Which produces output like:

Table: MyTable, Type: TABLE, Remarks: A simple test table

And to detail all the columns in a database (the result set is ordered by table name so all columns will be in table order) evaluating:

```
while (rsCol.next()) {
        System.out.println ("Table: " + rsCol.getString("TABLE_NAME") +
", Column: " +  rsCol.getString(COLUMN_NAME) +
                            ", Data type: " + rsCol.getString("TYPE_NAME")+
                            ", Size: " + rsCol.getString("COLUMN_SIZE") +
          ", Allow null: " +
                        rsCol.getString("IS_NULLABLE") +
", Remarks: " + rsCol.getString("REMARKS"));
     }
```

Producing output like:

```
Table: MyTable, Column: FirstField, Data type: char, Size: 10, Allow null:
YES, Remarks: null

Table: MyTable, Column: LastField, Data type: char, Size: 20, Allow null:
YES, Remarks: null
```

It is also possible to nest these code examples to list columns for each table individually (note the TABLE_NAME column is the third element in the rsTab result set):

```
    while (rsTab.next()) {
        // Get the columns for each named table
        rsCol = dbmd.getColumns(null, "%", rsTab.getString(3), "%");
        while (rsCol.next()) {
            …
            …
        }
    }
```

Another useful method is getIndexInfo e.g.:

```
// Get index details for the current table name in the rsTab result set
ResultSet rsIndex = dbmd.getIndexInfo(null, "%", rsTab.getString(3),
                    false, true);
     while (rsIndex.next()) {
```

```
            System.out.println ("Indexed Column: " +
                        rsIndex.getString("COLUMN_NAME") +
                        ", Ascending/Descending: " +
                        rsIndex.getString("TYPE_NAME"));
      }
```

Finally, it is also possible to get metadata about results sets themselves utilising the ResultSetMetaData class and this can also be used to find out about database tables as the following demonstrates:

```
      // Create an SQL statment
Statement query = conn.createStatement();

     // Set the query to get details for the current table name in the
     // rsTab result set
      String queryString = "SELECT * FROM " + rsTab.getString(3);
      ResultSet rs = query.executeQuery(queryString);

     // Get metadata about the result set
      ResultSetMetaData rsmd = rsCat.getMetaData();

     // Output table details
System.out.println ("TABLE: " + rsmd.getTableName(1) +
  " has " + rsmd.getColumnCount() + "columns");

     // Output the details of each column in the table
      for (int i = 1; i <= rsmd.getColumnCount(); i++){
            System.out.println ("Column: " + rsmd.getColumnName(i) +
                        ", Type: " + rsmd.getColumnTypeName(i));
}
√
```

This code produces output like:

```
  Table: MyTable has 2 columns,
  Column: FirstField, Type: STRING  Column: LastField, Type: STRING
```

3.8 mSQL Introduction

So far we have relied on the Microsoft ODBC drivers to access our databases. The next example removes this dependence and introduces another database system, mSQL.

Mini SQL, or mSQL, is a lightweight database engine designed to provide fast access to stored data with low memory requirements. As its name implies, mSQL offers a subset of SQL as its query interface. Although it only supports a subset of SQL (no views, sub-queries, etc.), everything it supports is in accordance with the ANSI SQL specification. The mSQL package includes the database engine, a terminal "monitor" program, a database administration program, a schema viewer, and a C language API. The API and the database engine have been designed to work in a client/server environment over a TCP/IP network.

mSQL will run on a variety of platforms. The most common is Unix. Others include Linux and MSWindows.

3.8.1 The mSQL Server (Database Engine)

The mSQL daemon, **msqld.exe**, is a standalone application that listens for connections on a well known TCP socket. It is a single process engine that will accept multiple connections and serialise the queries received. It utilises memory mapped I/O and cache techniques to offer rapid access to the data stored in a database. It also utilises a stack based machanism that ensures that INSERT operations are performed at the same speed regardless of the size of the table being accessed. Preliminary testing performed by a regular user of mSQL has shown that for simple queries, the performance of mSQL is comparable to or better than other freely available database packages. For example, on a set of sample queries including simple INSERTs, UPDATEs and SELECTs, mSQL performs roughly 4 times faster than University Ingres and over 20 times faster than Postgres on an Intel 486 class machine running Linux.

3.8.2 Mini SQL Specification

The mSQL language offers a significant subset of the features provided by ANSI SQL. It allows a program or user to store, manipulate and retrieve data in table structures. It does not support relational capabilities such as table joins, views or nested queries. Although it does not support all the relational operations defined in the ANSI specification, it does provide the capability of "joins" between multiple tables.
Although the definitions and examples on the next pages depict mSQL key words in upper case, no such restriction is placed on the actual queries.

3.8.3 The mSQL Driver Classes

You can obtain the mSQL driver classes from HHHHHHHH

In addition to the mSQL driver classes you will also need.

- Any Java VM supporting JDBC 1.2 or later (JDK 1.1 or later)
- The InfoBus JDK 1.1 Collections API
  (http://java.sun.com/beans/infobus/#DOWNLOAD_COLLECTIONS)

I installed the downloaded collection classes in my jdk directory as c:\jdk1.3\1.1collections

3.8.4 How to create a database with mSQL

Creating a database with mSQL is very simple. Let's assume that you want to create a new database named Contracts.
mSQL maintains its databases in sub directorys of the directory msqldb. With mSQL a database is effectively a directory. In fact a sub directory to ..\mSQL\msqldb. So create Contracts as a sub-directory to the directory msqldb.

You are now ready to add new tables to the Contracts database. To create a database table with mSQL all you have to do is use the msql command.

Let's assume that you want to create a new database table named Customerin the database named Contracts. At the command line, type: msql Contracts

In the *msql* utility, create the table in mSQL with the usual SQL syntax. Then type \g (mSQL's *go* command), and then type \q when you're ready to quit. The sequence of commands in the *msql* utility looks like this:

```
create table Customer (cnum integer,
first_name char(20), last name char(20) )
\g \q
```

At this point you'll be returned to the command line.

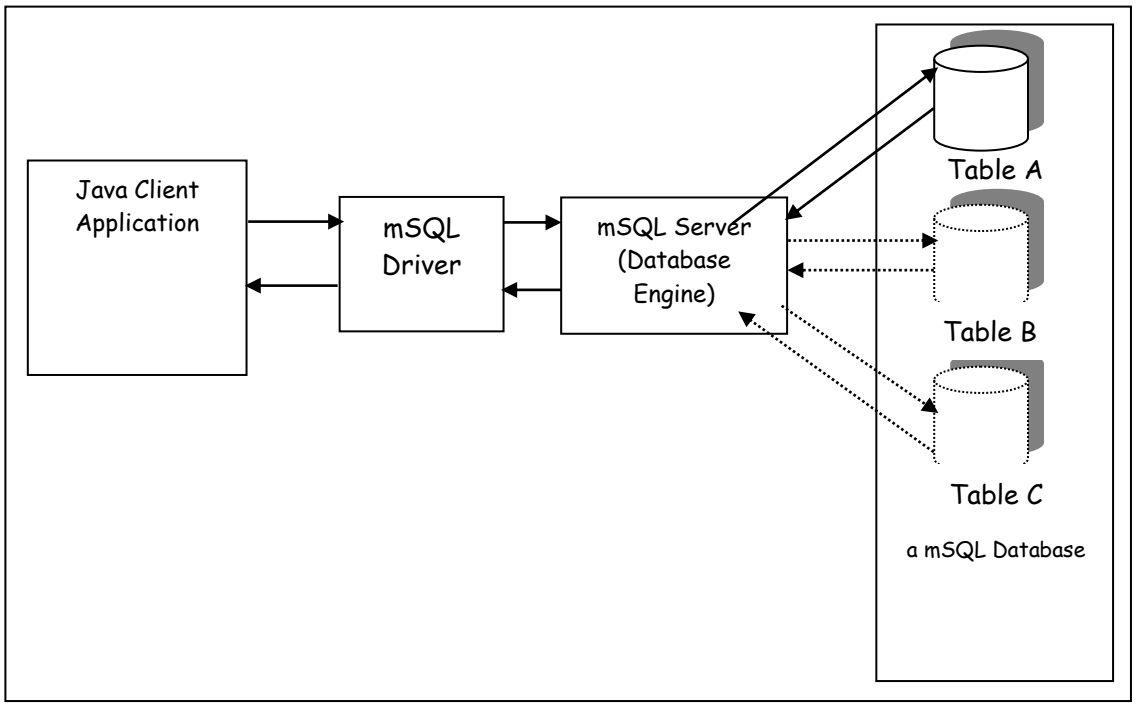
This command created a database table named *Customer* in the database named *contact_mgr. The Customer table contains three fields named cnum, first_name,* and *last_name.*
Other locations on the *Developer's Daily* web site
We invite you to visit other free educational resources on our web site: .Our Developer's Daily home page .*DevDirecto*~ -our index of resources for software developers .Our Java Education Center .Our Perl Education Center
.Our Unix Education Center
~ http.IIWWW.deV-=h/msql/dhTahle-create.html --27/09/01
Mini SQL (mSQL) -How to display the structure of a Mini SQL (mSQL) database table Page 1 of 1
How to display the structure ora Mini SQL (mSQL) database table by DeveloRer's Daily
Displaying the structure of a *Mini SQL* (mSQL) database table is very simple. All you have to do is use the relshow command.
Let's assume that you have a database named *contact_mgr* that contains a database table named *Salespeople.* To show the structure of the Salespeople table, just type this command at the Unix command line:

```
relshow contact_mgr Salespeople
```

Our sample *Salespeople* table contains just a few columns, and the output from relshow looks like this:

Database = contact_mgr Table = Salespeople
+ + + + + + I Field I Type I Length I Not Null I Unique Index I + + + + + + I snum I int I 4 I y I N/A I I first_name I char I 20 I N I N/A I I last_name I char I 20 I N I N/A I I sp_index I index I N/A I N/A I y I + + + + + +
At this point you'll be returned to the Unix command line.
Other locations on the *Developer's Daily* web site
We invite you to visit other free educational resources on our web site: .Our DeveloRer's Daily home Rage .*DevDirectoO!* -our index of resources for software develoRers .Our Java Education Center .Our Perl Education Center
.Our Unix Education Center
L *http:1 Iwww* .devdaily .com/dblmsq1/dbTable-display-structure.html & *27/09101*

3.8.5 Example 4 – A mSQL Database (Non Microsoft Database)



java application → Type 3 mSQL driver → mSQL Server → msQL database

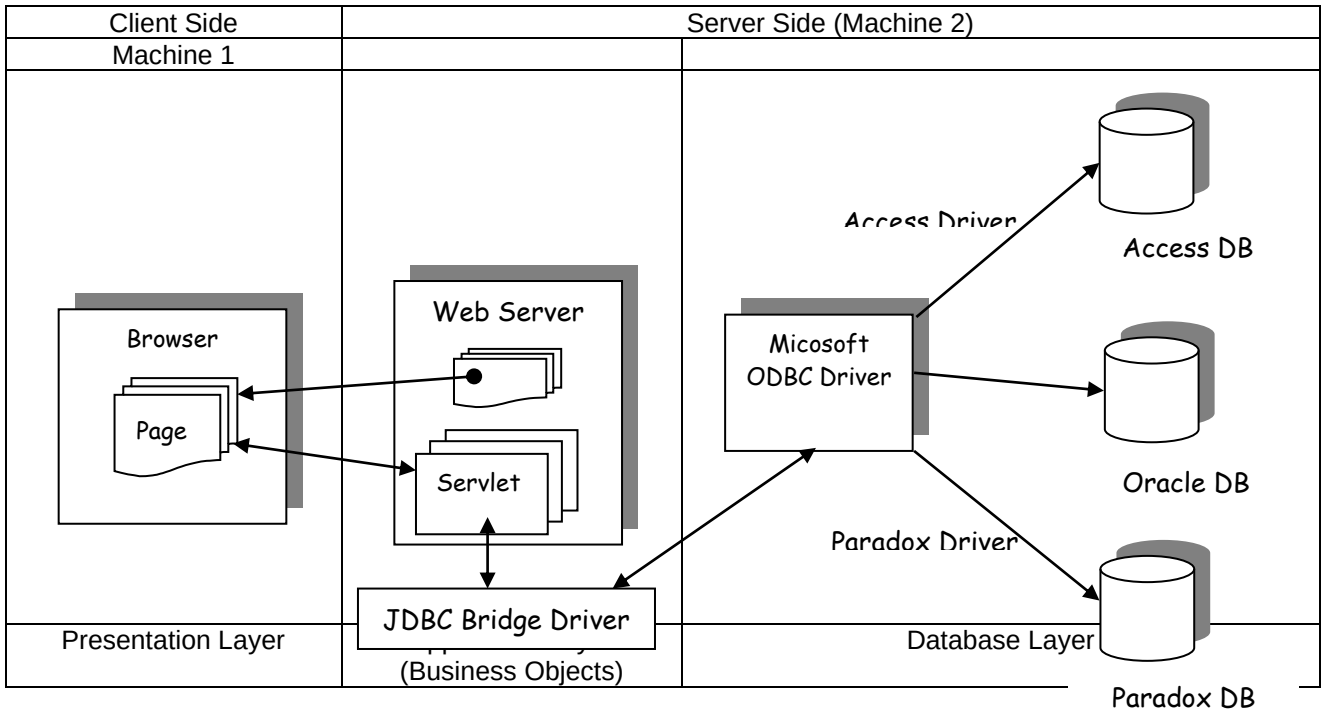
3.9 Middleware

Middleware is used to describe software (not hardware) that is used to provide connectivity between two (or more) other bits of software. A typical use is to connect client side software to old legacy systems and also into the world of objects via CORBA .
Typically a user at the client could make a request for some information and that single request could involve the middleware getting data from an old system (e.g. a big IMB mainframe connected via SNA) and combining it with the results of calling a business object written in C++ or Java and sending it all back.

However middleware is a pretty generic term and so some would say that the MQM software on its own is actually a bit of middleware.

3.10 JDBC and Servlets In A Distributed System

3.10.1 A Type 1 connection



- the web page that invokes the database servlet
- the database servlet
- a web server
- the JDBC bridge database driver
- the appropriate ODBC driver
- the database, identified with a DSN (probably using the Data Source Administrator from the win9x/NT control panel)
- in order to be recognised by the ODBC driver the database needs to be given a DSN
- the database client is the servlet
- the web server needs to be running
- the database servlet needs to be located in the directory
  webserverdirectory\sevlets\myDBExamples

- the web page (the html document) needs to be located in the public_html directory of the web server
- the form on the web page needs to POST to
  <form method="post" action="/servlet/myDBExamples.Type1DemoDB.class" name="">
- When using the JDBC you must allocate a DSN (Data Source Name) to your database. You can use the win9x/NT Data Source Administrator (from the control panel) to do this. Set up your DSN as System DSN. These data sources are local to a computer, rather than dedicated to a user. The system, or any user having privileges, can use a data source set up with a system DSN.

3.10.2 The Web Page

```
<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF">
<p>TYPE 1 JDBC CONNECTION</p>
<p> </p>
<p>Some descriptive text...</p>
```

```
<form method="post" action="/servlet/myDBExamples.Type1DemoDB.class" name="">
<input type="submit" name="runExample" value="Run Example">
</form>
</body>
</html>
```

3.10.3 The Database Servlet

```
// Type 1 Connection
// No ipaddress/port involved because a necessary restriction of a type 1
// connection is that the ODBC driver, the database and the client
// (in this case a servlet) all have to be on the same machine.
//
package myDBExamples;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;
import java.util.*;
public class Type1DemoDB extends HttpServlet {
    private String theDatabaseDriver;
    private String theDatabaseName;
    private String theDatabase;
    private Connection theConnection;
    private String theTableName;
    private Statement sqlStatement;
    private String numCols;
    private Vector theColNames;
    private String theErrorMessage = "No Errors reported...";
    public void init(ServletConfig config)
        throws ServletException {
        // init() is only executed once... so set up everthing with
        // the target Database...
        try {
            theDatabaseDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
            theDatabaseName = "OrderSysDB";
            theDatabase = "jdbc:odbc:" + theDatabaseName;
            theTableName = "tbl_article" ;
            Class.forName(theDatabaseDriver);
            theConnection = DriverManager.getConnection(theDatabase);
            // Create a sqlStatement object from theConnection...
            sqlStatement = theConnection.createStatement();
            this.readDatabase();
        }
        catch (ClassNotFoundException e) {
            System.out.println("From ClassNotFoundException : " + e.getMessage());
        }
        catch (SQLException e) {
            theErrorMessage = " Error while writing html back to the browser : " + e.getMessage();
        }
    }

    private void readDatabase() {
        try {
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theTableName);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
```

```
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            numCols = new Integer(nCols).toString();
            // and what are the column names...
            theColNames = new Vector();
            for (int col = 1; col <= nCols; ++col) {
                theColNames.addElement(rmd.getColumnName(col));
            }
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            System.out.println("From SQLException : " + e.getMessage());
        }
    }

    public void doPost(HttpServletRequest rq,HttpServletResponse rp)
    throws ServletException, IOException
    {
        try {
            rp.setContentType("text/html");
            PrintWriter browserOut = rp.getWriter();
            // Now return a completed page to the client...
            browserOut.println("<HTML>");
            browserOut.println("<BODY>");
            browserOut.println("<H4>Type 1 Connection to : " + theDatabaseName + "</H4>");
            browserOut.println("Table name = " + theTableName);
            browserOut.println("<br>Number of columns = " + numCols);
            Enumeration e = theColNames.elements();
            while (e.hasMoreElements()) {
                browserOut.println("<br>" + (String)e.nextElement());
            }
            browserOut.println("<br>" + theErrorMessage);
            browserOut.println("</BODY></HTML>");
            browserOut.close();
        }
        catch (IOException e) {
            theErrorMessage = " Error while writing back results : " + e.getMessage();
        }
    }
}
```

3.10.4 A Type 3 Connection

Uses a JavaSoft type 3 connection. The client is the servlet. The servlet uses a type 3 JDBC driver to talk to the database access server (DBAS). The DBAS translates requests from the client (the servlet) into vendor specific database protocol. The significant difference, bewteen a type 1 connection and a type 3 connection is that the restriction of having the database on the same machine as the client (ie the servlet) is now removed. This is much better. Hence the path to the database now has to include an ipaddress.

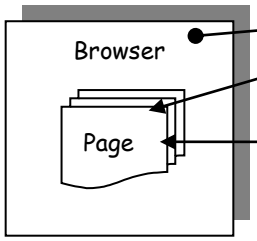
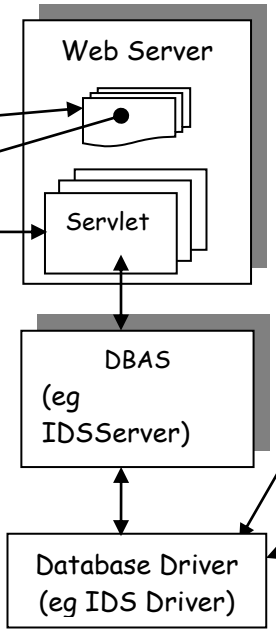
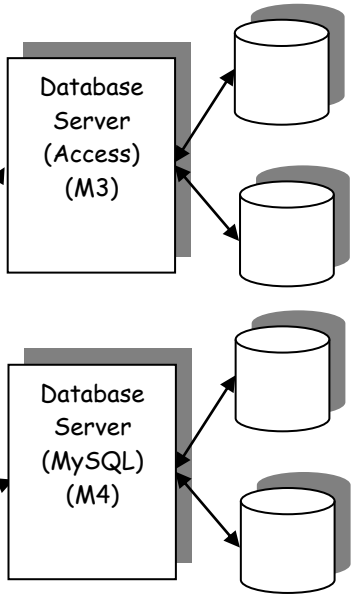
- you must have the IDS server running
- the IDS driver is located under the directory IDSServer\classes, so this path must be on the system classpath.

much of the code is the same as the previous example except for the following changes

- the web document should point to the type 3 class,
  <form method="post" action="/servlet/myDBExamples.Type3DemoDB.class" name="">

- the servlet code referencing the drivers and location of the database needs changing to

```
theDatabaseDriver = "ids.sql.IDSDriver";
ipAddress = "//127.0.0.1";
port = ":12";  // the IDSDriver port...
theDatabaseName = "OrderSysDB";
theDatabaseURL = "jdbc:ids:" + ipAddress + port + "/conn?dbtype=odbc&dsn=" + theDatabaseName;
```

<table>
<tr><td>Client Side</td><td colspan="2">Server Side</td></tr>
<tr><td>Machine 1</td><td>Machine 2</td><td>Machines 3 and 4</td></tr>
<tr><td> 
</td><td> 
</td><td> 
</td></tr>
<tr><td>• web page(s) design
• form construction
• linking to web server and servlets</td><td>• web server management
• servlet development
• linking to DBAS
• the Servlets interpret clients requests, send SQL queries to the database and send the results back to the clients</td><td>• IDS management
• Database Server management
• creation of the databases
• interpret the SQL queries and return the results to the web server servlets
• the DAS (Database Access Server) interprets SQL statements from the web server servlets and translates them into a form that the Database server understands</td></tr>
<tr><td>Presentation Layer</td><td>Application Layer
(Business Objects)</td><td>Database Layer</td></tr>
</table>

3.11 Applet Connection to a Database

Another connection architecture is for an applet to connect straight to the database, effectively cutting out the need for the cgi/servlet level. The business objects involved in the connection are transferred to the applet.

The IDS Driver classes directory (IDS/classes/ids) must be placed in the same directory as the applet class on the Web Server. This allows the applet to refer back to the Web Server from the client machine, and find the driver classes when it needs to access to them.

Checklist to get the system running…

- create the directory for the ‘web site’
  c:\JavaWebServer2.0\public_html\jeff
  (the directory jeff can be replaced with your choice…)

- in this directory place the ‘site’ home page html that embeds the database applet
  index.html

- create the directory that will contain all the applet bits
  c:\JavaWebServer2.0\public_html\jeff\AppletDBDemo

- copy the
  c:\IDSServer\classes\ids directory (and all its subdirectories)
  to the AppletDBDemo directory
  ie create the ids directory as follows
  c:\JavaWebServer2.0\public_html\jeff\AppletDBDemo\ids

- in the directory …\jeff\AppletDBDemo place all the applet classes
  AppletDBDemo.class
  AppletDBDemo$ButtonListener.class
  AppletDBDemo.html

- start the IDSServer with
  idss –con

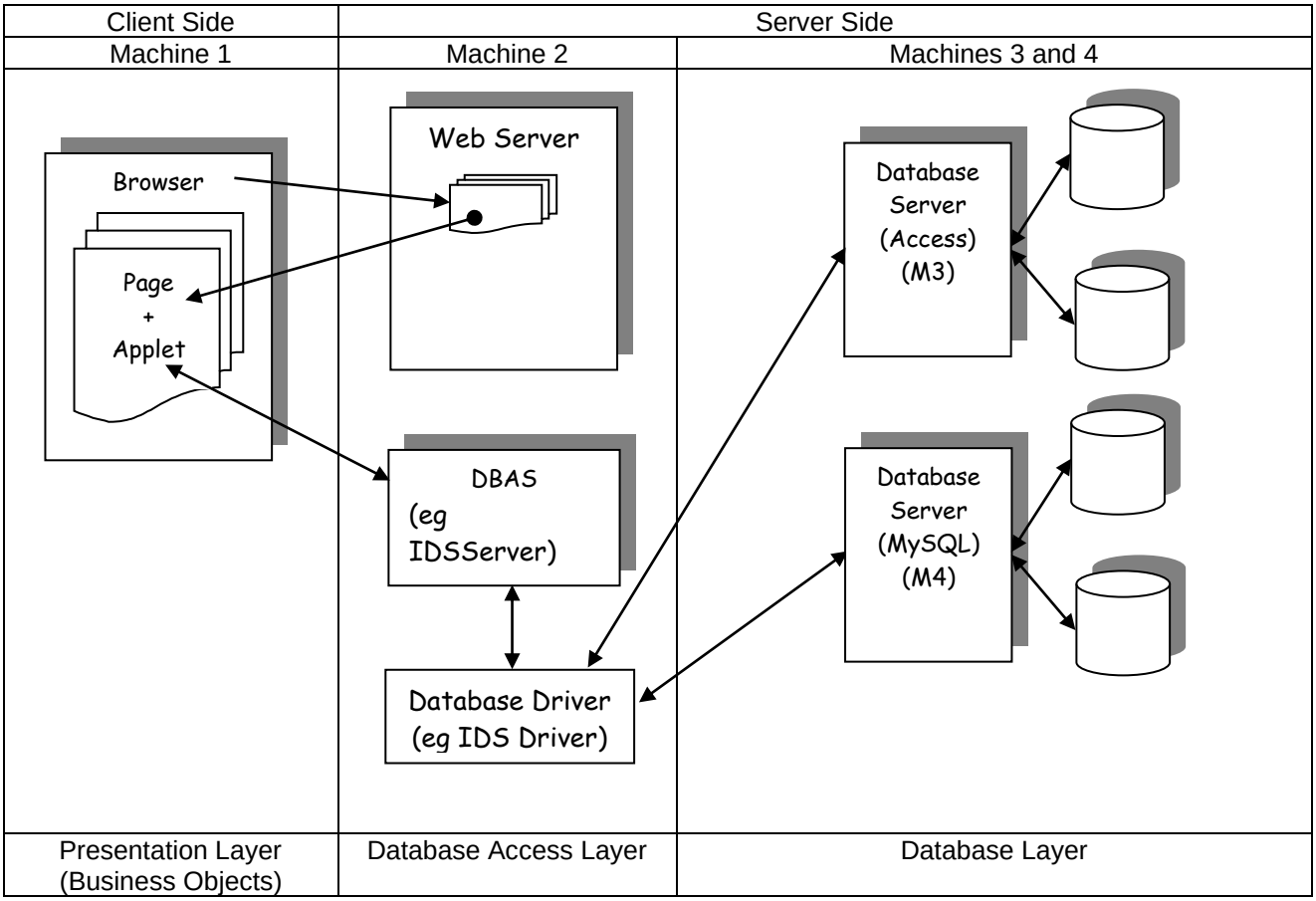
- start the JavaWebServer from the command line
  cd\JavaWebServer2.0\bin
  httpd
  then open a browser and run the server admin program by pointing the browser to
  http://localHost:9090/

- and finally point your browser to the site home web page with
  http://localHost:8080/jeff/

3.12 Registering a Database with the Microsoft ODBC

MyComputer/Control Panel/ODBC Data Sources (32 bit)
System DSN (Data Source Name)
Add
Select Microsoft Access Driver (*.mdb)
Finish
Enter a Data Source Name (eg MyDataBase)
Use the Select button to locate the database file
OK and exit

The selected database is now registered as an ODBC Data Source with the Microsoft O/S.



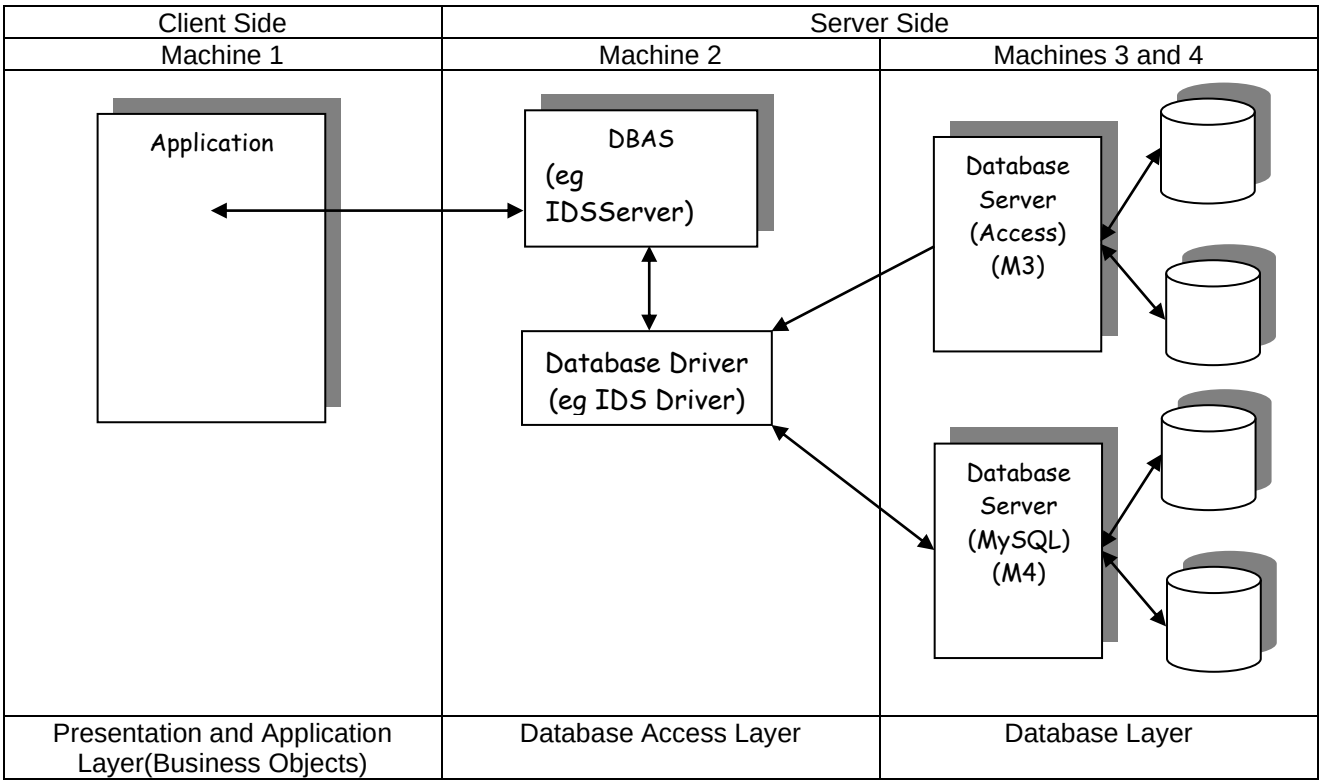

```
// Applet to Database Example
// Uses a Type 3 connection (... the IDS Driver)
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.sql.*;
import java.util.*;

public class AppletDBDemo extends Applet {
    // The GUI bits...
    private TextField theDatabaseBox;
    private Button submitBtn;
    private TextField numOfColsBox;
    private java.awt.List colsList;
    private Button closeDBaseBtn;

    // The DB connection bits
    private String ipAddress, port;
    private String theDatabaseDriver;
    private String theDatabaseName;
    private String theDatabaseURL;
    private Connection theConnection;
    private String theTableName;
    private Statement sqlStatement;
    private String numCols;
    private Vector theColNames;
```

```
    private String theErrorMessage = "No Errors reported...";

    public void init() {
        // init() is only executed once... so set up the GUI and
        // set up everthing with the target Database... unless you want
        // to be able to re-open the database, in which case you'll have
        // to shift the code to some openDatabase() method which you can recall
        try {
            this.setUpGui();
            theDatabaseDriver = "ids.sql.IDSDriver";
            ipAddress = "//127.0.0.1";
            port = ":12";  // the IDSDriver port...
            theDatabaseName = theDatabaseBox.getText();
            theDatabaseURL = "jdbc:ids:" + ipAddress + port + "/conn?dbtype=odbc&dsn=" +
            theDatabaseName;
            theTableName = "tbl_article";
            Class.forName(theDatabaseDriver);
            theConnection = DriverManager.getConnection(theDatabaseURL);
        }
        catch (ClassNotFoundException e) {
            System.out.println("From ClassNotFoundException : " + e.getMessage());
        }
        catch (SQLException e) {
            theErrorMessage = " Error while writing html back to the browser : " + e.getMessage();
        }
    }

    public void setUpGui() {
        this.add(new Label("Enter database name : "));
        theDatabaseBox = new TextField(30);
        this.add(theDatabaseBox);
        theDatabaseBox.setText("OrderSysDB");
        submitBtn = new Button("Submit");
        this.add(submitBtn);
        numOfColsBox = new TextField(10);
        this.add(new Label("                                    "));
        this.add(new Label("Number of Columns : "));
        this.add(numOfColsBox);
        colsList = new java.awt.List(10);
        this.add(colsList);
        closeDBaseBtn = new Button("Close Database");
        this.add(closeDBaseBtn);
        // Finally set the listeners...
        submitBtn.addActionListener(new ButtonListener());
        closeDBaseBtn.addActionListener(new ButtonListener());
    }

    private void readDatabase() {
        try {
            // Create a sqlStatement object from theConnection...
            sqlStatement = theConnection.createStatement();
            // Form the SOL query...
            ResultSet results = sqlStatement.executeQuery("SELECT * FROM " + theTableName);
            // Create a meta object for the results object
            ResultSetMetaData rmd = results.getMetaData();
            // Now the meta object can tell us a few things about the database...
            // like, how many columns are in the table...
            int nCols = rmd.getColumnCount();
            numCols = new Integer(nCols).toString();
```

```
            // and what are the column names...
            theColNames = new Vector();
            for (int col = 1; col <= nCols; ++col) {
                theColNames.addElement(rmd.getColumnName(col));
            }
        }
        catch (SQLException e) {
            System.out.println("From SQLException : " + e.getMessage());
        }
    }

    public void postResults() {
        String nextName;
        numOfColsBox.setText(numCols);
        colsList.removeAll();
        Enumeration e = theColNames.elements();
        while (e.hasMoreElements()) {
            nextName = (String)e.nextElement();
            colsList.add(nextName);
        }
    }

    private void closeDatabase() {
        try {
            colsList.removeAll();
            theColNames.removeAllElements();
            numCols = "";
            numOfColsBox.setText(numCols);
            sqlStatement.close();
            theConnection.close();
        }
        catch (SQLException e) {
            System.out.println("From SQLException : " + e.getMessage());
        }
    }

    private class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            String buttonString = evt.getActionCommand();
            if (buttonString.equals("Submit")) {
                readDatabase();
                postResults();
            }
            if (buttonString.equals("Close Database")) {
                closeDatabase();
            }
        }
    }
}
```

3.13 Application Connection to a Database

Another variation on the connection architecture is for an application to connect straight to the database. This architecture uses the Internet but not the web. The business objects involved in the connection are transferred to the application.



The previous code for the applet can easily be adapted to become an application.

- remove import java.applet.*;
- change the class header to
  ```
  public class ApplicationDBDemo extends Frame {
  ```
- change the applet init() to the application constructor
  ```
  public ApplicationDBDemo() {
     super();
  ```

- in the method public void setUpGui() add the line
  ```
  this.addWindowListener(new MyWindowCloser());
  ```
- add the window closer class
  ```
  private class MyWindowCloser extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
          System.exit(0);
      }
  }
  ```
- and finally add a main() method
  ```
  public static void main(String args[]) {
      ApplicationDBDemo demo = new ApplicationDBDemo();
      demo.setSize(250,300);
      demo.show();
  }
  ```

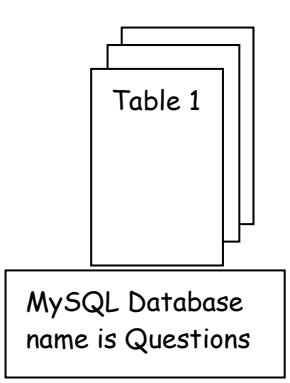
3.14 Atomic Transactions

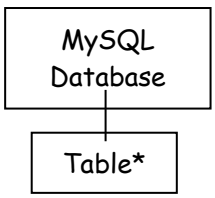
By default, a newly created database Connection object is in "auto commit" mode. That means that each update to the database is treated as a separate transaction and is automatically committed to the database. Sometimes, however, you want to be able to group several updates into a single "atomic" transaction, with the prop-erty that either all of the updates complete successfully or no updates occur at all. With a database system (and JDBC driver) that supports it, you can take the Con -
nection out of "auto commit" mode and explicitly call commit( ) to commit a batch of transactions or call ro 11 ba c k ( ) to abort a batch of transactions, undoing the ones that have already been done.
Example 16-5 displays a class that uses atomic transactions to ensure database consistency. The example is an implementation of the RemoteBank interface that was developed in Chapter 15, Remote Method Invocation. As you may recall, the RemoteBankServer class developed in Chapter 15 did not provide any form of persistent storage for its bank account data. Example 16-5 addresses this problem by implementing a RemoteBank that uses a database to store all user account information.

After the RemoteDBBankServer creates its Connection object, it calls setAuto- Commit( ) with the argument fa 1 se to turn off "auto commit" mode. Then, for example, the openAccount ( ) method groups three transactions into a single, atomic transaction: adding the account to the database, creating a table for the account history, and adding an initial entry into the history. If all three of these transactions are successful (i.e., they don't throw anyexceptions), openAccount( ) calls commit( ) to commit the transactions to the database. However, if anyone of the transactions throws an exception, the catch clause takes care of calling roll-back( ) to rollback any of the transactions that succeeded. All remote methods in RemoteDBBankServer use this technique to keep the account database consistent. In addition to demonstrating the techniques of atomic transaction processing, the RemoteDBBankServer class provides further examples of using SQL queries to inter-
act with a database.
Example 16- 5: RemoteDBBankServer java

```
import java.rmi .*;
import java.rmi .server.*;
import java.rmi.registry.*;
import java.sql.*;
import java.io.*;
import java.util.*;
import java.util.Date; II import explicitly to disambiguate from java.sql.Date
import Bank.*;
1**
* This class is another implementation of the RemoteBank interface.
* It uses a database connection as its back end, so that client data isn't
* lost if the server goes down. Note that it takes the database connection
* out of "auto commit" mode and explicitly calls commit() and rollback() to
* ensure that updates happen atomically;
public class RemoteDBBankServer extends UnicastRemoteObject
implements RemoteBank {
```

3.15 MySQL DataBases





3.15.1 A Short Tutorial on MySQL

refer to a particular table as Questions.table3. This would refer to table 3 in the Questions database.

mysqld - starts MySQL
mysqladmin -u root shutdown -- shutsdown MySQL

msql gives access to the editor

runs on port 3306
copy of the driver can be obtained from http://www.mysql.com/downloads/

manual.html is a must - Section 9 gives a good tutorial

Use the SHOW statement to find out what databases currently exist on the server:

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| test     |
| tmp      |
+----------+
```

Selecting a database...

```
mysql> USE menagerie
Database changed
```

to list all tables in a database...
Now that you have created a table, SHOW TABLES should produce some output:

```
mysql> SHOW TABLES;
+--------------------+
| Tables in menagerie |
+--------------------+
| pet                |
+--------------------+
```

To verify that your table was created the way you expected, use a DESCRIBE statement:

```
mysql> DESCRIBE pet;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| name    | varchar(20) | YES  |     | NULL    |       |
| owner   | varchar(20) | YES  |     | NULL    |       |
| species | varchar(20) | YES  |     | NULL    |       |
| sex     | char(1)     | YES  |     | NULL    |       |
| birth   | date        | YES  |     | NULL    |       |
| death   | date        | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

You can use DESCRIBE any time, for example, if you forget the names of the columns in your table or what types they are.

before a database can be accessed from a Java app it has to be created by the MySQL Administrator (me...). Open a dos window and with MySQL running start the editor with mysql and do
msql> create database JeffsDatabase; note the semi-colon.

databases are located in the directory --- c:\mysql\data

CREATE DATABASE [IF NOT EXISTS] db_name
CREATE DATABASE creates a database with the given name. Rules for allowable database names are given in section 7.1.5 Database, table, index, column and alias names. An error occurs if the database already exists and you didn't specify IF NOT EXISTS.

I used the following to grant JEFF access to all the databases in MySQL. Password for JEFF is set as pat.

C:\MySQL\bin>mysql --user=root mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.32-debug

Type 'help;' or '\h' for help. Type '\c' to clear the buffer

mysql> grant all privileges on *.* to JEFF@localhost identified by "pat" with grant option;
Query OK, 0 rows affected (0.11 sec)

mysql> grant all privileges on *.* to JEFF@"%" identified by "pat" with grant option;
Query OK, 0 rows affected (0.00 sec)

mysql> \q
Bye

Access Control info - see section 6.0

4 REMOTE METHOD INVOCATION

4.1 RMI Background

RMI is an acronym for Remote Method Invocation, a new package that was included in the release of the JDK 1. 1. RMI provides Java with the ability to execute methods of an object that exists in another virtual machine. The remote object may exist in another application on the same machine, on another machine on the local LAN or on a remote LAN on the Internet. RMI is based on the Remote Procedure Call (RPC) technology that evolved in the 1980s, which allowed procedures on remote machines to be executed as easily as procedures on the local machine. RMI expands on this principle, adding the object-oriented technologies that have evolved in recent years.

4.2 What is RMI ?

Distributed systems require that software running in different address spaces, potentially on different hosts, be able to communicate with other software. For a basic communication mechanism, the Java language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone.

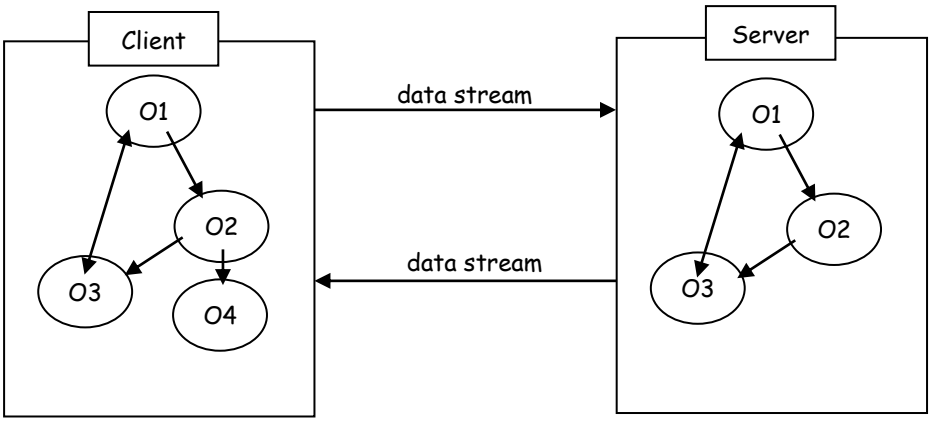
An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR.

RPC, however, does not translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require remote method invocation or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

The Java remote method invocation system has been specifically designed to operate in the Java environment. While other RMI systems can be adapted to handle Java objects, these systems fall short of seamless integration with the Java system due to their interoperability requirement with other languages. For example, CORBA presumes a heterogeneous, multilanguage environment and thus must have a language-neutral object model. In contrast, the Java language's RMI system assumes the homogeneous environment of the Java Virtual Machine, and the system can therefore take advantage of the Java object model whenever possible.

4.2.1 The Traditional Client-Server Model

The traditional client-server model is one where clients are processes (running programs) which reside on one or more host computers, and are able to communicate with a server process running a Java program on its host (which in general is different from the client hosts). The communication between clients and server in this model takes place through ports and sockets, while the messages that make up the communication take the form of streams of data. A message protocol must be established between client and server so that the client and server know how to respond to received messages. This means that the messages from the client are not method invocations in the 00 sense, but rather a stream of data that must be interpreted by the server before it can invoke methods on the objects it knows about. Communication takes places through the use of ports, Socket objects, ServerSocket objects and Stream objects.



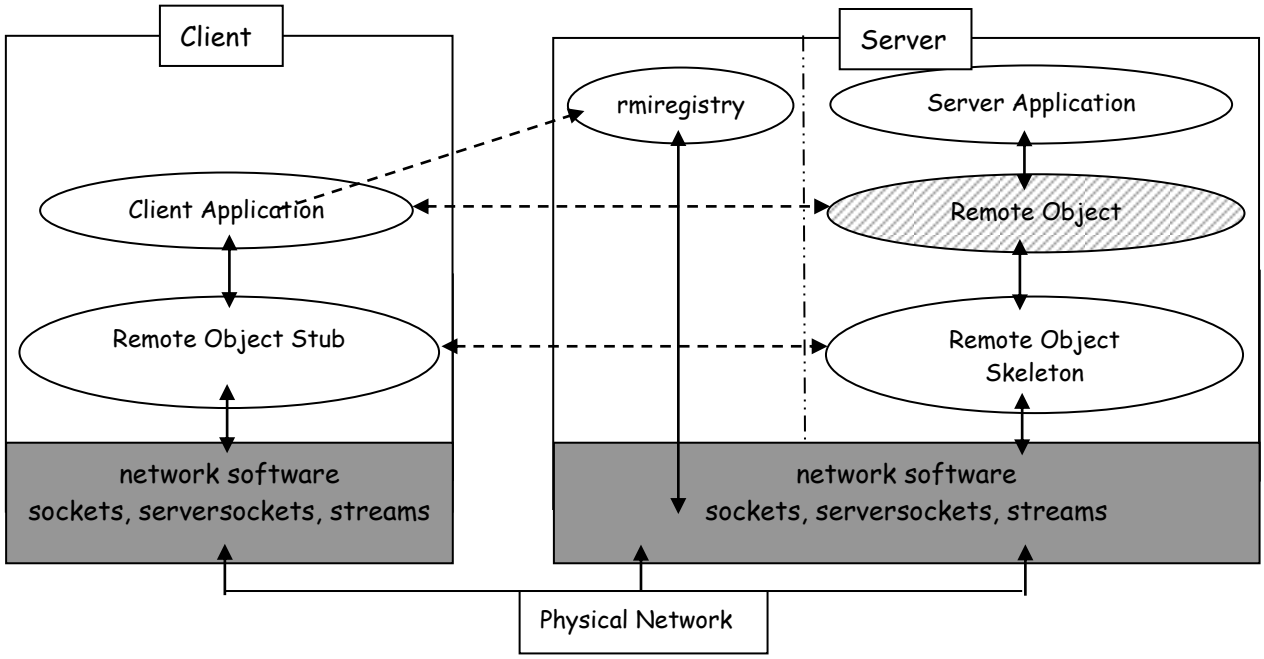
Socket level programming is viewed as a primitive mechanism that is prone to error and many experts argue that RMI should be the preferred mechanism. There are however drawbacks with using RMI techniques.

- Invoking remote methods has a high overhead. As larger, more complex objects are passed, a time penalty is incurred to convert the objects into the proper format required by the serialization methods.
- An RMI application is Java-specific. Unlike CORBA, there are no standards for using RMI with other platform-specific software. Currently, the remote methods must be called by a Java application.
- Development is more difficult. Using RMI to its fullest is sometimes difficult and requires lots of practice and forethought. A poorly designed RMI application can reduce performance to unacceptable levels.

4.2.2 RMI fits the OO Paradigm

In the 00 paradigm messages are sent to a receiver object. RMI implements this message passing mechanism by allowing a client object to send a message to a server object located on a (remote) server host. That is, the client invokes a method on the remote object. The execution of the method takes place on the remote host where the remote object resides. The difference between a client-server model as described above and the 00 paradigm might seem slight but it is significant. It is a non-trivial task to set up communication between clients and a server involving the creation of socket connections and the use of input and output message streams. What the RMI mechanism seeks to achieve is to allow developers to remain within the object-oriented paradigm rather than having to concern themselves with sending messages which are only indirectly connected to that paradigm.
RMI requires several components to function properly, including a remote interface, remote server, client, and naming registry. Each of these components defines a specific function and communicates using object serialization. Object serialization is a method of converting object hierarchies into byte streams for transportation. This allows Java applications that exist outside of the same virtual machine to communicate with one another.

4.3 How the RMI Mechanism Works



In the diagram above the dotted lines represent a virtual communication. The actual communication takes place between the stub and skeleton objects at the physical layer. This real communication is invisible to both the client and server applications software. It is handled by the Java RMI classes and the Java io and net classes (streams, sockets, serversockets etc).

Java's RMI approach is organized into a client/server framework. A local object that invokes a remote object's method is referred to as a *client object,* or simply a *client.* A remote object whose methods are invoked by a client object is referred to as a *server object,* or a *server.*

Java's RMI approach makes use of stubs and skeletons. A *stub* is an object that is downloaded by the client application. The stub object acts as a proxy for the remote server object. The stub provides the same methods as the remote server object. The client object invokes the methods of the stub object as if they were the methods of the remote server object. The stub then communicates these method invocations to the remote server object via a skeleton that is implemented on the remote host. The skeleton object is a proxy for the remote server object that is on the same host as the remote server object. The skeleton communicates with the stub of the client and marshalls method invocations from the stub to the actual remote object. It then receives the value returned (if any) by the remote method invocation and passes this value back to the stub. The stub, in turn, sends the return value on to the client object that initiated the remote method invocation.

Stubs and skeletons communicate through a remote reference layer. This layer provides stubs with the capability to communicate with skeletons via a transport protocol. RMI currently uses TCP for information transport, although it is flexible enough to use other protocols.

4.3.1 The Communication Process

A client communicates with a server in the following manner,

1. The client obtains an instance of the stub class. The stub class is automatically pregenerated from the target server class by the rmi compiler and implements all the methods that the server object implements.

2. The client calls a method on the stub. The method call is actually the same method call the client would make on the server object if both objects resided in the same JM

3. Internally, the stub uses a socket connection to the skeleton on the server. It marshalls all the information associated to the method call, including the name of the method and the arguments, and sends this information over the socket connection to the skeleton.

4. The skeleton demarshalls the data and makes the method call on the actual server object. It gets a return value back from the actual server object, marshalls the return value, and sends it back over the wire (ie the Sockets) to the stub.

5. The stub demarshalls the return value and returns it to the client code.

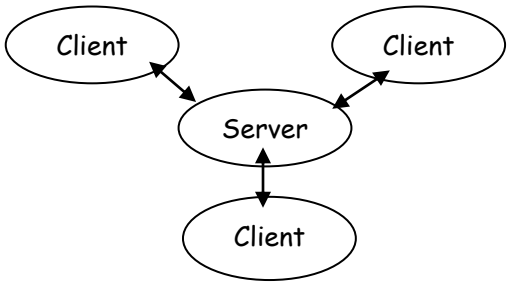
4.4 RMI Configurations

Three example rmi configurations are illustrated.

- Client and Server on the same machine. This is clearly an artificial use of rmi but its purpose is to demonstrate the basic rmi principles and show how an object (the client object) of one application (the client software) can communicate with an object (the remote object) of another application (the server software). It should provide you with a feel for the rmi processes involved. No security manager is necessary.

- Client application and server application on different machines. The client software is developed on the client machine. The client developer is aware of the remote object and its capabilities. The developer writes client code that sends messages to the remote object. The developer sees the remote object as offering some kind of service that the client software can use. When the client software is executed on the client machine it seeks out the required remote object on a remote server machine and sends/receives messages to/from it. In this configuration RMI uses serialisation to transfer the stub object from the server machine to the client machine. The client needs some degree of protection from a possible malicious stub class and so a Security Manager is now required.

- Bootstrapping a client application to client machines. The server maintains a set of remote objects accessed by a number of clients. Client software, residing on the server machine, is bootstrapped from the client machine by a generic bootstrap program that is posted from the server machine to the client machine. The only software that resides on the client machine is the bootstrap program and a Security Manager class. The client application is loaded remotely (ie on the server) but executed locally (ie on the client). The server is an 'object resource centre''. This is an easy way of distributing client software. Updates are easily handled. Again a Security Manager is required.



4.5 Client and Server on the same Machine

4.5.1 A Warning

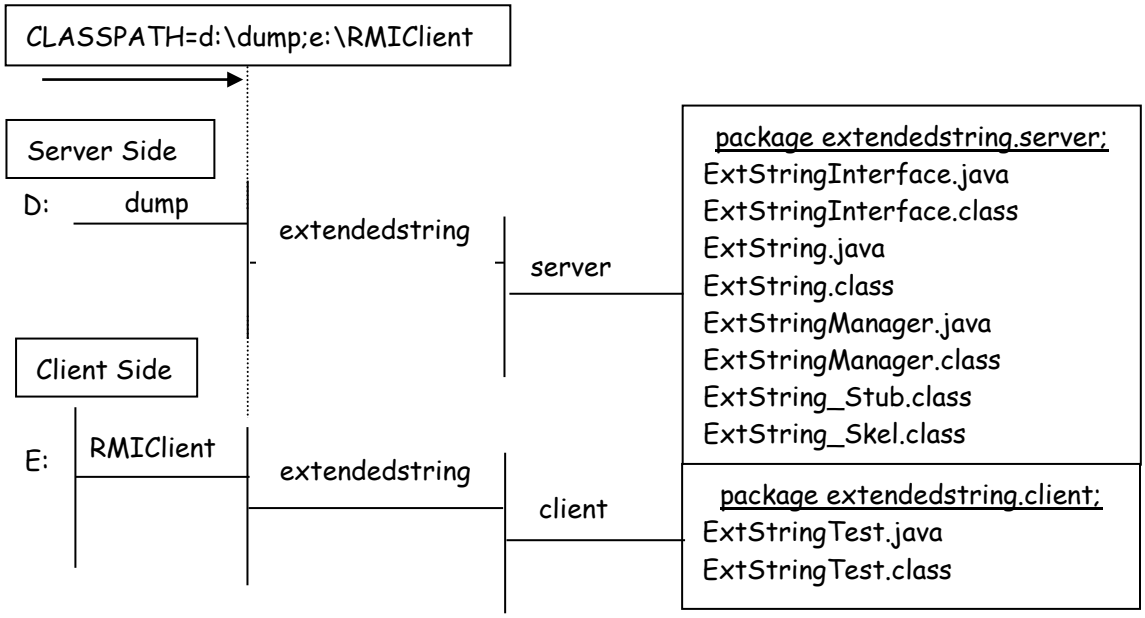
You must be totally conversant with the ideas and practice of the java classpath and the java package mechanism. If you are in any way unsure on any aspect of either you are advised to do some revision before attempting the practicals and exercises in this rmi section. Unless you fully understand setting and using the classpath and managing packages you will run into difficulties with this rmi work.

We demonstrate the development of a RMI service by creating a remote object that will provide extended services on a String object. The client of our RMI extended string service is able to call upon the remote object, stored on the server, to perform a number of operations (services) on a string object that are not provided by the java.lang.String class methods. In this introductory example both the client and the server will be applications and will run on the same machine, hence the use of the loopback address 127.0.0.1. We shall see later what additions are required to implement the system across a network such as the Internet.

4.5.2 Your System Classpath and The Working Directorys Structure

Developing a distributed system on a single machine often leads to a number of classpath/directory pitfalls. To make the development a little more realistic we can 'separate' the client and server software by using the following structure. If you don't have two drives on your system then at least you should follow the directory structure as far as possible on your single drive. For the demonstration example that follows this is how things will look when the system is complete. You will need to refer back to this diagram as the development of your rmi system progresses.

The example of rmi that we are about to develop uses the following directory/package structure. This package structure and the use of the CLASSPATH variable are vital to a successful implementation so you should take careful note of the layout.



Since the development and implementation of the RMI system is sensitive to correct directory and classpath settings I strongly suggest that you use the structure above while you work through these notes. When you have gained some experience managing RIM applications you can experiment with different package (ie directory) structures.

Since both client and server are on the same machine we will not concern ourselves at this point with a very important feature of RMI, the use of a Security Manager. When I discuss the more likely case that the client and server form part of a distributed system I will introduce and implement a Security Manager.

4.5.3 On The Server Side - Step 1 – Design the Interface of the Remote Object

The first step in implementing a client/server RMI application is to define an interface. This interface is responsible for defining the methods that may be invoked by an object in another virtual machine. This interface specifies the method names, parameters, and return values that are required. It is the building block for the application, since the remote server object must implement this interface to be considered a remote object.

```
package extendedstring.server;
import java.rmi.*;
public interface ExtStringInterface extends Remote {
    public int getNumVowels(String aString) throws RemoteException;
    public String padLeft(String aString,int len,char c) throws RemoteException;
    public String padRight(String aString,int len,char c) throws RemoteException;
    public int charCount(String aString,char c) throws RemoteException;
}
```

4.5.4 On The Server Side - Step 2 – Create the Remote Object's Class

Design and develop the class of the remote object. The class must implement the above interface. The class will define the interface methods and possibly additional (local) methods necessary for the remote object to

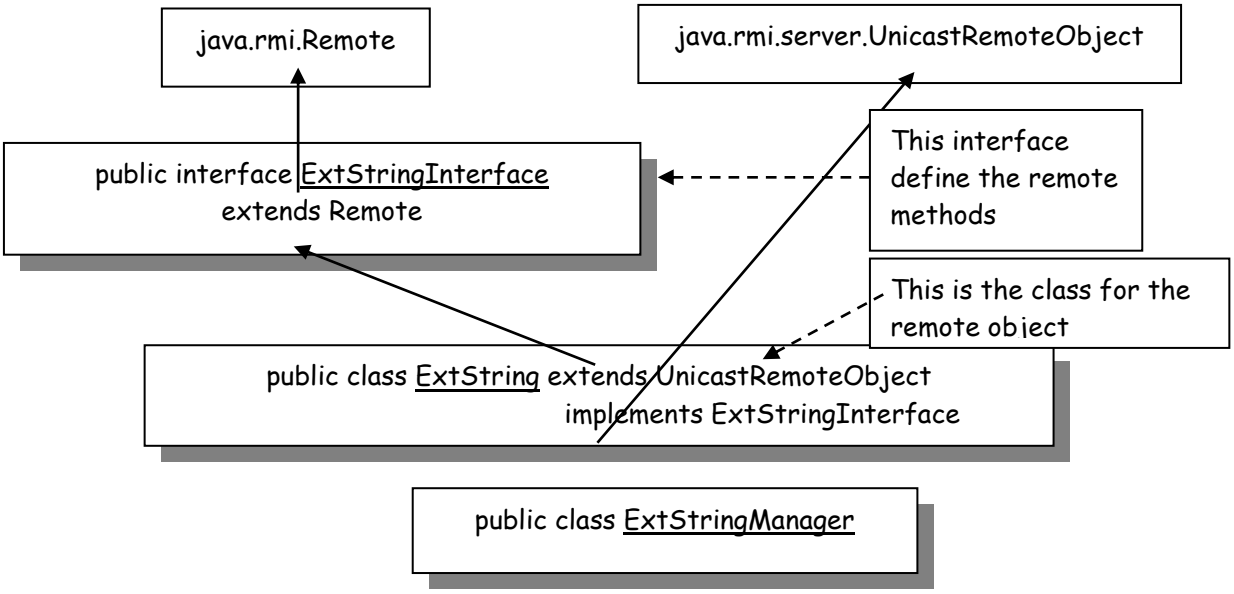
perform locally. The class is required to extend the UnicastRemoteObject class included in the java.rmi.package. This class provides the necessary functionality to facilitate remote computing. This is the class from which the remote object will be instanced. From the users point of view they will be communicating with an ExtString object for the services they require.

We use synchronized methods to prevent two clients getting access to the same method at the same time. I will discuss the issue of synchronisation in more detail later in the chapter.

```
package extendedstring.server;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public class ExtString extends UnicastRemoteObject implements ExtStringInterface {
    private StringBuffer workString;
    public ExtString() throws RemoteException {
        super();
    }
    public synchronized int getNumVowels(String aString) throws RemoteException {
        int count = 0;
        char c;
        for (int i = 0; i < aString.length(); ++i) {
            c = aString.charAt(i);
            if (this.isVowel(Character.toUpperCase(c))) ++count;
        }
        return count;
    }
    private synchronized boolean isVowel(char c) {
        return ((c == 'A') || (c == 'E') || (c == 'I') || (c == 'O') || (c == 'U'));
    }

    public synchronized String padLeft(String aString,int n,char c) throws RemoteException {
        int L = aString.length();
        int len = n - L;
        String packingStr = this.formPacking(len,c);
        workString = new StringBuffer(aString);
        workString.insert(0,packingStr);
        return workString.toString();
    }

    public synchronized String padRight(String aString,int n,char c) throws RemoteException {
        int L = aString.length();
        int len = n - L;
        String packingStr = this.formPacking(n,c);
        workString = new StringBuffer(aString);
        workString.append(packingStr);
        return workString.toString();
    }

    private synchronized String formPacking(int length, char c) {
        StringBuffer str = new StringBuffer(length);
        for (int i = 0; i < length; ++i) str.insert(i,c);
        return str.toString();
    }

    public synchronized int charCount(String aString,char c) throws RemoteException {
        int count = 0;
        for (int i = 0; i < aString.length(); ++i)
            if (aString.charAt(i) == c) ++count;
        return count;
    }
```

}

There could be additional methods over and above those defined in the interface and implemented above. These additional methods would be used locally by the remote object. They are not exported, ie made available to client objects since they do not appear in the remote object interface.



4.5.5 On The Server Side - Step 3 - Develop the ExtStringManager Class

If you like this is the 'server application' in our simple system. The class ExtStringManager contains the main() method and is the starting point for the server application. It creates an instance of the ExtString object and registers the object with the RMI naming service, using the name ExtendedString to name/identify the object with the naming service.

```
package extendedstring.server;
import java.rmi.*;
import java.net.*;
public class ExtStringManager {
    public ExtStringManager() {
        try {
            String objName = "ExtendedString";
            ExtString extStr = new ExtString();
            System.out.println("About to attempt to bind : " + objName + " to the naming server...");
            Naming.rebind("rmi://127.0.0.1/" + objName,extStr);
            System.out.println("Ok... " + objName + " is registered with the naming server...");
        }
        catch (Exception e) { e.printStackTrace(); }
    }

    public static void main(String args[]) {
        new ExtStringManager();
    }
}
```

the code Naming.rebind("rmi://192.168.0.1/" + objName,extStr);
registers the remote object with the application naming server. This naming server, also known as the registry, is a Java program that keeps a record of where remote objects are kept (it is informed by the server) so that clients can access this information. Once the remote object is bound to the registry with a name, the client object may locate it.
There are two methods that can be used to register an object with the application naming server (the rmiregistry). These are bind() and rebind() as used in the example. If an object with the same name already exists in the registry then rebind() with the same object name will simply replace the existing one, whereas

bind() will throw an AlreadyBoundException. That is a remote object with the same name is already in the registry.

4.5.6 On The Server Side - Step 4 - Compile All Server Side Classes at this stage

On the server side you should now compile the interface (ExtStringInterface), and the two classes, the implementation class (ExString) and the remote string manager class (ExtStringManager).

4.5.7 On The Server Side – Step 5 - Create the Stub/Skeleton Classes using the RMIC

The final development step on the server side is to create the proxy objects that will form the stub/skeleton layer for the rmi architecture. So you now must run the RMIC compiler on the ExtendedString.class. The RMIC compiler should be available in the bin directory of the JDK (..\bin\rmic.exe). It's probably best to use a command line window with something like

D:\dump > rmic extendedstring.server.ExtString

Note the current directory d:\dump (which must be on your classpath) and the fully qualified package name, which you must use, (extendedstring.server.ExtString) for the target class.
This produces the files for the proxy objects. You should find them in the server directory.

- ExtString_Stub.class
- ExtString_Skel.class

Ok so what have we done so far. We have,

- developed the remote object's interface
- developed the remote object's class that implements the methods of the interface
- developed the remote object manager class that creates the remote object and registers it with the server's naming service
- used the rmi compiler (rmic) to create the _Stub and _Skel class files

The server, working directory should now contain the following files,

server directory
ExtStringInterface.java
ExtStringInterface.class
ExtString.java
ExtString.class
ExtStringManager.java
ExtStringManager.class
ExtString_Stub.class
ExtString_Skel.class

4.5.8 Disappearing _Stub and _Skel files

One of the problems with RMI development as outlined above, is that as we develop/compile/recompile the system classes over and over again then this re-compilation process destroys the _Stub and _Skel. Hence you will have to use rmic after each system recompilation.

If you use one of the IDEs (such as jBuilder) you should make sure that the compile before run option is off. Otherwise when you run the registration program later you may want to recompile the RemoteStringInterface and the RemoteStringProcessor classes and consequently destroy the _Stub and _Skel classes.

4.5.9 Developing The Client Side

Things are considerably easier on the client side. All that is required here is to develop a client that creates a reference to a remote object. This reference is obtained by consulting the naming service at the server. Once this reference is obtained it represents the remote object within the client environment and can be sent messages just as any of the objects created solely on the client side. In other words it behaves just as any other object on the client side.

In our example, ExtStringTest makes a reference to ExtStringInterface and therefore requires access to ExtStringInterface for compilation so you will have to make sure that it has access to this class. If you have

correctly set your classpath environment variable as indicated at the beginning of this example then when you compile your ExtStringTest the java compiler will be able to locate the interface in the server package. If you are using an IDE such as Jbiulder then you will have to include a reference to the server package in the Required Libraries section of the Project Properties. Add it to the Jbuilder group.

In a 'real' application the interface would have to be made available to the client developer. So what would normally happen is that the client devloper would create a server sub-directory off the directory where the client software was being developed and place a copy of the interface in this directory. The copy would have to be obtained from the remote server developer. Note also that it is likely that the client would need access to documentation on the ExtStringInterface.class so that the method interfaces can be understood. This documentation would likely be in the form of some API html documents.

On the client side complete and then compile the client class ExtStringTest. Remember to do this in the client directory, e:/RMIClient/extendedstring/client.

```
package extendedstring.client;
import java.rmi.*;
import extendedstring.server.*; // we need access to the ExtStringInterface
public class ExtStringTest {
    private String theString;
    private String theServerID;
    private String theRemoteObjectName;
    private ExtStringInterface stubObject;

    public ExtStringTest(String aServerID, String aRemoteObjectName) {
        try {
            theServerID = aServerID;
            theRemoteObjectName = aRemoteObjectName;
            stubObject = (ExtStringInterface)Naming.lookup(theServerID + theRemoteObjectName);
        }
        catch (Exception re) {
            re.printStackTrace();
        }
    }

    public void start(String aString) {
        theString = aString;
        try {
            System.out.println("Starting the extended string processing on the string : " + theString);
            System.out.println(stubObject.padLeft(theString,25,'x'));
            System.out.println(stubObject.padRight(theString,25,'x'));
            System.out.println("Number of vowels = " + stubObject.getNumVowels(theString));
            System.out.println("Number of f's = " + stubObject.charCount(theString,'f'));
            System.out.println("Number of p's = " + stubObject.charCount(theString,'p'));
        }
        catch (Exception re) { re.printStackTrace(); }
    }

    public static void main(String args[]) {
        ExtStringTest est = new ExtStringTest("rmi://127.0.0.1/","ExtendedString");
        est.start("jeff allen");
    }
}
```

The client attempts to locate the name of the server object in the registry. If it is found, the registry will return a reference to the stub object. This stub will be used by the client object to invoke methods on the remote server object. The stub provides the necessary methods (streams/sockets etc) for communicating to the remote server.

Finally the client may now invoke remote methods. These methods, defined in the remote interface, are the key to the communications between the client and the remote object. In addition the remote object may be referenced by more than one client, giving a one to many relationship. This leads to a multithreaded system.

4.5.10 Running the System

There are various ways you might do this. One approach to running the system is to open 3 command windows…

1. to start the rmiregistry program
2. to start the server application, ExtStringManager
3. to start client application, ExtStringTest

If you are using an IDE it may be possible to run everything from within the IDE, including starting the rmiregistry program.

4.5.11 Start The Naming Service

Start the naming service software on the server. This is the java rmiregistry program. In a win9x/NT environment you can start the rmiregistry program with a command line,

```
d:\dump > rmiregistry
```

This should run as a background task. If you are using win9x/NT etc you can see the registry as a task by using the ctrl/alt/del key combination.

Another important point is that you should stop and restart the rmiregistry program as you devlop your rmi system. Experience shows that the registry can become ‘confused’ (another word for bug ?) by having to process repeated attempts at registering your rmi object. So clear (stop/start) the registry from time to time during devleopment.

The RMI registry is a Java program named rmiregistry which, when it is executing, maintains a list of references to objects that have been registered with it. An object is registered by a server using the rebind() or bind() methods from the Naming class (part of the java.rmi package). The first argument of the rebind/bind methods is a String that specifies,

- where the registry is located, by quoting the IP address of its host (localhost (127.0.0.1) can be used when the registry is on the same host as the server) and the port on which the registry listens for communications (by default this is 1099, and can be omitted)

- a string name be allocated to the object by which clients can gain access to the object; this will be the objects name in the naming server (the rmiregistry)

The second argument is a reference to the object itself. A client gains access to a remote object by executing the method lookup(), also from the Naming class. This method has a single String argument as in the rebind method that specifies where the registry is located (the registry host's IP address and port number) and the String name by which the registry knows the object. The registry program rmiregistry is part of the Java system. There is another useful method in the Naming class, namely list. This method returns a list of all the names of objects currently supported by the registry. It is important to recognise that the RMI registry is a non-persistent scheme. The registry forgets all its contents when the rmiregistry program stops.

4.5.12 Start the ExtStringManager to Create and Register The Remote Object

The final action on the server side is to execute the ExtStringManager class. As described previously this will create the remote object, register it with the server namimg service and wait for a client connection.
Start your server with,

```
d:\dump > java extendedstring.server.ExtStringManager
```

You should see the following in the command window

```
About to attempt to bind : ExtendedString to the naming server...
Ok... ExtendedStringis registered with the naming server...
```

4.5.13 Start the Client Software

Start you client with

```
e:\RMIClient > java extendedstring.client.ExtStringTest
```

When the client runs you should see the following output and responses from the remote object.

```
Starting the extended string processing on : jeff allen
xxxxxxxxxxxxxxxjeff allen
jeff allenxxxxxxxxxxxxxxxxxxxxxxxxxx
Number of vowels = 3
Number of f's = 2
Number of p's = 0
```

4.5.14 A Brief Step by Step Guide to RMI Success

Set your classpath=d:\dump;e:\RMIClient (or whatever….)

On the Server Side

1. Develop the Remote Object Interface
   public interface ExtStringInterface extends Remote {
2. Develop the Remote Object Class
   public class ExtString extends UnicastRemoteObject
   implements ExtStringInterface {
3. Develop the ExtStringManager Class (The server application…)
   public class ExtStringManager {
4. Compile All Server Side Classes
5. Create the Stub/Skeleton Classes using the RMIC
   D:\dump > rmic extendedstring.server.ExtString

On The Client Side

1. Make the server side interface, ExtStringInterface, available to the client developer. This is achieved by setting the classpath environment variable to to include the server directory path.
2. Complete and then compile the client class ExtStringTest. Remember to do this in the client sub-directory.

Running The System

1. Open 3 command windows…
   one for the rmiregistry program
   one for ExtStringManager
   one for ExtStringTest
2. Start The Naming Service
   d:\dump > rmiregistry
3. Start the RMIserver to create and register the remote object
   d:\dump > java extendedstring.server.ExtStringManager
4. Start the Client Software
   e:\RMIClient > java extendedstring.client.ExtStringTest

4.6 Client Application and Server Application on Different Machines

To implement the next rmi configuration you will need access to a Web Server and an Internet connection. The Web Server acts as the network server for the class files needed for the RMI client and Server. The client uses http protocol to contact the server and then gains access to the remote object through the server. The proxy object (the _Stub) class has to be transferred to the client machine from the server machine. See next section for further details.

4.6.1 Stub object Transfer and RMI Security

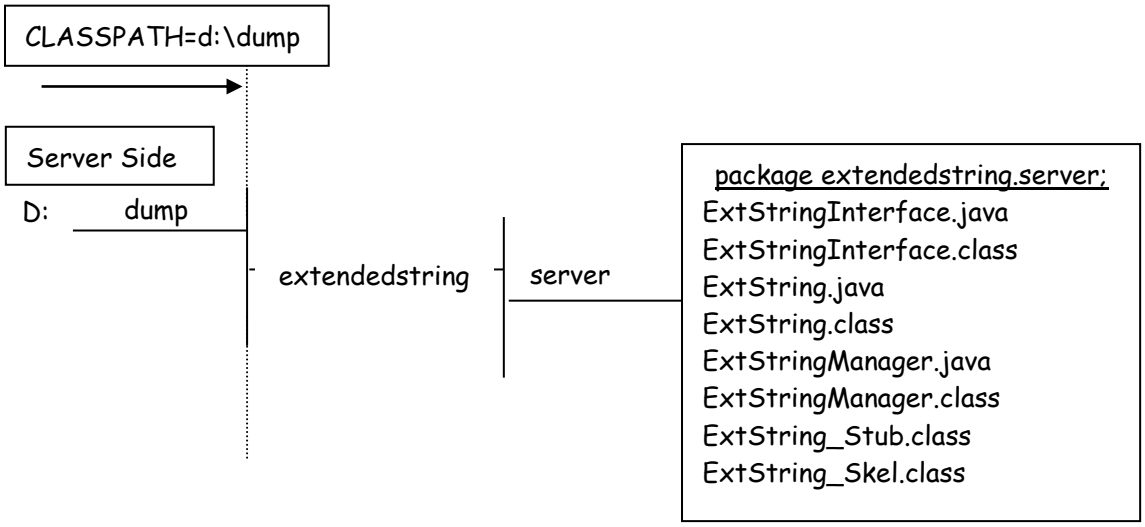
When the client application consults the rmi naming service on the server with a reference to a remote object the server uses serialisation to transfer the stub object to the client machine. Because a client may load an untrusted stub object, it should have a security manager installed to prevent a malicious (or just buggy) stub from deleting files or otherwise causing harm. A Security Manager is now required at the client side.

To relieve some of the restrictions placed on RMI applications by the RMISecurityManager an approach that is often used is to subclass the RMISecurityManager class and override the methods necessary to provide the correct implementation needed by the application. Note that removing some of the protection provided by the RMISecurityManager may have strong implications on an application and must be taken seriously. Java security is beyond the scope of these RMI notes, but it is crucial that a programmer understand the Java security model before modifying the security manager.

We present two RMI configurations. Both approches involve a security manager. The first approach involves overriding some methods of the RMISecurityManager class and the second approach involves using a substitute policy file.

4.6.2 Approach 1 - Overriding the RMISecurityManager Class

Developing The Server Side
The server side software was developed in the following structure on the server machine,

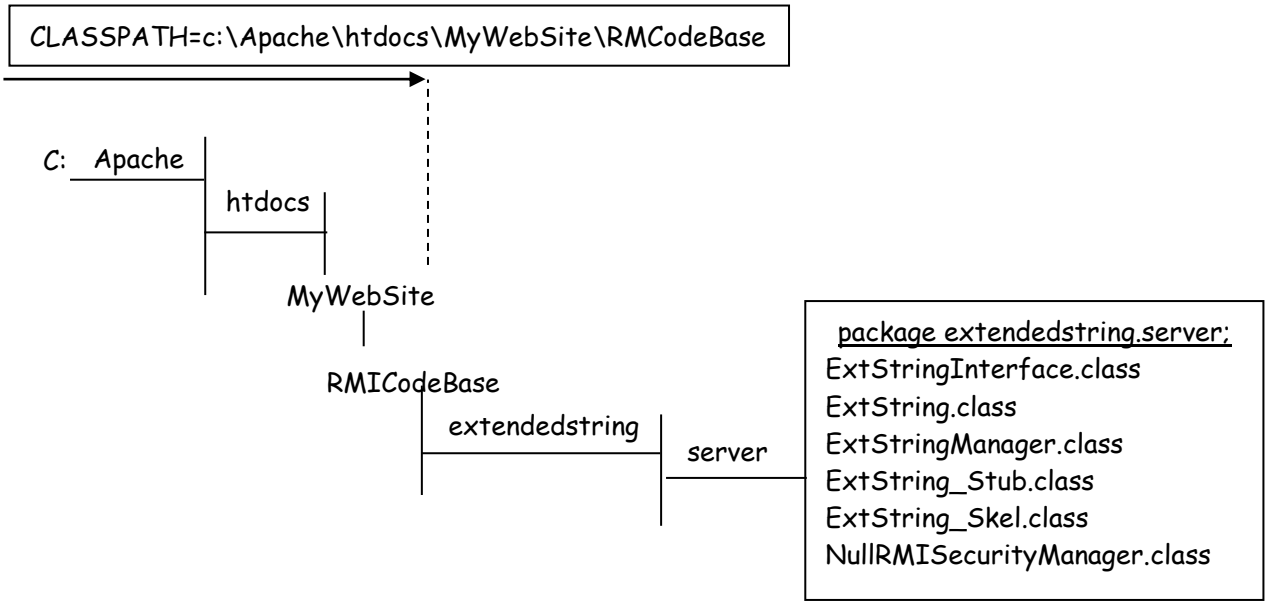


The change required in the server application, ExtStringManager.java, is as follows. The code changes are shown in bold type.

```
package extendedstring.server;
import java.rmi.*;
import java.net.*;
public class ExtStringManager {
    public ExtStringManager() {
        try {
            String objName = "ExtendedString";
            ExtString extStr = new ExtString();
            System.out.println("About to attempt to bind : " + objName + " to the naming server...");
            Naming.rebind("rmi://192.168.0.1/" + objName,extStr);
            System.out.println("Ok... " + objName + " is registered with the naming server...");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```

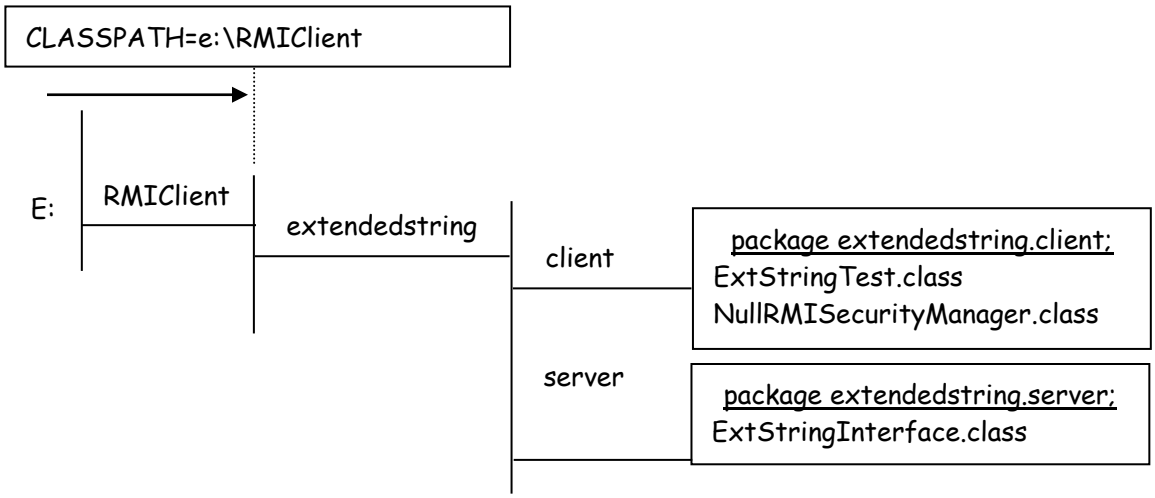
I used the Apache Web Server to act as the network server for the RMI application. The client uses http protocol to contact the server software through the Apache Server and the Apache server sends the stub class to the client.

I deployed the server software as follows,



Developing The Client Side

The client side software was developed in the following structure on the client machine,



We subclass the RMISecurityManager class as follows,

```
package extendedstring.client;
import java.rmi.RMISecurityManager;
import java.security.Permission;
/**
 * This class defines a security policy for RMI applications
 * The policy changes from RMISecurityManager are:
 *       Security Check                This Policy       RMISecurityManager
 * ---------------------------------------------------------------------------
 * Access to Thread Groups                YES                NO
 * Access to Threads                   YES              NO
 * Create Class Loader                    YES                NO
 * System Properties Access            YES                NO
 * Connections                         YES                NO
 */
```

```
public class NullRMISecurityManager extends RMISecurityManager {
    //   Loaded classes are allowed to create class loaders.
    public synchronized void checkCreateClassLoader() {
        // Provide null override
    }
    //   Connections to other machines are allowed
    public synchronized void checkConnect(String host, int port) {
        // Provide null override
    }
    //   Loaded classes are allowed to manipulate threads.
    public synchronized void checkAccess(Thread t) {
        // Provide null override
    }
    //   Loaded classes are allowed to manipulate thread groups.
    public synchronized void checkAccess(ThreadGroup g) {
        // Provide null override
    }
    //   Loaded classes are allowed to access the system properties list.
    public synchronized void checkPropertiesAccess() {
        // Provide null override
    }
    public synchronized void checkPermission(Permission p) {
        // Provide null override
    }
    public synchronized void checkAccept(String host,int port) {
        // Provide null override
    }
}
```

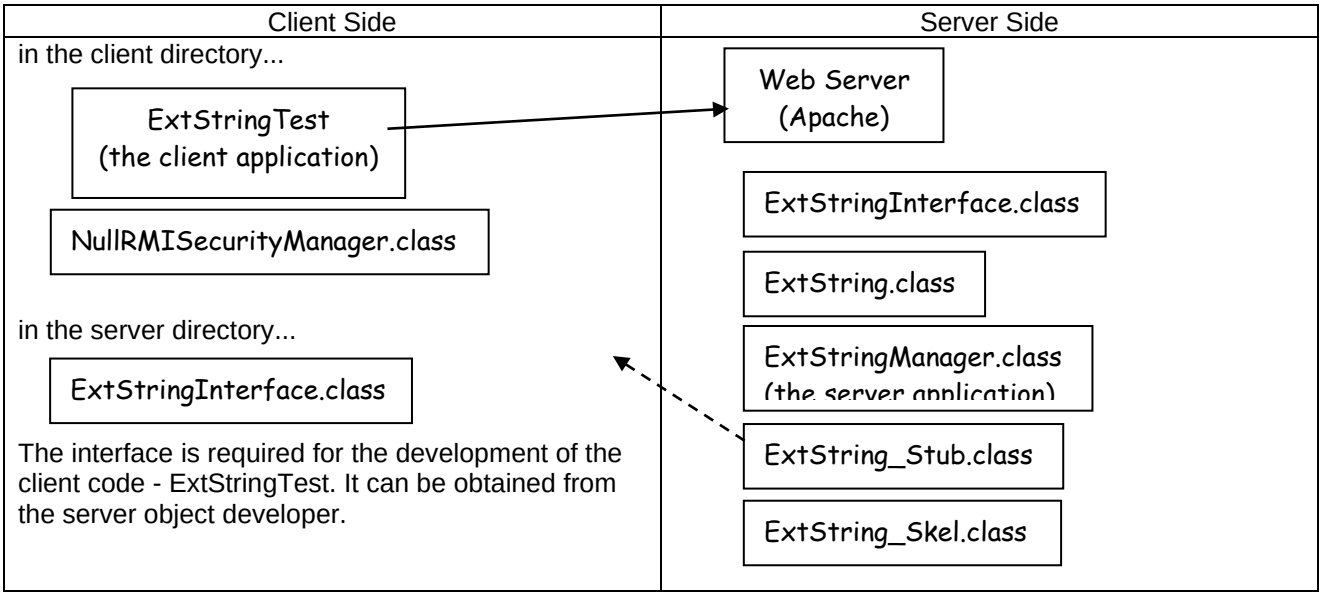
Any client wishing to access the server will need to have access to a copy of NullRMISecurityManager.class and will have to create an instance of NullRMISecurityManager and use this instance when using the System class to set the security manager in their software. So the only change we have to make to the client software is

- create an instance of the NullRMISecurityManager
- re-direct the client to the server machine with a new ip address

The changes are as follows,

```
public static void main(String args[]) {
    System.setSecurityManager(new NullRMISecurityManager());
    ExtStringTest est = new ExtStringTest("rmi://192.168.0.1/","ExtendedString");
    est.start("jeff allen");
 }
```

The architecture we employ is as follows,



The client contacts the Web Server and the server returns a copy of the stub class.

4.6.3 Running the System

This assumes you have created the _Stub and _Skel classes using the RMI compiler (rmic) and organised the classes into their appropriate directories as outlined above.

1. Start the server side system - on the server machine,

- Start the web server
- Open 2 command windows…
  - one for the rmiregistry program
  - one for ExtStringManager
- start the Naming Server – ie rmiregistry
- start the server application – ExtStringManager (to create and register the remote object) with

java^-Djava.rmi.server.codebase=file:c:\Apache\htdocs\MyWebSite\RMICodeBase\^
-Djava.rmi.server.hostname=192.168.0.1^extendedstring.server.ExtStringManager

The above should be regarded as one continuous line with ^ representing a space character.

- The java.rmi.server.codebase property specifies where the publicly accessible classes are located.
- The java.rmi.server.hostname property is the complete host name of the server where the publicly accessible classes reside.
- The class to execute – extendedstring.server.ExtStringManager.

2. Start the client application - on the client machine

- Start the client application with

Java^-Djava.rmi.server.codebase=http://192.168.0.1:80/MyWebSite/RMICodeBase/^
extendedstring.client.ExtStringTest

The above should be regarded as one continuous line with ^ representing a space character.

- The java.rmi.server.codebase property specifies where the publicly accessible classes for downloading are located.
- The class to execute – extendedstring.client.ExtStringTest

If all has gone according to plan you should see the same output as before,

Starting the extended string processing on : jeff allen
xxxxxxxxxxxxxxxjeff allen
jeff allenxxxxxxxxxxxxxxxxxxxxxxxxxx

Number of vowels = 3
Number of f's = 2
Number of p's = 0

Tip : it's a good idea to put the startup commands into a couple of batch files...

4.6.4 Approach 2 - Overriding The Java Policy File

An alternative approach to subclassing the RMISecurityManager class is to override the security policy file. The security policy file is a text file (it can be found and viewed in your jdk jre directory, ..\jre\lib\security) that, amongst other things, mediates access to RMI. Since this is a simple text file it can be easily modified. However perhaps a better/safer option is to override this system file with a policy file of your own and redirect the java security policy to this new file.

Since these notes are not concerned directly with java security issues we will use a very simple new policy file that gives full permissions to our client. Our simple replacement policy file is then,

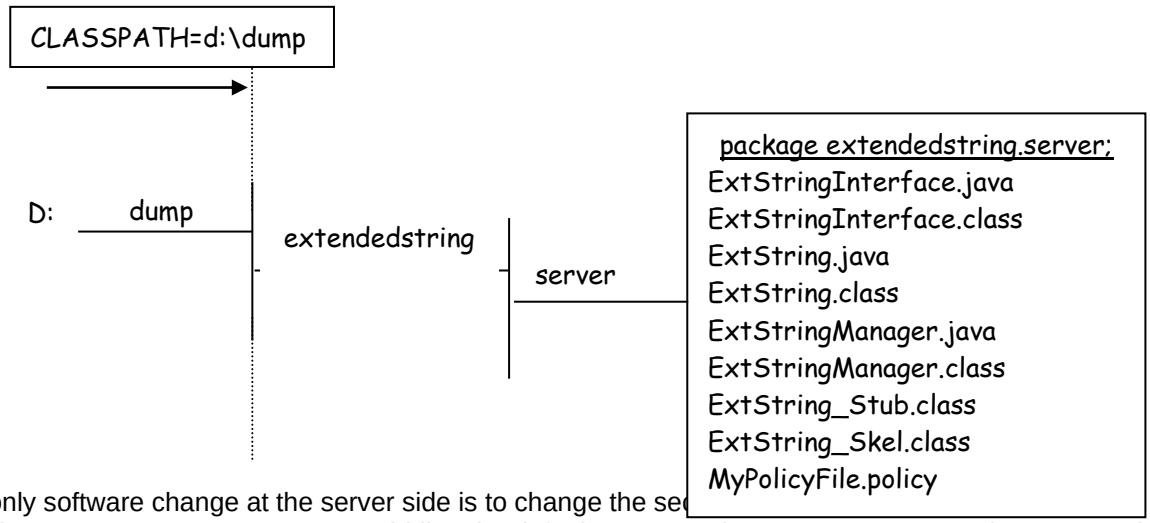
```
// This is a simple replacement version for the system security file
grant {
    permission java.security.AllPermission;
};
```

Save this as MyPolicyFile.policy in the server directory.

We can now dispense with our NullRMISecurityManager class and effectively replce it by our modified security policy file.

Developing The Server Side

The server side software was developed in the following structure on the server machine,

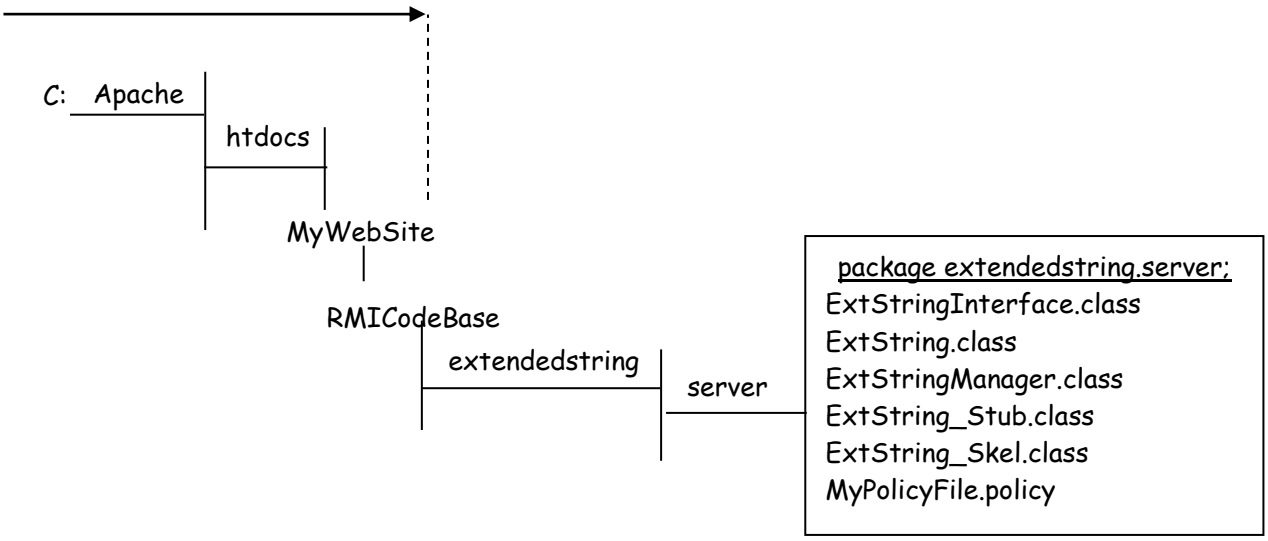


The only software change at the server side is to change the se
security manage. We are now not overidding the default RMI security manager. So ExtStringManager.java needs to be changed to

```
public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    new ExtStringManager();
}
```

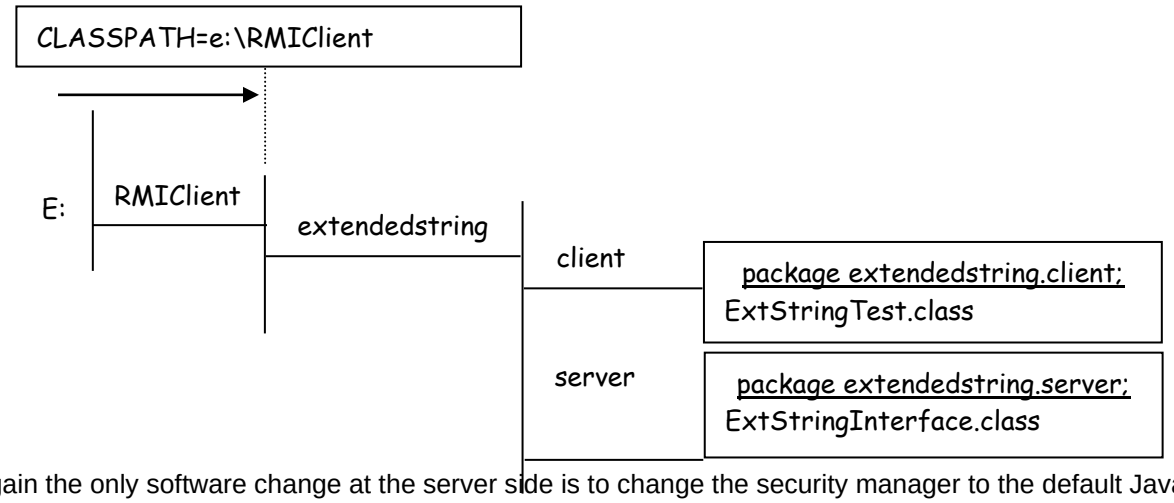
As before the server system needs to be deployed inside the Apache Web Server,

```
CLASSPATH=c:\Apache\htdocs\MyWebSite\RMCodeBase
```



Developing The Client Side

As before the client side software was developed in the following structure on the client machine,



Again the only software change at the server side is to change the security manager to the default Java RMI security manage. We are now not overidding the default RMI security manager. So ExtStringTest.java needs to be changed to

```
public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    ExtStringTest est = new ExtStringTest("rmi://192.168.0.1/","ExtendedString");
   est.start("jeff allen");
  }
}
```

4.6.5 Running the System

This assumes you have created the _Stub and _Skel classes using the RMI compiler (rmic) and organised the classes into their appropriate directories as outlined above.

1. **Start the server side system - on the server machine,**
   - Start the web server
   - Open 2 command windows…
     one for the rmiregistry program
     one for ExtStringManager
   - start the Naming Server – ie rmiregistry
   - start the server application – ExtStringManager (to create and register the remote object) with

java -Djava.rmi.server.codebase=file:c:\Apache\htdocs\MyWebSite\RMICodeBase\^
-Djava.rmi.server.hostname=192.168.0.1^
-Dava.security.policy=c:\Apache\htdocs\MyWebSite\RMICodeBase\extendedstring\server
MyPolicyFile.policy^extendedstring.server.ExtStringManager

The above should be regarded as one continuous line with ^ representing a space character.

- The java.rmi.server.codebase property specifies where the publicly accessible classes are located.
- The java.rmi.server.hostname property is the complete host name of the server where the publicly accessible classes reside.
- The java.rmi.security.policy property specifies the policy file with the permissions needed to run the remote server object and access the remote server classes for download.
- The class to execute – extendedstring.server.ExtStringManager.

2. Start the client application - on the client machine

- Start the client application with

Java^-Djava.rmi.server.codebase=http://192.168.0.1:80/MyWebSite/RMICodeBase/^
-Djava.security.policy=http://192.168.0.1:80/MyWebSite/RMICodeBase/extendedstring/server/
MyPolicyFile.policy^extendedstring.client.ExtStringTest

- The java.rmi.server.codebase property specifies where the publicly accessible classes for downloading are located.
- The java.rmi.security.policy property specifies the policy file with the permissions needed to run the client program and access the remote server classes for download.
- The class to execute – extendedstring.client.ExtStringTest

If all has gone according to plan you should see the same output as before,

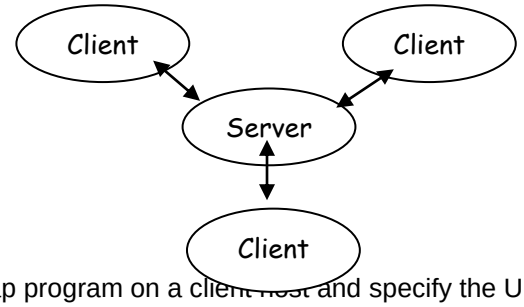
```
Starting the extended string processing on : jeff allen
xxxxxxxxxxxxxxxjeff allen
jeff allenxxxxxxxxxxxxxxxxxxxxxxxxxxx
Number of vowels = 3
Number of f's = 2
Number of p's = 0
```

Tip : it's a good idea to put the startup commands into a couple of batch files...

4.7 Bootstrapping a Client Application on Client Machines

The distribution of java classes using the applet based approach works fine. Classes are loaded on the Web Server and users download the java classes as part of a web page. The browser provides the run time environment. However there may be times when you want to run the client as an application rather than as an applet. In this section we look at a technology that can be used to distribute applications almost as easily as distributing applets. A bootstrap program is distributed to clients and this is used to execute the client software that actually resides on the server host.

In a fully implemented system the server could maintain a set of remote objects accessed by a number of clients. Client software, residing on the server machine, is bootstrapped from the client machine by a generic bootstrap program that is posted from the server machine to the client machine. The only software that resides on the client machine is the bootstrap program and a Security Manager class. The client application is loaded remotely (ie on the server) but executed locally (ie on the client). The server is an 'object resource centre'. This is an easy way of distributing client software. Updates are easily handled.

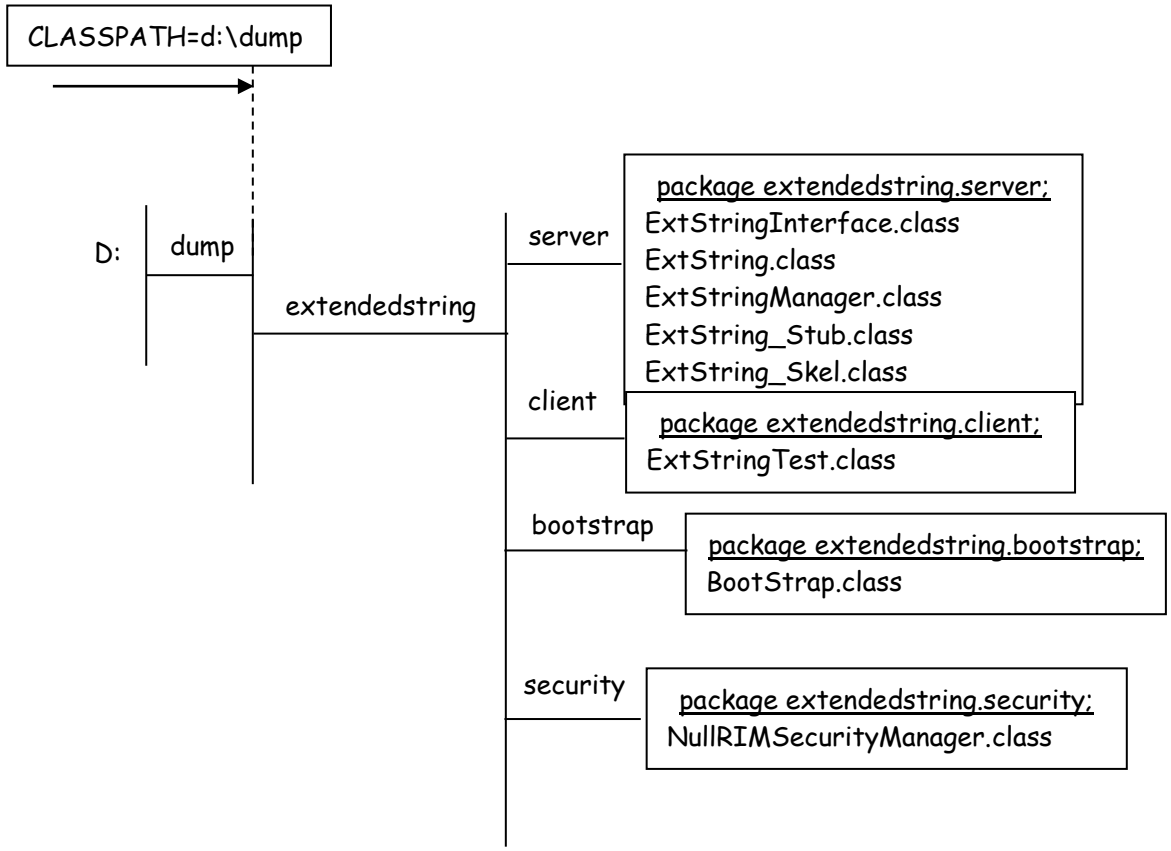


You run the Bootstrap program on a client host and specify the URL and name of a client application (that is, ExtStringTest). Bootstrap retrieves the client application from a remote Web server and executes it locally. The client application (running locally) uses RMI to invoke the methods of server objects (that is, ExtString) executing on the Web server. The Bootstrap program solves the problem of maintaining and distributing client programs. The clients are maintained and stored on a central Web server. They can be upgraded and modified without having to be installed on users' computers. You only need to distribute the Bootstrap program. It is a small program that can be used to load and execute clients from any host.

All classes are loaded from the URL by the client. In particular, the client application – ExtStringTest - is loaded remotely and executed locally. Any classes that are loaded by ExtStringTest are also loaded from the identified URL.

4.7.1 Development of The Software

This time all the development can take place at the server side. When completed the bootstrap software can be deployed at the client side of ths system.
The following structure was used on the server side to develop the software



4.7.2 The BootStrapClass

BootStrap.java is a simple console program. It is important to note that in the first line, it sets the security manager to an object of the NullRMISecurityManager class. This class overrides many of the methods of RMISecurityManager and supports a liberal policy for remotely loaded clients.

```
package extendedstring.bootstrap;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import extendedstring.security.*;
public class BootStrap {
    public static void main(String args[]) {
        System.setSecurityManager(new NullRMISecurityManager());
        System.out.println("Starting BootStrap");
        if (args.length != 1) {
            // a message to the user…
        }
        else {
            System.out.println("Attempting to load : " + args[0]);
            try {
                String urlString = "http://192.168.0.1/MyWebSite/RMICodeBase/";
                Class clientClass = RMIClassLoader.loadClass(urlString,args[0]);
                System.out.println("Ok loaded...");
                String tempStr = clientClass.getName();
                System.out.println("Name is = " + tempStr);
                Runnable clientInstance = (Runnable)clientClass.newInstance();
                clientInstance.run();
            }
            catch (SecurityException se) {
                System.out.println("Jeff-Security Exception...");
            }
            catch (Exception exc) {
                System.out.println("Jeff-Error in BootStrap...");
                exc.printStackTrace();
            }
        }
    }
}
```

The main processing performed by BootStrap.java takes place within the try statement. It invokes the static loadClass() method of the RMIClassLoader class to load the class named as a command line argument – in our case this is ExtStringTest. The class is loaded from the URL specified by the java.rmi.codebase property.

The loaded class – ExtStringTest - is assigned to the clientClass variable. An instance of the loaded class – ExtStringTest - is then created and cast as a Runnable object. This creates a separate thread of execution. That's why I will implement ExtStringTest as Runnable. The run() method of the newly created thread is invoked to cause the thread to be executed. There could be other ways of implementing ExtStringTest of course.

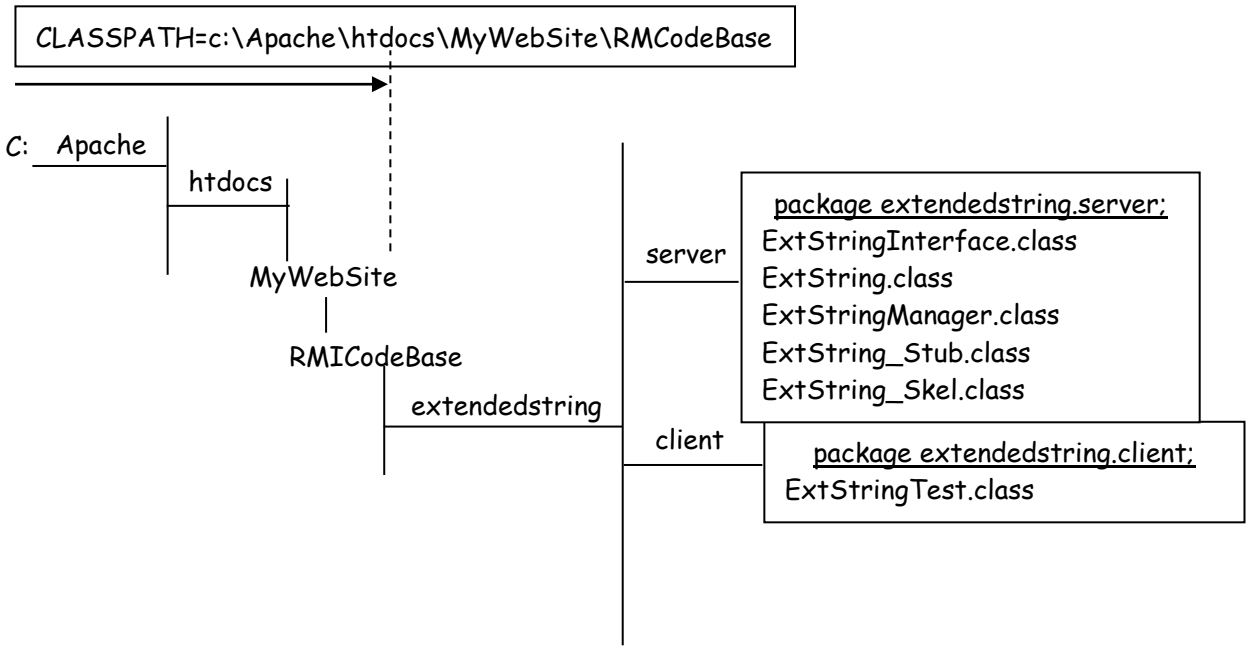
4.7.3 The Amended Client Software - ExtStringTest

```
package extendedstring.client;
import java.rmi.*;
import extendedstring.server.*; // we need access to the ExtendedStringInterface
public class ExtStringTest implements Runnable {
    private String theString;
    private String theServerID;
    private String theRemoteObjectName;
    private ExtStringInterface stubObject;
    public ExtStringTest(String aServerID, String aRemoteObjectName) {
        try {
```

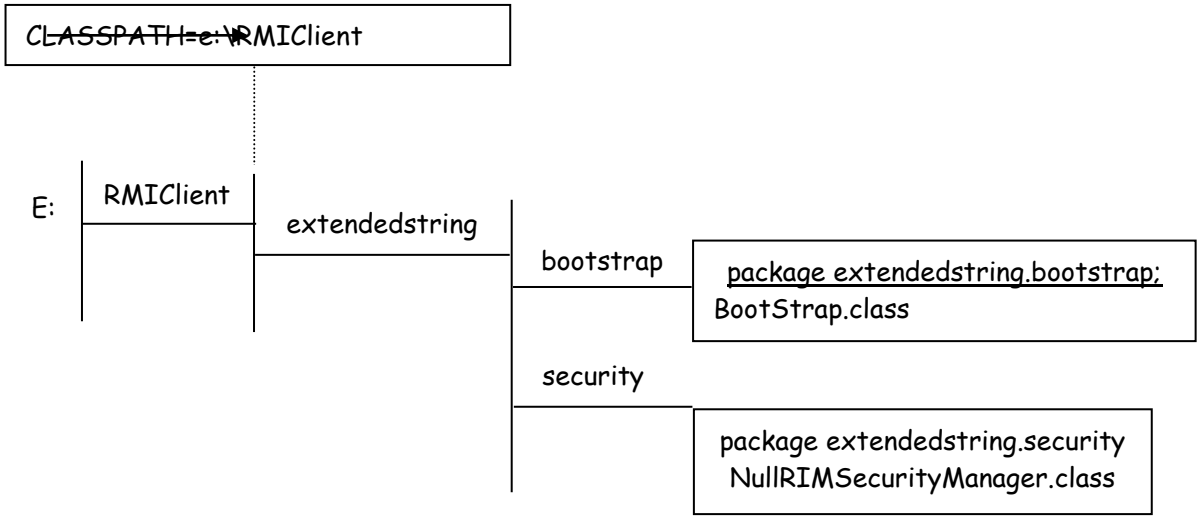
```
            theServerID = aServerID;
            theRemoteObjectName = aRemoteObjectName;
        }
        catch (Exception re) { re.printStackTrace(); }
    }

    public ExtStringTest() {
        // This null constructor is included to satisfy the newInstance() call
        // used in the BootStrap class
    }

    public void run() {
        try {
            theServerID = "rmi://192.168.0.1/";
            theRemoteObjectName = "ExtendedString";
            stubObject = (ExtStringInterface)Naming.lookup(theServerID + theRemoteObjectName);
            this.start("jeff allen");
        }
        catch (Exception re) { re.printStackTrace(); }
    }

    public void start(String aString) {
        theString = aString;
        try {
            System.out.println("Starting the extended string processing on the string : " + theString);
            System.out.println(stubObject.padLeft(theString,25,'x'));
            System.out.println(stubObject.padRight(theString,25,'x'));
            System.out.println("Number of vowels = " + stubObject.getNumVowels(theString));
            System.out.println("Number of f's = " + stubObject.charCount(theString,'f'));
            System.out.println("Number of p's = " + stubObject.charCount(theString,'p'));
        }
        catch (Exception re) { re.printStackTrace(); }
    }
}
```

4.7.4 Deployment of The Software

Using the Apache Web Server the server side of things can be deployed as follows,



and on the client machine we use a structure as follows,



4.7.5 Running the System

This assumes you have created the _Stub and _Skel classes using the RMI compiler (rmic) and organised the classes into their appropriate directories as outlined above.

1. Start the server side system - on the server machine,

- Start the web server
- Open 2 command windows…
  - one for the rmiregistry program
  - one for ExtStringManager
- start the Naming Server – ie rmiregistry
- start the server application – ExtStringManager (to create and register the remote object) with

java^-Djava.rmi.server.codebase=file:c:\Apache\htdocs\MyWebSite\RMICodeBase\^
-Djava.rmi.server.hostname=192.168.0.1^extendedstring.server.ExtStringManager

The above should be regarded as one continuous line with ^ representing a space character.

- The java.rmi.server.codebase property specifies where the publicly accessible classes are located.
- The java.rmi.server.hostname property is the complete host name of the server where the publicly accessible classes reside.
- The class to execute – extendedstring.server.ExtStringManager.

2. Start the client application - on the client machine

- Start the bootstarp program and hence the client application with

java^-Djava.rmi.server.codebase=http://192.168.0.1:80/MyWebSite/RMICodebase/^
extendedstring.bootstrap.BootStrap^extendedstring.client.ExtStringTest

The above should be regarded as one continuous line with ^ representing a space character.

- The java.rmi.server.codebase property specifies where the publicly accessible classes for downloading are located.
- The class to execute – extendedstring.client.BootStrap with its command line parameter ExtStringTest

If all has gone according to plan you should see the same output as before,

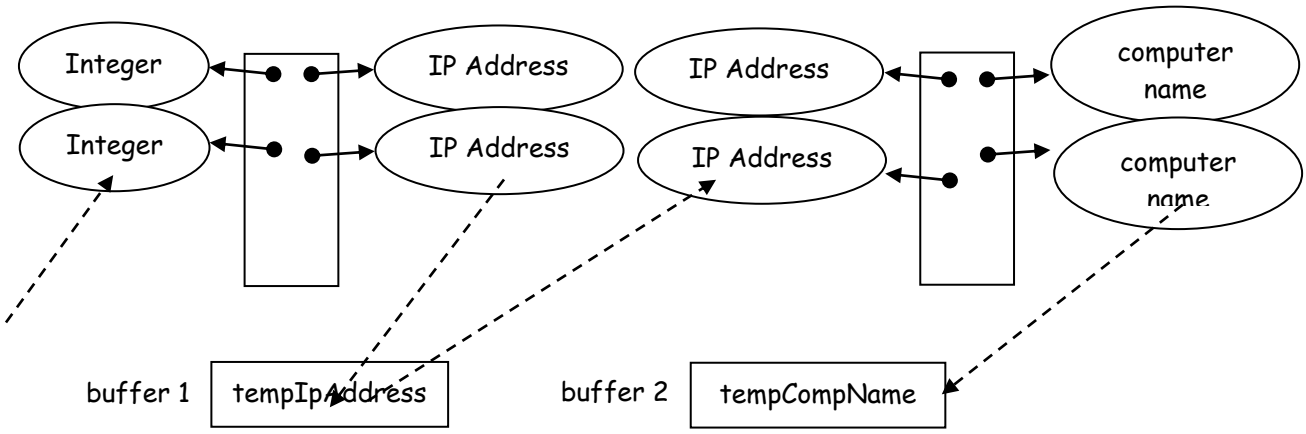
```
Starting the extended string processing on : jeff allen
xxxxxxxxxxxxxxxjeff allen
jeff allenxxxxxxxxxxxxxxxxxxxxxxxxxxx
Number of vowels = 3
Number of f's = 2
Number of p's = 0
```

4.8 Multiple Clients

Previously in this chapter I mentioned the problem of synchronising the methods of the remote object so that only one client at a time can access any particular method of the remote object. The following example demonstrates this aspect in more detail. Although synchronisation appears to solve the problem it actually only partially solves the problem and in fact introduces yet another problem. It does not alone solve the problem of deadlock. Typically this is where two (or more) threads are each waiting for the other to complete a task. Thread A cannot progress until thread B has completed, but thread B is waiting for thread A - deadlock. Futher features - wait() and notify() - are required to fully solve the deadlock problem (See Book 4 for details of these two methods).

The remote object provides a simple look up service. Given the number of a computer the renote object can provide details of the computer. A client can request the details of a computer by sending the remote object an integer. For example a typical request from a client would be for the remote object (the server) to return the details of computer number 3.

In providing this service the server object has access to two hashtables.



and two storage (buffer) areas.

When the server object receives a lookup request, it uses the passed integer to lookup the ip address in table 1. Suppose this is done with method m1(). It then stores this retrieved ip address in buffer 1. It then goes off and proceeeds to use this ip address to retrive the name of the computer from table 2, using say m2(). It then stores the retrieved computer name in buffer 2. Finally it retrieves both strings from the buffers and returns to the client object a concatenated string representing the details of the request computer.

Suppose now that the second search, m2(), is a lengthy process. While m2() is being executed another client comes along with another request. The search table 1 method, m1(), of the server executes and places the found ip address in buffer 1. When m2() returns it places the retrieved computer name in buffer 2. Clearly the contents of the two buffers are now at variance. If this situation is allowed to continue things just become chaotic as illustrated in the following outputs from an unsynchronised run of the system ManyClientsRMI. First of all compile the system with the getDetails() method of the NameGetter class set as not synchronised. ie with the signature of

public String getDetails(int n) throws RemoteException

and then run the system by using 5 command line windows

1. for the rmi compiler - rmic NameGetter
2. for the rmi registry - rmiregistry
3. for the server - java LookUpServer
4. for client number 1 - java LookUpClient 1
5. for client number 2 - java LookUpClient 2
6. (and more clients if you wish...)

Output should look something like the following.

Client Number 1	Client Number 2
Client 1 : Looking up details of computer number 2 123.4.5.4 : Earth ERROR -----> wrong computer retrieved... Client 1 : Looking up details of computer number 5 123.4.5.4 : Uranus ERROR -----> wrong computer retrieved... Client 1 : Looking up details of computer number 4 123.4.5.2 : Jupiter ERROR -----> wrong computer retrieved... Client 1 : Looking up details of computer number 2 123.4.5.3 : Earth ERROR -----> wrong computer retrieved... Client 1 : Looking up details of computer number 2 123.4.5.2 : Earth Client 1 : Looking up details of computer number 1 123.4.5.1 : Venus Client 1 : Looking up details of computer number 3 123.4.5.2 : Saturn ERROR -----> wrong computer retrieved... Client 1 : Looking up details of computer number 5 123.4.5.6 : Uranus ERROR -----> wrong computer retrieved... Client 1 : Looking up details of computer number 3 123.4.5.3 : Saturn Client 1 : Looking up details of computer number 0 123.4.5.0 : Mars Client 1 : Looking up details of computer number 1 123.4.5.1 : Venus Client 1 : Looking up details of computer number 5	Client 2 : Looking up details of computer number 1 123.4.5.0 : Venus ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 1 123.4.5.3 : Venus ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 2 123.4.5.0 : Earth ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 0 123.4.5.6 : Mars ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 2 123.4.5.2 : Earth Client 2 : Looking up details of computer number 4 123.4.5.5 : Jupiter ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 4 123.4.5.4 : Jupiter Client 2 : Looking up details of computer number 2 123.4.5.2 : Earth Client 2 : Looking up details of computer number 3 123.4.5.2 : Saturn ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 2 123.4.5.1 : Earth ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 1 123.4.5.3 : Venus ERROR -----> wrong computer retrieved... Client 2 : Looking up details of computer number 2 123.4.5.5 : Earth ERROR -----> wrong computer retrieved...

You should now change the method signature of the getDetails() method to

public synchronized String getDetails(int n) throws RemoteException

and repeat the exercise. This time the two clients should work together in synchronisation and the output should be consistent.

4.9 FAQs, Hints, Tips and TroubleShooting

4.9.1 How does it all work

Conceptually this is how it works,

- the client calls the Naming.lookUp() method
- the server sends a copy of the stub class file to the client.
- a stub object is created at the client
- the client application posts a message to this stub object
- the stub object marshalls the message and using the network layer sends the message to the server
- at the server side the skel object takes the message from the network layer
- the skel object marshalls the message and sends it to the 'real' remote object.
- the remote object sends the reply back to the skel object.
- the skel object marshalls the reply and uses the network layer to send the reply to the stub object on the client
- the stub object marshalls the reply and passes it on to the client object.

4.9.2 My _Stub and _Skel files keep disappearing – why

One of the problems with RMI development is that as we develop/compile/recompile the system classes over and over again then this re-compilation process destroys the _Stub and _Skel. Hence you will have to use/reuse rmic after each system recompilation..

If you use one of the IDEs (such as jBuilder) you should make sure that the compile before run option is off. Otherwise when you run the registration program later you may want to recompile the RemoteStringInterface and the RemoteStringProcessor classes and consequently destroy the _Stub and _Skel classes.

4.9.3 The rmi compiler complains it can't find a class

Its almost certainly something to do with your classpath setting and package name of your class. When you use RMIC make sure you are in the correct base directory and make sure that you use the fully qualified package name for the target class

4.9.4 I get a java.rmi.NotBoundException error

Make sure the server is running before you run the client. The remote object may not have been registered with the name server – ie with rmiregistry

The binding code may not have been executed eg

```
Naming.rebind("rmi://127.0.0.1/" + objName,extStr);
```

4.9.5 I get an Exception in thread "main" java.lang.NoClassDefFoundError error

Check you classpath – the JVM cannot find your class.

4.9.6 The permissions file cannot be found

Make sure you use the full path for the permissions file when setting the security,
-Djava.security.policy=d:\dump\server\permissions.txt

4.9.7 I got the following error message from JBuilder

Problem with server java.rmi.ConnectException: Connection refused to host: 192.168.0.1; nested exception is:

java.net.ConnectException: Connection refused: no further information

Cause : Its possible that the rmiregistry has not been started

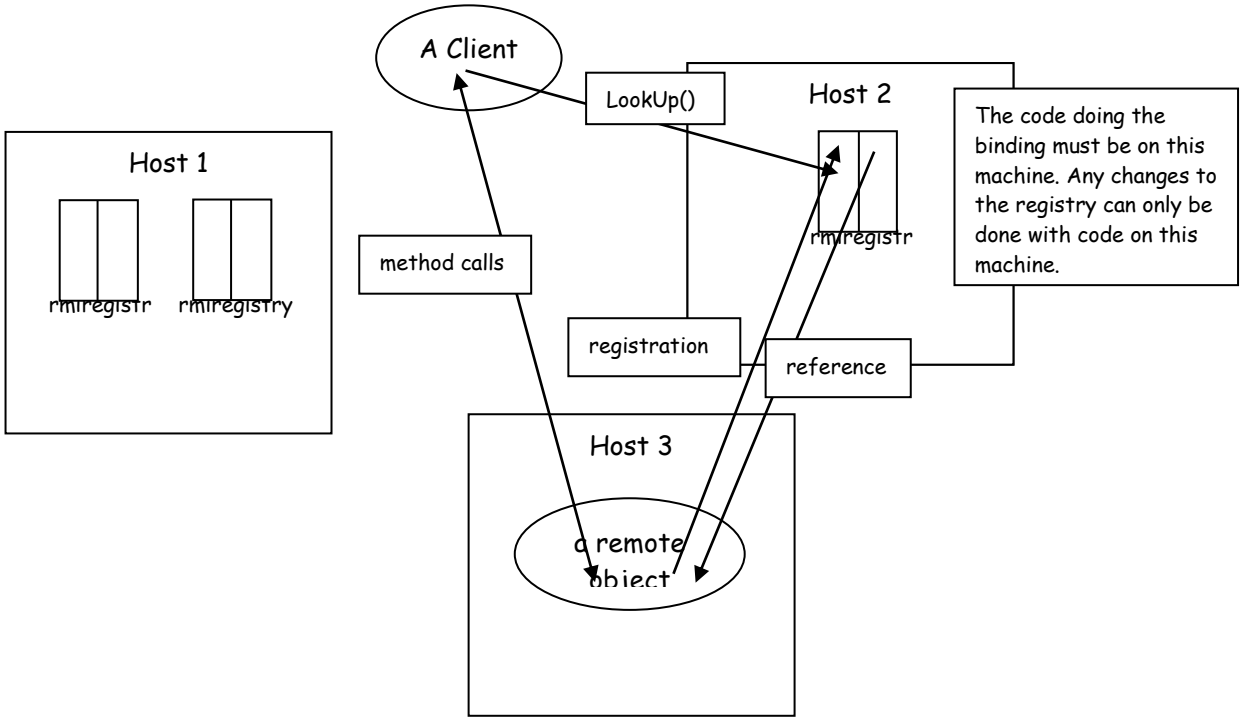
4.9.8 RMI References

An excellent reference is...
http://java.sun.com/docs/books/tutorial/rmi/

4.10 Some Extras

You can have more that one instance of the RMIREGISTRY running on the same machine. When the naming server is started (ie instanced) simply allocate it to a different port

rmiregistry 1099
rmiregistry 2099
rmiregistry 3099
etc...



Changes to the registry on a host can only be made by an application running on that host. This means a client can read the registry but it cannot write to the registry and hence corrupt any entries in the registry. For H3 to register an object with the registry on H2 the actual registration must take place on H2. This means the Naming.rebind() call must be made on H2. That is, the program doing the binding, must be on the same machine (ie in the same JVM) as the registry.

4.11 An Example of a Possible System

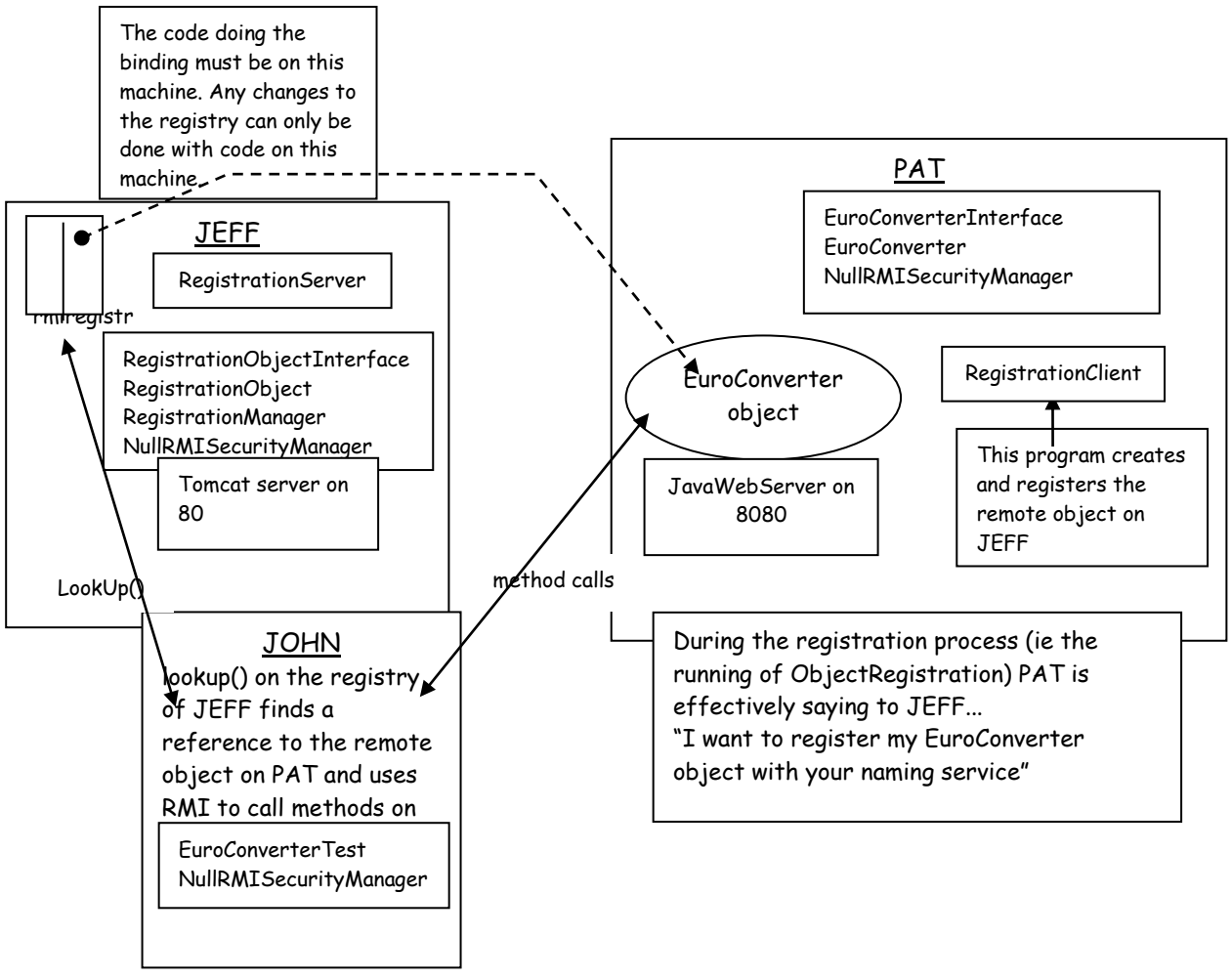
(Currently located in the ThreeWayLink under JavaExamples)

The scenario is...
- the naming service resides on computer JEFF
- the remote object resides on computer PAT
- computer PAT has previously registered the remote object with the naming service on computer JEFF
- a client on computer JOHN wants to access the remote object but doesn't know where the remote object is
- so the client first accesses the naming service on computer JEFF to obtain a reference to the remote object
- the client can then send messages to the remote object on computer PAT

Any server wishing to register a remote object with the registry on JEFF can only do so by using the registration software on JEFF. Servers wishing to register objects have to obtain a reference to the Register object on JEFF using RMI and use the remote object on JEFF to register their own remote object(s). JOHN is a client wishing to obtain services from the remote object on PAT.

Note the need to have a web server running on both JEFF and PAT. This is so that the RMI _stub files can be transferred from JEFF → PAT and from PAT → JOHN.

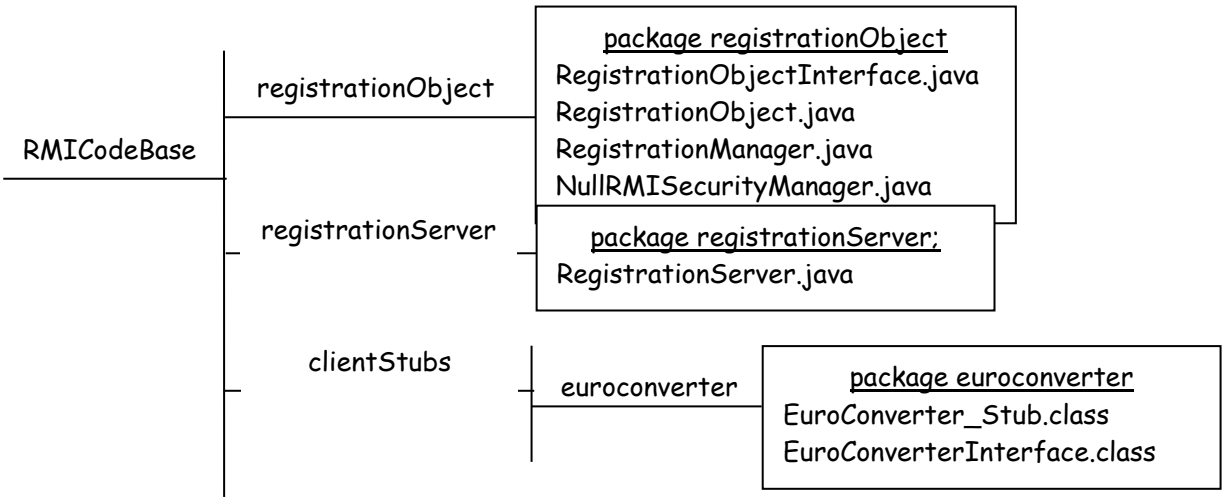
In what follows below I was running Tomcat as the web server on JEFF and JavaWebServer2,0 as the web server on PAT.



On JEFF

There are two main software components on JEFF.

1. The software related to the remote object (RegistrationObject) used by clients to register their own remote objects with the naming server on JEFF. This software is located in the directory registrationObject.
2. The server software - RegistrationServer. This software allows the client (PAT) to log onto JEFF and then to transfer some classes to support PAT's remote object when it is registered with the naming server on JEFF. This software is located in the directory registrationServer. The client software that communicates with this server is of course located on PAT. It logs onto JEFF, transfers the class files to the RegistrationServer on JEFF, which then places the class files in the directory clientStubs. The client on PAT then invokes the remote registrationObject on JEFF to carry out the registration of its own remote object with the naming server on JEFF.



- install the registration object component in tomcat\webapps\ROOT\MyWebSite\RMICodeBase\registrationObject
- install the RegistrationServer component in tomcat\webapps\ROOT\MyWebSite\RMICodeBase\registrationServer
- compile both components
- form stub + skel files for the registration software with rmic registrationObject.RegistrationObject from the RMICodeBase directory
- start the RMI naming service (rmiregistry)
- register the RegistrationObject object with the naming server by running RunObjectManager.bat from directory ..\RMICodeBase
- start the RegistrationServer with RunTCPServer.bat
- start the Tomcat web server

```
RunObjectManager.bat
java registrationObject.RegistrationManager
```

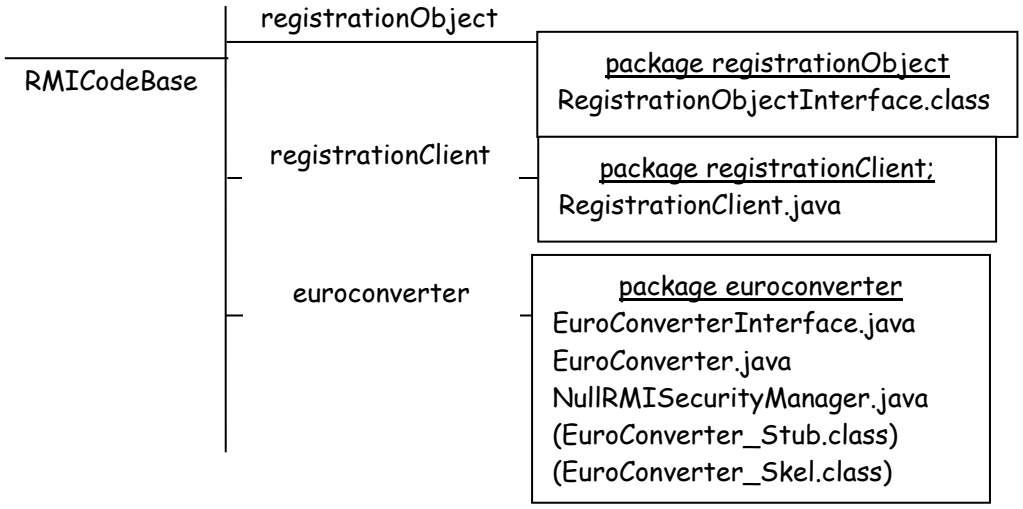
```
RunTCPServer.bat
java registrationServer.RegistrationServer
```

On PAT

There are two main software components on PAT.

1. The software related to PATs remote object (EuroConverter). This software is located in the directory registrationObject.
2. The client software - RegistrationClient. This software allows the client (PAT) to log onto JEFF and then to transfer some classes to support PAT's remote object when it is registered with the naming server on JEFF. This software is located in the directory registrationClient. It logs onto JEFF, transfers the class

files to the RegistrationServer on JEFF, which then places the class files in the directory clientStubs. It then then invokes the remote registrationObject on JEFF to carry out the registration of its remote object - EuroConverter - with the naming server on JEFF.



- install the remote object (EuroConverter) component in... JavaWebServer\public_html\MyWebSite\RMICodeBase\euroconverter
- install the RegistrationClient component in JavaWebServer\public_html\MyWebSite\RMICodeBase\registrationClient
- compile both components; the compilation of the remote object component will need access to RegistrationObjectInterface.class so a copy from JEFF must be available and placed in the directory ..\RMICodeBase\registrationObject
- form stub + skel files for the EuroConverter object with rmic euroconverter.EuroConverter from the RMICodeBase directory
- run the client software, RegistrationClient, to copy EuroConverter_Stub.class and EuroConverterInterface.class to the server and to register the EuroConverter remote object with the naming service on JEFF - use RunRegistrationClient.bat to do this.

RunRegistrationClient.bat

```
java -Djava.rmi.server.codebase=http://192.168.0.1:80/MyWebSite/RMICodeBase/^
registrationClient.RegistrationClient^192.168.0.1^6001^RegistrationObject^EuroConverter^
euroconverter^euroconverter\EuroConverter_Stub.class^
euroconverter\EuroConverterInterface.class
```

note : ^ represents a space... the above is considered to be all one line.

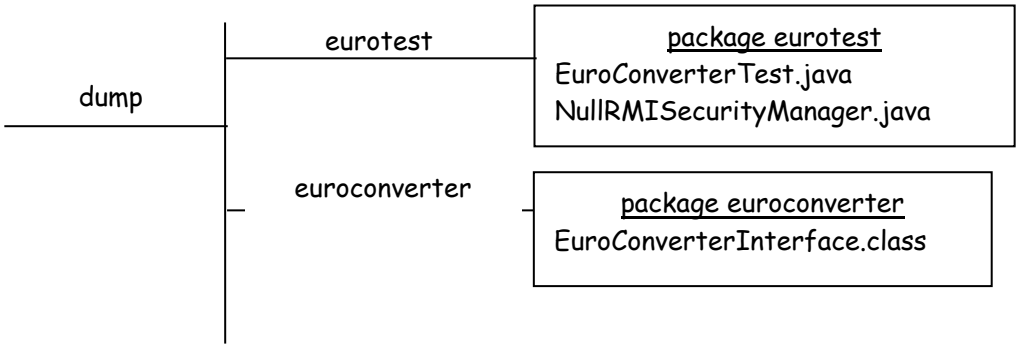
There are seven arguments passed to RegistrationClient. They are...
1. 192.168.0.1 - the IP address of the registration server (JEFF)
2. 6001 - the port
3. RegistrationObject - the name of the remote object on JEFF
4. EuroConverter - PATs remote object name
5. euroconverter - the path from RMICodeBase to the remote object on PAT
6. euroconverter\EuroConverter_Stub.class - a required file for JEFF to form the stub class for PATs remote object. Client that wish to access PATs remote object will be sent this when they access the naming server on JEFF looking for PATS remote object
7. euroconverter\EuroConverterInterface.class - as previous

- the EuroConverter remote object is now registered with the naming server on JEFF and is running on PAT
- start JavaWebServer

RegistrationClient does the following
- logs onto the RegistrationServer software at JEFF and copies the two class files EuroConvert_Stub.class and EuroConverterInterface.class
  from the client (PAT) to the server (JEFF), where the server then stores the two class files in the directory clientStubs.
- accesses the remote object at JEFF, RegistrationObject, and uses the registerObject() method to register the EuroConverter object with the naming server on JEFF.

On JOHN
- install the EuroConverter test software in some suitable directory eg c:\dump\eurotest
- compile the system; the compilation will need access to EuroConverterInterface.class from PAT so a copy must be available and placed in the directory c:\dump\euroconverter
- run the EuroConverterTest program with RunClient.bat from the c:\dump directory

RunClient.bat
```
java -Djava.rmi.server.codebase=http://192.168.0.2:8080/MyWebSite/RMICodeBase/
eurotest.EuroConverterTest
```
- 192.168.0.2 is the address of PAT (...where the remote EuroConverter object resides)



4.12 A Run Over The Internet

- I logged into my ISP and then used ipconfig from a command window to find the allocated session ip address for my machine JEFF.

4.13 Security Issues

The archetypal bad case is something like the following:

A hacker has written a program that scans the Internet looking for RMI registries. It does this by simply trying to connect to every port on every machine it finds. Whenever the program finds a running RMI registry, the program immediately uses the list() method to find the names of all the servers running on the registry. After which, the program calls rebind() and replaces each stub in the registry with a stub that points to his server.

The point: if you don't restrict access to a naming service, then your network becomes incredibly vulnerable. Even if each individual server is secure (e.g., each individual server requires the clients to log in), the naming service itself is a vulnerable point and needs to be protected.

The solution the RMI registry adopted was quite simple: any call that binds a server into the registry must originate from a process that runs on the same machine as the registry. This doesn't prevent hackers from finding out which servers are running, or calling methods on a given server, but it does prevent them from

replacing any of the servers, and thus prevents them from altering the structure of client-server applications, which depend on the registry.

The application (or just code) that creates and manages the remote object does not have to be on the same machine (ie in the same JVM) as the naming server - rmiregistry. But any software that does the binding etc of the remote object to the registry has to be on the same machine (ie in the same JVM) as the rmiregistry. This means that the remote object does not have to be on the same machine as the rmiregistry.

4.14 An Extract from the Excellent O'Reilly Book

This excellent description of the steps involved in implementing RMI is take from the first class series of O'Reilly books – Java Examples in a Nutshell

To develop a RMI-based application, you need to follow these steps:

- Create an interface that extends the java.rmi.Remote interface. This interface defines the exported methods that the remote object implements (i.e., the methods the server implements and that clients can invoke remotely). Each method in this interface must be declared to throw a java.rmi.RemoteException, which is the superclass of many more specific RMI exception classes. Every remote method must declare that it can throw a RemoteException because there are quite a few things that can go wrong during the remote method invocation process over a network.

- Define a subclass of java.rmi.server.UnicastRemoteObject that implements your Remote interface. This class represents the remote object (or "server" object). Other than declaring its remote methods to throw RemoteException objects, the remote object does not need to do anything special to allow its methods to be invoked remotely. The Uni castRemoteObject and the rest of the RMI infrastructure handle this automatically.

- Write a program (a "server") that creates an instance of your remote object. Export the object, making it available for use by clients, by registering the object by name with a registry service. This is usually done with the java.rmi.Naming class and the rmiregistry program. A server program may also act as its own registry server by using the LocateRegistry class and Registry interface of the java.rmi .registry package

- After you compile the server program with javac, use rmic to generate a "stub" and a "skeleton" for the remote object. With RMI, the client and server do not communicate directly. On the client side, the client's reference to a remote object is implemented as an instance of a "stub" class. When the client invokes a remote method, it is a method of this stub object that is actually called. The stub does the necessary networking to pass that invocation to a "skeleton" class on the server. This skeleton translates the networked request into a method invocation on the server object, and passes the return value back to the stub, which passes it back to the client. This can be a complicated system, but fortunately, application programmers never have to think about stubs and skeletons; they are generated automatically by the rmic tool. Invoke rmic with the name of the remote object class (not the interface) on the command line. It creates and compiles two new classes with the suffixes _Stub and _Skel. If the server uses the default registry service provided by the Naming class, you must run the registry server, if it is not already running. You can run the registry server by invoking the rmiregistry program.

- Now you can write a client program to use the remote object exported by the server. The client must first obtain a reference to the remote object by using the Naming class to look up the object by name; the name is typically an rmi:URL. The remote reference that is returned is an instance of the Remote interface for the object (or more specifically, a "stub" object for the remote object). Once the client has this remote object, it can invoke methods on it exactly as it would invoke the methods of a local object. The only thing that it must be aware of is that all remote methods can throw RemoteException objects, and that in the presence of network errors, this can happen at unexpected times.

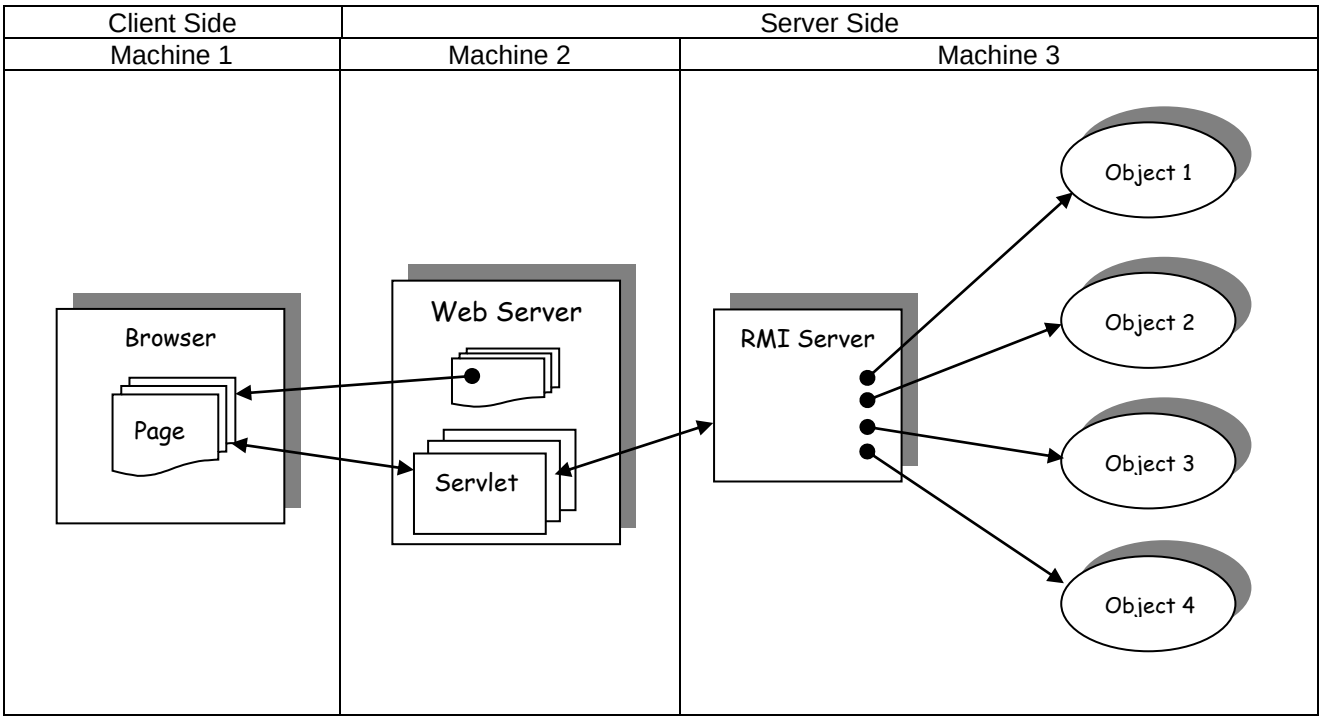
- RMI uses the serialization mechanism to transfer the stub object from the server to the client. Because a client may load an untrusted stub object, it should have a security manager installed to prevent a malicious (or just buggy) stub from deleting files or otherwise causing harm. The MISecurityManager class is a suitable security manager that all RMI clients should install.

- Finally, start up the server program, and run the client!

The three-tier application must have a communications layer between the business module and the user interface for the design to function properly. This usually means that an application level protocol must be defined; this perpetuates cumbersome design that leads to errors. With the addition of RMI, the application-level protocol goes away and an easier implementation may be used. Until recently, the three-tier model couldn't be combined easily with object-oriented design. However, with the addition of RMI to the Java package, a truly object-oriented distributed architecture design is available.

The RMI package allows Java developers to design three-tier applications without the worries of building a custom communications layer. In addition, the computing power that becomes necessary for large-scale applications may be moved to a server that is capable of handling the requirements. The user interface may be moved to a workstation that is equipped with less hardware, thus creating a thinner client. The RMI model brings the three-tier architecture to a more powerful state, and allows the computational requirements of an application to be placed on the server instead of the client. The client then becomes free to operate on more important tasks rather than take up the user's time by making computations that it may not be readily equipped to handle.

RMI in a Distributed System



5 JAVA NATIVE INTERFACE, LEGACY SOFTWARE AND RMI

Before we look at how we can get the JNI to work with RMI so that a remote object can be made to interface with some legacy software, I will provide a simple example of how the JNI works.

5.1 The Java Native Interface

The Java Native Interface (JNI) is the link between a Java application and native code. It allows you to invoke native methods from a Java application or applet, pass arguments to these native methods and use the results returned by the methods.

The JNI provides the capability to access native methods through shared dynamic link libraries, referred to as DLLs on Windows systems. The JNI allows Java programs to invoke the native method, pass arguments to the native method, and receive the results returned by the native method. The JNI also provides programmers with the capability to include the JVM in non-Java applications.

5.2 An Example of Accessing a C++ Native Method

The following example will demonstrate how we can use the JNI to access a C++ function from a Java application. Lets look at the C++ function first. It's a very simple function that takes an int argument and returns the argument doubled. Nothing fancy because it's the Java/C++ link that I am trying to demonstrate, not the complexity of the native code.

```
int doublit(int n) {
    return 2*n;
}
```

5.2.1 The Java Side of Things

On the Java side of things I need to develop a class where I can declare a method that will act as an alias for the C++ function doublit(). Here it is,

```
public class Alias {
    // doubler() is a Java alias for the C++ function doublit() which is the
    // native method we are really calling
    public native int doubler(int n);
    static {
        // The name of the target dll (legacy.dll) is the argument to loadLibrary()
        System.loadLibrary("legacy");
    }
}
```

The method doubler() is the Java alias for the C++ function doublit(). The Alias class is the class used to declare the doubler() method, Note how the doubler() method is declared. The method signature includes the native keyword and the body of the method, is replaced by a semicolon. You may recall that this is similar to the way in which we declare abstract methods in Java. Incidently, the choice of Alias for the class name is mine. There is no JNI requirement for it to be called this.

In addition, the loadLibrary() method of the System class is invoked to load the shared library file specified by the string "legacy". In Win9x and Windows NT, this causes the legacy.dll dynamic link library to be loaded. This dll will be produced as part of the C++ side development, later in these notes. The loadLibrary() method is invoked as part of a static initializer of the Alias class. A static initializer is used so that the library is loaded only once, when the Alias class itself is loaded. The legacy.dll will be produced when we compile the C++ code later in these notes.

Now that you've declared the method to be native, where is its actual implementation? The method's implementation will be included in a native library. It is the duty of the class in which this method is a member of, to invoke the library, so that its implementation becomes available to whoever needs it. The easiest way to have the class invoke the library is to add the following to the class,
static

```
{
  System.loadLibrary (nameOfLibrary);
}
```

A static code block is always executed once when the class is first loaded. You can include virtually anything in a static code block; however, loading libraries is the most common use for it. If, for some reason, the library fails to load, an UnsatisfiedLineError exception will be thrown once a method from that library is called. If the library loads fine, the JVM will add the correct extension to its name (.dll in Windows, and .so in UNIX).

In addition to the Alias class I will also develop a Java application, JNIClient.java that calls the C++ function. Here is the very simple class,

```
public class JNIClient {
    public static void main(String args[]) {
        int n = 3;
        int answer = new Alias().doubler(n);
        System.out.println("answer = " + answer);
    }
}
```

Notice that I do not refer to the C++ function directly but instead I use the Java alias method doubler().

The final thing to do on the client side is to use javah.exe to produce the C++ header file that will act as an interface between the Java aplication and the C++ native method.

You can create the header file with something like,

```
C:\jdk1A.l\jdg\ch59>javah -jni Alias
```

This causes the Alias.h file to be created. It is a C++ language header file. Heed the warning of the first line of the file,

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Alias */
#ifndef _Included_Alias
#define _Included_Alias
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Alias
 * Method:    doubler
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_Alias_doubler(JNIEnv *, jobject, jint);
#ifdef __cplusplus
}
#endif
#endif
```

The method signature parameters function as follows:

- JNIEnv *: A pointer to the JNI environment. This pointer is a handle to the current thread in the Java virtual machine, and contains mapping and other hosuekeeping information.
- jobject: A reference to the method that called this native code. If the calling method is static, this parameter would be type jclass instead of jobject.
- jint: The parameter supplied to the native method.

5.2.2 The C++ Side of Things - Implementing C++ Native Method

On the C++ side, I developed the following C++code,

```
//---------------------------------------------------------------------------
#include <jni.h>
#include "Alias.h"
//---------------------------------------------------------------------------
int doublit(int n) {
    return 2*n;
}

JNIEXPORT jint JNICALL
Java_Alias_doubler(JNIEnv *env, jobject obj, jint n) {
    return doublit(n);
}
// In a way… doubler() is a Java alias for the C++ function doublit()
//---------------------------------------------------------------------------
```

The file begins with the following two include statements:

```
#include <jni.h>
#include "Alias.h"
```

The first line includes the jni.h header file that can be found as ..\jdk1.3\include. This file contains a number of type and function definitions.
The second line includes the Alias.h header file that was generated in the Java side of things.

Following the include statements is the C++ language doublit() function which simple returne twice its argument.

```
int doublit(int n) {
    return 2*n;
}
```

And finally, we include an implementation for Java_Alias_doubler()

```
JNIEXPORT jint JNICALL
Java_Alias_doubler(JNIEnv *env, jobject obj, jint n) {
    return doublit(n);
}
```

The first parameter for every native method is a JNIEnv interface pointer. It is through this pointer that your native code accesses parameters and objects passed to it from the Java application. The second parameter is jobject, which references the current object itself. In a sense, you can think of the jobject parameter as the "this" variable in Java. For a native instance method, such as the displayHelloWorld method in our example, the jobject argument is a reference to the current instance of the object. For native class methods, this argument would be a reference to the method's Java class. Our example ignores both parameters.

This method is defined in the same manner as in the Alias.h header file. All it does is retrive the parameter n and pass it to the C++ function and then return the result to the calling Java code.

All that remains now is to compile the code so that it produces a shared library, ie a dynamic likn library if you are working with Win9x or NT. I used C++ Builder to develop the C++ code and the DLL option from the linker page. The output from my efforts was the file legacy.dll.

5.2.3 Running the System

To test the system all I had to do was to execute my JNICLient class.
To run the example, the Java virtual machine needs to be able to find the native library. Perhaps the easiest way to do this is to copy the dll into the same directory as the java program.

5.3 Running A C++ Program From a Java Program

Whilst I was developing the previous example it occurred to me what would happen if the C++ function we are aliasing happened to be the main() function. If we can execute the native main() function then clearly we can execute the C++ program from within the Java program. It turns out that I could do this. This is how I achieved it.
In my existing C++ program I changed the main() method to become the start() method and I then added the JNI directive to the existing C++ program. My C++ program finished up as follows,

```
#include <stdio.h>
#include <conio.h>
#include <jni.h>
#include "Alias.h"
//------------------- Function Prototypes ----------------------
    int func(int &, int &);
//--------------------------------------------------------------
void start() { // This was originally the main() function…
    int x = 7, y = 8, FuncVal;
    clrscr();
    printf("\nBefore call... x = %d, y = %d",x,y);
    FuncVal = func(x,y);
    printf("\nAfter call... x = %d, y = %d",x,y);
    printf("\nFunction Value = %d",FuncVal);
    getch();
}

int func(int &p, int &q) {
    p = 2 * p; q = 3* q;
    printf("\nIn function... p = %d, q = %d",p,q);
    return(p * q);
}

//---------------------------------------------------------------------------
JNIEXPORT void JNICALL
Java_Alias_cppMain(JNIEnv *env, jobject obj) {
    start();
}
//---------------------------------------------------------------------------
```

I called the program CppNative.cpp and used the C++ compiler to produce the dll, CppNative.dll.

I then produced the Java side classes

```
public class Alias {
  // cppMain() is a Java alias for the C++ main() function which is the
  // native method we are really calling.
  public native void cppMain();
  static {
    // The name of the target dll (CppNative.dll) is the argument to
loadLibrary()
    System.loadLibrary("CppNative");
  }
}
```

```
public class JNIClient {
  public static void main(String args[]) {
    new Alias().cppMain();
  }
}
```

and used javah to produce the following header file

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Alias */
#ifndef _Included_Alias
```

```
#define _Included_Alias
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Alias
 * Method:    cppMain
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_Alias_cppMain
  (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

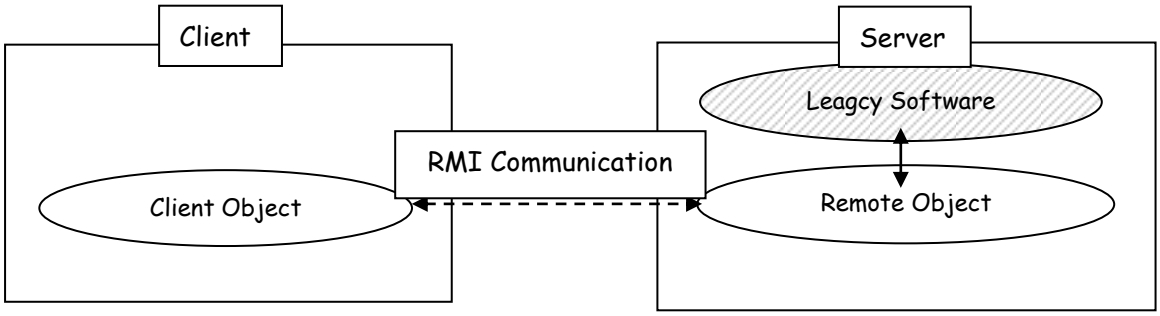
When I ran the JNIClient program the Alias object responded to the cppMain() method by executing the corresponding C++ function – the start() function – which effectively executed the C++ program.

I have used the word, program, in a rather loose sense in the above explantion, What I have really done is to run the dll library files CppNative... but it effecitively amounts to the same thing.

5.4 Legacy Software

Many commercial companies have a huge portfolio of software that has been in use for some time. Such legacy software cannot be instantly rewritten: often the software is the result of many tens of years of design and programming effort and many companies are keen to keep this software going, even when they migrate many of their applications to the Internet. Often a server will host this legacy software. A company that wishes to integrate such software with, say, a Web-based customer ordering facility faces a number of problems, for example the language used to develop the ordering application would certainly not be Java.

One solution to this problem involves a combination of RMI and JNI.



In the architecture I have outlined it's the client object that requires access to the legacy software. The remote server object – on the same machine as the legacy software – acts as a proxy for the client object. It receives messages/request from the client object, passes these on to the legacy software and then transmits the results back to the client object.

A java object on the server is consulted to access the native C++ code. The object then returns the results of its consultationto the calling object on the client.

6 SOAP

What is SOAP?
The Simple Object Access Protocol (SOAP) is an open XML based protocol allowing applications to communicate with each other over the Internet. Because SOAP is entirely XML based it is language, vendor and platform independent, and as it is designed to run on top of HTTP it can penetrate the firewalls of servers, which are frequently configured to reject all requests originating from the Internet not that are not HTTP based (usually on port 80) [1]. Effectively SOAP wraps new or existing software, running on any platform, in XML and makes them accessible over the Internet. SOAP is not object orientated itself although

it will work with both applications and individual objects (such as Java classes) written in object-orientated languages.
(
When a client wishes to invoke a method on a remote server it issues a remote procedure call (RPC) it places a representation of the method into an XML "Envelope" known as a payload to which are added the HTTP headers needed to send it to the server. The server will unpack the method and execute it on the target application (which must be registered with the server) and then package the resulting answer as an XML payload returned using HTTP. Essentially any software that is capable of exchanging data using the Internet and that can parse XML may be a SOAP client
(
There must also be code on the server that is responsible for understanding the SOAP request, invoking the specified method, building the response message, and returning it to the client. Applications or objects must register the methods they wish to expose with the server so that they may be invoked. Under the SOAP specification these implementation details are left to the developer. The following section on using SOAP will look at how this is achieved by Apache for Java
(

How SOAP is used in a distributed system.
This section looks at how SOAP is actually used in a distributed system. It will focus on how the Apache organisation implements its SOAP server as an extension of its Tomcat server, which runs Java servlets and Java Server Pages. The code used here will utilise the Apache SOAP toolkit that provides the classes required by the client to send and receive SOAP messages.

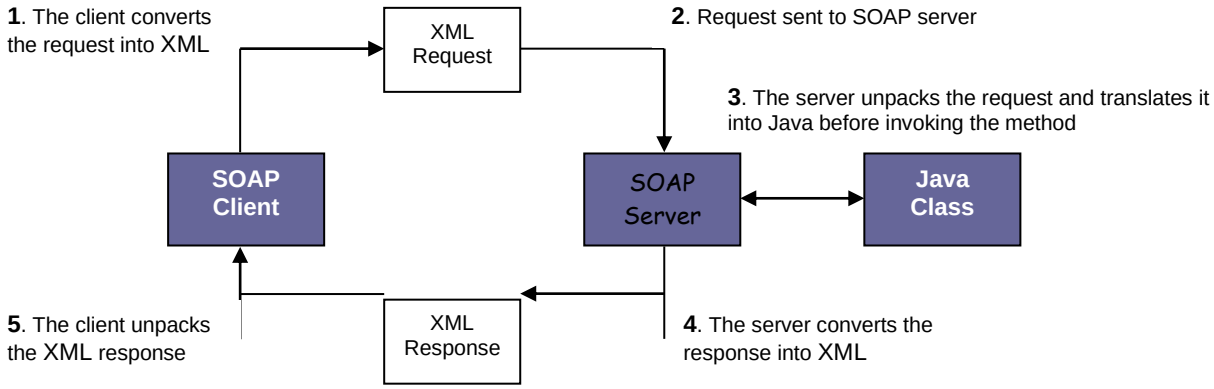


Fig 1. How a remote procedure call is made to a Java class using SOAP [2].
This section will look at how the method getQuote of the Java class StockQuoteService, which returns the current value of a given stock, may be accessed as a SOAP remote procedure. The outline code for StockQuoteService is [3]:

```
public class StockQuoteService {

  public float getQuote (String symbol) throws Exception {

      // code to find and return the value of a given stock

  }
}
```
(

Note that this class requires no additional code or compilation to allow methods to be called remotely, in fact all that needs to be done in order to expose any methods to make available as remote procedures is to register the class and methods with the server. Once the Tomcat server has been started (in this case receiving HTTP requests on port 8080) and the Apache SOAP server is running, this is done using a deployment descriptor written in XML:
(

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
```

```
            id="urn:m879-soapsample-quotes">
   <isd:provider type="java"
            scope="Application"
            methods="getQuote">
      <isd:java class="samples.stockquote.StockQuoteService"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
</isd:service>
```

(
Here the tag service provides a unique resource name (urn) with which clients can identify the service on the server. The tag provider details the methods (just one here) exposed by the service and the location of the class that provides them. The faultListener is a listener to which the server reposts any faults that occur (DOMFaultListener is the default). The deployment descriptor is registered with the server using the ServiceClientManager included in the toolkit:

java ServiceClientManager http://localhost:8080/soap/servlet/rpcrouter deploy dd.xml

(
This will invoke the deploy method of the servlet rpcrouter running on the Tomcat server which will register the service described by the deployment descriptor dd.xml. rpcrouter is the servlet provided by the Apache SOAP server, which is used to make the local Java call to StockQuoteService when the getQuote method is requested.

To call this remote method the client code looks like this:

```
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class QuoteFinder {

   // Create a call to the remote getQuote method
   // of the service m879-soapsample-quotes.
   Call call = new Call ();
   call.setTargetObjectURI ("urn:m879-soapsample-quotes");
   call.setMethodName ("getQuote");

// Set the parameter required by the getQuote method to find the value // of IBM's stock
   Vector params = new Vector ();
   params.addElement (new Parameter("symbol", String.class, "IBM", null));
   call.setParams (params);

   // Call the remote method
   Response resp = call.invoke
 ("http://localhost:8080/soap/servlet/rpcrouter", "");

   // Check the response.
   if (resp.generatedFault ()) {
     Fault fault = resp.getFault ();
     System.out.println ("Fault Code   = " + fault.getFaultCode ());
     System.out.println ("Fault String = " + fault.getFaultString ());
   } else {
     Parameter result = resp.getReturnValue ();
     System.out.println (result.getValue ());
   }
 }
}
```
(

I think this code is pretty much self-explanatory. The classes Call, Response and Fault are found in the packages supplied with the Apache SOAP server and write and parse the XML used in the messages automatically.
(
This is the request message produced by the client containing the HTTP message with the SOAP XML message as the payload [4]:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "urn:m879-soapsample-quotes"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:getQuote xmlns:m="urn:m879-soapsample-quotes">
         <symbol>IBM</symbol>
      </m:getQuote>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

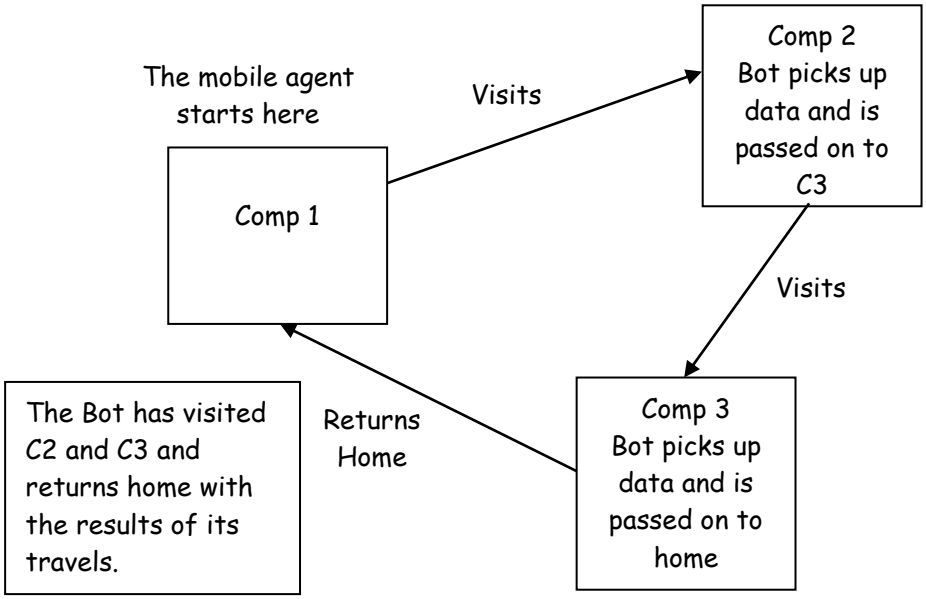
And this is the response message:
(

```
    HTTP/1.1 200 OK
    Content-Type: text/xml; charset="utf-8"
    Content-Length: nnnn

    <SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:getQuote xmlns:m="urn:m879-soapsample-quotes">
         <Price>34.5</Price>
      </m:getQuote>
   </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

(
The advantages of SOAP compared to CORBA and RMI
As already mentioned as it piggybacks HTTP, SOAP messages can access servers over the Internet without problems with firewalls, which hampers CORBA and RMI servers that run on non-HTTP ports. This allows SOAP clients to access remote methods on widely distributed systems and not just on intranets as its rivals are and so this makes it an obvious choice for business-to-business applications. Also because SOAP is run over HTTP it can take advantage of Secure Socket Layers to provide secure encryption of data sent over the Internet with out the need for additional features to be added to SOAP itself adding a level of security not available its competitors [4].
(
Like RMI but unlike CORBA, SOAP is a simple, lightweight way of providing distributed computing. Like CORBA but unlike RMI, SOAP works with any language. Because the SOAP protocol has been kept simple implementing language bindings between SOAP and software developed in a given language should not be a major effort [5].
(
SOAP does not require stubs and skeletons to be generated and so there is no need for the special description languages compilation of code to produce these, and no need to distribute stubs to the client as is true of both RMI and CORBA. The client computer does not need to use any kind of additional software at all other than that contained in the client code itself. In fact unlike with RMI and CORBA, SOAP clients need not even be conventional programs as long as they can send and receive the HTTP/XML messages.

(
Finally because it is based on XML, SOAP messages are extensible both in terms of the protocol itself and in the data types used in data transfer without these affecting earlier implementations of the protocol.

7 MOBILE AGENTS (BOTS)



The Bot travels around all registerd travel agents looking for any last minute flights to a particular destination. It reports back its findings.

- both C2 and C3 run the same software (the server software) but use different databases
- both C2 and C3 must agree to accept an agent and the to 'process' the agent when it arrives
- C2 and C3 may be regarded as 'servers' in the rmi architecture