



Passport to Intranet architectures and performances

November 98

A report by
TechMetrix Research

Foreword

The following pages are fact-orientated, rich in diagrams and detailed measures. Deliberately, the synthesis does not deliver a verdict: which is the best architecture offering the best compromise between performance and deployment constraints?

Nevertheless, thanks to the numerous measures, which illustrate this document, the facts are clear. If one wants to seize all of the opportunities offered by Internet technologies, the choice is clear: the HTML/HTTP combination.

Indeed, this rustic middleware, too often dismissed as a second-rate patch up, appears to be the only one that is really " all-terrain ". Not only is HTML+HTTP the only solution already available on the market which enables to reach the " deployment 0 " target, but furthermore, its network performance is homogeneous and predictable. Thus, in our various tests, HTML+HTTP never came in last in its group, but came in first twice.

Many lessons were learned during this campaign of tests, the first one being that there is a significant gap between the claims of the major actors (Microsoft, Sun, etc.) and the technical reality, as tested on a real network.

This is how we realized in concrete terms that the ActiveX-DCOM solution did not offer any real advantage (significant deployment constraints, even in a " pure " Microsoft context, for mediocre network performance). Not as optimized as the Java+Corba/IOP solution, the Java-RMI solution revealed itself disappointing

In the end, among the " object middlewares ", the best choice seems to be Corba/IOP. However, implementation is complex and distances it from the simplicity of the winning pair HTML+HTTP.

Lastly, the biggest surprise comes from the " traditional " client-server solution, which shows quite a decent level of performance, depending on the cases tested... Here, too, straightforwardness of operation plays its part (but not at the deployment level, where the constraints are the highest).

In short, the choice is simple: if you have to address a large group of users, favor the " pure " Web solution (HTML+HTTP). If you only have to satisfy a small group of users who express strong requirements in terms of user interface, you might still prefer the traditional client-server solutions, with which you will find the best guarantee of result...

Jean-Christophe Cimetiere (jcc@techmetrix.com) - CEO of TechMetrix Research.

Contents

- 1. INTRODUCTION..... 7**
- 2. INTRANET ARCHITECTURES: CONCEPTS..... 9**
 - 2.1. GENERAL FRAMEWORK..... 10
 - 2.2. ACTIVEX-DCOM..... 13
 - 2.3. JAVA RMI..... 16
 - 2.4. JAVA - IIOP (CORBA)..... 19
 - 2.5. HTML-HTTP..... 22
 - 2.6. CLIENT/SERVER..... 24
- 3. SYNTHESIS..... 27**
 - 3.1. DEPLOYMENT OF THE APPLICATIONS..... 28
 - 3.1.1. *Lessons*..... 28
 - 3.1.2. *Results of the tests about the deployment*..... 33
 - 3.2. PERFORMANCE MEASUREMENTS..... 35
 - 3.2.1. *Lessons*..... 35
 - 3.2.2. *Results of the measurements*..... 37
 - 3.3. COMFORT ZONES..... 40
 - 3.3.1. *ActiveX - DCOM*..... 41
 - 3.3.2. *Java - RMI*..... 42
 - 3.3.3. *Java - Corba*..... 43
 - 3.3.4. *HTML-HTTP*..... 44
 - 3.3.5. *Client/Server*..... 45
- 4. ANALYSIS OF PERFORMANCE MEASUREMENTS..... 47**
 - 4.1. ACTIVEX AND DCOM (MTS)..... 48
 - 4.1.1. *Deployment constraints*..... 48
 - 4.1.2. *Step 1: Deployment / Initialization*..... 49
 - 4.1.3. *Step 2: Retrieval of 50 lines*..... 52
 - 4.1.4. *Step 3: Retrieval of 5*50 lines*..... 54
 - 4.1.5. *Step 4: Insertion of 50 lines*..... 56
 - 4.1.6. *Step 1 to 4: Network utilization*..... 57
 - 4.1.7. *Step 5: Retrieval of 4 000 lines*..... 58
 - 4.2. JAVA RMI..... 59
 - 4.2.1. *Deployment constraints*..... 59
 - 4.2.2. *Step 1: Initialization / Deployment*..... 60
 - 4.2.3. *Step 2: Retrieval of 50 lines*..... 63
 - 4.2.4. *Step 3: Retrieval of 5*50 lines*..... 65
 - 4.2.5. *Step 4: Insertion of 50 lines*..... 66
 - 4.2.6. *Step 1 to 4: Network utilization*..... 67
 - 4.2.7. *Step 5: Retrieval of 4 000 lines*..... 68
 - 4.3. JAVA IIOP (CORBA)..... 69
 - 4.3.1. *Deployment constraints*..... 69
 - 4.3.2. *Step 1: Initialization / Deployment*..... 70
 - 4.3.3. *Step 2: Retrieval of 50 lines*..... 73
 - 4.3.4. *Step 3: Retrieval of 5*50 lines*..... 74
 - 4.3.5. *Step 4: Insertion of 50 lines*..... 75

4.3.6.	Step 1 to 4: Network utilization	76
4.3.7.	Step 5: Retrieval of 4 000 lines	77
4.4.	HTML-HTTP.....	78
4.4.1.	Deployment constraints.....	78
4.4.2.	Step 1: Initialization / Deployment.....	79
4.4.3.	Step 2: Retrieval of 50 lines.....	81
4.4.4.	Step 3: Retrieval of 5*50 lines	83
4.4.5.	Step 4: Insertion of 50 lines.....	84
4.4.6.	Step 1 to 4: Network utilization	85
4.4.7.	Step 5: Retrieval of 4 000 lines	86
4.5.	CLIENT/SERVER.....	88
4.5.1.	Deployment constraints.....	88
4.5.2.	Step 1: Initialization / Deployment.....	89
4.5.3.	Step 2: Retrieval of 50 lines.....	90
4.5.4.	Step 3: Retrieval of 5*50 lines	91
4.5.5.	Step 4: Insertion of 50 lines.....	92
4.5.6.	Step 1 to 4: Network utilization	93
4.5.7.	Step 5: Retrieval of 4 000 lines	94
5.	IMPLEMENTATION.....	95
5.1.	ACTIVEX-DCOM.....	95
5.1.1.	Implementation of the architecture.....	95
5.1.2.	Development of the client part	96
5.1.3.	Development of the server part.....	98
5.1.4.	Mechanism used for communication	99
5.1.5.	Development facility	100
5.2.	JAVA - RMI	101
5.2.1.	Implementation of the architecture.....	101
5.2.2.	Development of the client part	102
5.2.3.	Development of the server part.....	106
5.2.4.	Mechanism used for communication	108
5.2.5.	Development facility	110
5.3.	JAVA-IIOP (CORBA)	111
5.3.1.	Implementation of the architecture.....	111
5.3.2.	IDL definition of the objects.....	112
5.3.3.	Development of the client part	114
5.3.4.	Development of the server part.....	115
5.3.5.	Mechanism used for communication	116
5.3.6.	Development facility	116
5.4.	HTML-HTTP.....	117
5.4.1.	Implementation of the architecture.....	117
5.4.2.	Development of the client part	118
5.4.3.	Development of the server part.....	118
5.4.4.	Mechanism used for communication	119
5.4.5.	Development facility	119
5.5.	CLIENT/SERVER	120
5.5.1.	Implementation of the architecture.....	120
5.5.2.	Development of the client part	120
5.5.3.	Development of the server part.....	121
5.5.4.	Mechanism used for communication	121
5.5.5.	Development facility	121
6.	METHODOLOGY.....	123

1. Introduction

The aim of this study is to provide a pragmatic and realistic vision of the possible variants found in Intranet architectures. The tests designed to support the analysis were performed with a concern for direct application to the " real world ".

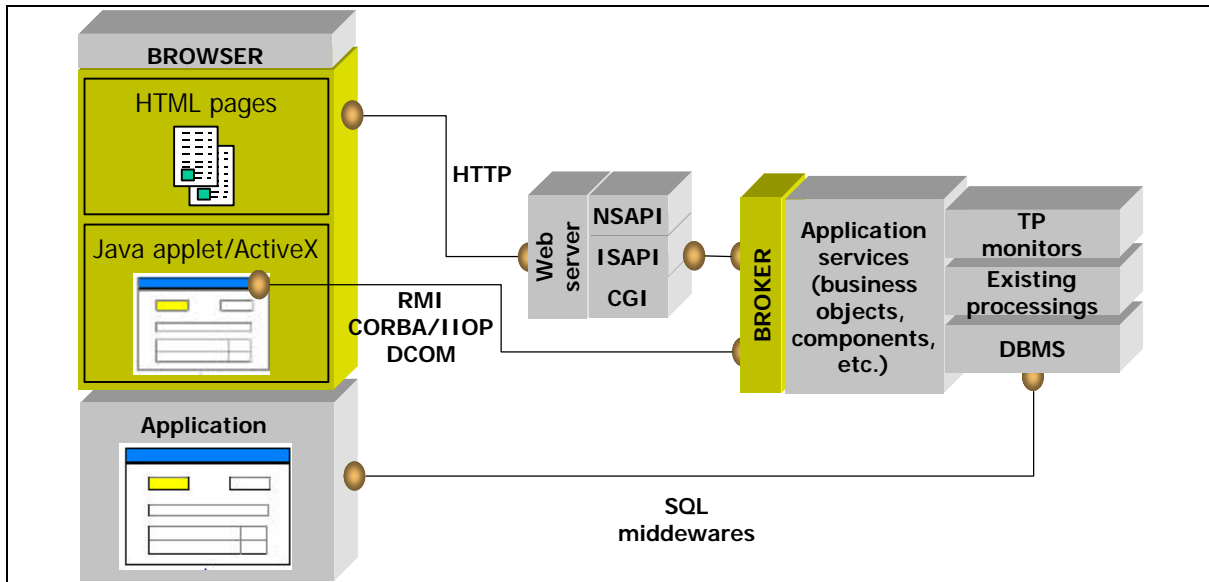
Even though our test application is rather simple, we are talking about true applications, which are to be deployed on heterogeneous platforms, representative of what one is to find in firms.

Depending on the type of application, the target and the operating constraints, the choices are numerous. Our comprehensive analysis will enable you to make a decision while having in hand all the assets necessary to maximize your investments.

2. Intranet Architectures: concepts

2.1. General framework

The architectures studied in this passport represent the models one may implement. This chapter will show the different Internet architectures conceivable, as well as the traditional client/server model.



General diagram of intranet and client-server architectures

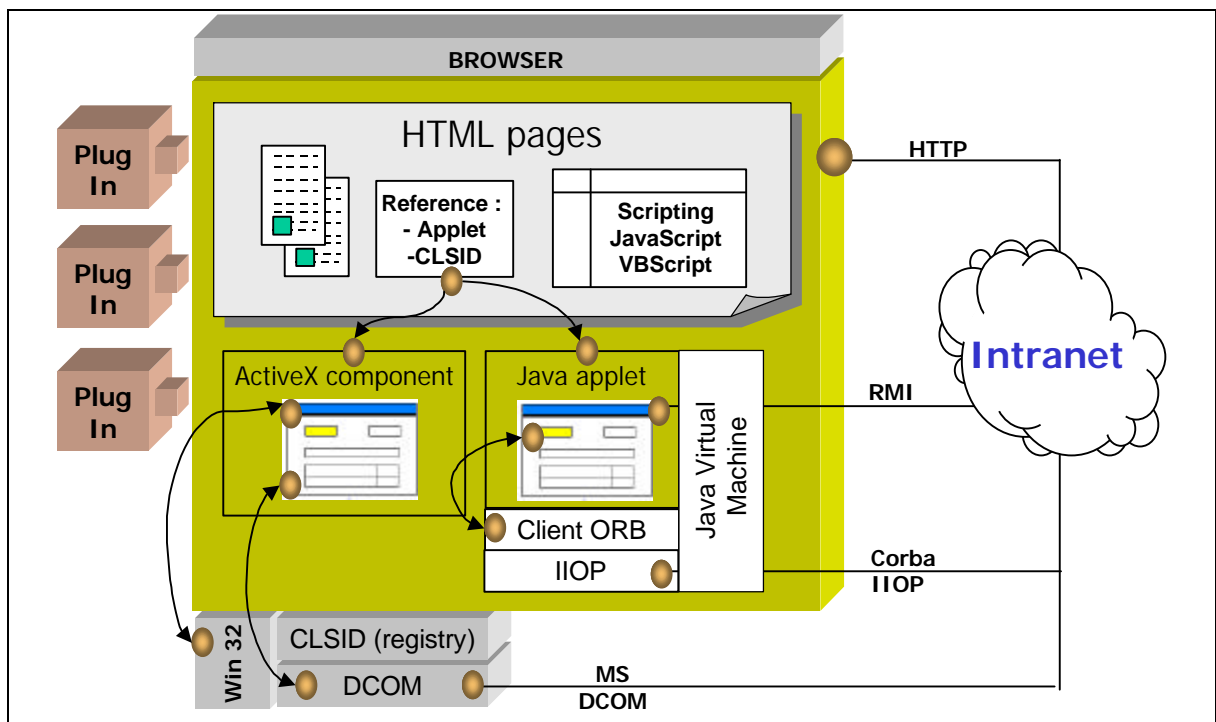
To deploy applications, intranet architectures rely on a browser, a sort of “ universal ” client, and on standard communication protocols. Through the tests carried out, this study will illustrate the constraints imposed by each architecture.

The client/server model relies on specific application deployment procedures and communication protocols.

■ **Synthetic presentation of the "universal" client: the browser**

We will only mention the two most popular ones: Netscape Navigator (or "Communicator" in its latest version) and Microsoft Internet Explorer. The main task of these two client tools is to interpret and to display HTML format pages. Today's reference is HTML version 3.2, as both Internet Explorer 3 and Netscape Navigator 3, currently the most widely used browsers, accept it.

Both Netscape Communicator 4.0 and Internet Explorer 4.0 offer extended features that are not yet standardized, notably concerning HTML. Communicator 4.0 also integrates IIOP, via Visigenic's ORB.



The browser, a platform for running applications.

Once installed on the client station, they allow for the visualization of the HTML pages sent by HTTP servers. Internet Explorer operates on Windows and Macintosh platforms (and now on Solaris Unix and HP-UX), whereas Netscape Navigator and Netscape Communicator cover all Unix clients, as well as OS/2, Macintosh and Windows platforms.

Apart from HTML, the page description language, the main browsers can each interpret a script language (JavaScript for Netscape, VBScript for Microsoft) which makes processing on the client station possible. These processings are directly linked to the HTML page, which contains the script.

The main browsers can also operate downloadable software components: Java Applets (for Netscape and Microsoft) or ActiveX components (Microsoft only). These components enable to bring the behavior of graphic interfaces to the browser (event-driven interfaces), to perform processings on the user workstation, or to maintain a connection with the processing server, so as to perform dynamic updates within the current HTML page.

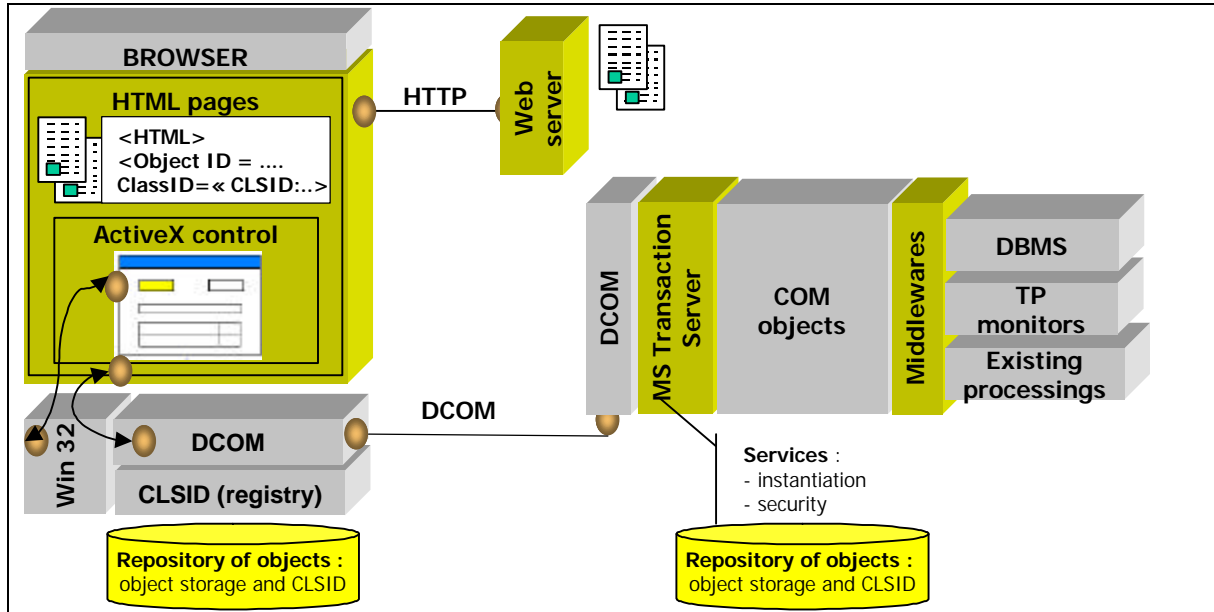
Both browsers support Java applets, as they both include a Java Virtual Machine (JVM) essential for the interpretation of the Java language. Internet Explorer 3 and Netscape Navigator 3 provide JVMs which rely on the JDK 1.02 (a stabilized version of a set of Java classes). As for Internet Explorer 4 and Netscape Communicator 4, they support the JDK 1.1.

Currently, ActiveX components are only supported by Microsoft environments. Therefore, one needs Windows to be able to use them. Netscape Communicator 4 requires the use of a plug-in to operate them, only in Windows, of course.

Plug-ins are risky bypass solutions, as they require a supplementary deployment step (physical installation on the client station). Plug-ins allow for the extension of browser features, by enabling it to perform new functions. Plug-ins must be specifically designed for a given browser, and must evolve with it. There is no standard to regulate their evolution.

2.2. ActiveX-DCOM

■ Architecture



ActiveX and DCOM communication principles.

The origin of the ActiveX model comes from OLE (Object Linking & Embedding). OLE-1 provides a way to produce more heterogeneous applications. OLE-2 introduces COM, the Component Object Model of Microsoft. COM then makes up a basis for inter-software interaction. DCOM (Distributed COM) is only an enhancement, which introduces the concept of physical separation of the object model, thus allowing the distribution of components between various computers.

The OLE stamp used to designate most technologies using COM. This label has since been replaced by the term ActiveX, which groups together all components working on the COM-DCOM model. The ActiveX components are not only visible graphical objects; they may also be technical or business components.

The use of ActiveX components in a browser is slightly different than their use inside a "traditional" Windows application. Firstly, the browser is able to automatically install the client components to be used. For this purpose, it downloads the installation files (.CAB files). The validity (as far as security is concerned) of the components is provided by a certificate-based system. As soon as the component has been downloaded, installed and referenced in the registry base, it may be used by the browser, or by any other application.

■ **Principles of operation**

A COM object (ActiveX component) provides services through methods, which are grouped into interfaces. It uses a standard interface (Iunknown) enabling the applications (or the other components) to know which other interfaces are available.

The standard interface of the COM components, Iunknown, is made up as follows:

Query Interface:

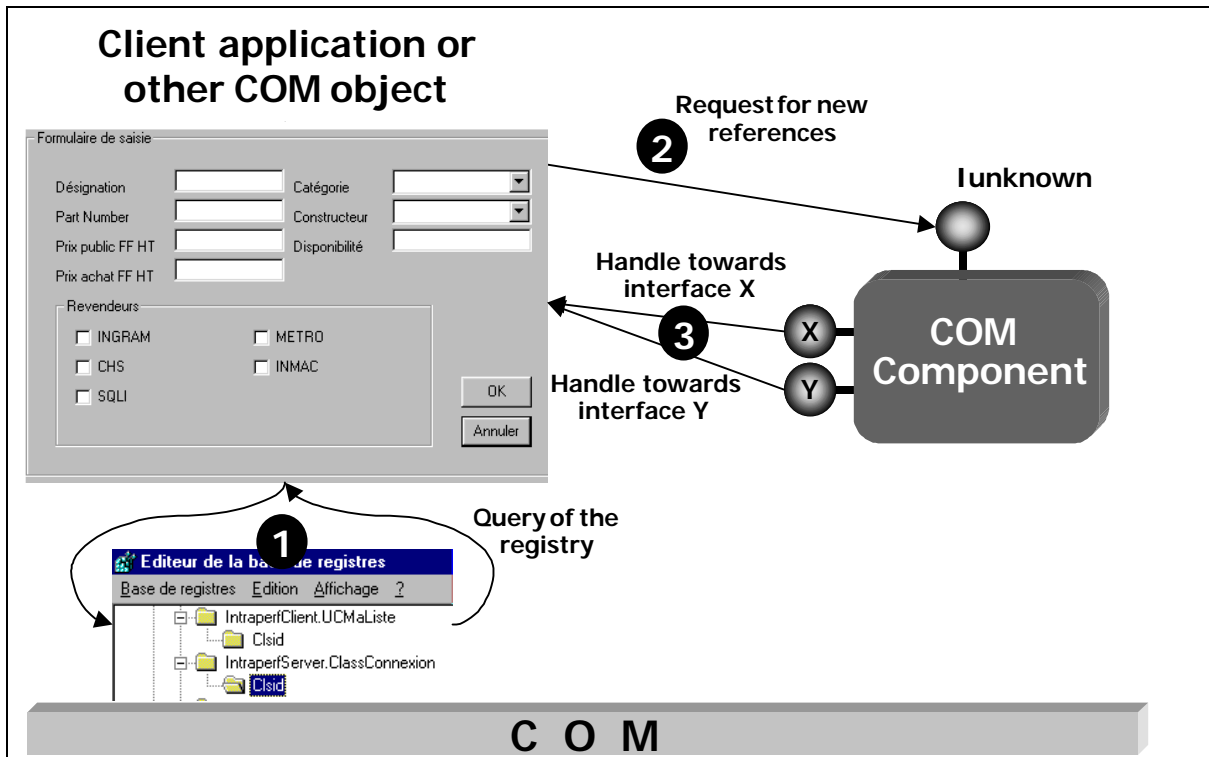
- Enables a client to request a handle towards another interface
- Helps solve the problem of version management

AddRef and Release

- Counts the number of object references
- Controls object life span

To put it simply, the client application will invoke the COM object:

- By using the information provided in the local registry base of each computer, (❶).
- It will then request an instance of the component (❷), by using the " CoCreateInstance " method with the reference of the object.
- In return, the client application receives a handle towards the Iunknown interface, which enables it to access the object interfaces (❸).



COM principles: interaction between application and component.

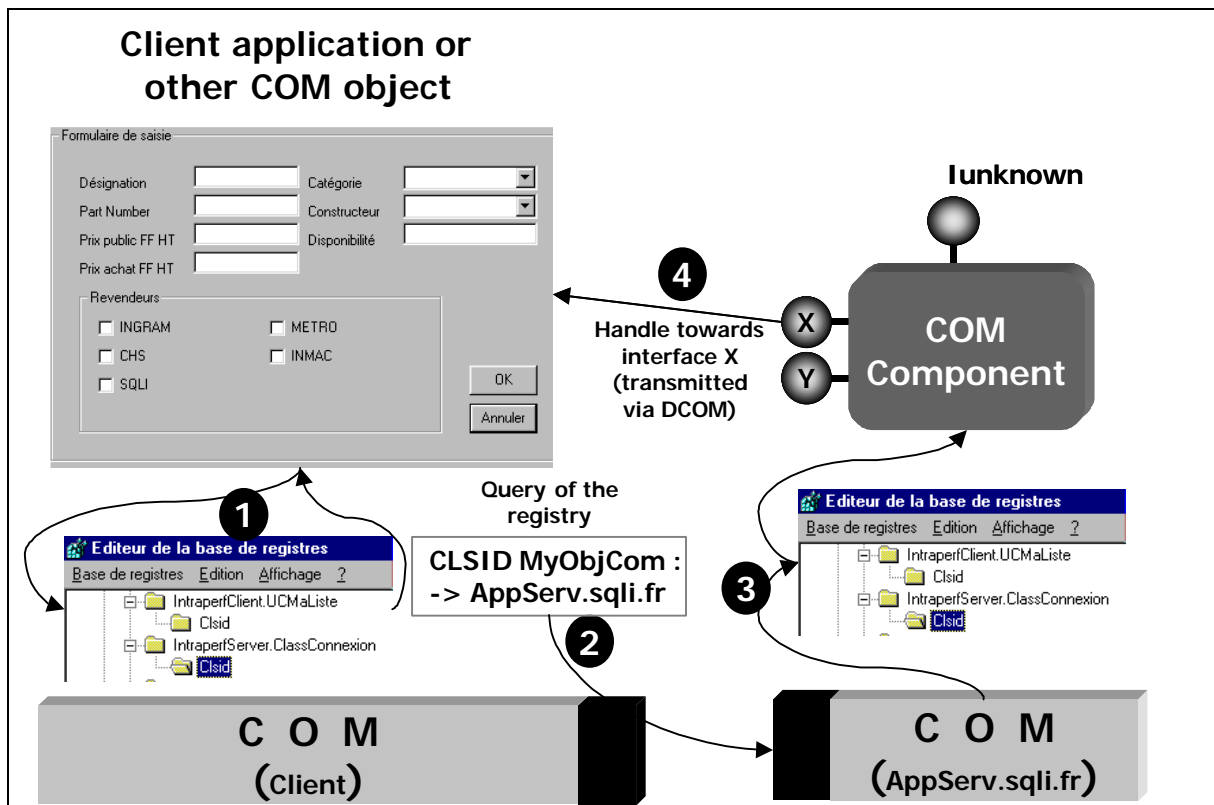
■ **From COM to DCOM**

In fact, DCOM (Distributed Component Object Model) is only an enhancement of COM, which gives it the dimension of a distributed system, enabling COM objects distributed over the network to communicate.

Microsoft claims in its DCOM white paper: " DCOM, the TCP/IP of objects ". While this vision is somewhat " biased ", one cannot deny the importance and the validity of this model.

The operation principles of DCOM are similar to those of COM. The client application will invoke the COM object:

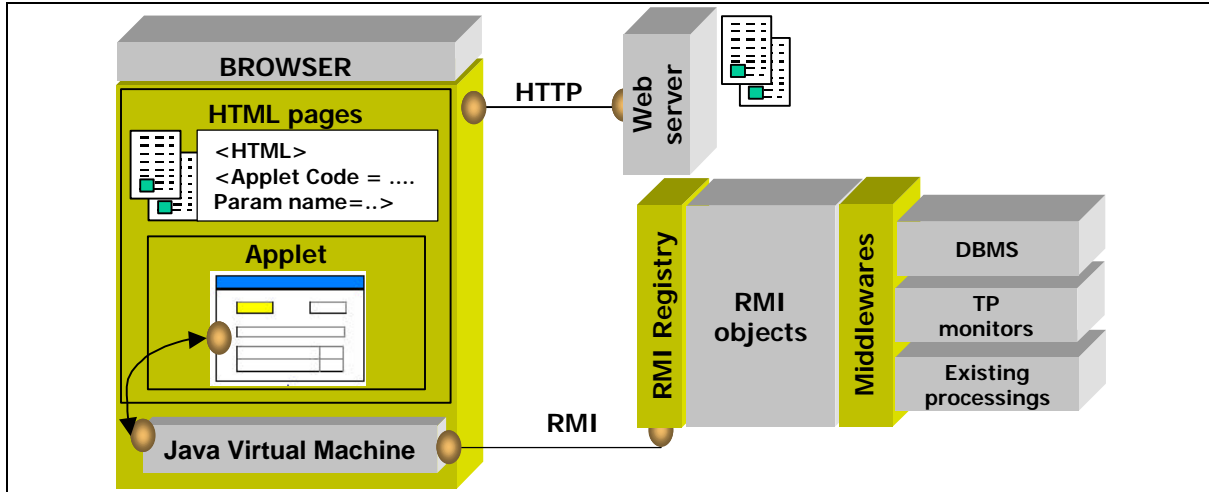
- It uses the information provided in the local registry base of each computer (❶).
- The register states that the object is remote; COM then transmits the location request to DCOM, which turns to the server designated in the registry (❷).
- On the server, COM takes care of instantiating the requested object, using the information contained in the server's registry (❸).
- Lastly, DCOM brings the pointer back over the interface created, and transmits it to the initial client (❹).



DCOM principles: interaction between application and remote component.

2.3. Java RMI

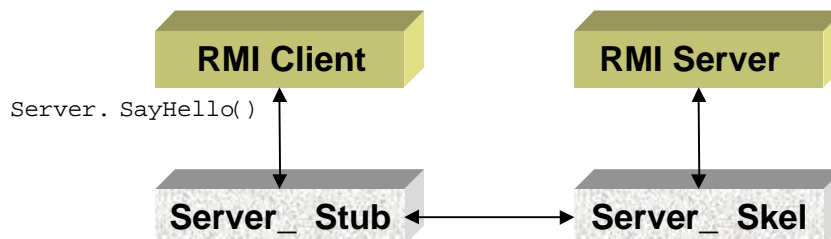
■ Architecture



RMI principles: interaction between applet and remote objects.

■ Principles of operation

RMI (Remote Method Invocation) provides a communication infrastructure allowing applications or Java applets to establish a dialogue. This middleware was born in JavaSoft laboratories, on the impulse of SUN. Totally linked to Java, RMI can be implemented rather simply. The architecture is based on the principle of *stubs and skeletons* (already used in RPC communications). The *stub*, loaded on the client station, plays the role of the server's proxy. Therefore, a remote method call is performed through the stub, and the server's location is made transparent to the developer. A call to a remote method will have the same syntax as a call to a local method. The server-side counterpart of the stub is the *skeleton*, which behaves like the client for the server.



Version 1.1 of the JDK (Java Developer Kit) introduces RMI and provides a conversion program able to generate these two classes from the server class programmed by the developer (*rmic*).

In parallel to this stub-skeleton mechanism, RMI uses the Java concept of interface. For each RMI server, the developer will have to provide 2 classes:

The server interface, which will provide descriptions (attributes, methods from my class...)

```
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

Server implementation (mechanics, such as database processings ...)

```
public class HelloImpl
    extends UnicastRemoteObject implements Hello
{
    private String name;
    ...
    public String sayHello() throws RemoteException {
        return "How do you do";
    }
    ...
}
```

The interface allows for the establishment of interface definition between the client and the server (equivalent to Corba's IDL). To make this interface definition available to the client, RMI uses a naming service, *RMIRegistry*. This registry answers connection requests to RMI servers, which registered themselves by returning their interface to the client. The calls are then performed through this interface:

```
...
try {
    Hello server = (Hello) Naming.lookup("//" +
        getCodeBase().getHost() + "/HelloServer");
    message = server.sayHello();
}
...

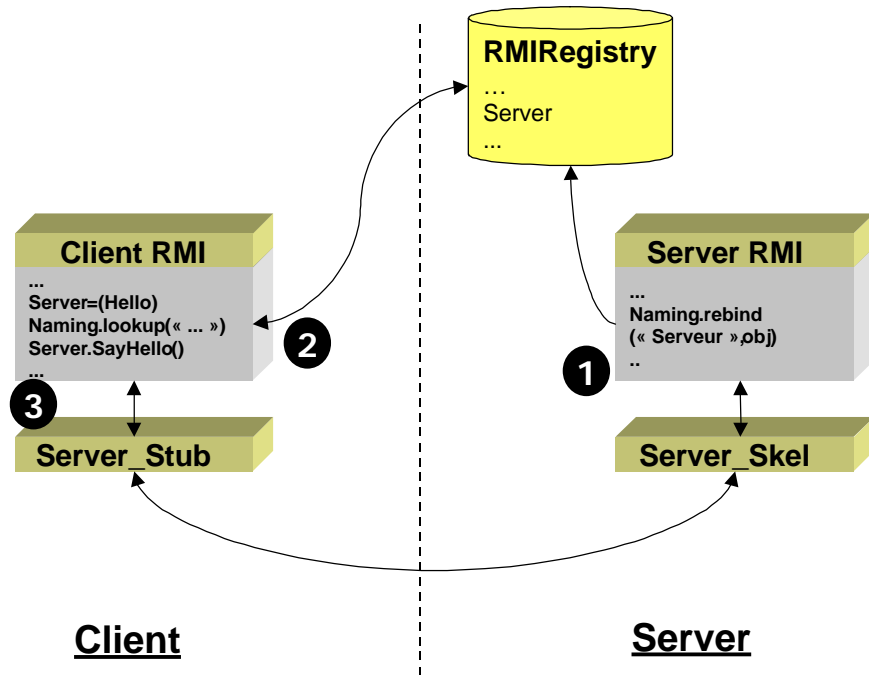
```

As the stub's loading is fully transparent, the communication is conveyed without any specific programming. The only part that needs to be developed is the registration of the server in the *RMIRegistry* as soon as it is launched:

```
public static void main(String args[])
{
    // A Security manager is mandatory
    System.setSecurityManager(new RMISecurityManager());
    try {
        HelloImpl obj = new HelloImpl("HelloServer");
        Naming.rebind("HelloServer", obj);
        System.out.println("server registered");
    }
    catch (Exception e) {
    }
}
```

NB: the *RMIRegistry* is an application which runs in memory, and which is launched in the same way as any other executable module.

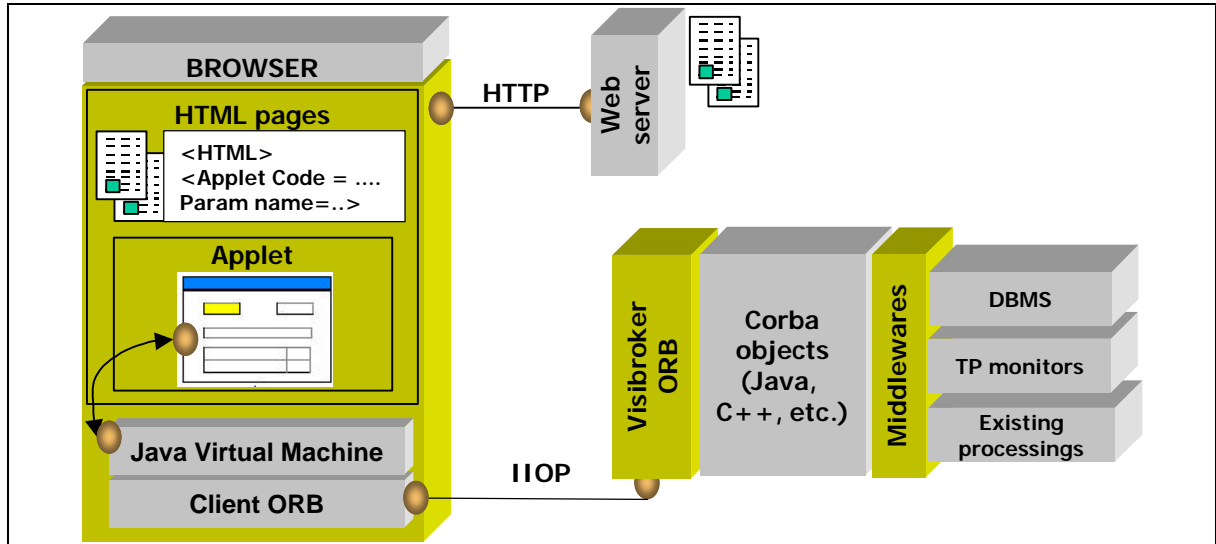
The diagram below shows the interaction between a server, a client and the RMIRegistry:



Today, RMI relies on RRL (Remote Reference Layer) to interface with transport layers. This layer may be replaced by IIOP in the next versions of JDK 1.2, so as to provide interoperability with Corba.

2.4. Java - IIOP (Corba)

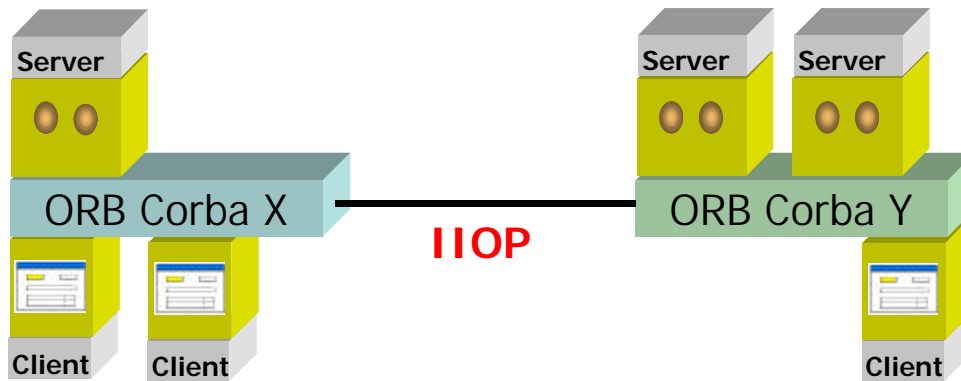
■ **Architecture**



Java - IIOP principles: interaction between applet and remote objects.

■ **Principles of operation**

CORBA (Common Object Request Broker Architecture) is a specification defined by the OMG in hopes of normalizing distributed object environments. While it is true that CORBA is only on paper, some editors (Iona, BEA, Visigenic - Inprise...) have implemented their own Corba ORBs. In fact, as its specifications are rather vague (and even, for some of the 15 Corba services, impossible to implement), the internal operation of each of the ORBs is proprietary. The role of the IIOP protocol (Internet Inter ORB Protocol) is to allow the communication between these various ORBs.



● Distributed Corba Object

As shown in the above diagram, each of the computers involved in a Corba application will have to run an ORB. For the servers, the installation is the same as for any piece of standard software. For the clients, one may choose between a traditional installation and the unloading of the ORB client through an applet during each application initialization.

As for RMI and DCOM, the communication architecture between the objects is based on the principle of *stubs and skeletons* (*proxy* and *stub* in the DCOM terminology). The *stub* loaded on the client station plays the role of the server's proxy. Therefore, a remote method call goes through the stub, and the server's location is made transparent to the developer. A remote method call will have the same syntax as a call to a local method. The server-side counterpart of the stub is the *skeleton*, which behaves like the client for the server.

Corba uses the IDL (Interface Definition Language) so that the client can know the public methods and attributes of a server. The use of this declarative language enables clients and servers written in different languages (Java, C, SmallTalk, etc.) to cooperate. Once the IDL definition of an object has been done, there are some compilers (for example, *idl2java* for Visibroker) which are able to automatically generate the interface according to the language chosen. One then only has to develop the object's implementation.

Example of an IDL file:

```
interface Automobile
{
    long color();
    long price();
};
```

In order to allow access to a Corba server during implementation, one will have to register it. An IOR (Implementation Object Reference) will be assigned to the object. This IOR will be a unique identifier, which, once obtained by the client, will be able to establish a dialogue with the server:

```

public class Server {
public static void main(String[] args) {
// ORB initialization.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
// BOA initialization.
org.omg.CORBA.BOA boa = orb.BOA_init();
Automobile best_bargain_now = (Automobile)
    new_example_Automobile("best_bargain_now");
// Exports the object created and waits for the queries
boa.obj_is_ready(best_bargain_now );
boa.impl_is_ready();
}}

```

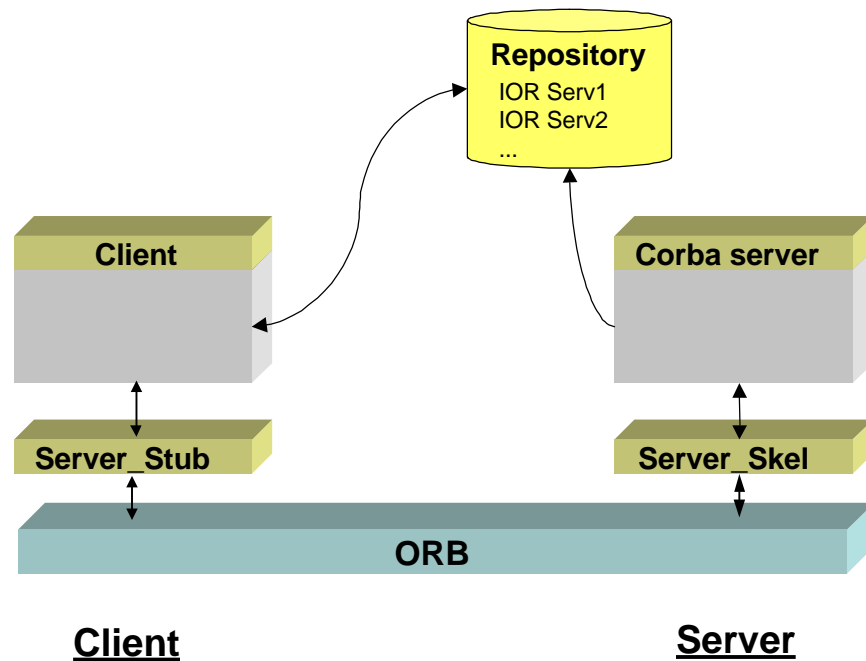
The client will then use the object's public methods after having established a connection with them through the IOR, as if they belonged to a local object:

```

public class Client {
public static void main(String[] args)
{
// ORB initialization.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
Automobile car =
AutomobileHelper.bind(orb,"best_bargain_now");
int prix = car.price();
System.out.println("The price is: " + price);
}}

```

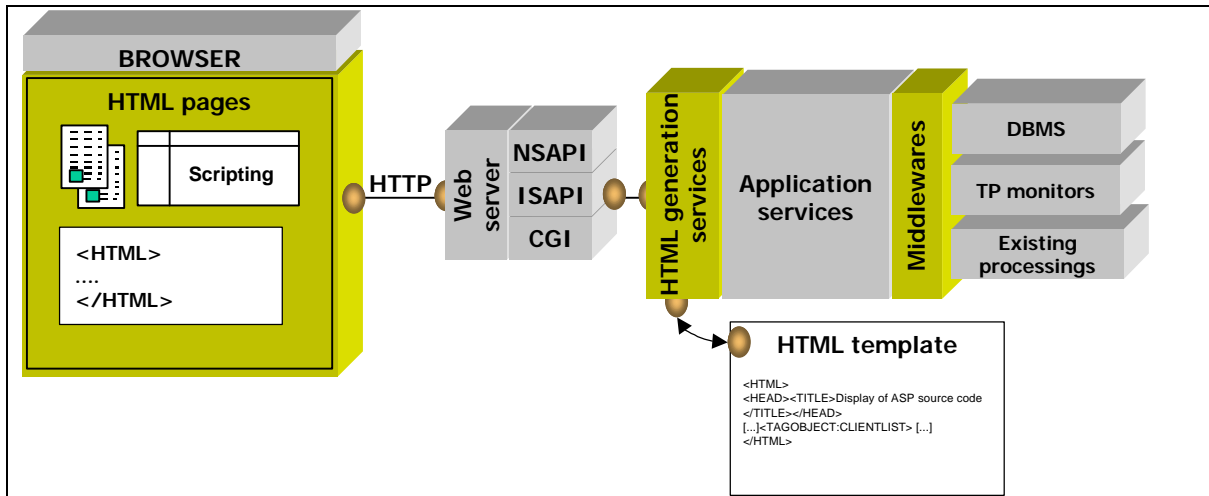
The diagram below shows the interactions between a Corba server and a client:



NB: Most vendors leave developers the choice between several IOR recovery mechanisms (bootstrapping, recovery on an HTTP URL, etc.)

2.5. HTML-HTTP

■ Architecture



HTML - HTTP principles: communication between component.

In its simplest form, the Web is characterized by the presence of an HTTP server (Web server), a browser able to interpret HTML page formatting language, and the HTTP communication protocol.

The HTTP protocol (Hyper Text Transfer Protocol)

The browser relies on HTTP protocol to communicate with the servers. This communication protocol is simple, and it is implemented over TCP-IP. So, whether over the Internet or on your local network, the browser will use this protocol to communicate with the server. The HTTP communication protocol operates in disconnected mode: the browser is only connected to the server during page download. Once the download is completed, the browser will disconnect itself from the server. This principle allows saving on network bandwidth.

■ **Principles of operation**

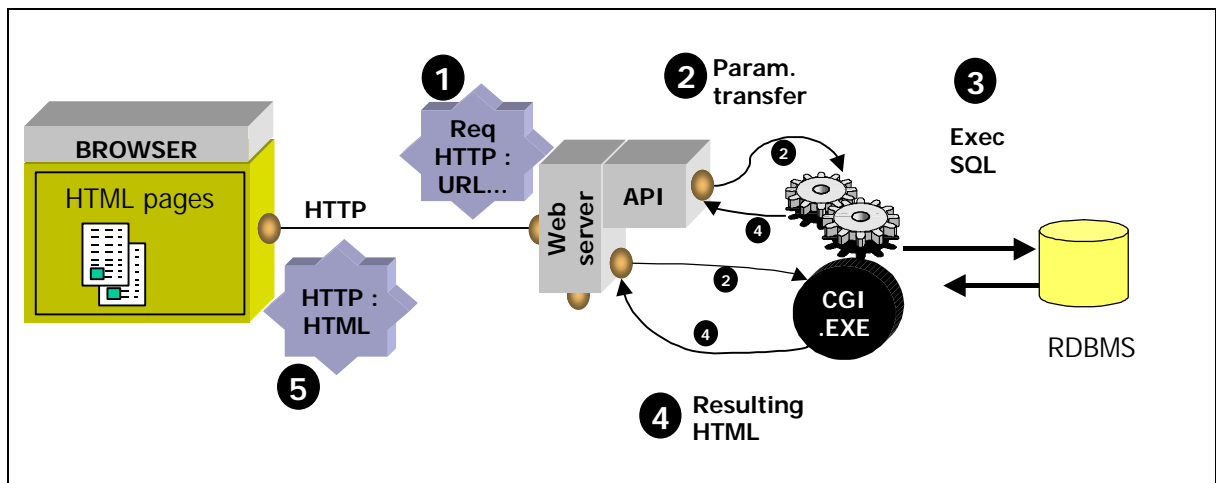
A WEB/HTML application is based on dynamic pages. The dynamic pages (applications) are called through a gateway. While the result cannot be anything other than an HTML page, the way to obtain it depends on the HTTP server and the development tool used. The applications may be called via a CGI (Common Gateway Interface), ISAPI (Internet Server API), NSAPI (Netscape Server API) or WRBAPI (Oracle Web Request Broker) gateway.

The main asset of the CGI gateway is its portability. All HTTP servers are able to run CGI modules. These executable modules, which are automatically installed by the development tool, are loaded in the server's memory while the application server is generating the page.

The NSAPI and ISAPI gateways (the HTTP-server API of Netscape and Microsoft, respectively) operate similarly to CGI, but they are libraries (DLL on NT) which take up less resources, and which are loaded directly on to the server's memory. Indeed, the library's code is only loaded in memory once. The various calls to the gateway only trigger the instantiation of the "Data" part in memory. This advantage is to be minimized: the NSAPI and ISAPI gateways respectively only operate on HTTP servers from Netscape and Microsoft. Oracle proposes an equivalent API, WRB, for its HTTP server (WebServer).

The operation principle is as follows:

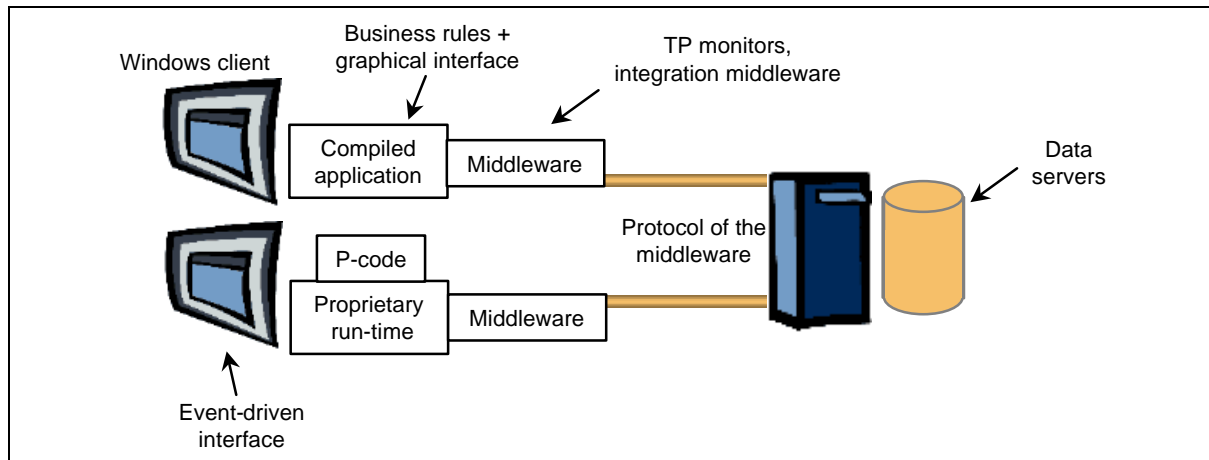
- The user clicks on the "Submit" button of an HTML form; the browser uses the Post method from HTTP, specifies the target URL and the HTTP headers. HTTP receives the call via a TCP/IP connection. The server analyzes the message and recognizes POST (1).
- The Web server launches a CGI program or transmits the request to an application server via an API (2).
- The CGI program or the application server performs the processings, for instance the connection to the DBMS (3).
- The CGI program or the application server generates the resulting HTML page (4).
- The Web server (HTTP) returns the results to the Web browser (5).



HTML - HTTP principles: on the fly HTML construction.

2.6. Client/server

■ First generation client/server architecture



First generation client/server architecture principles.

The first generation client/server architecture allows for the running of a client application with data (or processings) located on a server. The communication between the client and the server relies on proprietary protocols established by database vendors.

The entire application is located on the client station. This application relies on the middleware to access databases. Depending on the development tool used, the application is either compiled, or interpreted by a runtime module.

The database is used to store application data, but it also often makes some processings possible: the stored procedures.

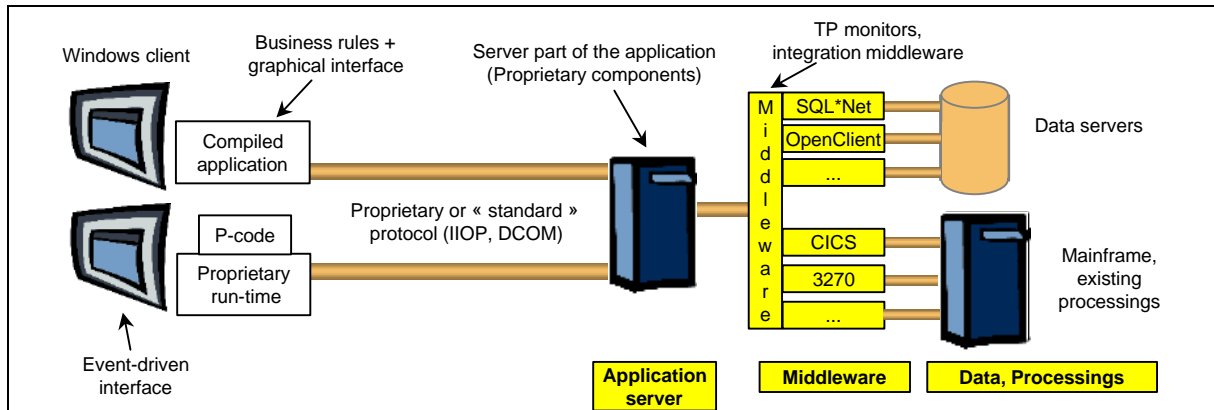
The middleware is based on a proprietary communication protocol. Nevertheless, the dialogue between the application and the database conforms to SQL language.

Before being able to run the application, it must be installed on the client stations. This involves installing the runtime module, the application's DLL, the application itself and the middleware. Next, the whole chain between the client and the server must be configured: the client middleware (possibly also ODBC), the runtime and the application.

A specialist must perform the entire installation stage. Therefore, widespread diffusion of the application is costly and time-consuming. What's more, the installation of new applications may generate conflicts with existing applications concerning the configuration of middleware(s) or the sharing of DLLs (version conflicts).

Similarly, application updates require the intervention of a specialist on each of the client stations.

■ **Second generation client/server architecture**



Second generation client/server architecture principles.

Compared with first generation client/server architecture, the second generation brought about the partitioning of application services (processings) on an intermediate server. This architecture lightened up the client station, by moving some processings from the client to the application server.

Generally, database access is only performed at the application server level. The client stations only access the various application services available on the intermediate server.

This solution thus simplifies application updates (in many cases, only the application services located on the server are updated), but the cost of the initial deployment remains the same. Indeed, communication between the client station and the intermediate application server continues to rely on proprietary protocols. The deployment constraints remain the same.

3. Synthesis

3.1. Deployment of the applications

3.1.1. Lessons

The first lesson one might learn from this test concerns application deployment. The key elements of each different model are as follows:

ActiveX-DCOM

In order to deploy components on the client station, one needs to know not only the type of operating system (16- or 32-bits Windows), but also the type of browser. This type of object is to be installed on the client station on a long-term basis. In addition to these deployment constraints, there is no doubt that this model is very difficult to maintain. For example, when the location of the DCOM server changes, it is necessary to update all of the registers on client stations.

Java RMI

It must be taken into account that the downloaded applet relies on a JDK version. Similarly, each browser puts forward its own "virtual machine" JVM (linked to a version of the JDK). As the RMI protocol is integrated into the JDK, the conformity between the base JDK of the applet and the JDK of the browser's JVM is enough to guarantee proper operation. On the other hand, this middleware has little to offer in terms of maintenance (workload, transparency of the server's location, etc.), because it lacks, among other things, a centralized location mechanism.

Java IIOP (Corba)

As with RMI, version conformity between the JDK of the applet and the browser plays a key role, but what's more, the components necessary for communication between Corba objects must be supported by the browser (Netscape version 4 integrates the client part of Visibroker's ORB). In practice, these components must be deployed locally, so as to ensure compatibility, and avoid systematically downloading the client ORB (as JAR, compressed files). Once this deployment stage has been completed, this middleware has limited maintenance constraints.

HTML-HTTP

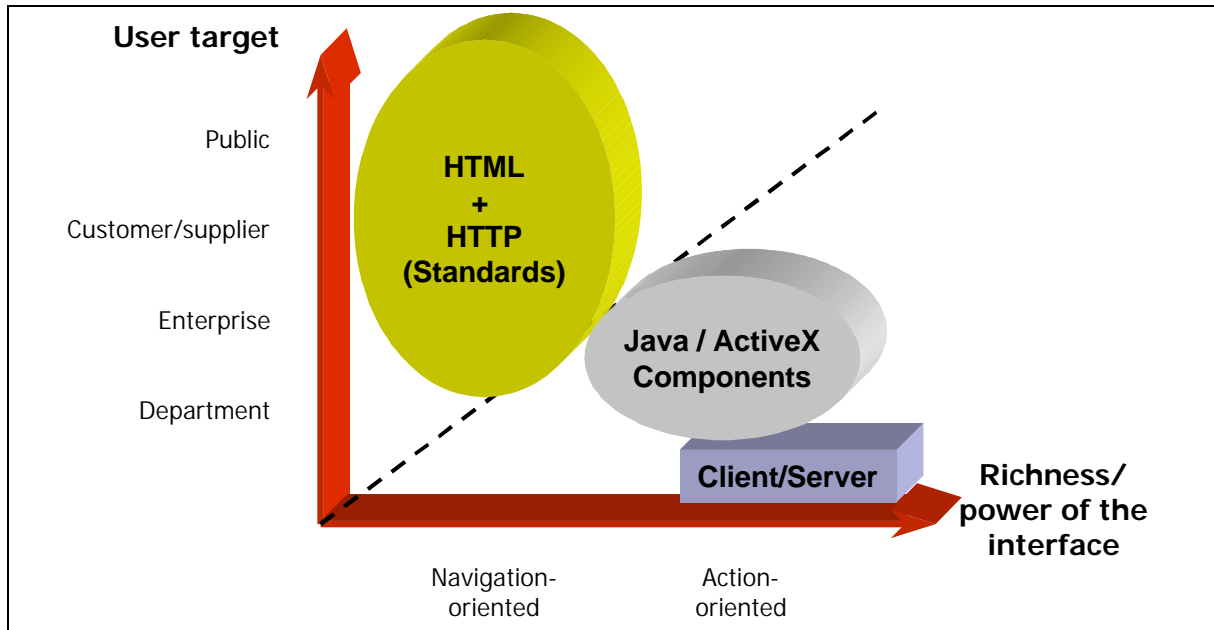
The HTTP transport protocol is standard. Its behavior remains equivalent, no matter the browser used. If one limits himself to HTML version 3.2, all browsers will interpret the pages normally. Operating discrepancies come about with client scripts that may be used, or with the new version 4.0 of HTML, which is not fully supported by the various browsers. Should there be a change in the server's configuration or a new version, only trivial modifications have to be made on the client station.

Client/server

The client/server model starts from the notion that the client is identified. From then on, a true deployment stage begins, with a step dedicated to the configuration and/or installation of runtime type components (DLL and EXE files) and middleware components (for example, SQL*Net to access an Oracle database). This deployment stage depends mostly on the operating system of the client station. Apart from file

deployment, one must also parameterize the middleware. This parameterization is to be repeated on all the client stations each time the server configuration is altered or a new application is deployed.

Our results enable us to deduce this first matrix representing the deployment as a function of the target and of the type of graphical interface, as follows:



Positioning of the technologies in relation to the target and to the user interface.

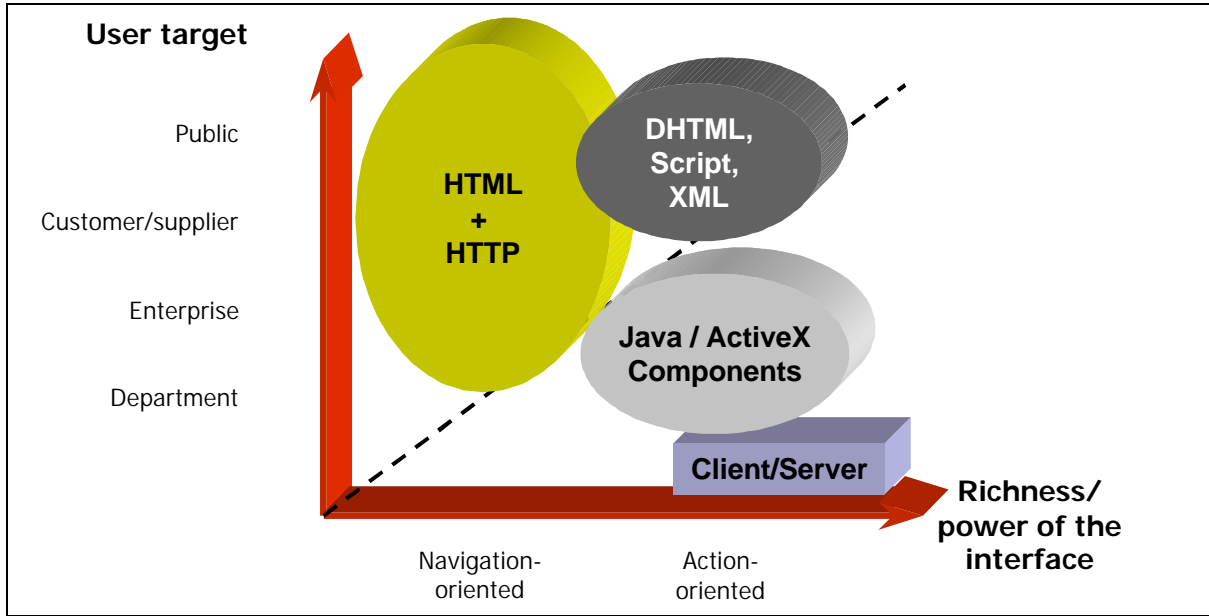
The higher the target (the number of users) is, the more difficult it becomes to predict which configuration they will use to access your applications. Considering this, only standard technologies such as HTML and HTTP will be able to fill your needs.

To express the degree of richness and power of the interface, one will consider it as more or less **Navigation-oriented (browsing)** or **Action-oriented (manipulation)**. Navigation is a distinctive feature of the hypertext interface. It provides the user with a very simple approach, but extensive interaction is limited. Manipulation is a distinctive feature of the windows-based graphical interface (like Windows). It allows the user to request very sophisticated actions.

The higher the degree of richness and power of the interface requested, and the farther one moves away from the standards, the more technical constraints one creates for deployment.

The evolution of standards, such as HTML 4, the scripting language (ECMA Script) and the Document Object Model (DOM), or even XML, pushes this set of technologies towards a higher-capacity graphical interface.

The diagram below shows which set of technologies will best answer the problems posed by very large-scale deployment together with a high degree of richness and power of the interface.

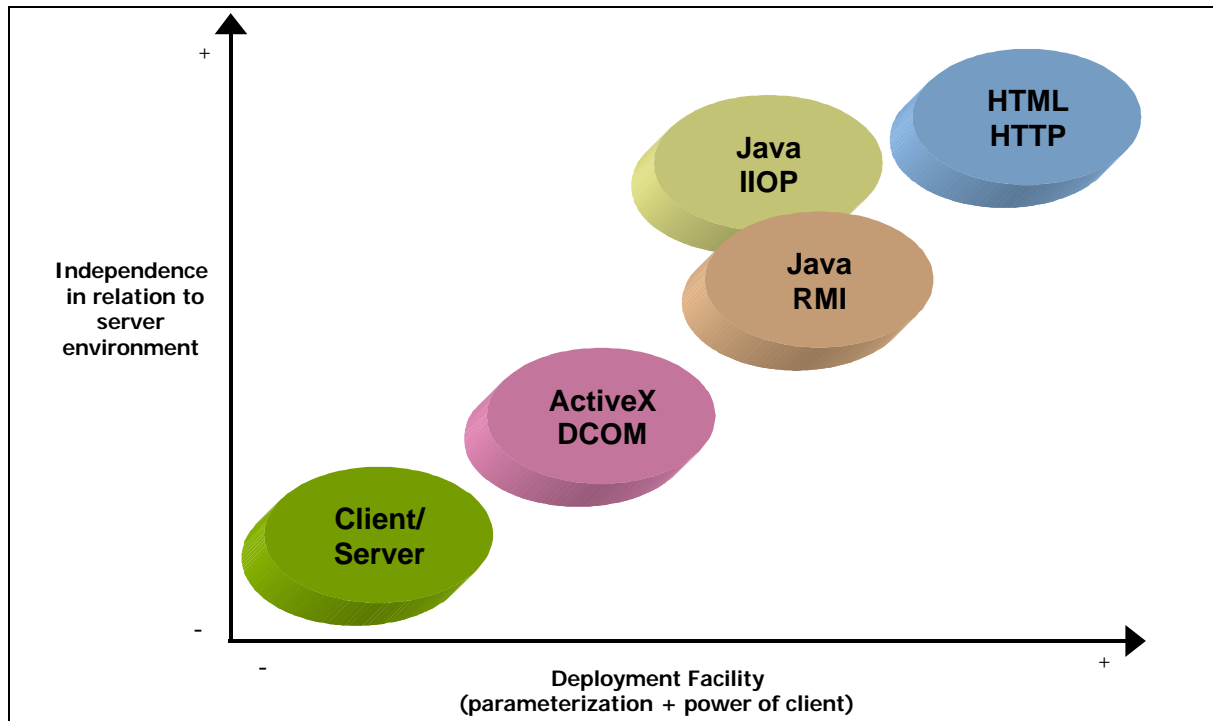


Evolution of Intranet technologies on the client side.

These results also show important differences in behavior when faced with deployment and maintenance constraints:

The *deployment constraints* include prior installation and parameterization, as well as the client configurations necessary for proper operation of the applications (OS, power, browser, etc.)

The *maintenance constraints* cover the re-parameterization tasks of the client stations to be performed when the server configuration is modified, and when a new release of the application is implemented.



Positioning of the technologies in relation to deployment and maintenance constraints.

The **Client/Server** model is, on the whole of these two axes, the least favorable one. This positioning was easily foreseeable. In fact, this architecture was designed to provide users with a rich interface, and not to ease deployment and maintenance tasks.

ActiveX-DCOM is definitely not the right solution to the constraints of Client/Server model. Indeed, the client ActiveX only adequately deploys on very specific configurations (which are not manageable with numerous stations). Moreover, as the object reference is hard-coded in the registry of client stations, the architecture is rendered quite rigid in relation to the server environment. A true object directory (Active Directory from NT5) will perhaps bring greater flexibility.

The use of Java, and consequently of the JVM, in the **Java-RMI** model, ensures independence in relation to operating systems. The deployment constraints are limited to the availability of an adequate version of JVM (JDK 1.1.x) and enough power on the client station for smooth operation of Java applets or applications. Should server configuration evolve, this architecture will show its limits. For instance, a Java-RMI application is only allowed to establish a dialogue with the very station running the RMI registry with which it is interacting. This prohibits, in particular, workload balancing over several stations.

Java - IIOP is quite close to the Java-RMI model. Nevertheless, this model adds a few deployment constraints. Indeed, ORB installation on client stations is mandatory in order to obtain optimal application operation (in return, no parameterization is necessary). As far as independence in relation to the server environment is concerned, Java-IIOP is fairly flexible. Indeed, Corba localization mechanisms (still quite proprietary for the moment in regard to market solutions) render the server world transparent. Note: Contrarily to RMI, Corba is independent of the development language. This possibility to choose creates additional deployment solutions.

HTML-HTTP is a pair of true standards, endorsed and used by all actors of software industry. In fact, this solution is the most flexible, the most all-terrain.

3.1.2. Results of the tests about the deployment


















The following tables show the result achieved by each architecture according to the browser and platform used. These results were obtained through the tests we ran. In the cases where the application did not run, there might exist bypass solutions (installing a plug-in, downloading a more recent version of the browser or a patch, etc.), but we wanted to show the most realistic synthesis as possible.

■ **Deployment on Windows 95 and NT 4**

Browser - Architecture	Internet Explorer 3	Internet Explorer 4	Netscape 3	Netscape 4
ActiveX - DCOM			Does not support ActiveX	(Operates with the Ncompass plug-in)
Java RMI	JDK 1.0		JDK 1.0	
Java IIOP (Corba)	JDK 1.0	Deployment of ORB	JDK 1.0	
HTML-HTTP				
Deployment apart from browser				
C/S *				

(*): In order to run the client/server application, one needs to install the application and its components (runtime and DLL), as well as the SQL*Net middleware from Oracle.

■ **Deployment on Windows 3.11**

Browser - Architecture	Internet Explorer 3	Internet Explorer 4	Netscape 3	Netscape 4
ActiveX - DCOM	 32 bits ActiveX / DCOM component	 32 bits ActiveX / DCOM component	 32 bits ActiveX / DCOM component	 32 bits ActiveX / DCOM component
Java RMI	 JDK 1.0	 Bad RMI implementation	 JDK 1.0	 Bad AWT implementation
Java IIOP (Corba)	 JDK 1.0	 ?	 JDK 1.0	 Bad AWT implementation
HTML-HTTP				
Deployment apart from browser				
C/S *	 (providing the application is compiled in 16 bit mode, and SQL*Net 16 bits is installed)			

(*): In order to run the client/server application, one needs to install the application and its components (runtime and DLL), as well as the SQL*Net middleware from Oracle.

3.2. Performance measurements

3.2.1. Lessons

Depending on the type of processing performed by the application, at least one architecture model emerges as the least bandwidth consuming. In fact, all of the models studied seemed to be at ease in at least one test scenario. What's more, the network-related measurements taken are more or less closely linked to the way programs are designed and written. It is therefore impossible to generalize and establish an "official" ranking.

However, some lessons might yet be learned:

ActiveX-DCOM

The use of network bandwidth is reasonable, except during component download and installation (this step only performed once). Relying on DCOM, the network traffic between the component and the application server remains at an acceptable level. However, the main drawback is that traffic volume is difficult to predict. DCOM is a rather rich, but obscure protocol for the developer.

Java RMI

Apart from the initialization stages (applet loading), the consumption of network bandwidth is rather good. The way one programs is important, as a great freedom (and responsibility) is given to the developer. However, the systematic exchange of the interface definition (the description of the public attributes and properties of the remote object, equivalent to the IDL with Corba) makes this model verbose in case of recurrent calls of the same methods.

Java IIOP (Corba)

The consumption of network bandwidth is rather good, and very homogeneous in all the scenarios (apart from initialization). As for RMI, the results concerning the network depend heavily on the options chosen during the object design. If the components are reused enough, without being too generic, this model will be optimal, in particular with recurrent calls. Besides, the mechanisms used for exchanging the interface definition are especially suitable for this mode of operation (a single exchange for n calls).

HTML-HTTP

The measurements are good in most cases. One of the main aspects of this model, is the ease with which the bandwidth consumption can be anticipated. As only HTML pages are sent to the client via the HTTP protocol, it is easy to calculate in advance the flows that will be generated. As the HTTP protocol is standard, its behavior during the transfer of an HTML page is constant and foreseeable. Besides, this model is the only one among those tested in which data and layout are mixed. If it proves necessary to optimize the bandwidth consumption, this will be mostly achieved through formatting constraints.

Client/server

On the whole, the network measurements taken for this model are satisfactory, except in one case. However, changing the programming logic may easily optimize this step (Step 4). This is one of the main difficulties experienced with traditional client/server technologies: the way the application is developed has a strong impact on the consumption of network bandwidth. And only thorough measurements, as the ones we conducted, highlight communication excesses before deployment on a network with low or average traffic.

3.2.2. Results of the measurements

■ **Step 1: Initialization**

Reminder

During this step, the application is launched and positions itself on the main menu. The connection to the database is set up, and if necessary, the application server initializes a new user session.

Architectures	Step 1 Initialization	
	Nb frames	Total nb bytes
ActiveX-DCOM (installed component)	19	4 172
Java RMI	195	38 194
Java IIOP (Corba) Installed ORB	78	34 158
HTML-HTTP	10	1 737
C/S	45	4 883

For the ActiveX-DCOM model, the physical, permanent deployment of all client components is its only mode of operation.

For Java IIOP, we pre-installed the ORB on the client station. This prevents downloading the client portion of the ORB each time the application is run. Nevertheless, the applet, which then uses the client ORB, is always downloaded at each subsequent use of the browser.

These steps of the station's configuration are described in detail in chapter 4, Analysis of performance measurements (p 47).

The HTML-HTTP option is the one that consumes the least network bandwidth at this stage. Indeed, all initialization tasks take place on the server side; the user only receives one HTML page, of which the only two pictures used are located in the cache memory. On this subject, even though the pictures are stored in the browser's cache, an exchange with the Web server is performed all the same, to check the validity of the picture contained in the cache.

For the C/S and ActiveX-DCOM architectures, the exchanges only concern the connection with the server. The C/S application directly connects to Oracle DBMS via SQL*Net, while the ActiveX component connects to the application server (an ActiveX server component) via DCOM.

For Java RMI and Java IIOP architectures, the browser first downloads the applet (it is very similar in both cases). This explains the higher number of frames and volume of these two architectures. The applet then establishes a connection with its server via RMI or IIOP.

■ **Steps 2 and 3: Loading 50 lines**

Reminder

During these steps, the application displays the first 50 lines resulting from a SQL query, then loads and displays the next 50, repeating this operation five times.

Architecture	Step 2 Loading of 50 lines		Step 3 Loading of 50 lines (5 times)	
	Nb frames	Total nb bytes	Nb frames	Total nb bytes
ActiveX-DCOM	20	14 880	55	67 650
Java RMI	20	9 588	60	30 025
Java IIOP (Corba)	46	33 787	49	32 635
HTML-HTTP	18	7 248	90	36 499
C/S	29	10 071	24	15 884

The above architectures obtain rather homogeneous results. A detailed analysis of each architecture provides all the explanation needed to understand the mechanisms involved.

One may note, for the HTML-HTTP model, an exact 1 to 5 factor between the results obtained in Step 2 and Step 3. This confirms the "predictable" aspect of this model. For the object-middleware architectures (DCOM, RMI and Corba), the ratio is different. Indeed, the first call uses up the most bandwidth, as the exchange of the interface definition (a description of the public attributes and properties of the remote object) and stub loading on the client station come before the sending of the data.

■ **Step 4: Insertion of 50 lines**

Reminder

During this step, the application displays a form in which the user inputs information. When the user validates the transaction, the DBMS processes one insert in the PRODUCT table and five inserts in the PRODUCTDEALER table. Once the transaction has been completed, a message stating that transaction was accurately carried out is sent to the user.

This operation is repeated 50 times.

Architectures	Step 4 Inserting 50 products	
	Nb frames	Total nb bytes
ActiveX-DCOM	112	78 008
Java RMI	260	65 063
Java IIOP (Corba)	168	31 680
HTML-HTTP	552	106 875
C/S	38 808	341 701

In this type of processing, little information is forwarded, but numerous round trips are performed, as intensive data input is simulated.

The ActiveX-DCOM, Java RMI and IIOP architectures operate in a "remote object method call" communication mode. So, each transaction triggered generates only a single procedure or method call, with the transaction values as parameters. This is the reason for which these architectures obtain the best scores, with a special mention for Corba.

With HTML-HTTP, the number of frames and the volume exchanged are considerable. Here, despite the fact that one only needs to transmit the parameters to be input, formatting information is also returned at each transaction.

For the Client/Server application, the bad results displayed are due to the programming method used. The application was designed in a standard way, and in this case, each time a transaction is validated, it is the client station, which sends the 6 SQL Insert queries to the DBMS. This explains the high volume of data exchanged. In this case, the use of a stored procedure would allow for a reduction in the number of network exchanges.

■ **Step 5: Loading 4000 lines**

Reminder

During this step, the application displays, all at once, 4000 lines resulting from a SQL query. The 4000 lines are retrieved from the client station.

Architecture	Step 5 Loading of 4000 lines	
	Nb frames	Total nb bytes
ActiveX-DCOM	1 211	1 124 750
Java RMI	514	414 406
Java IIOP (Corba)	573	529 455
HTML-HTTP	555	450 215
C/S	250	172 935

Although this type of processing is hardly recommendable when developing applications, it enables one to clearly understand how the various models operate.

There is little difference between the HTML-HTTP, Java RMI and IIOP models; they all offer fair network bandwidth consumption.

The ActiveX-DCOM model obtained a bad result. Depending on the parameter transfer method, various results are obtained. It is only through testing that one will find the best way to operate.

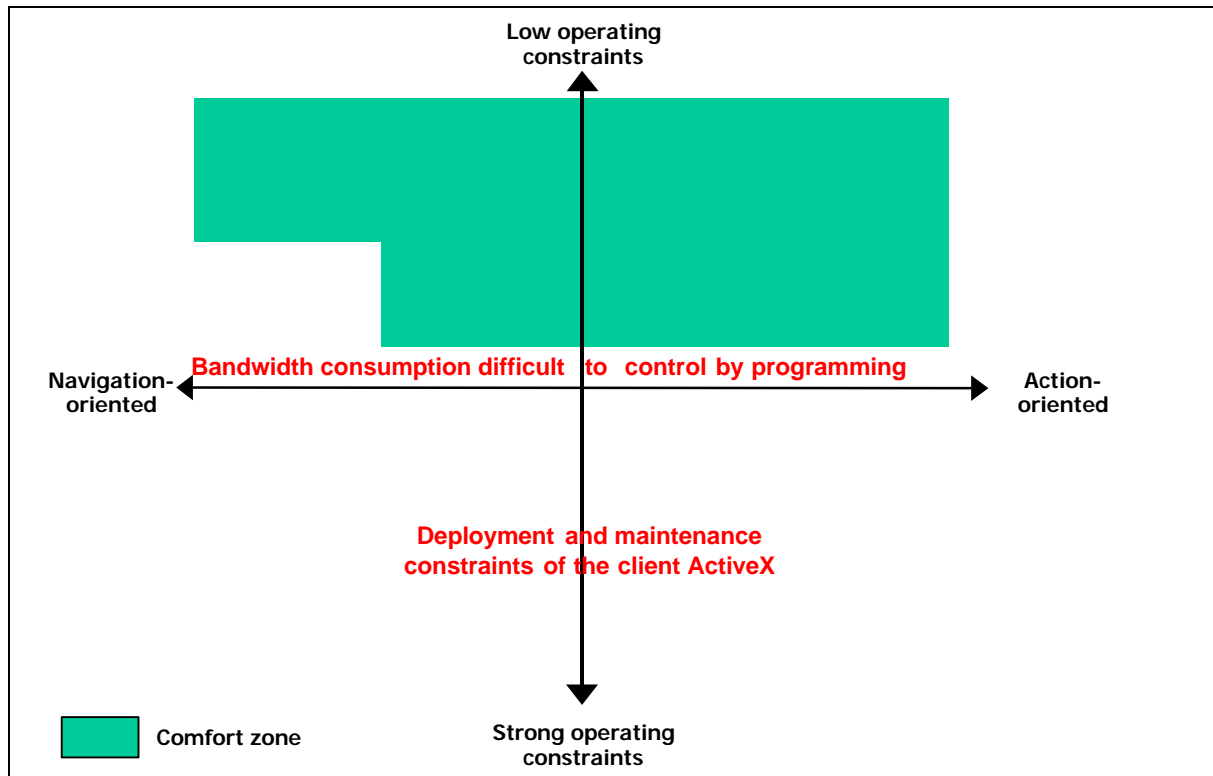
Lastly, the Client/Server model shows here one of its assets, notably regarding its direct communication capability with the DBMS. The application developed uses one of the optimization techniques suggested by Oracle (the Array Fetch) to retrieve the lines resulting from a SQL query. The results speak for themselves; the efficiency cannot be disputed.

3.3. Comfort zones

We determined the comfort zone of each of the architectures. The surface area covered by each model on a two-axis graph represents this comfort zone:

- The axis **Operation constraints** groups together deployment constraints (parameterization, installation, hardware and software configuration of client stations), maintenance constraints on client stations in case of modification of the server configuration, and network bandwidth consumption.
- The axis **Orientation of the application** expresses the nature of the application. Two metaphors linked to the graphical interfaces can be distinguished: the navigation-oriented and the action-oriented interface. A navigation-oriented interface is very simple to use, with applications that guide the user. An action-oriented interface involves a very strong interaction between the user and the interface, with sophisticated event-driven mechanisms.

3.3.1. ActiveX - DCOM

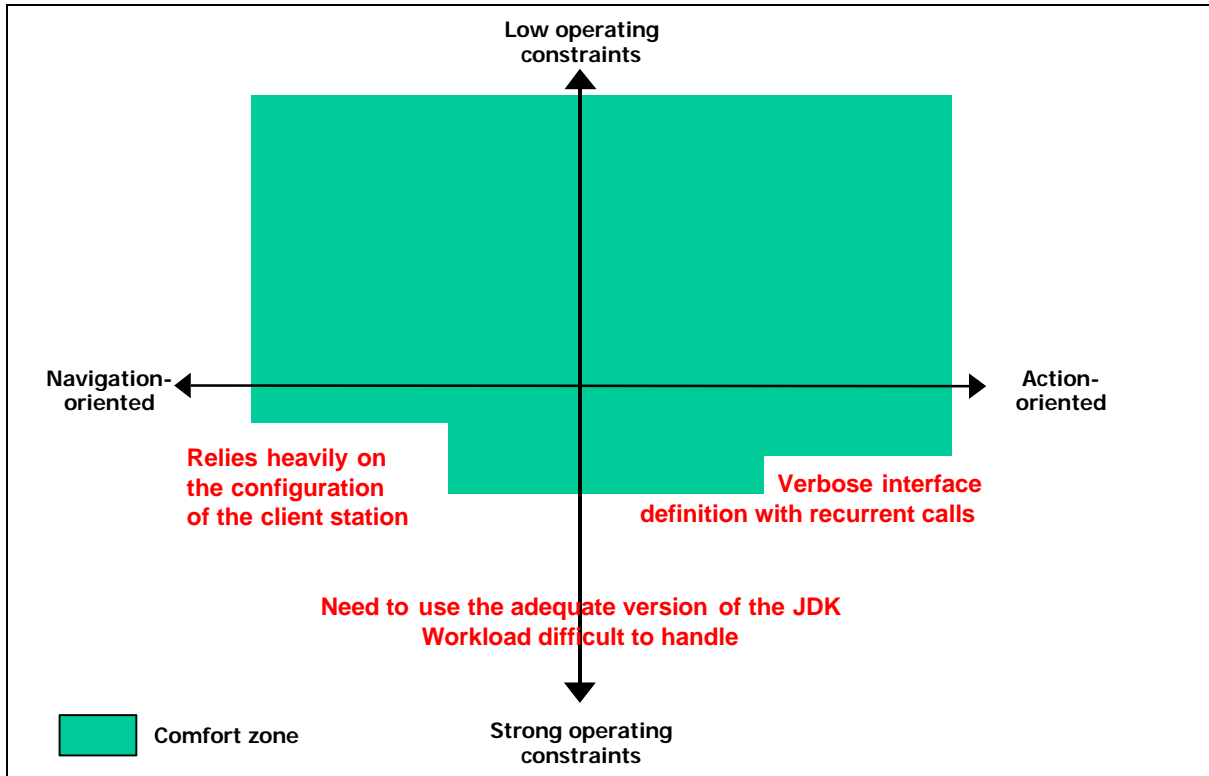


Comfort zone of the ActiveX-DCOM model.

- This model will not suit the environments with average operating constraints because:
- ActiveX clients are compiled components needing a specific operating system to run on (today, the only viable ones are the 32 bit Windows OS).
 - Installation of ActiveX requires the use of a specific (Microsoft) browser or a specific configuration (low reliability plug-ins).
 - The server must be referenced in the registry of each of the client stations, this thus renders maintenance difficult when the server environment is modified.

What's more, as it is difficult to control network traffic by programming, it is not easy to implement applications involving massive display.

3.3.2. Java - RMI



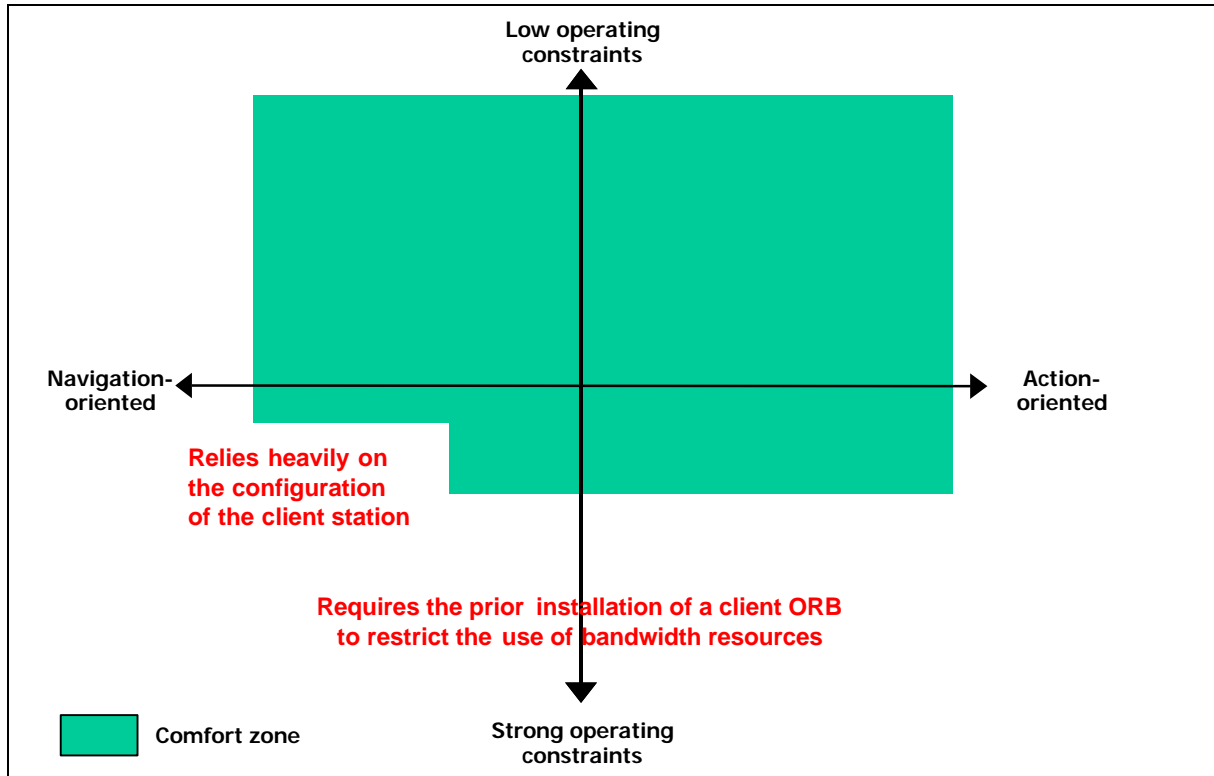
Comfort zone of the Java-RMI model.

Java-RMI is at ease with average operating constraints. However, when the operating constraints are high this model is not suitable, as the application deployment (performed at each initialization) requires an adequate version of a JDK to be pre-installed (the one of a browser, among others), and as the initialization stages are rather bandwidth-consuming. What's more, the RMI architecture limits workload capabilities.

Applications retrieving a lot of data will run into JVM performance problems on the client stations. For action-oriented applications performing recurrent calls, the systematic exchange of the interface definition generates useless traffic.

3.3.3.

Java - Corba

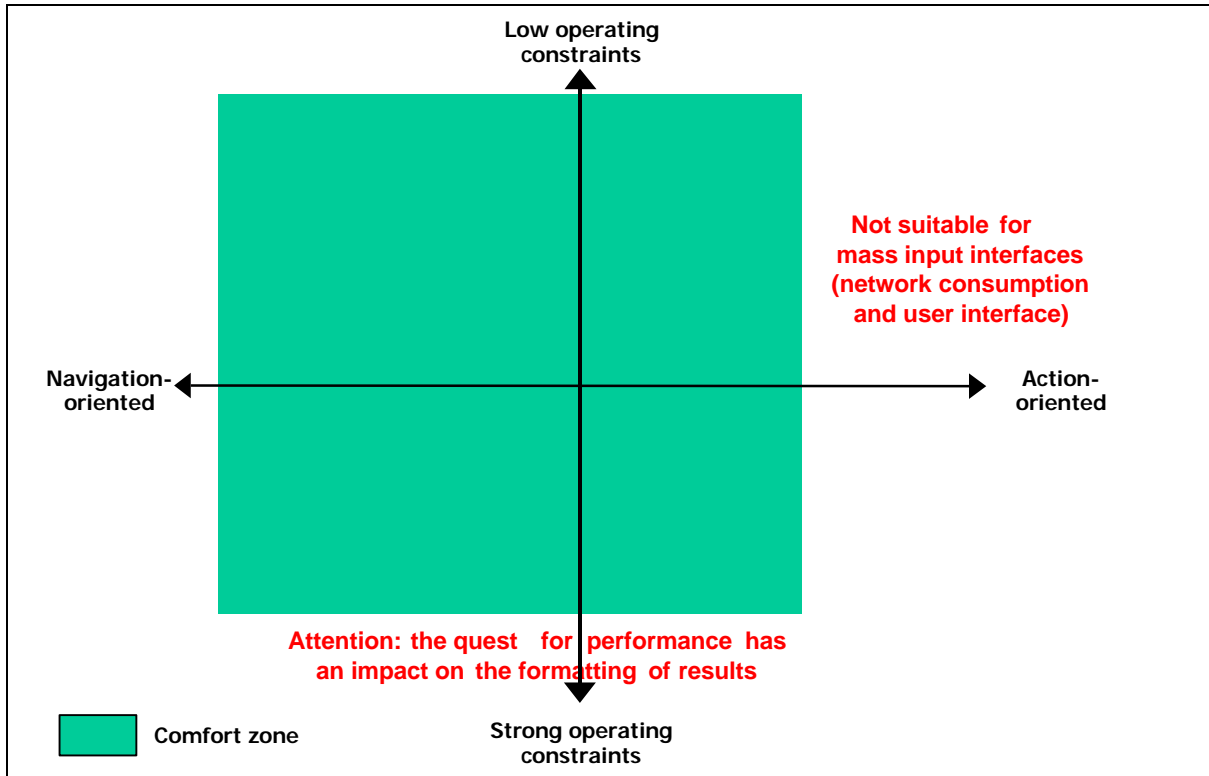


Comfort zone of the Java-IIOP model.

Java-IIOP is in its comfort zone when there are average operating constraints. However, the installation of the ORB on the client stations is almost mandatory if one wants to eliminate the traffic generated by its systematic loading.

As with RMI, the applications retrieving a lot of data are quite slow due to the JVM client's formatting performance.

3.3.4. HTML-HTTP



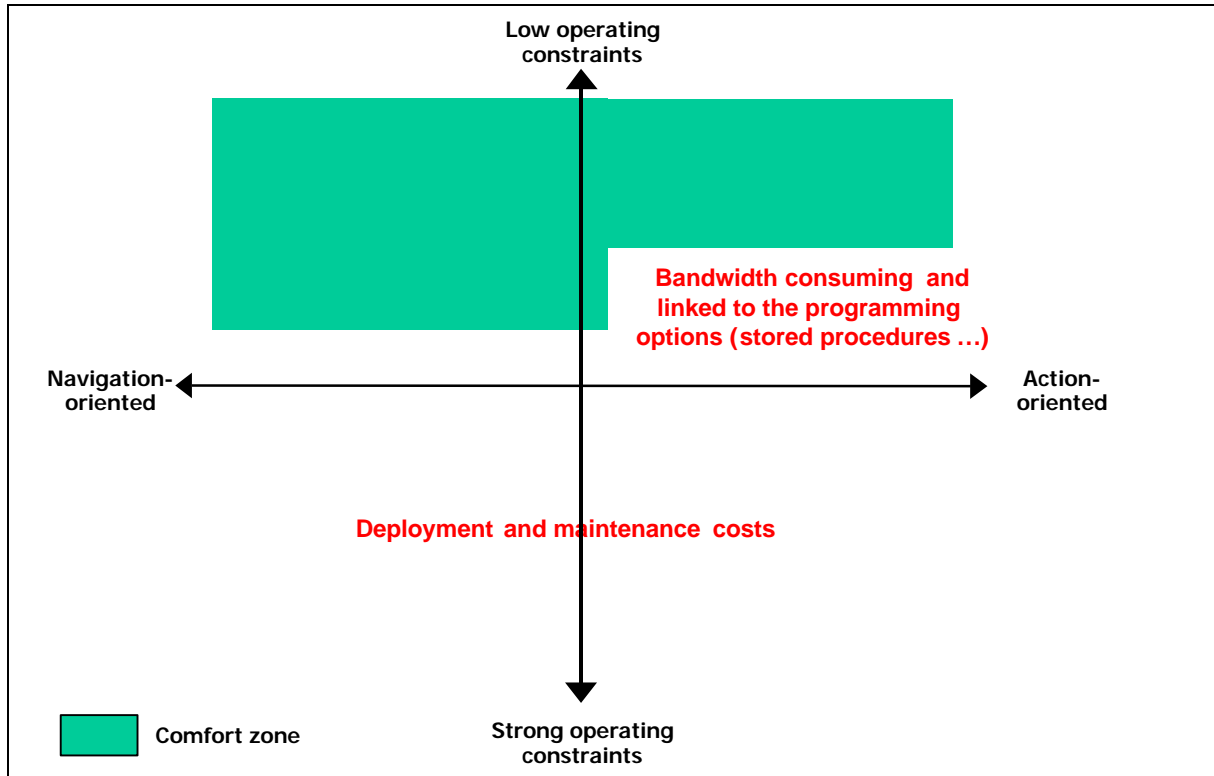
Comfort zone of the HTML-HTTP model.

HTML-HTTP is quite at ease, whatever the operating constraints. However, on low-traffic networks, the quest for performance will necessarily imply a simplification of the graphical interface.

If HTML-HTTP is very well suited to display-oriented applications and management type update applications, this model is not appropriate for mass modification applications. Indeed, in this case, the user interface is not productive (no keyboard shortcuts, no tables allowing input, etc.) and the consumption of network bandwidth is excessive (single-file input).

3.3.5.

Client/Server



Comfort zone of the Client/Server model.

The client/server model is extremely costly in terms of deployment and maintenance. Indeed, the configuration of the client station relies heavily on the server configuration, and application updates require once again deploying files on every single station.

Once these costs have been taken into account, this model behaves very well during massive displays, both at the network level and regarding the formatting time on the client station. However, network bandwidth consumption is not quite as good as with action-oriented applications. Furthermore, this consumption is strongly linked to programming options and the database middleware used.

4. Analysis of performance measurements

4.1. ActiveX and DCOM (MTS)

4.1.1. Deployment constraints

The application is deployed in the browser in the form of an ActiveX component. When the application is first used, the component will install itself physically on the user station. To do this, the browser will download an initial installation file (with the extension .CAB) which includes the component itself, the configuration data (which will be transferred to the workstation registry) and the references of the extra files that will be needed by the component to operate.

In this case, the ActiveX component was developed with Visual Basic. The component consists of an OCX file. It needs a few DLL to run (they are grouped in various CAB files, which are detailed below), as well as another OCX component, the one used to display the grid which shows the data. Concerning questions of security, ActiveX components and Java applets (which operate in an isolated, but restricted universe) have different strategies. Non-aggressiveness is validated by a certifying third party (Verisign, for example) which delivers a signed and authenticated certificate. As our component was not certified, we had to specify the lowest possible values for the security parameters of the browser in order to deploy it during the tests.

Once the files necessary for the operation of our component are downloaded, the elements used by our ActiveX component are referenced in the registry base using the logical name of the class (CLSID).









The client object reference (IntraperfClient.UCMylist) contains the

information needed by the browser to instantiate the ActiveX component. The component will then request the instantiation of the server component (IntraperfServer.ClassConnexion). The information about its actual location is stored at the level of the registry:

If it is stored locally, the registry provides the name of the corresponding DLL, if it is stored remotely, the registry gives the address of the computer on which the ActiveX server is really located.

Moreover, it is one of the strong constraints related to the use of ActiveX server components, because if one is to change the location of a component, one will have to update the registry of all the client stations that use it.



Windows 3.11				Windows 95, NT			
IE 3	IE 4	Net 3	Net 4	IE 3	IE 4	Net 3	Net 4
 32 bits ActiveX component	 32 bits ActiveX component	 32 bits ActiveX component	 32 bits ActiveX component			 Does not support ActiveX	 (Operates with the Ncompass plug-in)

4.1.2. Step 1: Deployment / Initialization

One of the features of ActiveX components is that they are physically and durably installed on the user station. We decided to measure and to analyze the installation and initialization stages separately.

■ Installation of the ActiveX components

Component installation begins by downloading an HTML page, which can be found within the body of the HTTP frame:

```
Server: Microsoft-IIS/4.0
Date: Sun, 10 May 1998 12:59:43 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Sun, 10 May 1998 12:59:43 GMT
ETag: W/"b04539c14c7fbd1:b7b"
Content-Length: 706
<HTML>
<OBJECT ID="UCMyList" WIDTH=503 HEIGHT=287
CLASSID="CLSID:9791EE38-CC92-11D1-8EF1-0008C728C347"
CODEBASE="IntraperfClient.CAB#version=1,0,0,0">
</OBJECT>
</HTML>
```

The various elements needed for proper operation are downloaded, then installed locally:

Type of request	Protocols	Nb frames	Frame size (bytes)	Actual size of the CAB (bytes)
HTML page				
	TCP	5	300	
	HTTP	2	1492	
CAB 1				18269
INTRAPERCLIENTDCOM.CAB	TCP	7	420	
	HTTP	13	19406	
CAB 2				88034
MSRACLI.CAB	TCP	32	1920	
	HTTP	65	91808	
CAB 3				7726
REMOTEREG.CAB	TCP	3	180	
	HTTP	6	8316	
CAB 4				103211
MSFLXGRID.CAB	TCP	50	3000	
	HTTP	78	112068	
CAB 5				690589
MSVBVM50.CAB	TCP	303	18180	
	HTTP	481	718222	
Total		1045	975312	907829

■ **Initialization and launching of the application**

Time on the client side:

Browser loading time: 3 seconds

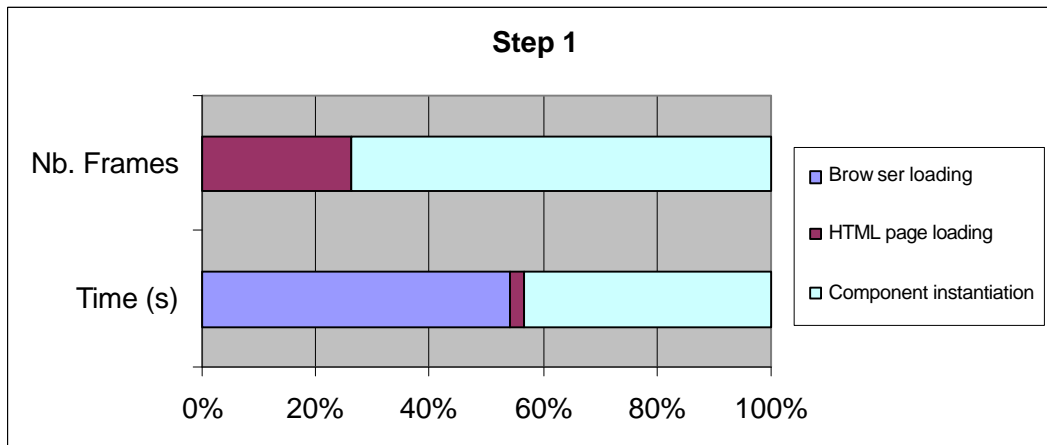
HTML negotiation: 0.15 of a second

Instantiation of the ActiveX component and connection to the ActiveX server: 2.4 seconds

Network load: 19 frames - 4 172 bytes

This initialization includes, on the client side, browser launching, instantiation of ActiveX through an HTML page, and the connection to the ActiveX server, via DCOM.

It can be broken down as follows:



The above diagram shows the amount of time used up by each task in terms of network consumption and time.

The number of frames is rather low, no optimization can be considered at this stage. Concerning the global response time at the client level, the most sensitive part is the instantiation of the ActiveX component. At this stage, the only possible optimization would consist in developing the component with Visual C++, instead of Visual Basic.

Break down of the step

Browser loading takes 3 seconds. The server is not solicited during this stage.

Frame 1 to 4: The first request received by the server concerns an HTML page over port 80, the one of the Web server, after the TCP/IP negotiation (3 negotiation frames)

Frame 5: The page requested is immediately returned by the Web server. This page contains information concerning the ActiveX component to be instantiated. The ActiveX component is loaded in memory by the browser. Its instantiation takes about 0.2 of a second. The HTML page contains the following information:

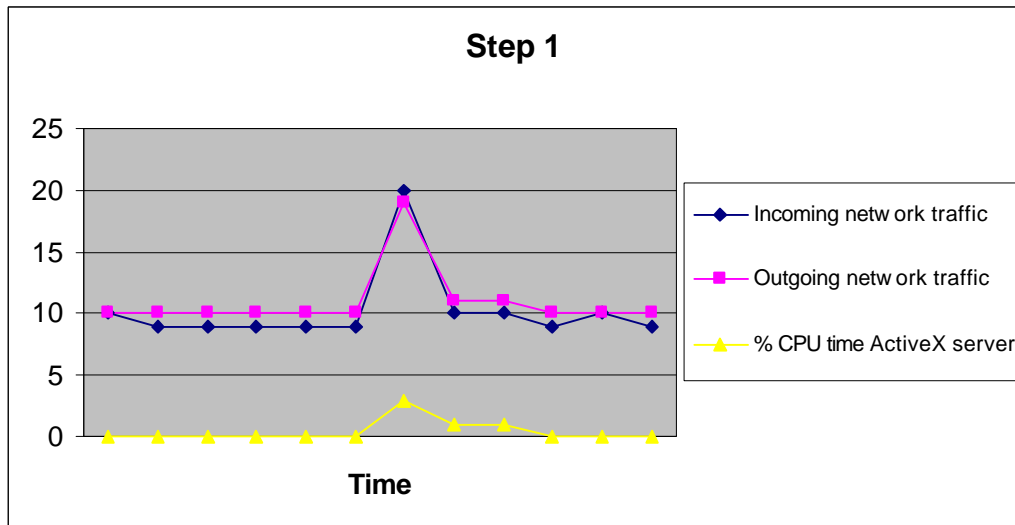
```

Server: Microsoft-IIS/4.0
Connection: keep-alive
Date: Tue, 28 Apr 1998 22:43:58 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Tue, 22 Apr 1998 5:00:55 GMT
Content-Length: 182
<HTML>
<TITLE>ActiveX DCOM</TITLE>
<BODY>
<OBJECT ID="UCMyList" WIDTH=503 HEIGHT=287
CLASSID="CLSID:9791EE38-CC92-11D1-8EF1-0008C728C347"
CODEBASE="IntraperfClientDCOM.CAB#version=1,0,0,0">
</OBJECT>
</BODY>
</HTML>
    
```

Frame 6 to 19 (≈ 2.4 seconds): This is where the connection with the remote ActiveX server takes place. After this exchange, a physical connection is open between the client ActiveX component and the remote server. This connection is established over TCP/IP communication ports (for example, 1727 on the server side and 1742 on the client side, but these ports may vary between two sessions.)

During this step, the exchanges over the network are dedicated to the establishment of a connection between the component and the ActiveX server. The following diagram represents the development of these operations, as seen from the server side.

The percentage of CPU time taken up by the ActiveX server increases when the client sends a connection request.

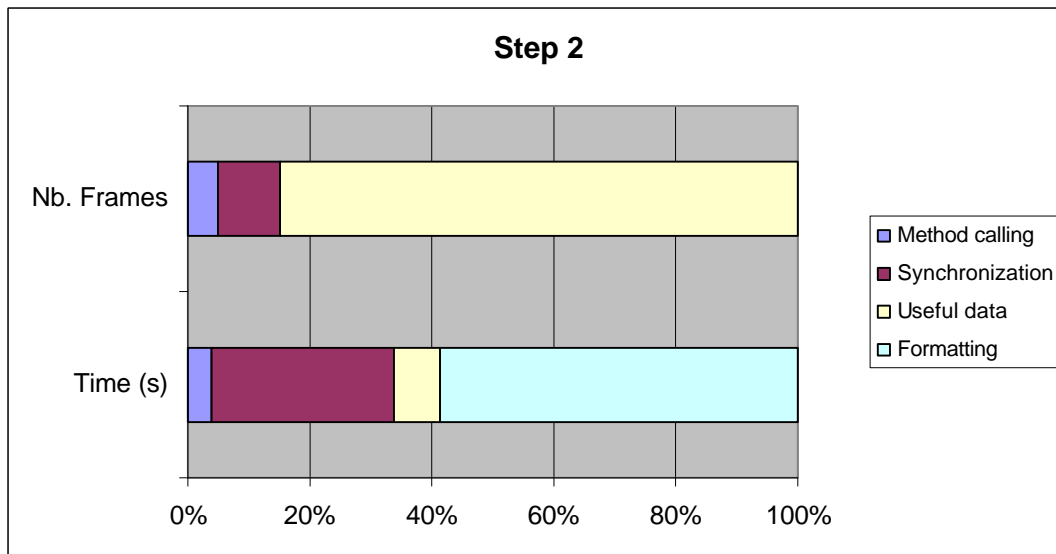


4.1.3. Step 2: Retrieval of 50 lines

Time on the client side: 2.6 seconds

The processings on the server side, including the exchanges on the network, last 1.2 seconds. The rest of the time is used by the ActiveX component to format the result in the table.

Network load: 20 frames - 14 880 bytes



One can distinguish four types of frames:

- 1 frame for DCOM method calling (request).
- 2 frames for "ping/working". The first one (ping) is sent by the client and inquires about the status of its request. The second is sent by DCOM while the ActiveX server processes the request.
- 13 data exchange frames, one may note that the result frames are well filled in (1048 bytes).
- 4 acknowledgment frames sent by the client (around 1 acknowledgment frame for each 3 data frames sent).

During this step, the overall processing time is fairly well distributed between the client and the server. Choosing to use a sophisticated grid to format the data has a strong impact on client-side response time. This type of component has very rich formatting capabilities, but does not perform extremely well. The graph shows what proportion of time is dedicated to the formatting tasks during this step (about 60%).

Break down of the step

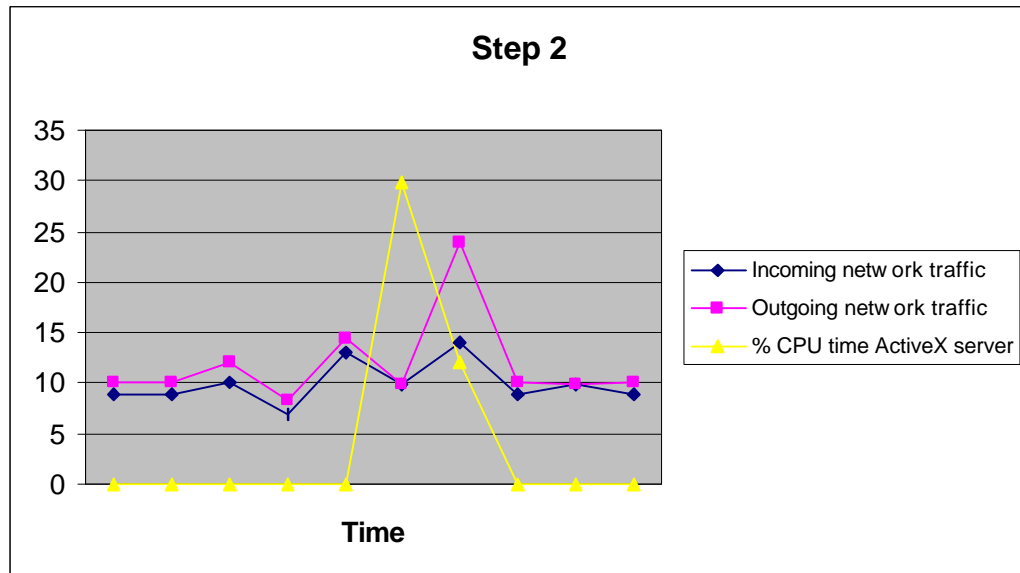
Frame 1: The ActiveX component requests the remote method operation, using a communication port negotiated during the establishment of connection to the server (in step 1).

Frames 2-3: Roughly one second after the method call (first frame), the ActiveX component performs a "ping", to inquire about the status of its request, DCOM answers that the ActiveX server is busy ("Working").

Frames 4 to 6: The ActiveX server begins answering the request. Frames 4 and 6 do not contain any useful data. These are only formatting frames. Frame 5 is an acknowledgement.

Frames 7 to 20: 11 frames are sent by the server (1048-byte frames all filled with useful data, except the last one) and three frames are sent by the client (152-byte acknowledgement frames).

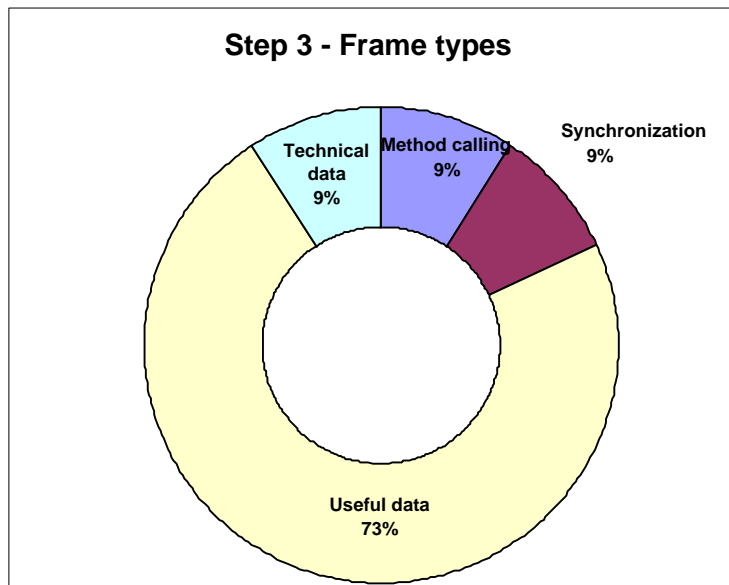
On the server side, the moment when the ActiveX server receives the service requested is easily distinguished (peak of the curve % CPU time ActiveX Server). As soon as the ActiveX server has finished its DBMS query, it returns the result to the client. This corresponds to the peak of the curve " Output network traffic " .



4.1.4. Step 3: Retrieval of 5*50 lines

Time on the client side: 1 second per fetch, 5 seconds in total

Network load: 55 frames - 67 650 bytes



During this step, the four types of frames in the above diagram are exchanged. One frame corresponds to each method call. For each result returned by the server, there is a technical data frame (object references, associated rights, etc.) and eight useful data frames. Regarding the technical data, little information is available, all the more so since only a very small part travels unscrambled. The useful data (data coming from the application server), however, is not coded and thus easier to read. Lastly, for each set of result frames (technical data and useful data) there is an acknowledgement frame (synchronization).

The resulting frames of this step are very well filled in, as they are all roughly 1476 bytes.

Something seems paradoxical when we compare with the previous scenario (operation of the query and retrieval of the first 50 lines): in principle, there should be the same number of frames and an identical filling rate for data retrieval. However, this is not the case. When the following 50 lines are requested, the exchanges are optimal while for the first 50 lines the exchanges were not as good.

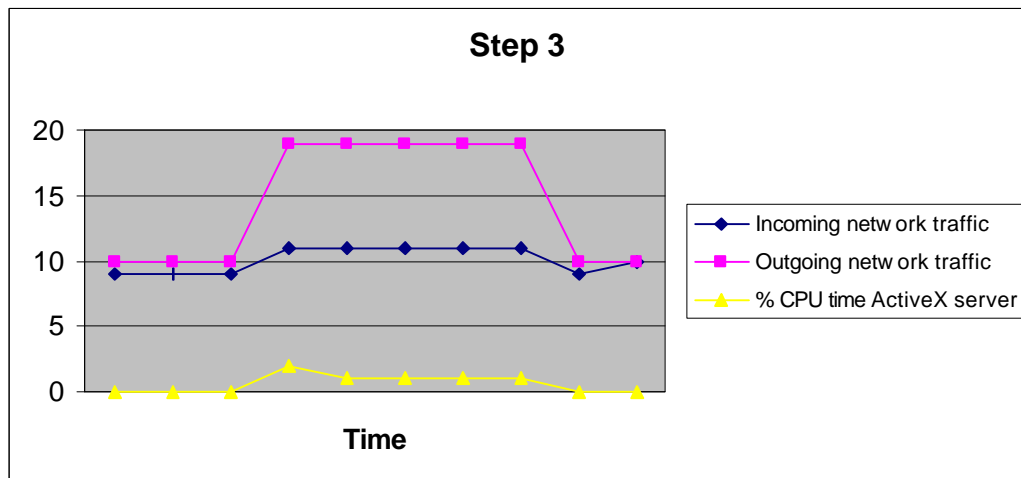
The table below compares the part common to steps 2 and 3:

Retrieval of 50 lines	Step 2	Step 3
Name of the method (parameter and return variable)	.PartialProductList (nbvalues As Long) As Variant	.FollowingList (nbvalues As Long) As Variant
Total number of frames	17	10
Number of system data frames	2	1
Number of useful data frames	11	8
Average size of useful data frames (bytes)	1028	1476
Number of synchronization frames (acknowledgement)	4	1

The only difference between the two scenarios is that two methods with different names are called. However, these two methods operate according to a same model, that is, one input parameter (an integer, which specifies the number of lines, wanted) and an output table (the table's size is dynamically specified by the server).

Therefore, even if in principle one codes along the same lines, the communication may not be identical. In this case, this situation has very little impact on the response times. The simplicity of using DCOM communication with Visual Basic (the development tool used for these tests) presents a drawback: its optimization is impossible.

From the server-side, the sending of lines to the client can be seen on the curve showing the output traffic. One may also note that the activity of the ActiveX server is rather low. Indeed, the result of the query was entirely built during the previous step. Here, each time the method ".FollowingList " is called, the ActiveX server only sends back to the client, a portion of the resulting table that it keeps in memory.



4.1.5. Step 4: Insertion of 50 lines

Time on the client side: 0.2 of a second per insert, 10 seconds in total.

In fact, the server side processing corresponding to the insertion query lasts only 0.013 of a second. The rest of the time is used for information refresh and the input of data at the level of the ActiveX component in the browser.

Network load: 112 frames - 78 008 bytes

One can distinguish two types of frames:

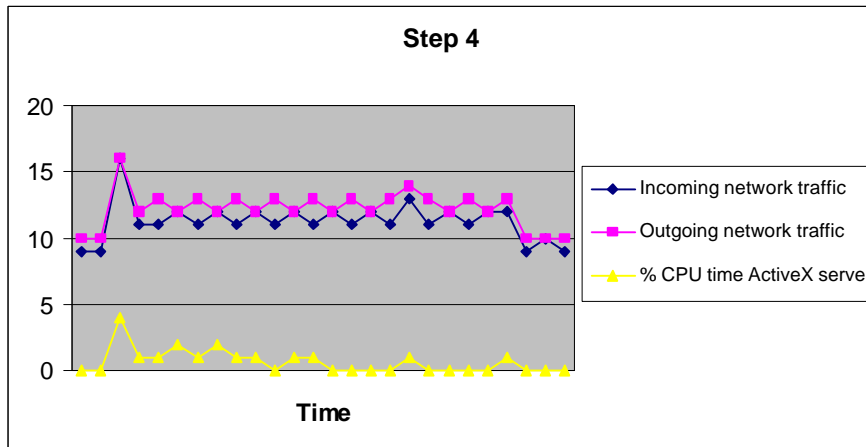
- the method calling frames
- the result frames.

Break down of the step

Frames 1 to 12 (about 0.2 of a second): these exchanges correspond to the initialization of the input interface. Several method calls are performed to fill in the lists from which the automated testing script will pick up the values.

Frames 13 to 112 (about 10 seconds): these 100 frames go by pairs, one remote method calling frame, followed by a result frame, with the status of the request (success or failure). Between the request and the answer, only 0.013 of a second goes by, during which the ActiveX server sends six Insert type SQL queries to the DBMS.

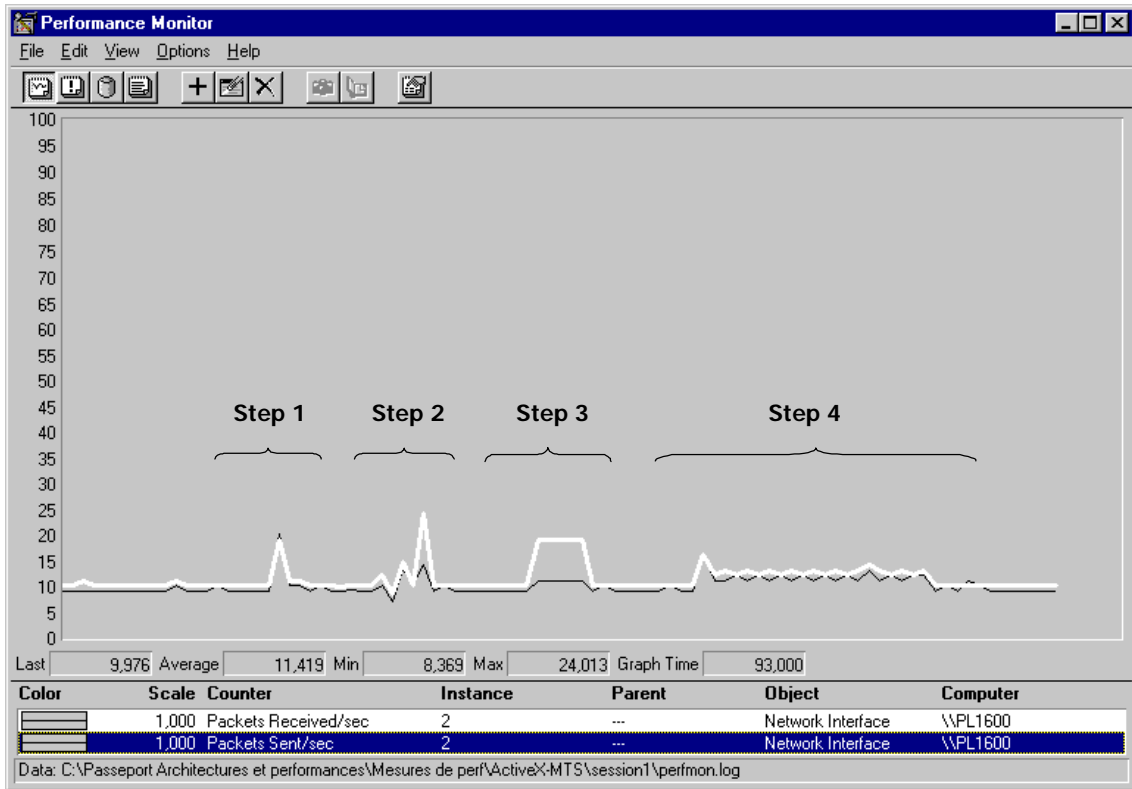
The first exchanges concerning the initialization of the input interface, in which the client requests three value lists, can be seen on the server side. This corresponds to the first peak of the network traffic curves, as well as the peak of the curve % CPU time taken by the ActiveX Server. Afterwards, the network exchanges remain constant during the entire value insertion stage. Similarly, the activity of the ActiveX server remains low during this stage that shows repeated method calls.



4.1.6. Step 1 to 4: Network utilization

This diagram shows the utilization of the network, as seen from the server. The bold curve represents the traffic from the server to the client. The other curve represents the traffic flowing in the opposite direction.

The measurement unit is the number of frames exchanged between the client station and the server per second.



During step 1, one can clearly see the initialization of the application, when the ActiveX control connects to the server. During step 3, which consists in retrieving five series of 50 resulting lines, one may note the proportion of method calls (lower curve) and the sending of the result by the ActiveX server (upper curve). Lastly, during step 4, "remote service call"-type dialogue shows a very steady consumption of network bandwidth.

On the whole, the network bandwidth consumption is reasonable, except during component download and installation (this step is not represented on this diagram). Relying on DCOM, the network traffic between the component and the application server stays at an acceptable level.

The main drawback, however, is that the volume of this traffic is not predictable, as shown in the comparison between the method calls of steps 2 and 3. DCOM is a rather rich, but obscure protocol for the developer.

4.1.7. Step 5: Retrieval of 4 000 lines

The processing time needed on the server side to run and answer the SQL query is very short: 1.5 seconds. The transfer time of the information over the network is also very short: 1.1 seconds. On the other hand, the formatting time on the client side is considerable: about 85 seconds were needed on our client computer (Pentium 266 MHz - 64 Mb RAM) to display all the data and hand control over to the user.

Network load: 1211 frames - 1.124.756 bytes

Two lessons can be learned from this step:

- The formatting of a large number of lines (even if this type of programming is to avoid) is very costly for ActiveX components, even more when one uses sophisticated graphical objects to display them (in our test, a grid). This formatting time is highly dependent on the client station.
- A relatively large volume of data is transferred. The "technical" encapsulation of the information is strong. However, this is the developer's only solution should he wish to receive the information "safe and sound". Also, there will be no opportunity to optimize with a 4GL at the level of Visual Basic (used for this test). Developing in C++ provides greater latitude, but is limited by the basic operation of DCOM.









In short, the 3 sub-steps are:

- *The calling of the remote method and the processing on the server side:* the results obtained are quite satisfactory. They only depend on the server station.
- *The data transfer stage:* this stage can hardly be compressed, as if DCOM mechanisms are easy to use, they also are quite opaque. This is accentuated by the use of the Visual Basic 4GL. The development of components under Visual C++ offers more opportunities, but it is also more complicated.
- *The formatting stage on the client station:* this stage depends entirely on the power of the client station.

4.2. Java RMI

4.2.1. Deployment constraints

The application is deployed in the browser in the form of applets. The first deployment constraint is the availability of a browser compatible with JDK 1.1.x. Indeed, RMI appeared as a result of this JDK version. In theory, this is the only limitation, and the applets should be able to run properly, no matter the operating system of the client station.

Windows 3.11				Windows 95, NT			
IE 3	IE 4	Net 3	Net 4	IE 3	IE 4	Net 3	Net 4
 JDK 1.0	 Bad RMI implementatio n	 JDK 1.0	 Bad AWT implementatio n	 JDK 1.0		 JDK 1.0	

In fact, during deployment, we encountered problems on the type 4 browsers in the Windows 3.11 environment. Indeed, the IE4 version we used prohibits RMI communication (this is a bug). The release of Netscape 4 used in Windows 3.11 did not allow us to correctly instantiate our graphical applet listing products (this is an implementation problem of the AWT).

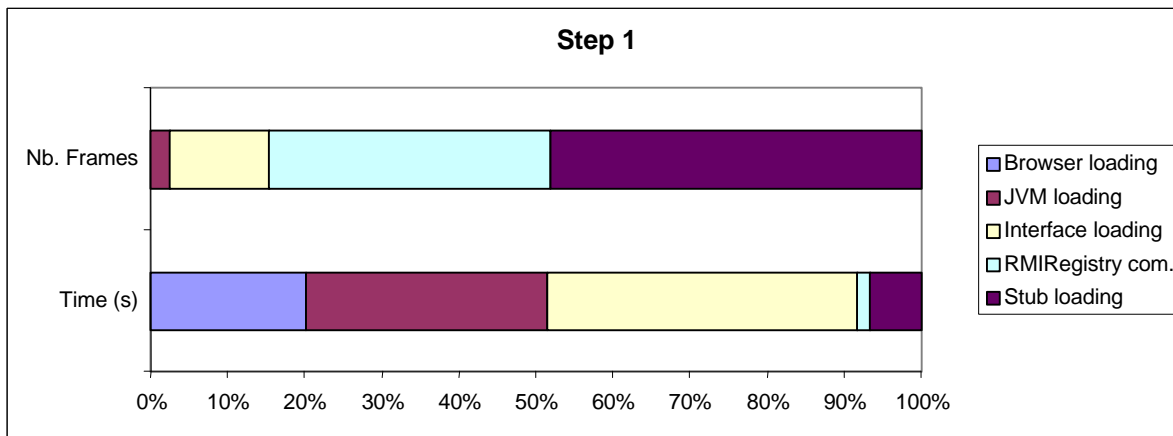
4.2.2. Step 1: Initialization / Deployment

The operation of RMI does not require any preliminary deployment. All the communication elements will be downloaded during the applet call.

Time on the client side: 15 seconds

This initialization includes, on the client side, browser launching, loading of the graphical applet through an HTML page, and connection to the RMI server.

Network load: 195 frames - 38 194 bytes

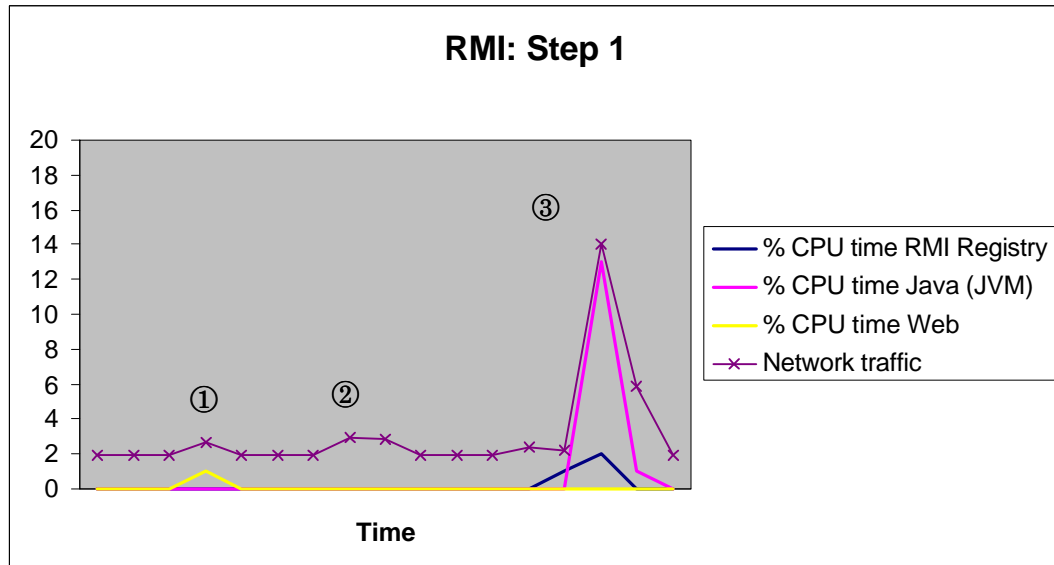


The above diagram shows the amount of time used up by each task in terms of network consumption and time. This analysis enables us to break these tasks up into three categories:

- *The incompressible tasks, not linked to the network:* the tasks performed on the client station belong to this category (loading of the browser and launching of the JVM).
- *The hardly compressible tasks, moderately linked to the network:* interface loading uses the network only moderately, but takes up a large part of the total time. Indeed, the instantiation of the graphical applets uses up client resources. The only way to compress this task is to optimize the programming of the graphical applets.
- *The incompressible tasks, strongly linked to the network:* the two RMI tasks belong to this category (establishment of the communication with the RMIRegistry and stub loading). As they generate a lot of network frames, these tasks may be quite slow on a low-traffic network.

In conclusion, this step depends little on the programming options chosen. The main factors of efficiency are network flow and the power of the client station.

The following diagram shows the measurements taken on the server side during Step 1:



Break down of the step

Browser loading takes 3 seconds. The server is not solicited during this stage.

Frame 4 (≈ 3 s): The first request received by the server concerns an HTML page over port 80, the one of the Web server, after the TCP/IP negotiation (①).

Frame 5 (≈ 3 s): The Web server immediately returns The page Web server. From then on, the server will only be solicited to provide Java applets.

```
Server: Microsoft-IIS/4.0
Connection: keep-alive
Date: Tue, 28 Apr 1998 21:23:35 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Tue, 21 Apr 1998 10:20:33 GMT
ETag: "9066721ef6dbd1:1a94"
Content-Length: 370
<HTML>
<TITLE>HTML test page</TITLE>
<BODY>
<APPLET CODE      = "Client.class" ...
<PARAM NAME="RMISERVER" VALUE="pl1600.sqli.fr">
</APPLET>
</BODY>
</HTML>
```

Frame 7 (≈ 7.5 s): The client requests the `Client.class` applet. This request takes place 4.7 seconds after the previous frame. This period corresponds to the launching of the JVM on the client station, which comes in addition to the loading of the browser (②).

Frame 30 (≈ 12.2 s): All the applets of the graphical interface have been retrieved on the client station.

Frame 31 (≈ 13.8 s): The graphical interface applets are instantiated on the client station, and a first request is directed towards RMIRegistry (port 1099, by default) (③).

Frame 66 (≈ 14 s): RMIRegistry indicates that the stub `PasseporpPerfDisp_stub.class` is needed. A dialogue over port 80 then begins, with the aim of loading this stub. Starting with frame 72, the RMI exchanges between the client station and the server are performed on a session port assigned by RMIRegistry (here, port 2013). The server is solicited, as seen by the jump in the curve representing the activity level of the RMI server (driven by the JVM).

Frame 108 (≈ 14.5 s): New communication starts with RMIRegistry (port 1099).

Frame 146 (≈ 14.7 s): RMIRegistry indicates that `PasseporpPerf_stub.class` is needed. A dialogue over port 80 then begins. The client applet communicates in RMI with our server, always over port 2013. In fact, from this moment on, it will use only this port for the dialogues of the next steps.

Frame 195 (≈ 15 s): The initialization stage is finished.

4.2.3. Step 2: Retrieval of 50 lines

To summarize, both server-side and network side performance is excellent during this step. The bottleneck is the client station, with an average performance.

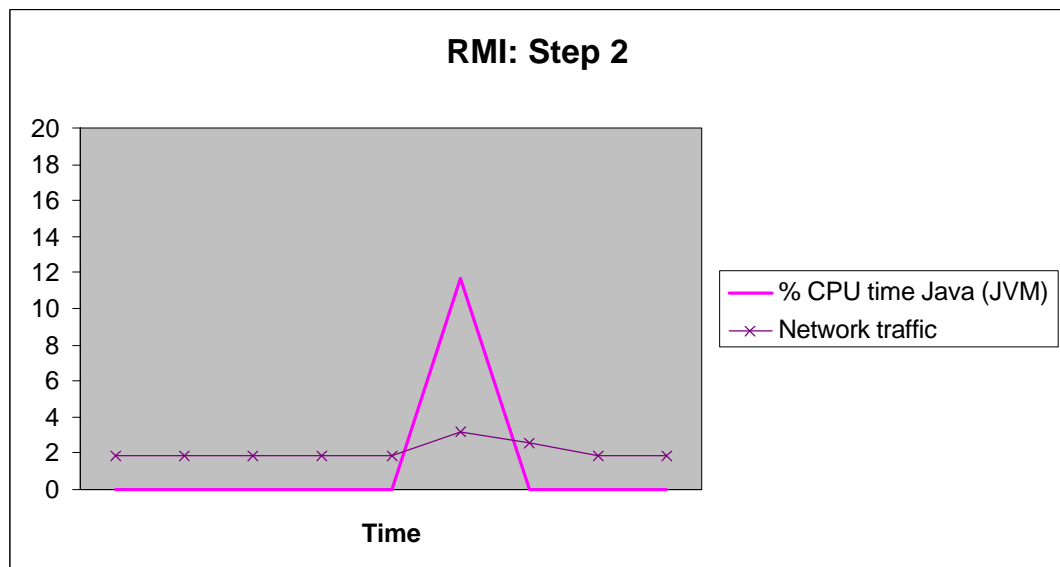
Time on the client side: 1.9 seconds

In fact, the server side processings last only 0.443 of a second. The rest of the time is used by the browser to format the result and to instantiate the business objects.

Network load: 20 frames - 9 588 bytes

One can distinguish four types of frames:

- A frame for RMI method calling.
- A few frames for exchanging business objects in relation to our programming choice.
- A few frames describing the objects exchanged. These frames correspond to an IDL description of the objects used in the interface of PasseportPerf.
- Data exchange frames with the RMI protocol. Among the features of these frames is a satisfactory filling rate (for a maximum of 1024 bits) and the generation of a few synchronization frames (around two frames out of three contain data during the dialogue).



As shown in this diagram, only the RMI-server processing uses up CPU time (apart from the database). The RMI Registry and Web server processing no longer have a role in communication.

Break down of the step

Frame 1: The browser requests the class `Product.class` (business object) and the server returns it in a single frame.

Frame 3: The client calls the method `GetProduct`. This request is performed over session port 2013.

Frames 5 and 6: The server sends a number of descriptions from the objects or data exchanged. All dispatching is prefixed with the IDL keyword. Example of the Product object:

```
IDL
Cat: Category
Manuf: Manufacturer
Designation: java/lang/String Availability:
java/lang/String
PartNumber: java/lang/String
PurchasePrice: java/lang/String
RetailPrice: java/lang/String
Dea: java/util/Vector;
```

Frame 7: From this frame, the server returns the data.

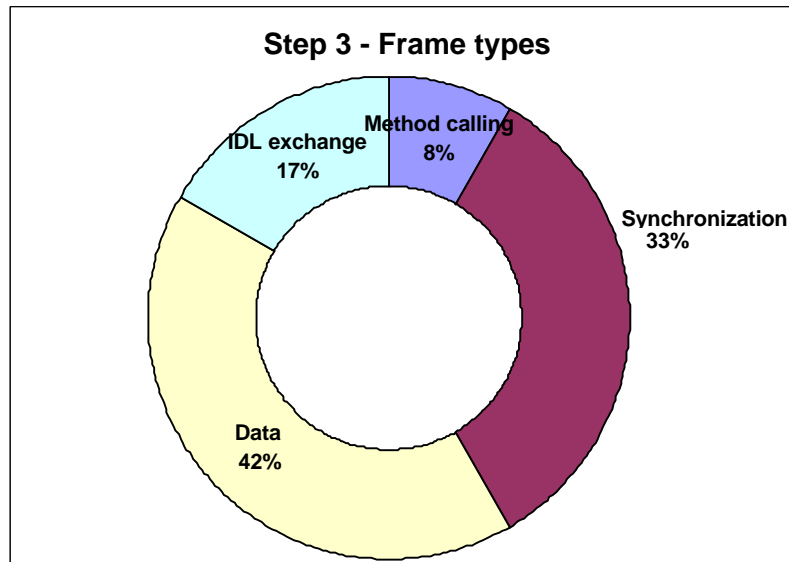
Frame 15: The browser requests two new classes, `Category.class` and `Manufacturer` (business objects), and the server returns them in a single frame.

Frame 20 (= 0.443 s): The transfers are finished.

4.2.4. Step 3: Retrieval of 5*50 lines

Time on the client side: 1 second per fetch

Network load: 60 frames - 30 025 bytes



During this step, the four types of frames in the above diagram are exchanged. RMI does not fill the frames to their maximum capacity. Indeed, no frame contains more than 1024 bits of data. Besides, if the RMI method calls require little bandwidth, the protocol consumes more synchronization requests and, above all, more IDL exchanges. Indeed, for each method call performed, two IDL frames are sent by the client station.

4.2.5. Step 4: Insertion of 50 lines

Time on the client side: 0.7 of a second per insert

In fact, the server side processings only last 0.443 of a second. The rest of the time is used by the browser to format the result and to instantiate the business objects.

Network load: 260 frames - 65 063 bytes

One can distinguish four types of frames:

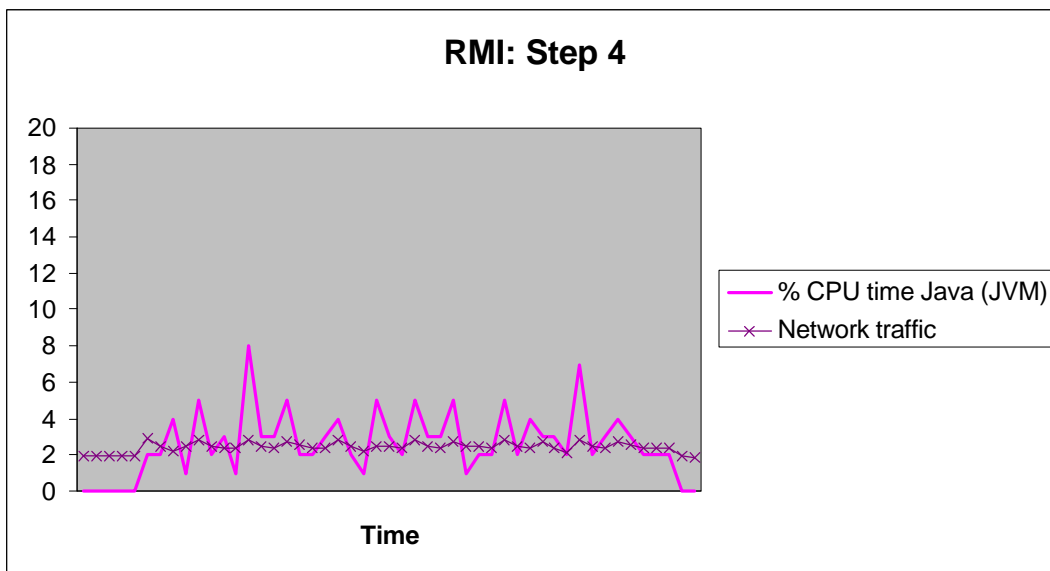
- A frame for RMI method calls.
- A few frames for exchanging business objects in relation to our programming choice.
- IDL exchange frames related to the operation of the RMI protocol.
- Data exchange frames over the RMI protocol. The features of these frames include a satisfactory filling rate (for a maximum of 1024 bits) and generation of a few synchronization frames (around two out of three frames contain data during the dialogue).

This architecture is efficient for performing inserts. Indeed, the client station only has to send the data through method calls. This reduces the volume of information exchanged. What's more, RMI makes the transfer of objects as values possible, which makes for clean and simple programming, as shown in our example:

```
Server.insertProduct(tmpProduct)
```

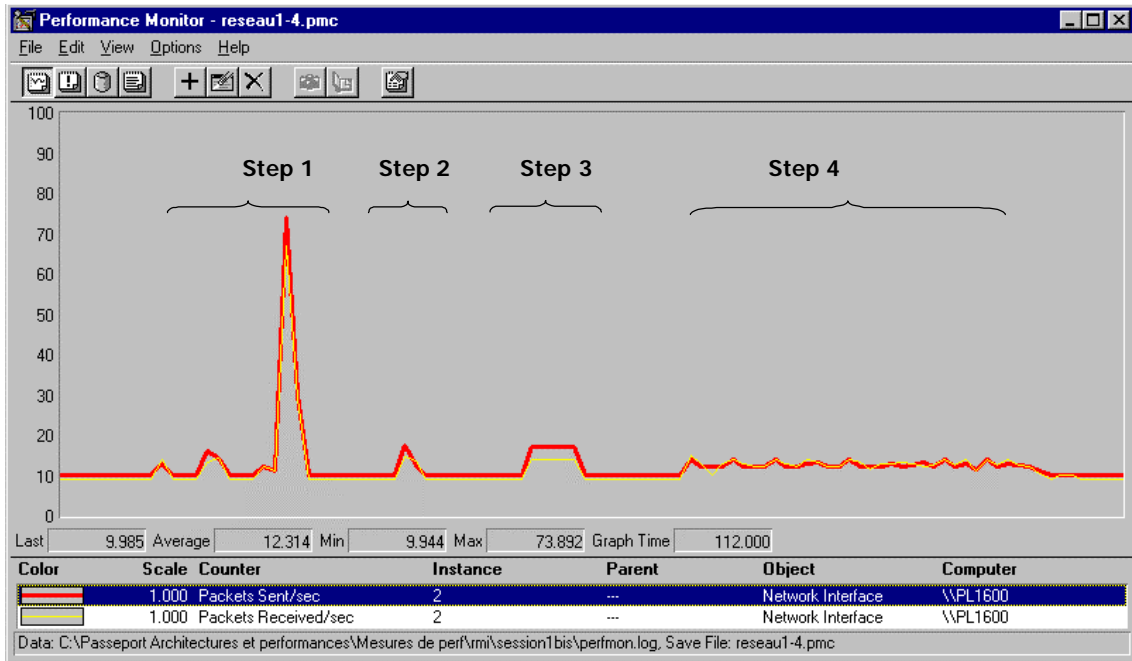
where tmpProduct is an instance of the Product object.

However, the protocol is rather verbose. The main reason being that the IDL is systematically exchanged each time a method call is made.



4.2.6. Step 1 to 4: Network utilization

This diagram shows the utilization of the network, as seen from the server. The bold curve represents the traffic from the server to the client. The light curve represents the traffic flowing in the opposite direction.



One can see that for steps 1 and 4, incoming traffic uses up approximately the same amount of bandwidth as outgoing traffic. Logically, the fetches (step 2 and 3) generate more outgoing traffic than incoming traffic on the server (even if the latter remains high, in particular with IDL exchanges). The initialization step 1 clearly consumes the most bandwidth. Apart from this step, the protocol uses little bandwidth. However, compared to similar architectures, RMI is especially verbose during exchanges from the client to the server (inserts, for example).

4.2.7. Step 5: Retrieval of 4 000 lines

The server side processing is very short, only 3.6 seconds, in comparison with the network transfer time and client-side formatting time.

Two lessons can be learned from this step:

- The formatting of a large number of lines (even if this type of programming is to be avoided) is very costly for Java applets. Indeed, between the end of the data retrieval on the client station and the applet refresh, there was a wait of more than three minutes on our client station (Pentium 266 MHz - 64 Mb RAM). This formatting time heavily depends on the client station (on a Pentium 90 - 64 Mb RAM, there is a wait of more than ten minutes). The browser is also a factor, as Netscape 4 is 40% slower than IE4 on the same computer.
- The data exchanges and their duration depend on the programming options chosen and the browser used. This application implements the serialization mechanism, which enables the transfer of an object by value (in our case, a product object vector). In fact, browser JVMs react differently to this mechanism. Transfer time is multiplied by 15 with Netscape 4 compared to IE4, while the number of frames is around 20% higher.

In short, the 3 sub-steps are the following:

- *The RMI connection stage and processing on the server side:* the results obtained are quite satisfactory, they only depend on the server station.
- *The data transfer stage:* This stage depends enormously on the programming options (serialization of objects or passing of parameters), as well as on the browser used (IE4 behaves better than Netscape 4 concerning serialization).
- *The formatting stage on the client station:* this stage heavily depends on the power of the client station. The choice of the browser is also important.

4.3. Java IIOP (Corba)

4.3.1. Deployment constraints









The application is deployed in a browser in the form of applets. Graphical classes of the Java JDK 1.1.x were used for development. Deployment is therefore limited to browsers supporting this JDK version.

For the Corba client, we used Visibroker ORB, from Visigenic. There are two deployment possibilities:

- The first one consists in deploying the three Java archive files (.JAR) on the client stations, and specifying each of these files in the CLASSPATH environment variable:
CLASSPATH=vbjapp.jar,vbjorb.jar,vbjtools.jar
- The second one consists in adding the ARCHIVE=vbjapp.jar, etc ... keyword inside the APPLET tag of the HTML page. The Java classes will then be loaded during initialization.

Note: the Netscape 4 browser is supposed to provide the Visigenic files during its installation procedure. In fact, we did not succeed in using them - probably because the version used on the server side differs from the one provided by Netscape.

The table below summarizes the results obtained:

Windows 3.11				Windows 95, NT			
IE 3	IE 4	Net 3	Net 4	IE 3	IE 4	Net 3	Net 4
 JDK 1.0	 ?	 JDK 1.0	 Bad AWT implementation	 JDK 1.0	 ORB to be deployed	 JDK 1.0	

During deployment, we encountered problems on the type 4 browsers in the Windows 3.11 environment. As with the RMI tests, the Netscape 4 version used in Windows 3.11 did not allow us to adequately instantiate our graphical applet listing products (this is an implementation problem of the AWT). Concerning IE4 in Windows 3.11, there was an unidentified, serious bug, which prevented the application from running.

4.3.2. Step 1: Initialization / Deployment

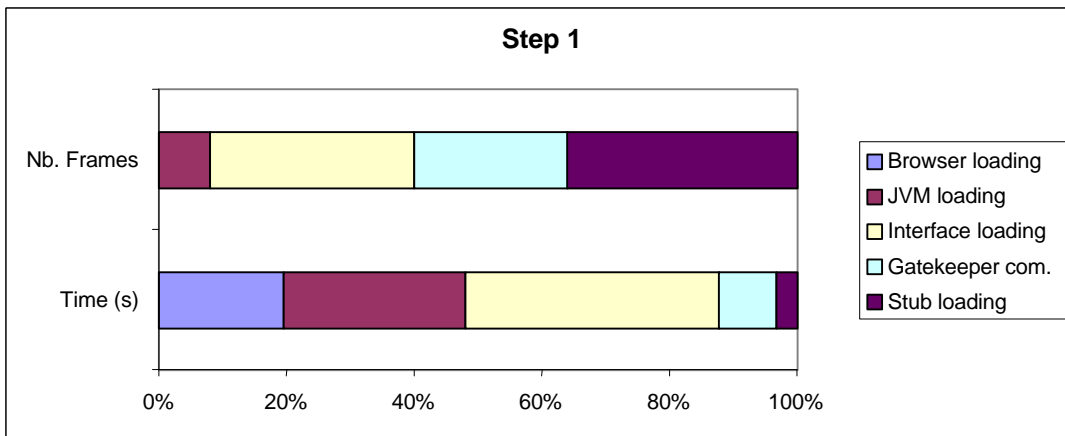
The deployment option chosen has an enormous impact on this step. In the case of the loading of the client ORB from the Web server (use of the ARCHIVE keyword), the client and the server exchange 2,9 Mb in 1128 frames. If the client already possesses the client ORB in .JAR files, the dialogue is limited to 34 Kb in 78 frames !

The second option was chosen for the continuation of the test.

Time on the client side: 15.5 seconds

This initialization includes, on the client side, browser launching, graphical applet loading through an HTML page, and connection to the Corba server.

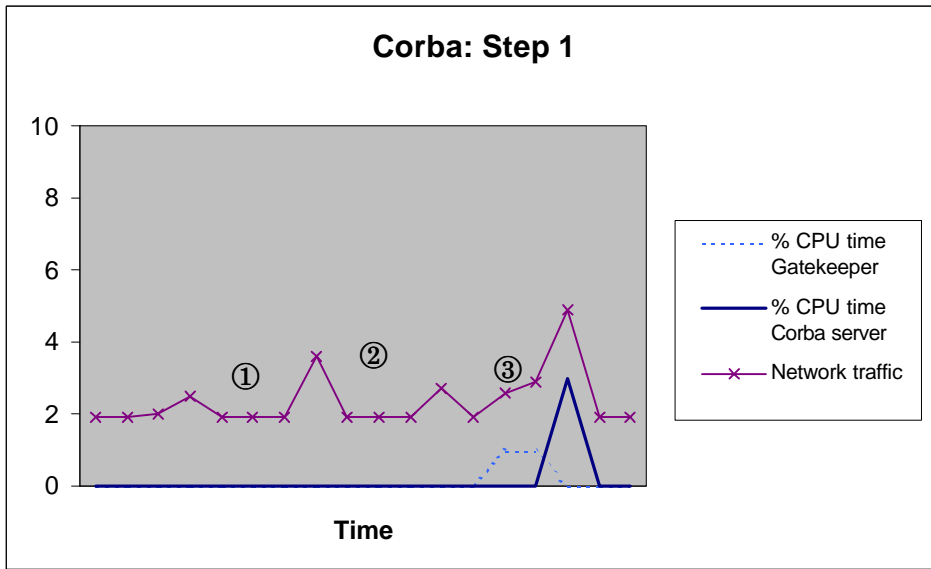
Network load: 78 frames -34 158 bytes



The above diagram shows the amount of time used up by each task in terms of network consumption and time. This analysis allows us to break these tasks up into three categories:

- *The incompressible tasks, not linked to the network:* the tasks performed on the client station belong to this category (loading of the browser and launching of the JVM).
- *The hardly compressible tasks, moderately linked to the network:* interface loading uses the network only moderately, but takes up a large part of the total time. Indeed, the instantiation of the graphical applets uses up client resources. The only way to compress this task is to optimize the programming of the graphical applets.
- *The incompressible tasks, strongly linked to the network:* the two Corba Visibroker tasks belong to this category (establishment of communication with Gatekeeper and stub loading). As they generate a little more than half the network frames, these tasks may be quite slow on a light traffic network. Nevertheless, the excellent filling rate of the frames limits their number in comparison with RMI.

In conclusion, this step depends little on the programming options chosen. The main factors of efficiency will be network traffic flow, the power of the client station and the prior installation of ORB on the client station.



Break down of the step

Browser loading takes 3 seconds. The server is not solicited during this stage (①)
 Frame 4 (≈ 3 s): The first request received by the server concerns an HTML page over port 80, the one of the Web server, after the TCP/IP negotiation.
 Frame 5 (≈ 3 s) the Web server immediately returns The page Web server. From this point on, the server will only be solicited to provide Java applets.

```

Server: Microsoft-IIS/4.0
Connection: keep-alive
Date: Tue, 28 Apr 1998 21:23:35 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Tue, 21 Apr 1998 10:20:33 GMT
ETag: "9066721ef6dbd1:1a94"
Content-Length: 370
<HTML>
<TITLE>HTML test page</TITLE>
<BODY>
<APPLET CODE = "ClientApplet.class">
<param name=org.omg.CORBA.ORBClass
value=com.visigenic.vbroker.orb.ORB>
<param name=ORBgatekeeperIOR
value="http://mdw.sqli.fr:2999/gatekeeper.ior">
</APPLET></BODY>
</HTML>
    
```

Frame 7 (≈ 7.5 s): The client requests the ClientApplet.class applet. This request takes place 4 seconds after the previous frame. This period corresponds to the launching of the JVM on the client station. It comes in addition to browser loading (②).

Frame 31 (≈ 11.8 s): All graphical interface applets have been retrieved on the client station.

Frame 32 (≈ 13.6 s): The graphical interface applets are instantiated on the client station, and a first query is directed towards Gatekeeper (port 2999, by default) (③).

Frame 50 (≈ 15 s): Gatekeeper's IOR is returned, followed by frames containing the IDL definition of the server.

Frame 51 (≈ 15 s): An IIOP dialogue starts between the client and the Corba server on the session port 2271 of the server ().

Frame 55 (≈ 15 s): The last application classes are retrieved, as well as the necessary stubs.

Frame 78 (≈ 15.5 s): The initialization stage is finished.

4.3.3. Step 2: Retrieval of 50 lines

Time on the client side: 1.9 seconds

In fact, the server side processings only last 0.494 of a second. The rest of the time is used by the browser to format the result and to instantiate the business objects.

Network load: 46 frames - 33 787 bytes

One can distinguish 4 types of frames, which appear in the following sequence:

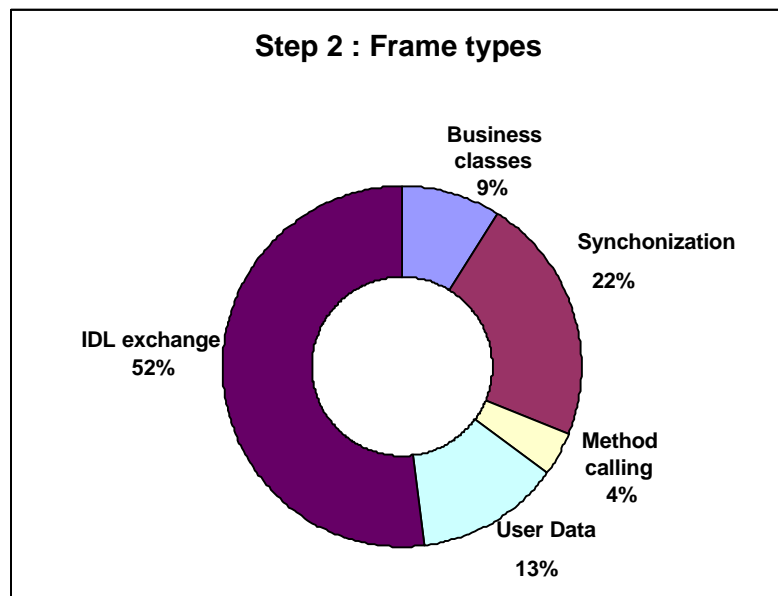
- A few retrieval frames used for a business object class
- A frame for IIOP method calls:

```

GIOP
...
IDL:SQLObject/Connexion:1.0
GetProduct
    
```

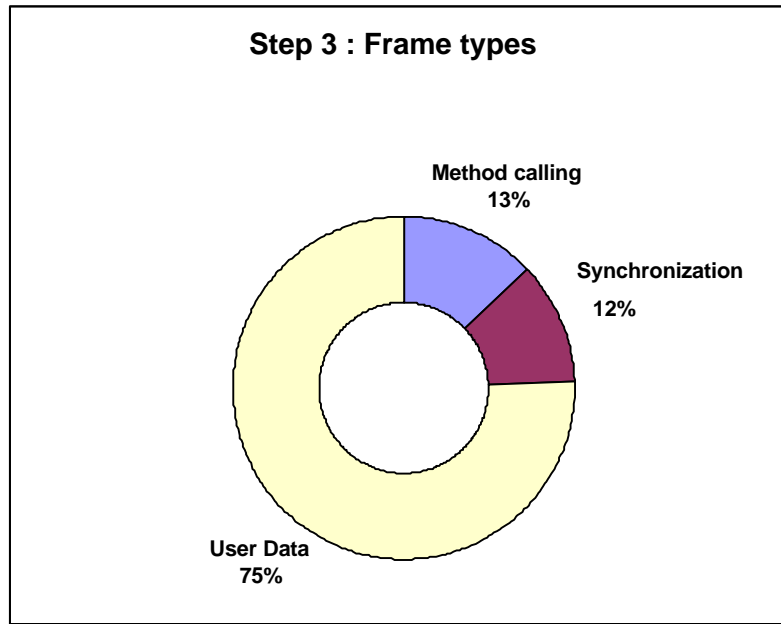
- Frames containing data (the filling rate is very good)
- Frames transmitting the Java classes containing the IDL description of each of the objects used (in the case of Visibroker, these classes are automatically created from the IDL description with the Caffeine compiler `idl2java`).

In conclusion, this step provides excellent server-side performance. On the network side, part of the load is to be attributed to IDL transfers. These transfers are only performed once, the first time the method is called (in contrary to RMI). The bottleneck is the client station, with an average performance.



4.3.4. Step 3: Retrieval of 5*50 lines

Network load: 49 frames - 32 635 bytes



During this step, the time used up on the server side is 3.480 seconds. The server is now in charge only of sending the data to the clients, as the IDL were already exchanged during the previous step. This allows for a reduction in the number of frames.

4.3.5. Step 4: Insertion of 50 lines

Time on the client side: 0.7 of a second per insert

In fact, the server side processings only last 0.2 of a second. The rest of the time is used by the browser to format the result and to instantiate the business objects.

Network load: 168 frames - 31 680 bytes

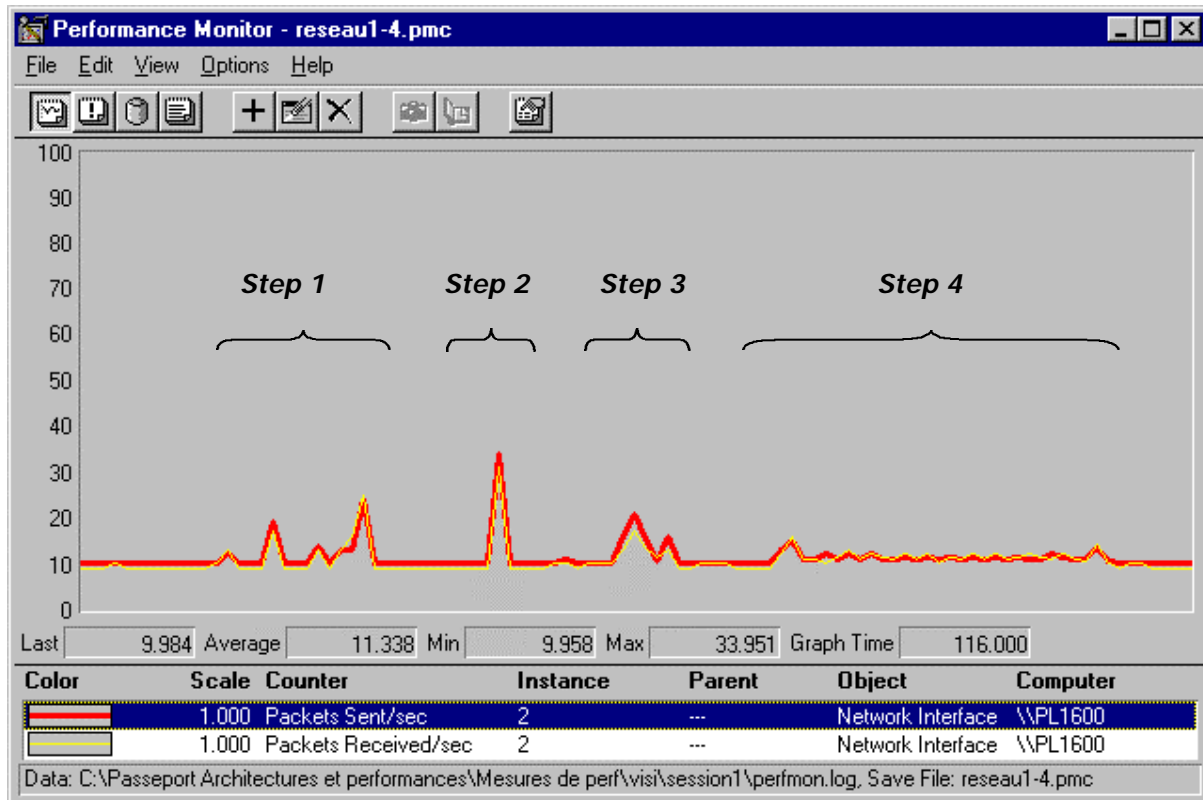
One can distinguish two types of useful frames:

- `InsertProduct` method call frames
- Data exchange frames in the IIOP protocol (which generates synchronization frames).

IIOP allows for the limitation of exchanges. Indeed, the IIOP dialogue is made up of only useful frames (method calls and data).

4.3.6. Step 1 to 4: Network utilization

This diagram shows the utilization of the network, as seen from the server. The bold curve indicates the traffic from the server to the client. The light curve represents the traffic flowing in the opposite direction.



Throughout the four steps, the incoming traffic takes up approximately the same amount of bandwidth as outgoing traffic.

4.3.7. Step 5: Retrieval of 4 000 lines

The server side processing is very short, only 5.5 seconds, in comparison with the network transfer time and client-side formatting time.

As for RMI, formatting time is very important, even if the choice of transferring information differs. Indeed, our RMI model uses object serialization (transfer of objects by value) for dialogues between the client and the server. Our CORBA model uses the transfer of parameters in the form of a table (CORBA does not support serialization). Our measurements highlight the important role of the client station and the browser:

- The formatting takes up 2 minutes 35 seconds on a Pentium 266 MHz - 64 Mb RAM, Netscape 4. IE4 is twice as slow.
 - On a Pentium 90 MHz - 64 Mb RAM, the formatting is two and a half times slower.
- Concerning the browsers, we obtain results diametrically opposed to those measured with RMI. Indeed, concerning parameter transfer in the form of a table (our CORBA application), Netscape is faster than IE4. On the contrary, for object serialization (our RMI application), IE4 has the best results.









In short, the 3 sub-steps are the following:

- *The IIOP connection stage and processing on the server side:* the results obtained are quite satisfactory. They only depend on the server station.
- *The data transfer stage:* this step is fast and takes up few network resources.
- *The formatting stage on the client station:* this stage heavily depends on the power of the client station and on the browser chosen.

4.4. HTML-HTTP

4.4.1. Deployment constraints

The application can be deployed in any browser. No additional module is necessary. Application deployment is reduced to the bare minimum: the presence of a Web browser on one's workstation.

Windows 3.11				Windows 95, NT			
<i>IE 3</i>	<i>IE 4</i>	<i>Net 3</i>	<i>Net 4</i>	<i>IE 3</i>	<i>IE 4</i>	<i>Net 3</i>	<i>Net 4</i>
							

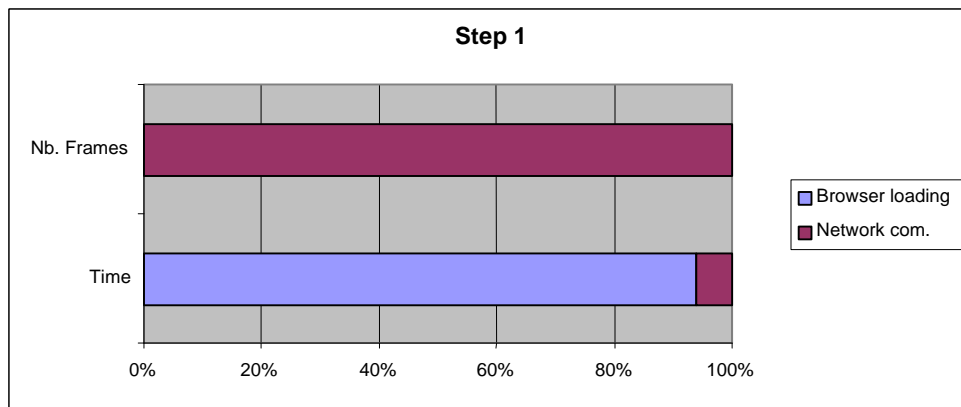
Whether it is a Netscape or Microsoft browser, the test application operates in the same way. The network flow is the same. In terms of performance, the use of a more powerful client station allows for the shortening of browser loading time, and the time needed for the formatting of HTML pages in the browser. Changing browsers does reduce the number of frames or the size of the packets sent over the network. The browsers tested show very similar behavior concerning the management of HTTP protocol.

4.4.2. Step 1: Initialization / Deployment

Time on the client side:

This time represents the browser's loading time on the client station and the time taken up by the loading of the application's welcome page. When using Internet Explorer from Microsoft, this time is equal to 3.2 seconds.

Network load: 10 frames - 1737 bytes



The above diagram shows the amount of time used up by each task in terms of network consumption and time. This analysis enables us to break these tasks up into two categories:

- *The incompressible tasks, not linked to the network:* this is the case for the loading of the browser on the client station.
- *The hardly compressible tasks, moderately linked to the network:* the welcome page call and its display by the browser.

In conclusion, this step depends mainly on the size of the HTML page retrieved.

Break down of the step

Browser loading takes 3 seconds. The server is not solicited during this stage.

Frames 1 to 3: TCP-IP negotiation, aimed at establishing a relationship between the browser and the Web server. Each frame is 60 bytes, giving a total volume of 180 bytes.

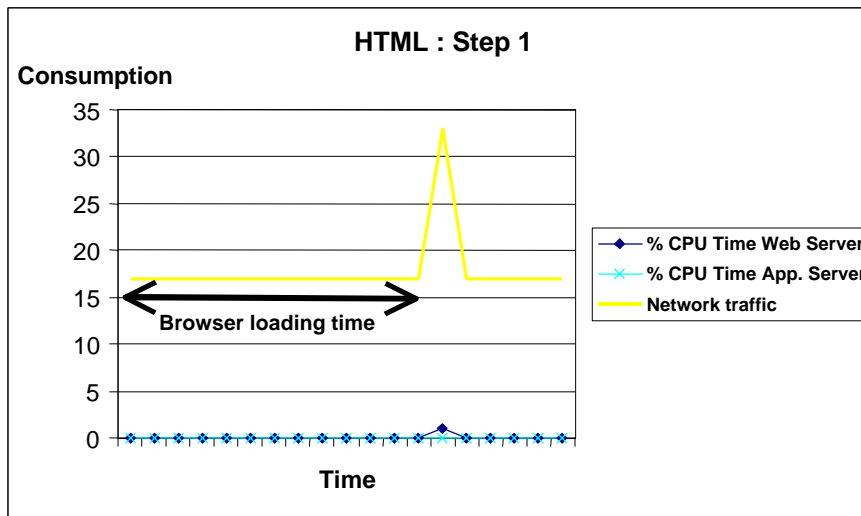
Frame 4: The first request received by the server concerns an HTML page over port 80, using the GET keyword of HTTP protocol. The size of this frame is equal to 396 bytes (keyword GET, URL requested, release of HTTP protocol 1.1, date of the equivalent page located in the client station's memory cache, and information about the client station: browser+OS).

Frame 5: The server immediately returns the requested Web page. It begins by first returning a 181-byte frame describing the " mime-type ". In this case, the mime-type is " text/html ".

Frame 6: This frame returns the HTML flow representing the page.

Frames 7 to 10: These four frames allow for the completion of TCP-IP communication. Each frame is 60 bytes, giving a total volume of 240 bytes.

The following diagram shows resource consumption of the various server-side processes. We can see that the application server is not in demand at all, and that the Web process (the HTTP server) works to provide the application welcome page. As already described above, the total network traffic represents only 10 frames.



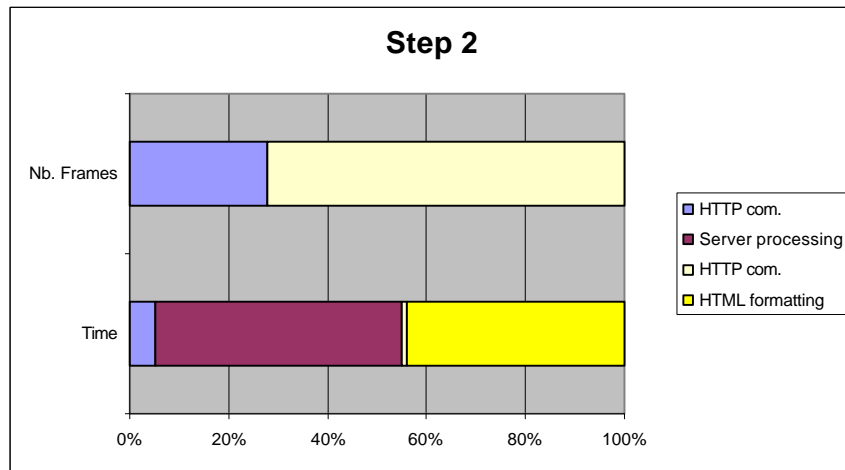
4.4.3. Step 2: Retrieval of 50 lines

Time on the client side:

After the browser has requested the page, the application server runs the SQL query, formats the result and returns the page to the browser. The formatting of the resulting HTML page takes 0.88 of a second, and the entire dialog over the network takes 0.126 of a second.

Even though the network dialogue is hardly compressible, HTML page size has a relative influence. However, the server time (the time needed to run the SQL query and to format the data in the HTML page) has a strong impact on global response time. This server processing time depends on the development tool and method used.

Network load: 18 frames - 7 248 bytes



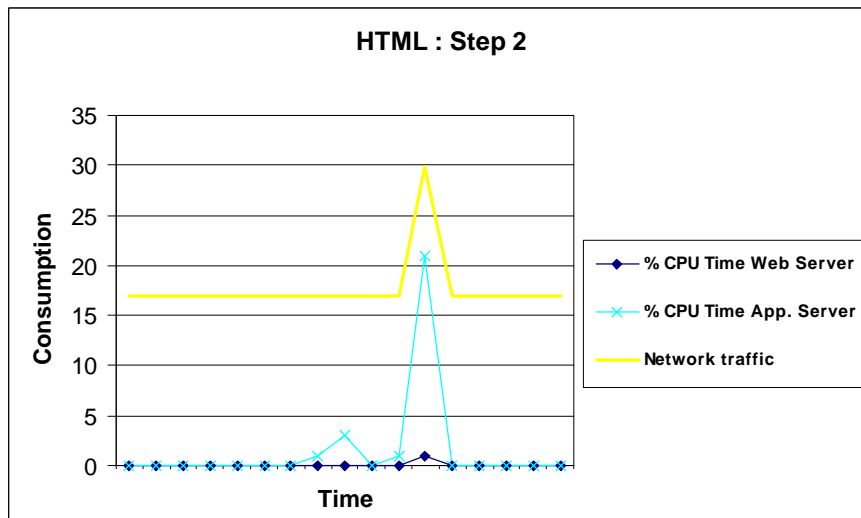
The portion of time used by the application server varies depending on the development tool and method.

This low bandwidth use can be divided up as follows: 11 frames of 60 bytes each, for negotiating the communication, and seven frames for HTTP communication, with an average size of 941 bytes each (6588 bytes in 7 frames).

The HTTP dialogue is always broken down in the same way: connection; sending of a page request to the server (with possibly some parameter transfer); acknowledgement from the server, which then processes the request, and returns the answer in the form of one " mime-type " frame and several frames containing the HTML page. A few acknowledgement frames make the synchronization of exchanges between the client and the server possible. Afterwards, four frames are used to complete the TCP-IP communication.

Break down of the step

- Frames 1 to 3: The client performs a TCP-IP connection with the Web server over port 80. This connection lasts 3 frames of 60 bytes each (180 bytes altogether).
- Frames 4 and 5: the client sends a GET query to the HTTP server, which acknowledges it (2 frames, 471 bytes altogether).
- It is at this moment that the application server connects to the database (the awakening of the Oracle process in the diagram above), performs the SQL query, and formats the results in the HTML page.
- Frame 6: The server answers the client by describing the " Mime type " of the answer: " text/html " for an HTML page (1 frame, 181 bytes long).
- Frames 7 to 14: The transfer of the HTML page from the server to the client is performed. This transfer requires five useful frames and three acknowledgement frames (for a total of 8 frames and 6176 bytes).
- Frames 15 to 18: TCP-IP disconnection (4 frames, 240 bytes altogether).



4.4.4. Step 3: Retrieval of 5*50 lines

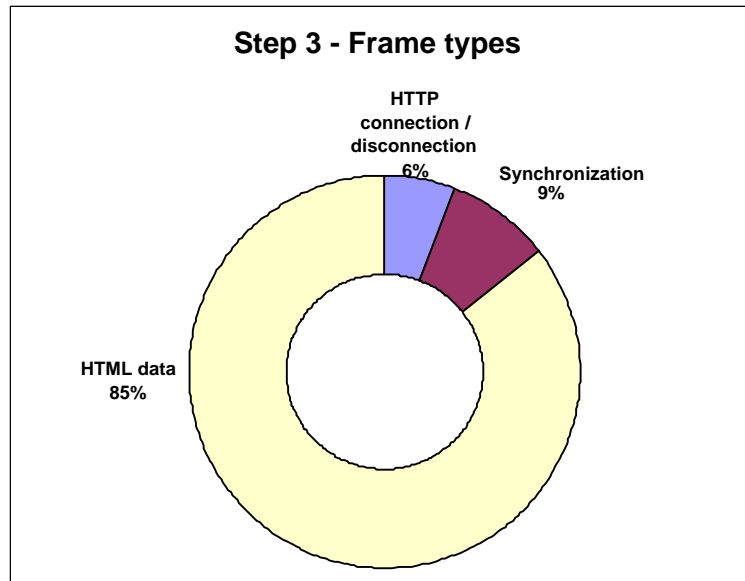
Time on the client side:

Less than 1 second for each new 50-line page (in fact, 0.930 of a second on average).

On our 10Mbit/s network, the time needed for connection, request sending and processing, and answer return represents only 0.35 of a second. The remaining part of the time is used for result formatting by the browser.

On a low traffic network, the transfer time is longer, but only 18 frames are exchanged. The browser formatting time remains constant.

Network load: 90 frames – 36499 bytes for all five pages, each of which retrieves 50 lines



During this step, the same HTTP communication is reproduced five times. 18 frames are exchanged between the client and the server.

Break down of the step

Frames 1 to 3: The client performs a TCP-IP connection with the Web server over port 80. This connection lasts 3 frames of 60 bytes each (180 bytes altogether).

Frames 4 and 5: the client sends a GET query to the HTTP server, which acknowledges it (2 frames, 636 bytes altogether).

At that time, the application server runs the SQL query allowing for the retrieval of the 50 requested lines, and formats the result.

Frames 6 to 14: the Web server answers the client with a frame describing the " mime type " of the answer, and the following frames include the HTML page (9 frames, 6251 bytes altogether).

Frames 15 to 18: these frames make TCP-IP disconnection possible (4 frames, 240 bytes altogether).

4.4.5. Step 4: Insertion of 50 lines

Time on the client side: 1.2 seconds per insert

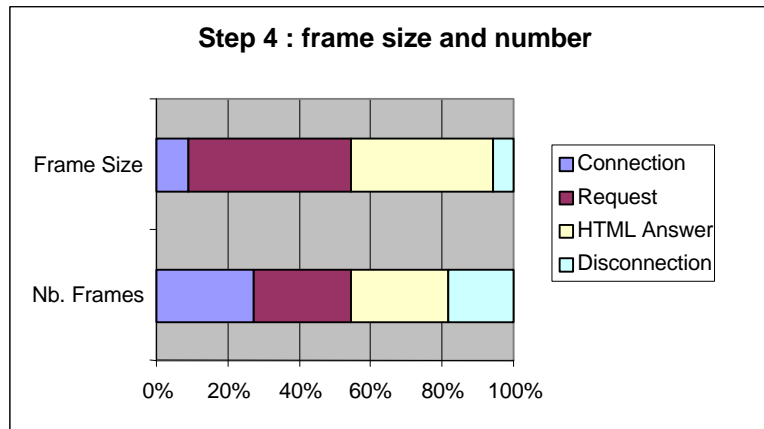
Sending of the query, its processing on the server and answer return represent only 0.3 of a second on our local network. The remaining nine-tenths of a second are dedicated to the display of the HTML page by the browser.

Network load: 552 frames - 106875 bytes exchanged

The first HTTP dialogue downloads the form used for information input. This page contains several pieces of data from database reference tables. The following process is then repeated 50 times: form field input in the browser and INSERT query in the database.

Break down of the step

Each INSERT query always generates the same scheme of network resource consumption.



Frames 1 to 3: The client performs a TCP-IP connection with the Web server over port 80. This connection requires 3 frames of 60 bytes each (180 bytes altogether).
Frames 4 to 6: the client performs a POST query with the adequate form parameters to the HTTP server, which acknowledges it. The client then returns a frame to the server specifying that the POST query is finished (3 frames, 926 bytes altogether).

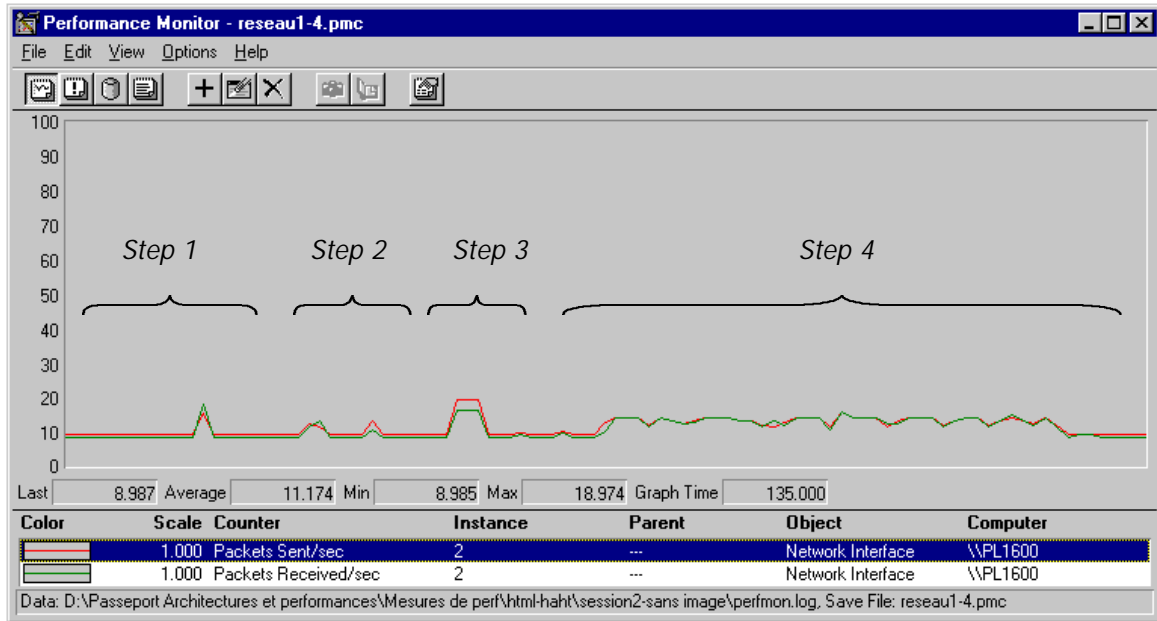
At that time, the application server runs the SQL INSERT query, and formats the result.

Frames 7 to 9: the Web server answers the client with a frame describing the " mime type " of the answer, and the following frames include the HTML page (3 frames, 808 bytes altogether).

Frames 10 and 11: these frames make TCP-IP disconnection possible (2 frames, 120 bytes altogether).

4.4.6. Step 1 to 4: Network utilization

This diagram below shows network utilization, as seen from the server. The bold curve represents the traffic from the server to the client. The light curve represents the traffic flowing in the opposite direction.



For all of the tests, the HTTP protocol shows a very homogeneous bandwidth use.

Logically, this protocol is particularly suitable for low traffic networks and/or when the client-server interactions are not intensive. Indeed, during step 4 many forms are sent to the server from the same client station, and the consumption of network bandwidth is higher than with connected mode protocols. In this case, this model forces useful data transport each time and layout becomes a handicap.

4.4.7. Step 5: Retrieval of 4 000 lines

Time on the client side:

The time necessary for connection to the Web server and the request of a server service represents only 0.121 of a second. After that, the biggest part of the work on the server side consists in running the query and setting up the HTML answer page, which represents nearly 450 Kb. It takes 3 seconds to transfer this page to our local network. The browser then needs 5.5 seconds to format the page received.

This test shows that no matter the size of the page returned by the Web server, the time needed to format the page remains satisfactory, even with a very high volume of data to be displayed.

Network load: 555 frames - 450215 bytes exchanged

During this exchange with very large pages (around 450 Kb), the time needed for connection, disconnection and query is negligible compared with the duration of the HTML page transfer. The frame filling rate for the answer was satisfactory: an average of 821 bytes per frame.

Break down of the step

Frames 1 to 3: The client performs a TCP-IP connection with the Web server over port 80. This connection lasts 3 frames of 60 bytes each (which makes 180 bytes altogether).

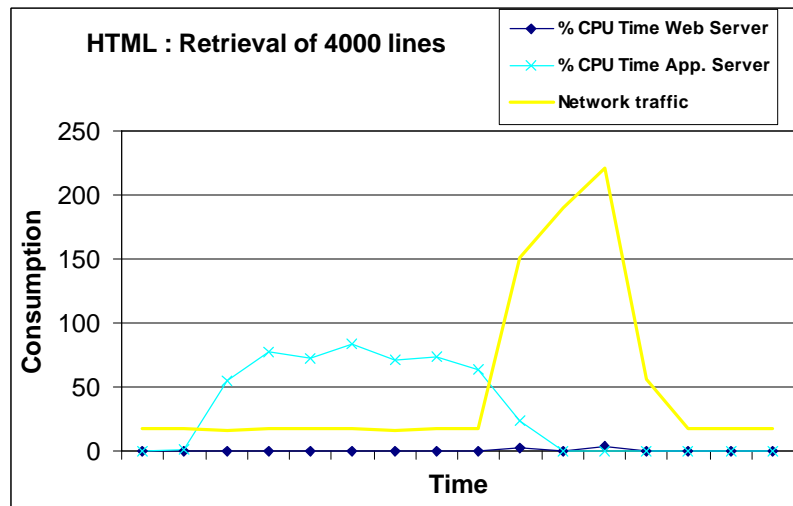
Frames 4 and 5: the client sends a GET query to the HTTP server, which acknowledges it (2 frames, 463 bytes altogether).

At that time, the application server runs the SQL SELECT query, and formats the result.

Frame 6: The server answers with a first frame describing the " Mime type " of the answer (1 frame, 181 bytes long).

Frames 7 to 553: They contain the HTML page itself and the client acknowledgements (547 frames, 449271 bytes altogether).

Frames 554 and 555: these frames allow for TCP-IP disconnection (2 frames, 120 bytes altogether).



The server side processing is long in comparison with the network transfer time and client-side formatting time. However, this time can certainly be optimized, as it depends heavily on the development method used.



4.5. Client/server

4.5.1. Deployment constraints

The application is deployed by installing, on each client station:

- the run-time of the development tool
- the application libraries
- the database middleware
- the ODBC driver (optional, depends on the tool used)

The middleware (SQL*Net in the case of Oracle) must be parameterized and a test enabling to check the proper operation of the whole carried out. As this installation and configuration stage may interfere with other elements on the client station, a specialist must absolutely perform it. This installation and configuration stage is not within reach of novice users.

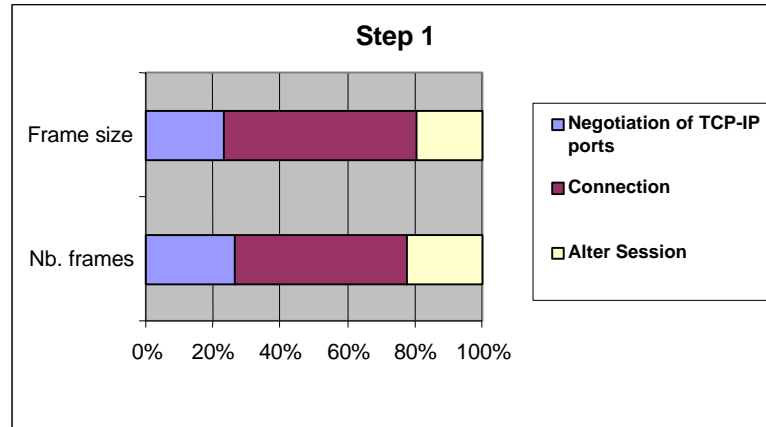
Windows 3.11	Windows 95, NT
 (providing the application is compiled in 16 bit mode, and SQL*Net 16 bits is installed)	

4.5.2. Step 1: Initialization / Deployment

Time on the client side:

This time represents the time needed by the client station to load the application. The application then loads the main window by connecting to the Oracle database. Nearly 8 seconds are necessary to perform all of the above.

Network load: 45 frames - 4883 bytes



Break down of the step

Application loading takes 6 seconds. The server is not solicited during this stage.

Frames 1 to 12: the SQL*Net communication ports are negotiated. This stage begins with a TCP-IP connection (3 frames, 120 bytes altogether). Afterwards, there are several exchanges to negotiate the SQL*Net communication port (9 frames, 1002 bytes altogether).

Frames 13 to 35: the SQL*Net connection. This step begins with a TCP-IP connection over the ports negotiated (3 frames, 120 bytes altogether). The connection string is then transmitted to the server. Once it has been accepted, the client sends the information from its station to the server (20 frames, 2690 bytes altogether).

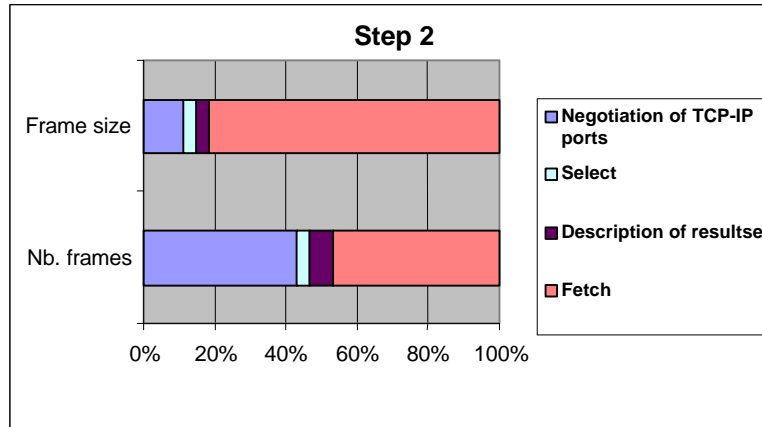
Frames 36 to 45: These 10 frames enable the client to send an ALTER SESSION query to the server, and to then exchange parameters (10 frames, 951 bytes).

4.5.3. Step 2: Retrieval of 50 lines

Time on the client side:

The response time needed to make the SELECT query and display the first 50 lines (partial fetch) is close to 4 seconds.

Network load: 29 frames - 10 071 bytes



Most of the communication is dedicated to data retrieval. There is not a lot of " technical " dialogue.

Break down of the step

Frames 1 to 12: the SQL*Net communication ports are negotiated. This stage begins with a TCP-IP connection (3 frames, 120 bytes altogether). Afterwards, there are several exchanges to negotiate the SQL*Net communication port (12 frames, 1122 bytes altogether).

Frame 13: the client sends a SELECT query to the database server (1 frame, 379 bytes altogether).

Frames 14 and 15: The server returns the description of the query's results (result set) to the client. The client then acknowledges (2 frames, 339 bytes).

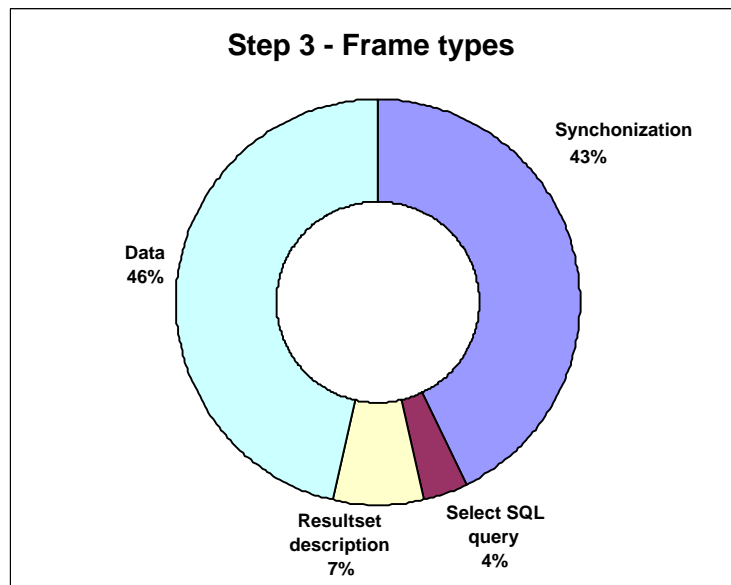
Frames 16 to 29: the server returns the query result to the client (13 frames, 8231 bytes altogether).

4.5.4. Step 3: Retrieval of 5*50 lines

Time on the client side: Less than 1 second for each packet of 50 lines (in fact, 0.747 of a second on average).

This time includes the database fetch request, network data transfer and data display in the client application.

Network load: 24 frames - 15884 bytes altogether to retrieve 50 lines five times.



During this step, very few frames are used to transmit data.

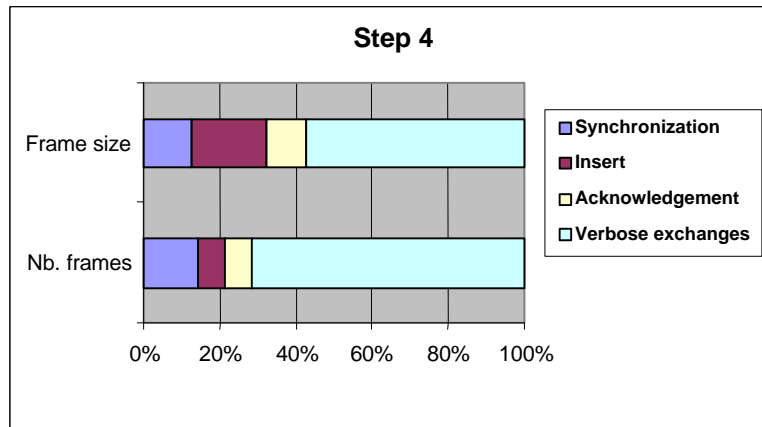
The client station sends a frame to request the fetch, and the data server answers by transmitting the 50 lines in two frames (with a size of 1514 bytes and 641 bytes, respectively). From time to time, the client station returns acknowledgements to the server (8 frames, 60 bytes each).

4.5.5. Step 4: Insertion of 50 lines

Time on the client side: 0.4 of a second per insert

This time is very short. Unfortunately, this step uses up a lot of bandwidth (size and number of frames).

Network load: 3808 frames - 341701 bytes exchanged



Break down of the step

Each INSERT query always generates the same scheme of network resource consumption.

Frames 1 and 2: the client and server are synchronized (2 frames, 145 bytes altogether).

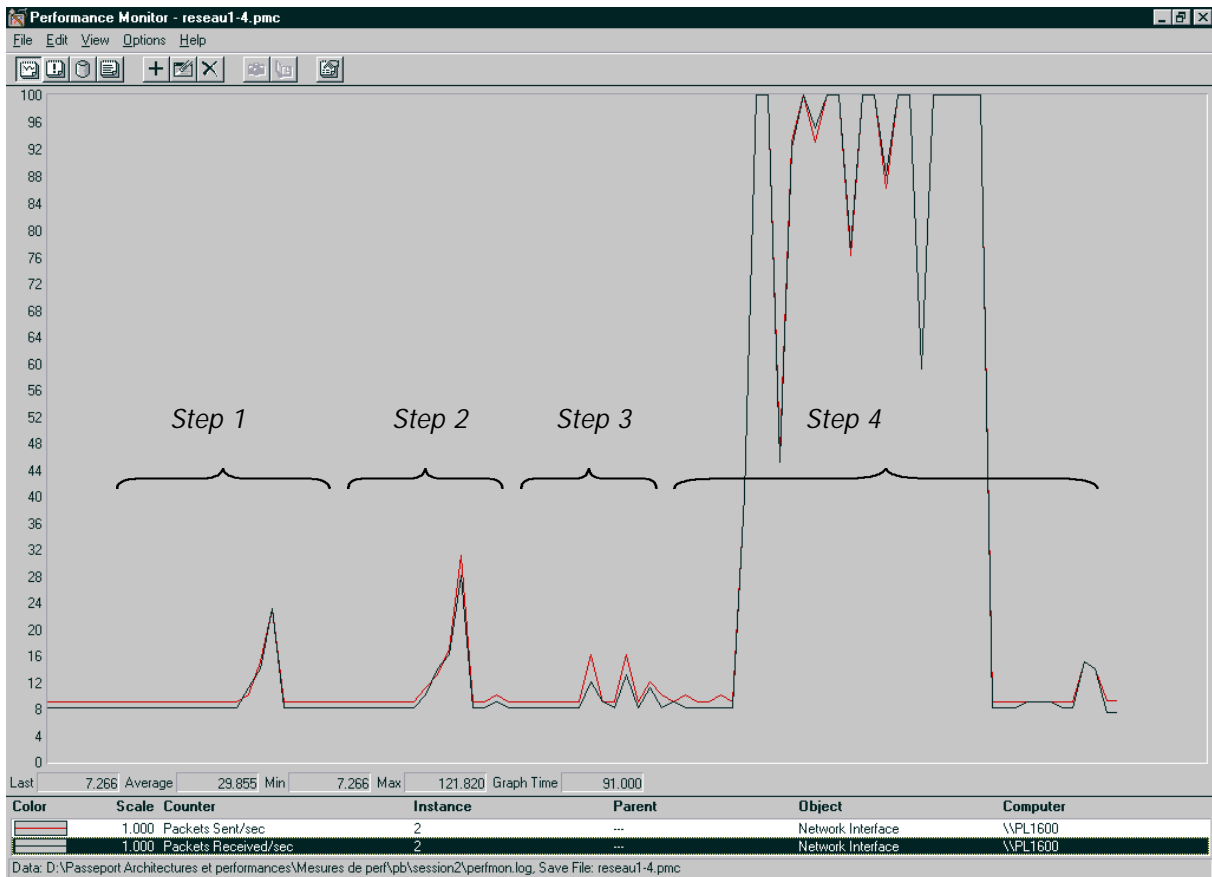
Frame 3: the client sends an INSERT query to the server (1 frame, 225 bytes).

Frame 4: after receiving the INSERT query, the server returns an acknowledgement (1 frame, 124 bytes).

Frames 5 to 15: the client and server exchange synchronization frames: one of 69 bytes from the server to the client, and another one of 71 bytes in the opposite direction.

4.5.6. Step 1 to 4: Network utilization

The diagram below shows the utilization of the network, as seen from the server. The bold curve represents the traffic from the server to the client. The light curve represents the traffic flowing in the opposite direction.



A global observation of network bandwidth consumption of the client/server model shows that during the three first steps, the consumption is reasonable. Only the last step (4) shows an over-consumption. This excess can, of course, be explained by analyzing the programming methods.

Even so, it is only through measurement analysis at the network level that the developer may discover this problem. Indeed, on a high traffic network (10Mb or more), like the one used during the tests, user response time is not affected by this high bandwidth consumption.

4.5.7. Step 5: Retrieval of 4 000 lines

Time on the client side:

It takes around 1 second to display the 4000 lines in the Windows window. This response time is very good, and can be explained in two ways. The first factor is the network data transfer time. At this level, the application takes full advantage of the mechanisms provided by the DBMS to retrieve a large set of data. The second factor is linked to the formatting, which depends on the graphical object used to display the data retrieved.

Network load: 250 frames - 172935 bytes exchanged

During this step, the database server is not in very high demand. However, there is a lot of network communication between the client and the server, but only the data are sent to the client station, and the network frames are optimized well.

Break down of the step

Frames 1 to 7: the client station sends a SQL SELECT query to the database server. The server returns column descriptions and prepares the result (7 frames, 1211 bytes altogether).

Frames 8 to 250: the server transmits the data to the client with the help of 1514-byte and 641-byte frames. For every two or four frames sent from the server to the client, the client returns a 60-byte acknowledgement frame to the server. The average filling rate of these 243 frames is 707 bytes per frame.

5. Implementation

5.1. ActiveX-DCOM

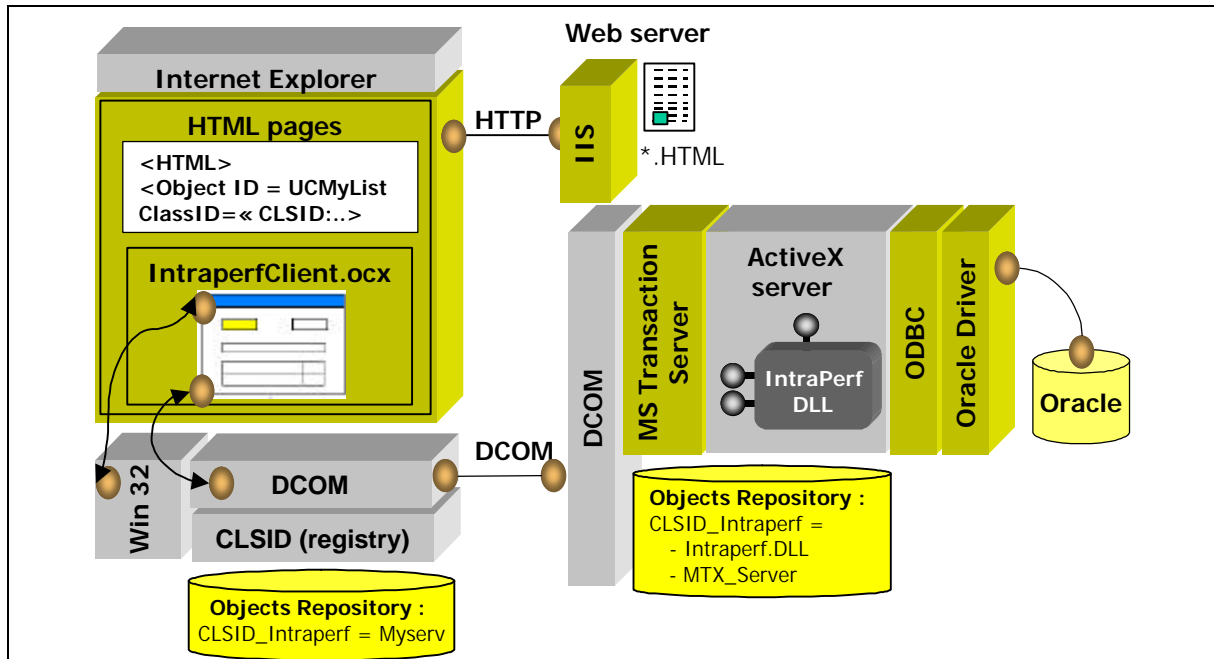
5.1.1. Implementation of the architecture

To develop the test application, we chose Microsoft's Visual Basic 5.0. Visual Studio also offers the tools Visual C++ and Visual J++ which make the implementation of the ActiveX-DCOM model possible.

Moreover, other tools like Delphi (Inprise) or PowerBuilder (Sybase) may be used to develop applications operating with a Microsoft architecture.

The application relies on two components:

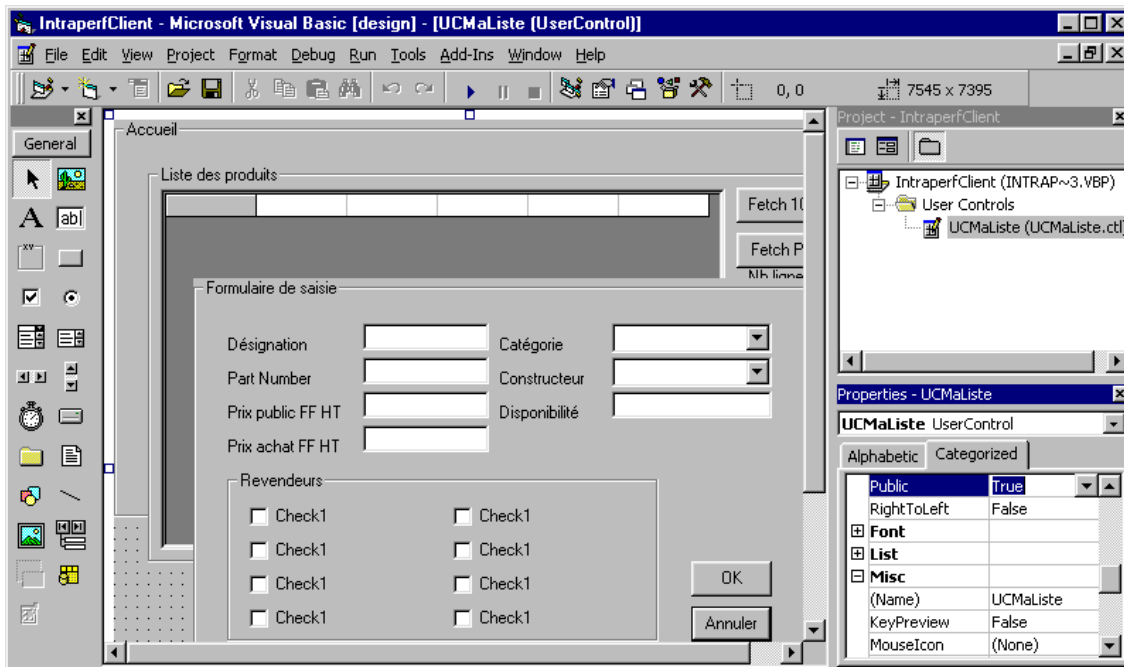
- IntrapertClient.OCX: the ActiveX component used for graphical display
- IntraPerf.DLL: the ActiveX server component that includes the processings performed by the application.



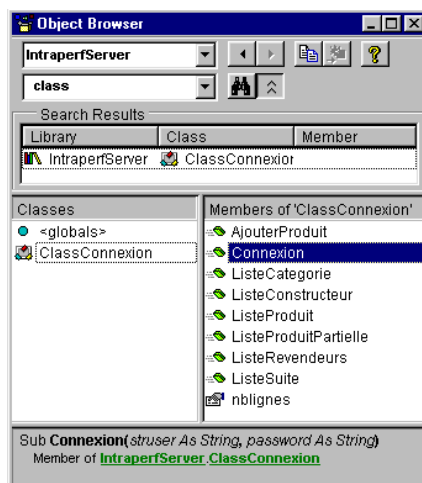
Architecture implemented for the tests

5.1.2. Development of the client part

The client part of the application is made up of an ActiveX component. As the application has only three windows (one for the menu, one for list mode display and one for data input), we grouped them into a single component.



Development of the component interface in Visual Basic.



The client component references the server object at the project level, which gives it access to the list of available classes. At the initialization stage, the client component runs the following code:

```
Set myconnec = New ClassConnection
myconnec.Connection "user", "password"
```

This method call creates an instance of the server object (IntraperfServer.DLL) which contains the "ClassConnection" class. The client then invokes the "Connection" method, which is run by the instance of the server object, and connects to the DBMS.

For the client application, the fact that the server (IntraPerfServer.DLL) is located on the station used for development, or on a remote computer, does not change anything. The Windows registry contains information about the actual location of the server object, but also the full definition of the classes, methods and attributes.

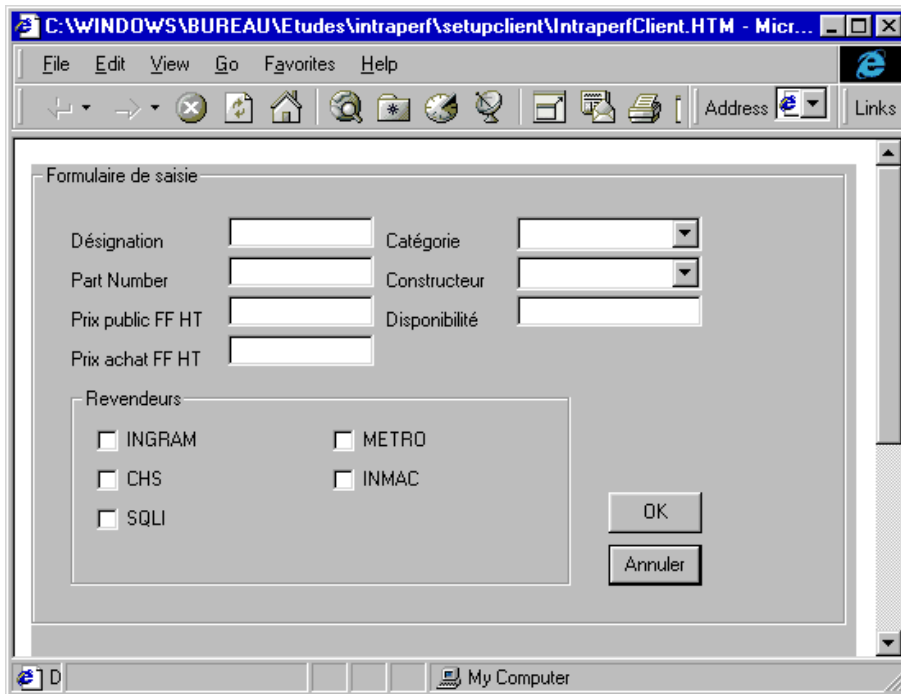
The client component contains neither business rule nor action. In fact, all user actions that require data access come through the server. Therefore, the code comes down to method calls or is related to graphical layout.

Once the component has been set up, it must be compiled in order to generate the file (OCX) which will be used. The components compiled with Visual Basic require a number of DLL files in order to operate. A wizard provided with Visual Basic enables one to build an automatic installation procedure through an intranet (a detailed list of the files is provided in the paragraphs relating to the analysis of performance measurements).

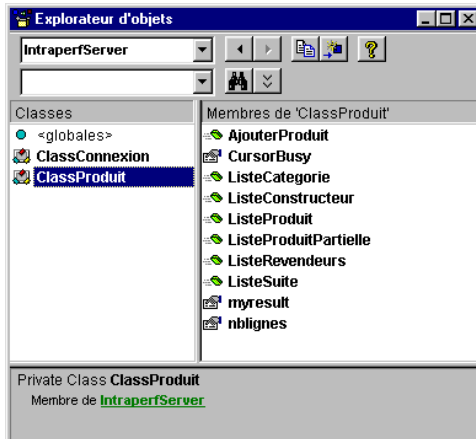
Once the installation process is completed (when all the files have been downloaded and the registry's configuration successfully updated), the browser may instantiate the ActiveX component, by using the information provided by the HTML page:

```
<HTML>  
<OBJECT ID="UCMyList" WIDTH=503 HEIGHT=493  
CLASSID="CLSID:3D40D2D2-FF06-11D1-B71B-444553540000"  
CODEBASE="IntraperfClient.CAB#version=1,0,0,0">  
</OBJECT>  
</HTML>
```

The display of the component gives the following result:



5.1.3. Development of the server part



The first step of server development consists in defining the business object, which will enable us to fill the application's requirements. In view of the simplicity of the specifications, a single business object was built: ClassProduct.

This object contains the methods necessary for the operation of the application. It sends SQL queries to the DBMS, and manages transactions during the data insertion stage (sending of the SQL Insert queries).

It possesses seven methods and three attributes with the following roles:

Methods	Role
PartialProductList	Creates the data list from a SQL Select query. Returns a given number of lines passed on as a parameter (n). Returns a table of n lines.
FollowingList	Returns the continuation of the previous list, with a given number of lines passed on as a parameter (n). Returns a table of n lines.
ProductList	Creates the data list from a SQL Select query. Returns all the lines of the result (4000 lines). Returns a table of 4000 lines.
CategoryList, ManufacturerList, DealerList	Return the data enabling one to fill in the list boxes and sequence of checkboxes on the insertion form.
AddProduct	Sends the SQL insert orders from the parameters received, by managing the validity of the DBMS transaction. Returns a status code about the validity of the transaction. (Used in scenario 4)
Attributes	Role
CursorBusy	Indicates that the result of a SQL query was created, but has not yet been entirely sent to the client.
MyResult	Attribute containing the current data set.
NbLines	Attribute containing the number of lines of the current data set.

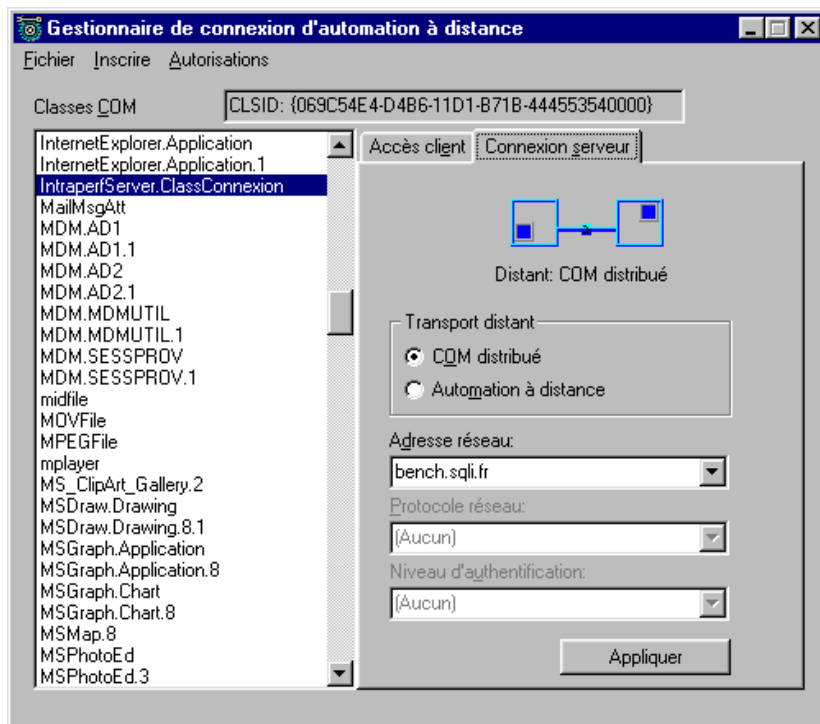
The other class " ClassConnection " has a technical vocation, notably handling database connection.

For the management of exchanges with the database, our application relies on the Remote Data Object (RDO) model from Microsoft, which itself is based on ODBC.

5.1.4. Mechanism used for communication

For the ActiveX control (IntraperfClient.OCX) instantiated by the browser, the location of the server object (Intraperf.DLL) is transparent. The only information it needs to know is the name of the class to which it refers.

The information about the actual component location is stored at the client station registry level. The screen shot below provides an " ergonomic " representation of the registry base, concerning the location of the components.



One can see that the " IntraperfServer " object is defined as a distributed DCOM object. Therefore, from this moment on, DCOM is in charge of transmitting the requests sent to the object to the server where the ActiveX server component is really located (for example, in this case, on the " bench.sqli.fr " server).

The ActiveX-DCOM section of Chapter 2, Intranet Architectures (p. 11) explains the COM/DCOM communication principles in detail.

5.1.5. Development facility

Developing an application relying on the ActiveX-DCOM model is rather easy. The main problem will be the design aspect. Indeed, as it is mandatory to separate the interface and the processings, the design becomes more difficult. An object approach is unavoidable, and this is true no matter the intranet model.

The development tool is another key factor affecting development facility. Microsoft leaves the choice of tool open, one can choose between Visual Basic, Visual C++ and Visual J++.

Visual Basic (version 6 now available) offers a good compromise between development facility and operating speed. The complex mechanisms underlying the use of a distributed object model are hidden, but Visual Basic does not provide control or optimization possibilities.

Lastly, even if the development stage is easy, one must not forget that component development, especially on the client side, is very restrictive. This point must be carefully studied before choosing this architecture.

5.2. Java - RMI

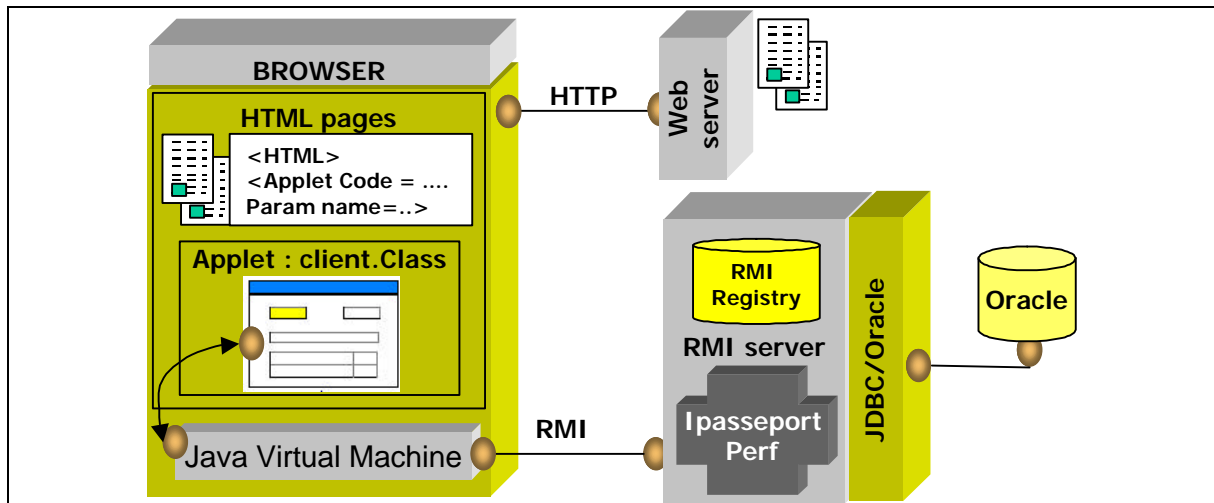
5.2.1. Implementation of the architecture

RMI, or Remote Method Invocation, is a 100% pure Java protocol which makes it possible to call methods from remote Java objects. The term " remote object " can be used to designate an object located in a process other than the calling object, or an object located on another computer. RMI operates exclusively over the TCP/IP protocol, which is the standard Internet protocol.

It is a three-tiered architecture. The application is made up of three modules: the database management system (Oracle in this case); a server module, named " server " below, which is in charge of answering client requests; and a client module, named " client " below, which presents the user interface. The communication between the database and the server takes place via the Oracle implementation of JDBC specification. As with ODBC, this specification establishes a protocol for accessing database management systems and performing queries. Lastly, RMI ensures the communication between the client and the server.

The server displays all the methods necessary to run the application for the client, for example, how to know the total number of suppliers, how to retrieve a certain number of products, etc.

A fourth mechanism, linked to RMI, comes into play in this implementation: a naming service, enabling the client object to obtain a reference to the desired server object. This service is provided by SUN as an integral part of RMI.



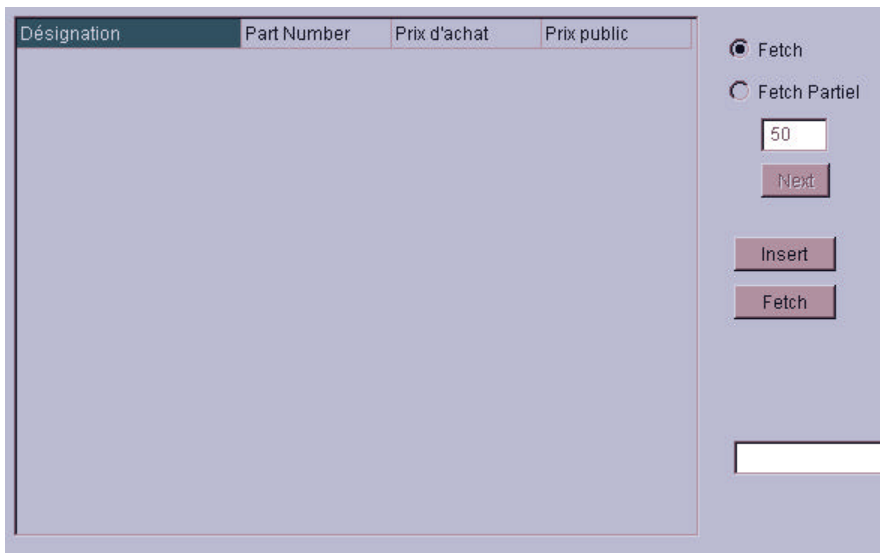
Architecture implemented for the tests

5.2.2. Development of the client part

The interface may be built with any JAVA interface generator. We only used Jbuilder for displaying product query results in a grid. The remaining part only uses AWT, the standard graphical library of the Java Development Kit.

Graphical interface design is simple and stripped to the bare minimum. Only the buttons necessary for the transfer of all, or a portion, of the products are present. There is also a button permitting the user to insert a product.

This screen looks like this:



The central part is a grid used for displaying the products. The available commands are displayed on the right.

The two radio buttons enable the user to select the type of fetch to be performed. With a partial fetch, the text field below is activated, and one may specify the number of records one wishes to retrieve. The Next button will only be activated if the partial fetch has been selected and if the user has already pressed the Fetch button. This button allows for the display of the following records. The Insert button, described below, displays a dialog box used to perform product insertion. The last text field has no impact on the application.

The following code is run when the user presses the Fetch button:

```
try{
    resetGrid();           // Empties the grid
    Checkbox
tmpCB=checkboxGroup1.getSelectedCheckbox();
    int fetchNb=0;
    if(tmpCB==PartialfetchCB){
        // If one wants to perform a partial fetch
        try{
            // retrieve the number of records ?
```

```

        fetchNb=(new
Integer(textField1.getText()).intValue());
        } catch(NumberFormatException nfE) {
            nfE.printStackTrace();
        }
    }
    Vector v=myPasseportPerf.getProducts(fetchNb);
    endOfRecordset=(fetchNb!=0 && fetchNb>v.size());
    NextBtn.setEnabled(!endOfRecordset);
    for(int i=0;i<v.size();i++){
        Product tmpProduct=(Product)v.elementAt(i);
        appendProduct(tmpProduct);
    }
} catch (RemoteException e2) {
    e2.printStackTrace();
}
}

```

The call to the server is printed in bold characters above. Note that at the end of the records, the Next button is disabled. The last loop displays the products in the grid.

One may also press the Next button to retrieve the following records. This button operates in a similar way, apart from the call to getNextProducts(). Its code is as follows:

```

try{
    int fetchNb=0;
    try{
        fetchNb=(new
Integer(textField1.getText()).intValue());
        } catch(NumberFormatException nfE) {
            nfE.printStackTrace();
        }
    }
    Vector v=myPasseportPerf.getNextProducts(fetchNb);
    endOfRecordset=(fetchNb!=0 && fetchNb>v.size());
    NextBtn.setEnabled(!endOfRecordset);
    for(int i=0;i<v.size();i++){
        Product tmpProduct=(Product)v.elementAt(i);
        appendProduct(tmpProduct);
    }
} catch (RemoteException e2) {
    e2.printStackTrace();
}
}

```

Inserting a product

This part is also very interesting, as it displays a dynamically created dialog box after the information has been retrieved in the database. To provide a better understanding, here follows a screen shot:

Désignation Catégorie

Part Number Constructeur

Prix Public HT Disponibilité

Prix Achat HT

Revendeurs INGRAM SFE Allium TechData
 CHS Illion Métrologie TWC

All of the text fields allow the user to input information. This dialog box first fills in the combo lists containing the categories and the manufacturers, after performing the retrieval in the database, then dynamically builds the list of check boxes after the transfer of the dealers from the base. This last method uses the check boxes, as there might be several dealers for a given product.

This dialog box is created as follows:

```
[...] Code used to design the other items
Vector categories=null,manufacturers=null,dealers=null;
try{
    categories=thePasseportPerf.getCategories();
    manufacturers=thePasseportPerf.getManufacturers();
    dealers=thePasseportPerf.getDealers();
} catch(RemoteException e){
    e.printStackTrace();
}
[...]
Category tmpCat;
// Filling the list of categories
for(int i=0;i<categories.size();i++){
    tmpCat=(Category)categories.elementAt(i);
    cat.add(tmpCat.Nom);
    hCategories.put(tmpCat.Nom,tmpCat);
}
[...]
// Filling the list of manufacturers
// Same method as for categories
add(new Label("Dealers"), new XYConstraints(10, 150,
100, 20));
Dealer tmpDea=null;
int lig=0,col=0;
int dlig=30,dcol=110;
// Creating the check boxes for the dealers
for(int i=0;i<dealers.size();i++){
    tmpDea=(Dealer)dealers.elementAt(i);
    Checkbox tmpCB=new Checkbox(tmpDea.Name);
    add(tmpCB, new XYConstraints(120+col*dcol,
150+lig*dlig, 100, 20));
    cbDealers.addElement(tmpCB);
    if(col++==3){
        col=0;
        lig++;
    }
    hDealers.put(tmpDea.Name,tmpDea);
}
```

Then, when the user presses the validation button, the Product object is created from the information entered, then transmitted to the server via the call to insertProduct().

5.2.3. Development of the server part

The server is made up of two parts: the first part is a simple load balancing mechanism, and the second part is the server object, strictly speaking.

■ *Load balance*

Load balancing is performed in a trivial way, by instantiating a server object for each client connected. This solution has both advantages and disadvantages. The main advantage is that each client possesses a simple environment with which the other clients do not interfere. The main disadvantage is the heavy load the server must withstand with a large number of clients. However, this drawback has no importance in our example, as we want to study network load and network bandwidth consumption. A production application would thus preferably be hosted on a system with load balancing capabilities. Our mechanism is named dispatcher here. A client object must establish a connection with a dispatcher server object, which in turn will instantiate a server object and return the reference to the client.

The dispatcher interface is as follows:

```
public interface IPasseportPerfDispatcher extends
Remote
{
    public Remote connect() throws RemoteException;
    public void disconnect(int ID) throws
RemoteException;}
```

The connect() method instantiates a server object and associates a cookie (or identifier) to it. This cookie is then used during client disconnection: disconnect() disallows the server object thus identified.

■ *Server object*

The server object created establishes a connection to the database and waits for the client queries. Its interface is as follows:

```
import java.rmi.*;
import java.util.Vector;
import Product;
public interface IPasseportPerf extends Remote
{
    public Vector getProducts(int Number) throws
RemoteException;
    public Vector getCategories() throws
RemoteException;
    public Vector getDealers() throws RemoteException;
    public Vector getManufacturers() throws
RemoteException;
    public long getNbProducts() throws
RemoteException;
    public boolean insertProduct(Product product)
throws RemoteException;
    public Vector getNextProducts(int Number) throws
RemoteException;
```

```

    }
    public int getID() throws RemoteException;
}

```

The constructor of the object implementing this interface opens the connection to database.

■ *Methods of product management*

Three methods are provided to manage the products: `getProducts()`, `getNbProducts()` and `getNextProducts()`. They allow one to carry out the fetch and partial fetch operations. One may perform a fetch by passing the argument 0 to the `getProduct()` function. One may perform a partial fetch by passing a positive number on as an argument; the function will then return a vector containing the first n products. Nevertheless, the server will retrieve all products in the base, and position a cursor at the place considered. A call to `getNextProducts()` will then enable one to retrieve the following n products.

Methods of managing other data

The `getCategories()`, `getDealers()` and `getManufacturers()` methods return a vector containing the corresponding objects. These functions always return all the corresponding data contained in the database.

Inserting a new product

The `insertProduct()` method makes the insertion of a new product in the database possible. To provide a better understanding of the operation it is necessary to show the structure of the object produced in detail:

```

public class Product implements java.io.Serializable
{
    public long ID;
    public String Designation;
    public String PartNumber;
    public String RetailPrice;
    public String PurchasePrice;
    public String Availability;
    public Category Cat;
    public Manufacturer Manuf;
    public java.util.Vector Dea;
    [...]
}

```

The `insertProduct()` method will then insert this product in the database by randomly positioning the simple fields (String type), and only inserting the ID for complex fields (Category and Manufacturer). For dealers, as is typical with this kind of relationship, there is a table, which manages the relationship between the dealer tables and product tables. This method will therefore perform as many insertions as there are objects in the Dea vector.

5.2.4. Mechanism used for communication

Here, the only mechanism used for communication between the client and the server is RMI, and information transfers are performed via " serialization ". In other words, the arguments and return objects will be transmitted via RMI, which will in turn format them so that they can be transported over the network.

Apart from transmitting raw information during frame analysis, RMI also transferred information about the object types. So, when simultaneously transferring many objects, some extra weight in the frames, due to type information, can be noticed.

The client is a JAVA applet, which displays the user interface and manages the connection with the server. A JAVA applet manages a few standard events, like " init ", which is called after the applet has been instantiated. This method is therefore used to create the connection.

The connection is performed in two steps:

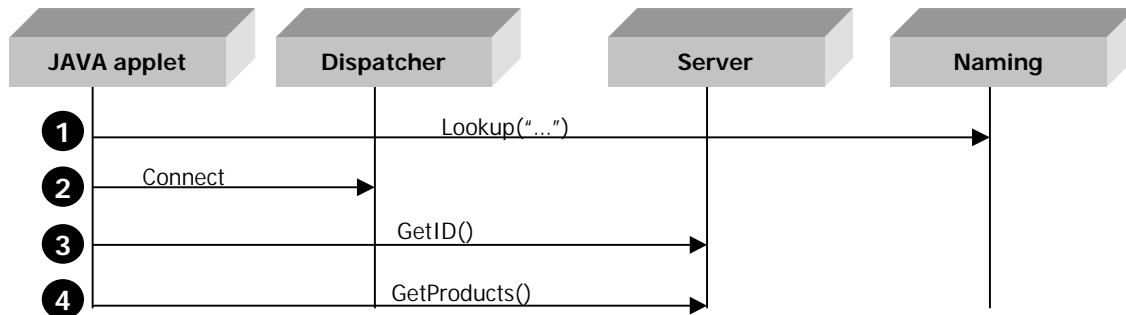
- First, a variable, which is the same type as the server interface, is declared.

```
| IMyServer theServer ;
```
- A connection with the naming service must then be established, by passing on the object name on which one wants to retrieve a reference. Note that the reference to the naming service is performed via the well-known URL system:

```
| rmi:///<Name or IP address of computer>/<Object name>
```
- This provides the following code, if our object is named EtudePerf:

```
| TheServer = Naming.Lookup("rmi://1.1.1.1/EtudePerf")
```

In our example, the client establishes a connection with a dispatcher in order to retrieve a valid reference about the object that will provide the data. The dynamic schema of this connection is as follows:



This schema generates the following code:

```

IPasseportPerf myPasseportPerf;
IPasseportPerfDispatcher myPasseportPerfDispatcher;
try{
    myPasseportPerfDispatcher=(IPasseportPerfDispatche
r) \
Naming.lookup("rmi://" +serverIP.trim()+"/PasseportPerfD
ispatcher");
    myPasseportPerf=(IPasseportPerf)myPasseportP
erfDispatcher.connect();
    passeportPerfID=myPasseportPerf.getID();
} catch(RemoteException e) {
    System.err.println("RemoteException");
    e.printStackTrace();
}
catch(java.net.MalformedURLException ex) {
    System.err.println("MalformedURLException");
    ex.printStackTrace();
}
catch(NotBoundException ex) {
    System.err.println("NotBoundException");
    ex.printStackTrace();
}
}

```

If everything went well, we are now able to call methods from the server via its interface:

```
| myPasseportPerf
```

5.2.5. Development facility

RMI is very well integrated into Java language and its frameworks. This degree of integration makes for a greater ease of development than with COM or Corba.

The following points must be considered:

- RMI does not require using an IDL language other than Java.
- RMI makes object transfer by value possible, relying on Java's standard serialization mechanism.
- In addition to transferring object instance by value, RMI allows the exchange of the classes themselves. If necessary, this exchange is automatically triggered by RMI.
- Object lifecycle is automatically managed by RMI, thanks to a distributed garbage collector, which tolerates partial anomalies.
- Based on standard Java types, RMI does not require the definition of a specific mapping.

These are major features for a distributed object system. Not only are they powerful, but they also make building applications easier.

However, RMI only allows for communication between Java objects. But if one must work in this environment, RMI will be, in principle, a better choice than CORBA.

RMI can nevertheless be criticized on two points, which weigh down its utilization considerably:

- As with Corba and COM, developing under RMI means using a special compiler, which generates stub codes. This is a consequence of the static type definition system found in Java.
- All distributed objects must be inherited from a specific class (the RemoteObject class); this poses many problems. In particular, this means that one must determine whether a class is to be distributed or not while it is being developed, which is regrettable.

RMI will undergo some major functional changes with the JDK 1.2 such as a system for automatically activating and disabling objects.

It is easy to develop small, simple applications that use RMI. However, as is the case with other distributed object systems, the use of RMI for enterprise-level applications remains very complicated. One needs to take difficult problems into account: deadlocks, reentry, multi-process cooperation, management of the accidental disappearance of a system component, management of network hitches, performance problems, referential integrity not provided by the system, integration of object distribution and multi-thread programming, etc. These problems clearly necessitate the help of a developer highly experienced in distributed objects.

5.3. JAVA-IIOP (Corba)

5.3.1. Implementation of the architecture

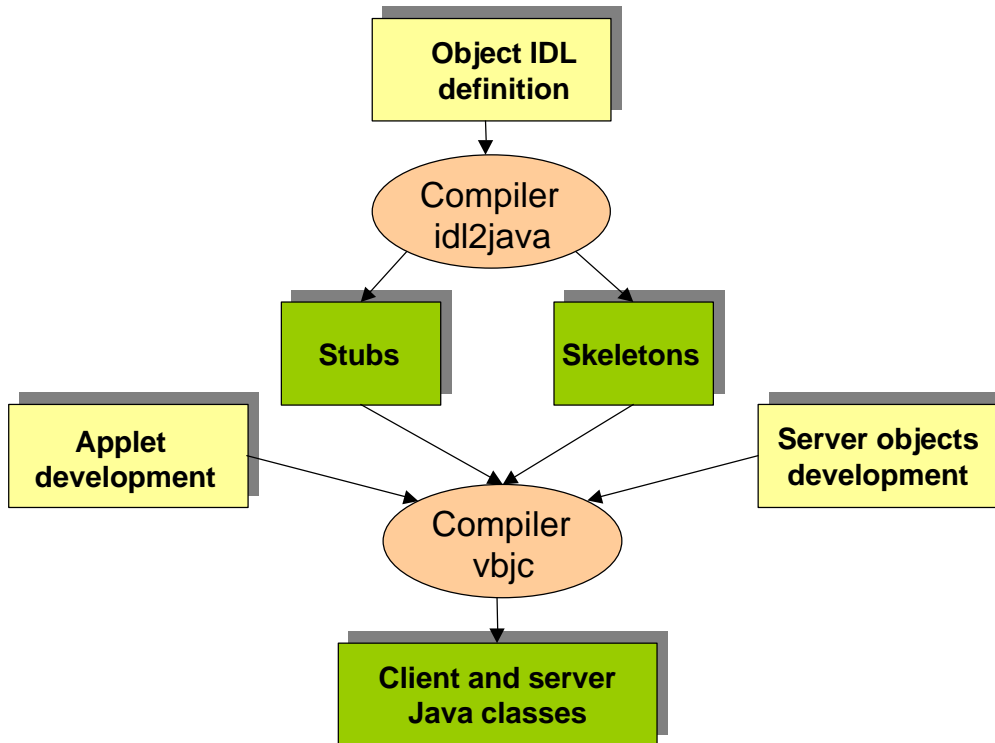
The test application was developed in Java, with the JDK 1.1.5 in Windows NT 4.0. The ORB selected is from Inprise (ex-Borland), namely Visibroker for Java 3.1. This product was added to Inprise's catalogue after the takeover of Visigenic.

Visibroker complies with the OMG CORBA 2.0 specification. It implements a portion of Corba's 15 services. As with many ORB editors, it introduces a number of mechanisms to facilitate the developer's work (launching of server objects when invoked, access to a server object through an URL, etc.) and the administrator's work (the Gatekeeper option may act as a proxy for Corba objects, so as to comply with firewall security policies, for instance). The ability to establish a dialogue with objects from other editors is guaranteed by the respect of IIOP protocol.

Visibroker provides several ways in which a client can connect to a Corba server. We made it a rule to comply with the following constraints:

- the server was launched and is waiting for connection requests,
- the client must be able to bind itself to the server, whatever its location over the network.

The technical solution chosen was to use the *Smart Agents* from Visibroker and to rely on standard stub and skeleton mechanisms detailed in chapter 2. This mechanism is an extension of Corba specifications and enables one to automatically obtain the reference of an object over the network (IOR). Once this choice was made, development could start.



The above diagram shows the various development stages. The light rectangles represent the code writing steps: definition of object IDL (Interface Definition Language), writing of the client applet and Corba server. The dark rectangles show the various classes that were automatically generated by the Visibroker tools.

5.3.2. IDL definition of the objects

One of the essential components of CORBA specifications is certainly the Interface Definition Language (IDL). The IDL interface is the core of any CORBA object, as it is what enables each object to communicate its methods and attributes to the other objects through the Object Request Broker (ORB).

This language enables one to associate the CORBA types to the ones of the language chosen to develop the objects, and to leave the choice of a development language up to the developer. The IDL language is based on the lexical rules of the C++. New keywords were introduced, to take the concept of distributed objects into account.

Here follows the IDL file, which defines the interface of the Dispatcher and Connection server objects. This file is called SQLObject.idl:

```

module SQLObject {
interface Connection {
struct Categorie {
    long ID;
    string Name;    };
struct Manufacturer {
    long ID;
    string Name;    };
struct Dealer {
    long ID;
    string Name;    };
typedef sequence<Dealer>TabDea;
typedef sequence<Manufacturer>TabManuf;
typedef sequence<Category>TabCat;
struct Product {
    long ID;
    string Designation;
    string PartNumber;
    string RetailPrice;
    string PurchasePrice;
    string Availability;
    Category Cat;
    Manufacturer Manuf;
    TabDea Dea;    };
typedef sequence<Product>Recordset;
TabDea getDealers();
TabCat getCategories();
TabManuf getManufacturers();
Recordset getProducts(in long Number);
Recordset getNextProducts(in long Number);
long insertProduct(in Product product);
long getID();
long close();    };
interface Dispatcher{
Connection connect();
long disconnect(in long ID); };
};

```

Once defined, the *idl2java* compiler creates the classes necessary for Corba infrastructure (including stubs and skeletons).

5.3.3. Development of the client part

The entire functional code of the RMI applet was used again and did not need any further development. Only the communication mechanisms with the server were modified. Code modifications concerned two points: access to the ORB, and the use of object tables instead of vectors to transmit the results of the method callings (Cf. Development of the server part).

The code needed to interface with the server can be broken down as follows:

- Initialization of the ORB

```
| org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this)
```

- Localization of the server object with Visibroker's *Smart Agents* mechanism. In order for this mechanism to operate, an *osagent* must be running on one of the network's stations, ready to answer this type of request. If the *osagent* knows the `SQLObjectManager` object, it will send its IOR to the client. This IOR can locate the server over the network in a unique way.

```
| server=  
| SQLObject.DispatcherHelper.bind(orb, "SQLObjectManager")  
| ;
```

- Object method calls can be performed transparently: in this case, the `connect()` method of the `Dispatcher` object, described in the IDL.

```
| manager=server.connect();
```

5.3.4. Development of the server part

The implementations of the Dispatcher (*DispatcherImpl.java*) and Connection (*ConnectionImpl.java*) objects use 95% of the code from the RMI version. Changing from one version to another is thus very fast. The only difference between the two technologies is the way in which data is transported between the client and the server. As the Java vectors do not correspond to any IDL type, the solution used in the RMI version could not be kept. We thus relied on tables as the return type of the various methods of our objects. These implementation files contain the business logic of the application.

A third file, *Server.java*, is necessary. Its role is to record the Corba server object on the *osagent* and to instantiate it, so that it becomes accessible to the requests from the clients. In operation, this is the server object that is run on the server station (*vbj Server*).

```
public class Server {
public static void main(String[] args) {
// ORB initialization.
org.omg.CORBA.ORB orb =
org.omg.CORBA.ORB.init(args,null);
org.omg.CORBA.BOA boa = orb.BOA_init();

// Creation of the Dispatcher object.
SQLObject.Dispatcher manager = new
DispatcherImpl("SQLObjectManager");

// Indication of the object's proper instantiation
boa.obj_is_ready(manager);
System.out.println(manager + " is ready.");

// Waiting for requests
boa.impl_is_ready();
}
}
```

The name used to reference the Dispatcher object is **SQLObjectManager**, which is found in the client's code.

5.3.5. Mechanism used for communication

The communication relies on IIOP and is performed via stubs and skeletons. We chose a type of communication corresponding to our specifications, but more complicated mechanisms do exist:

- **The dynamic invocation:** a client program may obtain the object's interface at runtime and build its requests dynamically.
- **The automatic activation:** it allows for the activation of a server only when a client accesses it. Visibroker provides an Object Activation Daemon (OAD) for this purpose.
- etc.

5.3.6. Development facility

Apart from the need for structuring in regards to object development, Corba introduces an additional problem: the choice of a communication mechanism. Indeed, the various mechanisms available will throw the inexperienced developer off. Optimal use of Corba will require a perfect knowledge of the implementation provided by the chosen editor.

Development with CORBA is not immune to some heaviness. Its complexity is more or less equivalent to DCOM's, except that high-level graphical tools, which often hide these problems (like Visual Basic), are not yet widely available.

CORBA is intended to make the implementation of multi-language distributed object systems possible. It defines an object model and a map between this object model and the one of the various languages used for implementation. This map may be more or less successful, depending on the cases, and it often introduces difficulties. Putting oneself inside the framework of an abstract object model allows for interoperability between various concrete models, but also brings some complexity at the conceptual level.

The specification of a distributed object must be performed in IDL language, which is specific to CORBA. A compiler is provided with this language to generate stub codes, as with DCOM and RMI.

Corba specifies numerous services and API, of varying quality, which are not all implemented by the ORB. Besides, each ORB implements extensions to CORBA in a proprietary way.

Note that all ORBs are not equivalent, and that one will have to determine, for each context, which ORB offers adequate functions.

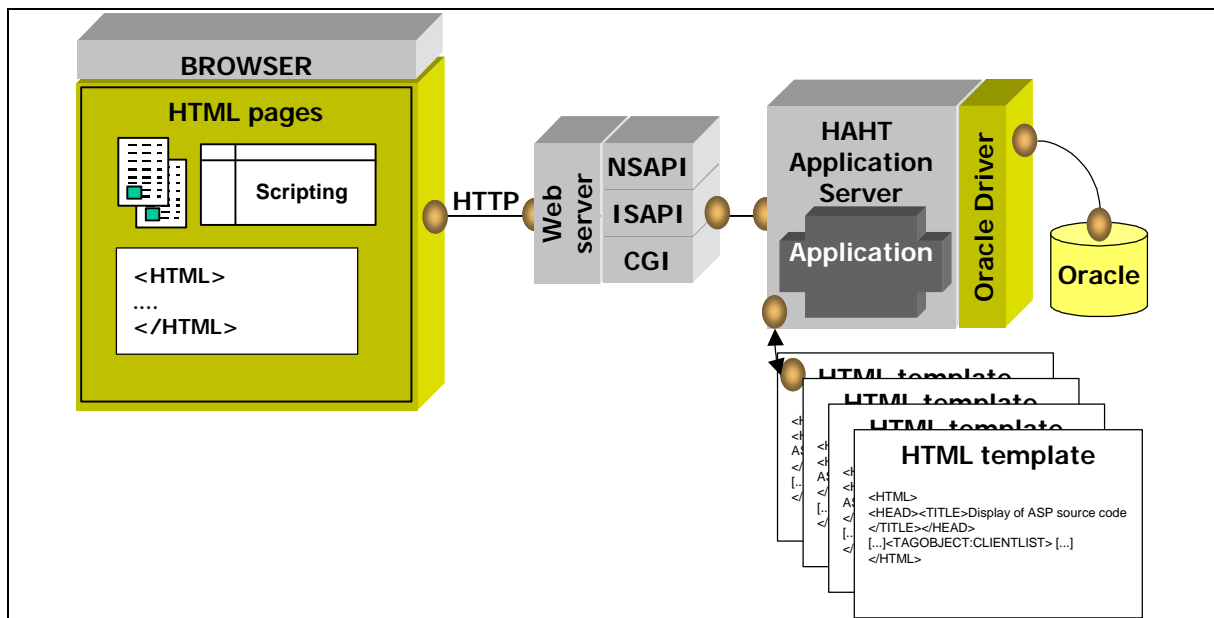
Producing robust, true enterprise-scale applications centered on Corba is difficult, and requires the help of specialists.

5.4. HTML-HTTP

5.4.1. Implementation of the architecture

In order to produce the test application, we had to choose a development tool. The HAHTSite product, from HAHT Software, was adopted because of its qualities in terms of ease and speed of development, as well as for its position on the market as a "purely HTML" tool. **Choosing this product or another one has no impact on network performance, inasmuch as the tool used leaves you in control during the generation of HTML pages.**

This is a traditional intranet architecture, based on an HTTP server. HAHTSite's server environment is made up of a CGI/ISAPI or NSAPI gateway (depending on the Web server used) and an application server ("Application Server"). The CGI/ISAPI/NSAPI gateway provides the link between the Web server and the application server.



Architecture implemented for the tests

5.4.2. Development of the client part

The development of the graphical interface in HTML was first performed statically, with a simple HTML editor. The pages produced were then imported into the development tool.

5.4.3. Development of the server part

The server part concerns DBMS access, and the few business rules of our application. As the tool used does not provide a true " object " approach, no prior modeling is necessary.

To develop database access, HAHTSite offers high-level components called Widgets. Indeed, the DB Table Widget performs a SELECT type SQL query in a database and displays its result set as an HTML table. Very easy to use , this component has nonetheless a few deficiencies, notably at the performance level.

Therefore, we developed this query with the help of HAHTTalk, the language of HAHTSite. This language is derived from Visual Basic for Application, and is trimmed with numerous Web-orientated libraries. The code necessary to perform a SELECT query is as follows:

```
Dim retvar as integer
Dim SQL As String
Dim Study As Database
Dim g_DynasetEval As Dynaset

retvar = GetConnection("Study",Study)
SQL="Select DESIGNATION,PN,RETAILPRICE,
PURCHASEPRICE,AVAILABILITY from PRODUCT"
Set g_DynasetEval=Study.CreateDynaset(SQL)
g_DynasetEval.Movefirst
print "<table><tr>"
While(Not g_DynasetEval.EOF)
    print "<td>"
    print g_DynasetEval.Fields(0).value
    print "</td><td>"
    print g_DynasetEval.Fields(1).value
    print "</td><td>"
    print g_DynasetEval.Fields(2).value
    print "</td><td>"
    print g_DynasetEval.Fields(3).value
    print "</td><td>"
    print g_DynasetEval.Fields(4).value
    print "</td></tr>"
g_DynasetEval.MoveNext
Wend
print "</table>"
```

In this case, the written code includes the database connection and communication aspects, as well as layout information (HTML). This development method is quite similar to the " Scripting " principle also found in the following development tools: Microsoft Visual Interdev, Allaire ColdFusion, etc.

5.4.4. Mechanism used for communication

Communication between the client and the server relies on HTTP. This part is transparent for the developer.

Management of the user context is handled by the application server, which assigns a unique identifier to each user of the application. This identifier is exchanged between the browser and the application server through the use of cookies or coded URLs.

5.4.5. Development facility

The difficulty of HTML Web application development is linked to the development tool used, but also to the variety of languages one will have to master.

Indeed, besides the language used by the development platform (VBA for HAHTSite, Java for NetDynamics or SilverStream, VBScript/JavaScript for Visual InterDev, etc.), the developer will need to know HTML, as well as the subtleties of the client-side scripting language, namely JavaScript.

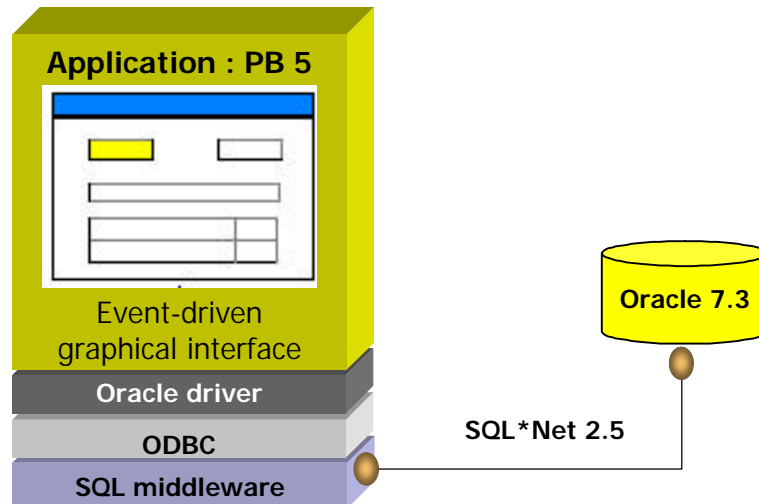
Lastly, the biggest challenge remains the design of the graphical interface and the application's logic.

5.5. Client/Server

5.5.1. Implementation of the architecture

The client/server architecture application was developed with PowerBuilder 5, a reference in the world of 4GLs. Even if version 5 of PowerBuilder allows for the development of three-tiered architecture applications with the help of a proprietary RPC system, we preferred a two-tiered architecture, which better represents the majority of client/server applications in Windows.

PowerBuilder relies on painters, which provide a graphical environment and permit the creation and management of such various objects as windows or data access objects.



5.5.2. Development of the client part

PowerBuilder's development philosophy, like many other client/server tools is based on very high-level graphical objects which are widely open to parameterization and can handle data coming from a DBMS.

This approach enables one to quickly build applications that interact strongly with the database. However, separation of the layout and the processing parts of the application are not possible with this approach (RAD-oriented).

The principle used to develop this application is as follows:

- Selection of a data source via wizards
- Parameterization of the display (in the form of a list, etc.)
- Integration of the graphical object in the application.

With PowerBuilder, the DataWindows object handles both database access and layout functions (lists, master/detail relationships, forms, graphs, etc.)

This object has extensive features. In particular, it manages the partial display of a data list without any programming (this corresponds to scenarios 2 and 3 of our tests).

5.5.3. Development of the server part

Within the context of this study, this section has no object for the two-tiered client/server model, as the only server component is the database.

Nevertheless, we may note that the databases have played the role of the application server for a long time, through the stored procedures.

5.5.4. Mechanism used for communication

Exchanges between the client and the DBMS rely on the middleware provided by the database vendor. For our tests, we used SQL*Net 2.3 to communicate with an Oracle 7.3 database.

5.5.5. Development facility

This aspect is strongly linked to the development tool. But generally speaking, due to the maturity of the client/server development tools, it is easier to develop a Windows application using a relational DBMS than any intranet application. This is always true for small- to medium-size applications.

On the other hand, the time "lost" during the development stage with intranet applications may be regained quickly during the deployment stages, a characteristically weak point of the client/server models.

6. Methodology

6.1. Test platform

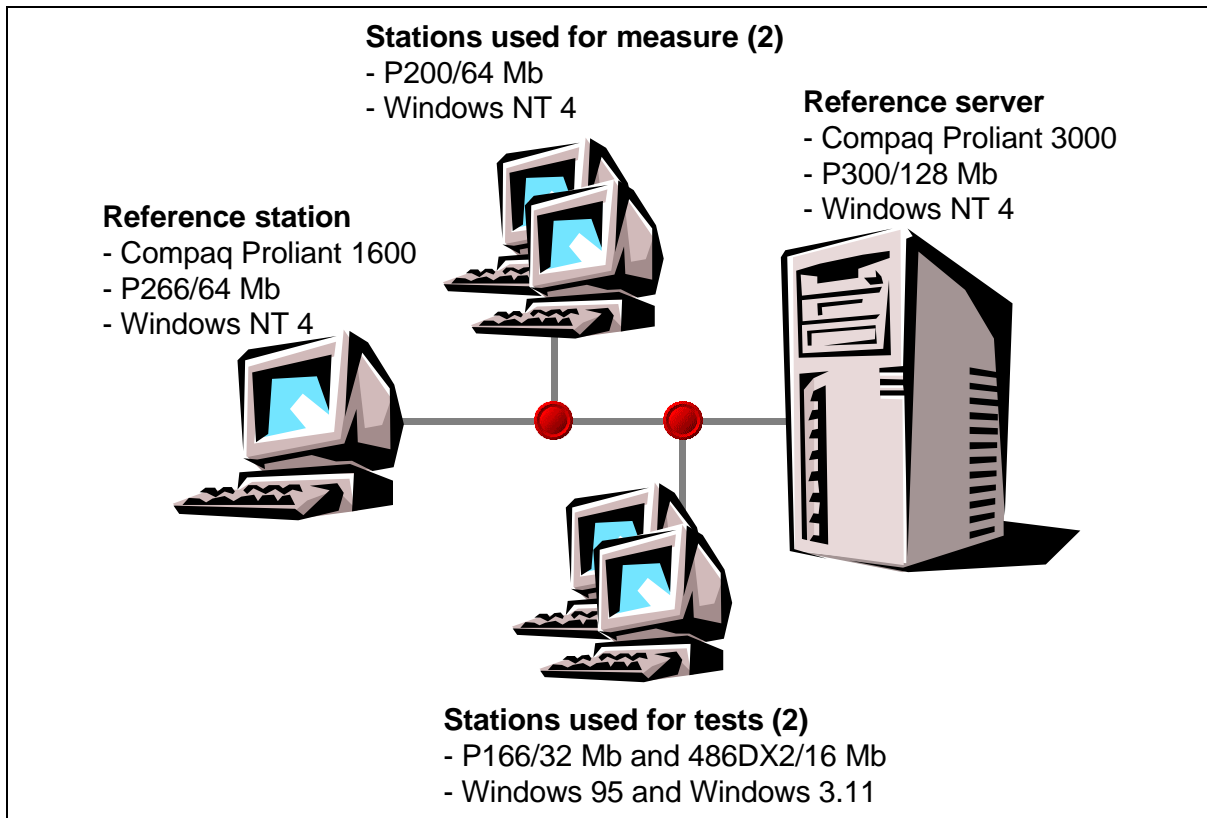


Diagram of the test platform

■ Network

Our network is made up of a 10-Mb hub, isolated from all networks. Each computer (client and server) is connected to the network with a fixed TCP/IP address. The services launched at the network level are limited to the bare minimum, so as to generate as little traffic as possible.

■ Server

Reference server	
Manufacturer/ Model	COMPAQ PROLIANT 3000
Processor	Pentium 300 MHz
Memory	128 Mb
Hard Disk	6 Gb
System	Windows Server NT 4.0, SP3
Web server	IIS 4.0 - Option Pack
DBMS	Oracle 7.3

■ **Clients**

Reference station

This station runs the test scenarios at the browser level. It also collects information about the user-level response times.

Manufacturer / Model	<i>COMPAQ PROLIANT 1600</i>
Processor	<i>Pentium 266 MHz</i>
Memory	<i>64 Mb</i>
Hard Disk	<i>4 Gb</i>
System	<i>Windows Server NT 4.0, SP3</i>
Browsers	<i>Internet Explorer 4.0.2 Netscape Communicator 4.06</i>

Station used for measures (2)

These stations are used to record network communication measures, as well as those of the server.

Processor	<i>Pentium 200 MHz</i>
Memory	<i>64 Mb</i>
Hard Disk	<i>2 Gb</i>
System	<i>Windows Workstation NT 4.0, SP3</i>
Software	<i>Performance Monitor SMS Network Monitor Visual Test</i>

Station used for tests (2)

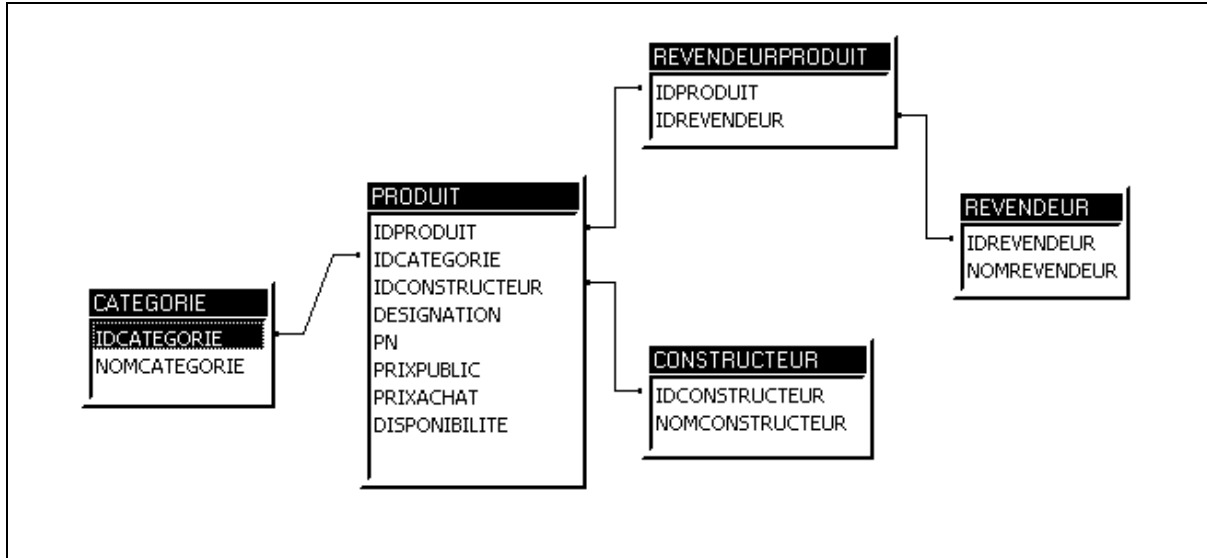
These stations are used to validate the deployment of the applications in multiple configurations.

Processor	<i>Pentium 166 MHz / 486DX66</i>
Memory	<i>32 Mb / 16 Mb</i>
Hard Disk	<i>1.5 Gb / 500 Mb</i>
System	<i>Windows 95 / Windows 3.11</i>
Browsers	<i>Internet Explorer 3.0.2 Internet Explorer 4.0.2 Netscape Navigator 3 Netscape Communicator 4.06</i>

6.2. Specifications of the application

The application chosen for these tests is deliberately very simple, and uses mechanisms found in any business application. It involves simple and mass displays, as well as data input and transaction activation.

■ A database with the following data model



The database is implemented via Oracle 7.3.1 DBMS in Windows NT. The base is loaded with the following volumes of data:

Tables	
PRODUIT (product)	4000 lines
CATEGORIE (category)	12 lines
CONSTRUCTEUR (manufacturer)	12 lines
REVENDEUR (reseller)	6 lines
REVENDEURPRODUIT (resellerproduct)	20000 lines

■ An application used for display and input:

- a menu
- list display, either partial or full, in a table
- data input in a simple form

Depending on the architecture chosen, the interface is implemented differently. However, the application logic remains the same. In the end, one obtains a version of the interface with the following technologies:

- HTML

6.3. Method used for tests and measurements

■ *Test scenario*

Step 1: Initialization

- The browser or the application (for the client/server tests) is loaded.
- The application opens (downloading of an HTML page and/or of the applet and ActiveX components)
- Connection is established with the application server, which connects to the database.

Step 2: Partial fetch

- The first 50 lines are loaded from the database (1 Select SQL query)

Step 3: Partial fetch, 5 times

- The following 250 lines are loaded, 50 lines at a time (5 Select SQL queries)

Step 4: Data insertion

- The data insertion window or form is loaded
- 50 products are inserted (50 times 1 Insert Product query + 5 Insert ProductDealer queries, which makes 300 Insert queries)

Step 5: Fetch of 4000 lines

- 4000 lines are loaded from the database (1 Select SQL query)

■ *Test scripts*

The test scripts, which drive the applications, were developed with the Visual Test tool (Microsoft/Rational). For each previously defined scenario, a script is recorded at the level of the browser (HTML pages, applet and ActiveX) or the Windows application. These scripts permit the measurement of response times at the client station level, and the synchronization of other measurement tools over the network, namely Network Monitor and Performance Monitor.

■ Network Monitor

Network Monitor is a tool from Microsoft, which is included in the SMS package. It is a trace and measurement tool, which works at the network level (commonly called a network "sniffer"). This tool captures the traffic between several computers over a network. To limit the risk of measurement errors at the network level to a minimum, the test platform is a fully isolated, independent network. Only the mandatory services are activated.

■ Performance Monitor

Performance Monitor is a measurement tool provided with Windows NT, which analyzes computer behavior according to such numerous parameters as memory usage, the percentage of CPU time required by an application, the volume of input/output at the level of network interfaces, etc.

This tool provides a graphical representation of application behavior on a Windows NT computer.

■ Measurements

Even though the main objective of this study was to define the bandwidth use over the network with several architectures, we performed performance measurements at the three following levels:

- On the client station, we measured the response times, as seen by the user.
- Over the network, we measured the amount of information exchanged.
- On the server station, we measured the load generated by the client requests.

Crosschecking these measurements provides us with a thorough analysis of the mechanisms implemented in these different architectures.

REPORT PERFORMED BY

TechMetrix Research

SQLI's
Department for Research & Development

Frédéric BON
Jean-Christophe Cimetière
Philippe Mougín
Christophe Noblanc

IN PARTNERSHIP WITH

ALKANE

European headquarters
TechMetrix / SQLI
55/57, rue Saint-Roch
75001 Paris
France

tel: +33 1 44 55 40 00
fax: +33 1 44 55 40 01
email: r&d@sqli.fr
Web: www.sqli.fr

US Branch
TechMetrix Research
6 New England Executive
Park, Suite 400
Burlington, MA 01803

Tel: (781) 270-7486
fax: (781) 270-7489
email: info@techmetrix.com
Web: www.techmetrix.com

ALKANE
18, rue du Docteur Roux
75015 Paris

Tel.: (33) 1 44 49 26 86
Fax: (33) 1 44 49 26 87
Email: info@alkane.com
WEB: www.alkane.com

Publication date: November 1998

Copyright:

Any publishing of written material, musical composition, drawing, painting, or any other production, wholly or partly printed or engraved, contrary to the laws and regulations relating to copyright, is an infringement of copyright; and any infringement of copyright is a misdemeanor.