THE ART OF DISTRIBUTED DEVELOPMENT OF MULTI-LINGUAL THREE-TIER INTERNET APPLICATIONS

Dragomir D. Dimitrijević
Consultant

I INTRODUCTION

In this article we describe the author's experience with the unconventional development of Internet applications. They were developed for Credit Suisse bank as a joint cooperation between the bank as the customer, KJsoft GmbH as their contractor, and the author as a subcontractor. Software was developed in a distributed fashion without any physical access to the production site.

Due to the very strict bank's security rules, previously developed applications used by the newly developed applications were not available for installation on the remote development site. For that reason, simple stubs were developed to emulate the behavior of previously developed but unavailable, CORBA (Common Object Request Broker Architecture) and database applications. In addition, the application had to support multiple spoken languages, thus the developed software had to be internally independent of any particular spoken language.

In this article we describe a number of usefull tips and tricks of trade that may be helpful to an intendend reader. In Section II, we describe the three-tier system architecture. In Sections III and IV we describe development of CORBA and database portions of the applications. In Section V, tips on multi-lingual application development are given. The conclusion is given in Section VI.

II SYSTEM ARCHITECTURE

Figure 1 depicts the three-tier system architecture typical for Internet applications. Users use web browsers to access various online banking applications via the Internet. Applications are executed by a web server. An example of such an application is quotation of currency exchange rates. The user selects desired currencies and a branch of bank on a query input form and submits the query. The web server accepts the query, processes it, and returns the response back to the user's browser. Depending on the particular application, the web server may consult with a CORBA application server and/or a database server. The response is returned in the user's language of choice (German, French, Italian, or English).

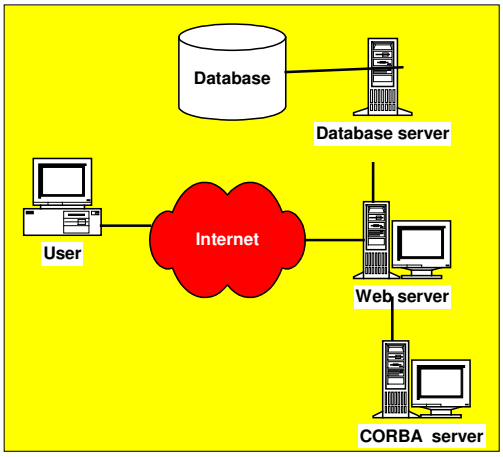

Figure 1: System architecture

Per bank's internal software development standard, all Internet applications, executed by the web server, are written using Java™ programming language and Java servlets. Although people from the Microsoft™ camp will most certainly disagree, this is a de facto standard for writting serious Internet applications.

All servers in the production environment run Sun Microsystems Solaris™ UNIX operating system. The web server is NES™ (Netscape Enterprise Server) with the addition of JRun™ engine for running Java servlets. The database server is Oracle™. The CORBA application server is IONA OrbixWeb™. CORBA clients use an internally developed API (Application Programming Interface) and wrapper Java clases developed on top of OrbixWeb.

The challenge in this project was to develop software in a distributed fashion without any physical presence at the production site, while still adhering to the very strict bank's security rules.

First, per bank's development protocol, software developers do not have direct physical access to the production system. Instead, the developed software is handed over to the production system staff for final testing and installation on the production system.

Second, for security reasons, the already developed software cannot be taken out of the bank's premises for installation on a remote development system. It means that copies of the database and the applications running on the CORBA server were not available at the development site. Instead, stubs had

to be developed to emulate behavior of CORBA and database servers.

The entire software described in this article was developed on a single Windows NT Workstation™. The CORBA server was one that comes with the JDK (Java Development Kit). The database was Microsoft Access™. The web server was Apache with the JServ engine for running Java servlets. Software was developed usjing Oracle JDeveloper™ IDE (Integrated Development Environment). Obviously, the development and the production environments were very different, which was one of the development challenges.

The only contact between the development and the production sites was over the phone and via e-mail, thus software was developed solely in a telecommuting fashion. Once the database and CORBA stubs were set-up on the development site, and a basic skeleton of the application was set-up on the production site, it was easy to gradually build the application on the development site and test it on the production site. Software was shipped via e-mail in the form of compiled JAR (Java Archive) files and static text, HTML (Hyper-Text Markup Language), and graphic files.

Throughout the rest of this article, we will describe some of the tricks of trade used to overcome the development challenges.

III CORBA IMPLEMENTATION

CORBA standard was, at least in theory, developed to standardize invocation of remote procedures in networks. However, this is far from reality. In theory, the development of code which invokes remote, already developed procedures, involves the following steps:

1. Use the remote procedure's IDL (Interface Definition Language) specification, and an IDL compiler to generate API stub code for invoking remote procedures in the desired programming language (Java, C, C, C++).

2. Develop code for initiating the ORB (Object Request Broker) within the application.

3. Develop code for invoking remote procedures from within the application.

However, in practice, there are a number of problems:

- CORBA applications developed in different programming languages may have problems talking to each other even when the development tools and the underlying libraries are produced by the same vendor.

- Java API specification is developed to standardize API and IDL stubs of all Java applications, thus maximize code portability. However, vendors of CORBA development tools like IONA did not adhere to this standard, so software developed using JDK and it's IDL compiler and CORBA name server cannot run using IONA's CORBA name server.

- Even different CORBA development tools of the same vendor, like IONA's OrbixWeb and Orbix 2000, are not mutually compatible, and require different application code.

Fortunately, the differences and incompatibilities in the application code apply mostly to a relatively small portion of the ORB initiation code. For that reason, it was possible to develop code using JDK CORBA environment and port it to the OrbixWeb production environment as follows:

1. Use JDK IDL compiler to compile IDL specification and generate Java stubs for the development environment.

2. Develop and test the application using the ORB initiation code appropriate for the JDK environment.

3. Use OrbixWeb IDL compiler to compile IDL specification and generate Java stubs for the OrbixWeb production environment.

4. Replace the ORB initiation code with the code needed for the OrbixWeb production environment.

Once this porting procedure is established, delivery of code modification is very efficient with a little help of code building scripts.

The other problem with the development of CORBA code was unavailability of the original CORBA application server. This problem was solved by developing a simple stub application server which emulates responses of the real application server. The stub server loads test data from a text file and upon request passes it to the CORBA client, i.e., to the requesting servlet in this case.

IV DATABASE IMPLEMENTATION

Portability of JDBC (Java Database Connection) code is significantly better than CORBA related code. As long as database operations are restricted to standard SQL (Structured Query Language) and free of triggers and stored procedures, developed Java code runs on virtually any database. Porting to a different database type performed by simply specifying a differen database source and driver in a database configuration textual file such as:

```
JDBCDriver = sun.jdbc.odbc.JdbcOdbcDriver
JDBCConnectionURL = jdbc:odbc:DbSource
```

The above two lines define database source named DbSource defined in the Windows ODBC (Open Database Connection) manager and the Sun Microsystems' JDBC-ODBC bridge database driver. By modifying the two lines in the database configuration file, one can switch from, e.g., Microsoft Access database at the development site to Oracle database at the production site. As a matter of fact, this approach is so convinient that the author used it in many other Java projects that involved databases. Microsoft Access allows quick prototyping and modification. Once the database design is finalized, the database can be ported to Oracle using an Oracle database porting tool. In addition, this approach

allows the use of a laptop computer for demonstration of work in progress at a customer's site.

This approach was used in the project described in this article to create the database stub which emulates behavior of the database at the production site. Since the application used a small subset of tables and fields in the actual database, replication of their structure at the development site was quick and easy.

When it comes to database Internet applications, another trick worth mentioning is the use of database connection pools. A typical servlet-based Internet application that uses a database involves three steps when a servlet is invoked: connecting to the database, accessing data, and disconnecting from the database. Connecting to the database is a time-consuming operation. For that reason, pools of pre-established database connections are maintained. Each servlet maintains a connection pool which consists of a configurable number of pre-estabkished database connections. Instead of waiting for the connection to be established, a database request takes an already established connection from the pool, uses it, and later returns it back to the pool for further reuse. The use of connection pools significantly improves the application's performance. Oracle's JDeveloper IDE comes with a library that implements a connection pool manager. However, the author uses one of many connection pool implementations available for download from the Internet.

V MULTI-LINGUAL IMPLEMENTATION

Software described in this article had to support four languages, i.e., the user had to be able to submit queries and receive responses in the selected preferred language. Figure 2 depicts a general appearance of the user interface displayed in the user's web browser.

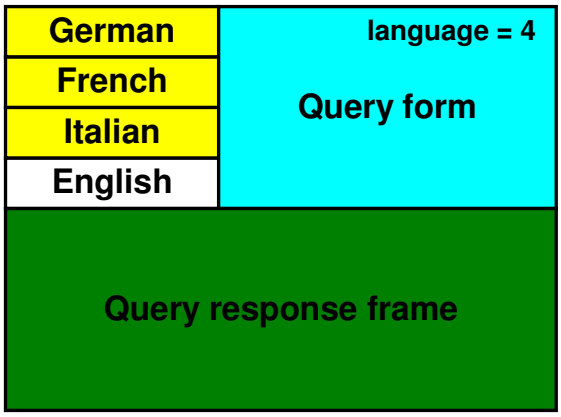

Figure 2: General appearance of the user interface

The user interface consists of two HTML frames. The upper frame is a static HTML page, written in the user's language of choice. It contains the language selection menu on the left side, and the query input form on the right side. The query form contains a hidden form parameter `language` which defines the form's language. For example, `language=4` corresponds to English language. This parameter is submitted together with other query parameters so that the servlet knows it has to respond in English language.

The lower frame contains response to user's queries. Responses are dynamically generated by Java servlets.

Figure 3 shows the organization of Internet directories which contain different file types and files which correspond to different languages. A good initial organization of directories allows easy maintenance at a later time.

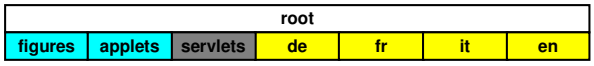

Figure 3: Organization of the Internet directories

The `root` directory contains static HTML files common to all languages such as the frameset HTML file which holds the two frames shown in Figure 2. The `figures` directory contains graphic files common to all languages. The `applets` directory contains Java applet classes and JAR files that contain applets.

The `servlets` directory is not a physical directory. It is a virtual directory mounted onto the web server's directory system so a servlet is invoked as it is in a real directory such as `http://www.aaa.ch/servlets/rates?p=a`. The `rates` is a servlet's external name followed by zero or more query parameters, depending on the query and the method used to invoke the servlet (PUT or GET).

The four remaining directories, `de`, `fr`, `it`, and `en` contain static HTML and graphic files which correspond to German, French, Italian, and English languages respectively. Names and functions of files they contain are same except that their contents are written in different languages.

In order to make software internally independent of any particular language, all language-dependent strings are referenced using their internal language-independent symbolic names. To achieve this, we used one instance of `java.util.Properties` class ([1]) for each language known to the application. The `Properties` class contains a set of name-value pairs of strings. A method `getProperty(String name)` and the string's symbolic name are used to retreive string in a specific language.

An instance of the `Properties` class may be loaded from a plain text property file which contains lines that have a form `name=value`. In order to improve readability of property files, we used a convention that property names have a form `function.name`. For example, properties

```
title.RPT_TITLE=Currency Exchange Rates
label.COUNTRY=Country
code.0119=Yugoslavia
error.DB_ERROR=Database error
```

define a report title, a label, a country name internally referenced using country's ISO (International Standards Organization) currency code, and an error message. Property files for other languages have the right-hand side of the equality sign translated to the corresponding language. During the initialization, an application loads one property file (vocabulary) for each language it supports, and creates an array of instances of `Properties` that correspond to

supported languages. Language-specific strings are retreived using the internal string name and the language code received from an HTML query form. The following code may be ised to translate internal language-independent string names to language-specific strings:

```
private Properties[] vocab[4];
String getString(String name, int lang){
  String r;
  if (lang>=1 && lang<=4) {
    r=vocab[lang-1].getProperty(name);
    if (r!=null) return r;
  }
  // Return untranslated name if
  // undefined language or
  // undefined string requested
  return name;
}
String getTitle(String name, int lang){
  return getString("title."+name, lang);
}
String getLabel(String name, int lang){
  return getString("label."+name, lang);
}
String getCountry(String name, int lang){
  return getString("code."+name, lang);
}
String getError(String name, int lang){
  return getString("error."+name, lang);
}
```

Using the above code, the four examples of strings may be translated to English as:

```
int lang=4;
String t=getTitle( "RPT_TITLE", lang);
String l=getLabel( "COUNTRY", lang);
String c=getCountry( "0119", lang);
String e=getError( "DB_ERROR", lang);
```

Each of the four methods invoked above will append an appropriate prefix to the internal string name and retreive its translation from the instance of `Properties` vocabulary which corresponds to English language code. This approach allows clean and orderly control of application's multi-lingual behavior. Furthermore, even in case of single-language applications, fine tunning is easier since strings displayed in browsers are not hard-coded, thus they are accessible for quick modification and customization in property files.

The property files may be stored in and retreieved from the file system. This approach allows customers, i.e., system administrators of the production site, to edit them and fine tune appearance of strings in multiple languages. The other approach is to put property files in the application's class hierarchy and pack the entire class hierarshy in a JAR file. Instances of `Properties` are then loaded as application's resources. In such a way, all application's classes and propertiy files can be delivered as a single JAR file which simplifies delivery and installation of the application. In case of multi-lingual applets, this is the preferred way to deliver applets to the user's browser.

VI CONCLUSIONS

In this article we have described the author's experience with the distributed development of multi-lingual three-tier Java/ CORBA/database applications. The challenge was to develop applications at a remote development site in a purely telecommuting fashion, i.e., without any physical access to the production site, and in some cases without all software and hardware components needed to replicate the production system at the development site. We believe that tips and tricks of trade described in this article could be of great use to other software developers.

REFERENCES

[1] -, *Java™ 2 Platform, Standard Edition, v1.2.2 API Specification*.

Abstract: In this article we describe author's experience with the distributed development of multi-lingual three-tier Java/ CORBA/database Internet applications. We believe that the described tips and tricks of trade may be of great use to readers who are involved with Java applications development.

THE ART OF DISTRIBUTED DEVELOPMENT OF THREE-TIER INTERNET APPLICATIONS, Dragomir D. Dimitrijević.