An Evaluation of the Java-based Approaches to Web Database Access*

Stavros Papastavrou[1], Panos Chrysanthis[1], George Samaras[2], Evaggelia Pitoura[3]

[1] Dept. of Computer Science, University of Pittsburgh
{stavrosp, panos}@cs.pitt.edu
[2] Dept. of Computer Science, University of Cyprus
cssamara@cs.ucy.ac.cy
[3] Dept. of Computer Science, University of Ioannina
pitoura@cs.uoi.gr

Abstract. Given the undeniable popularity of the Web, providing efficient and secure access to remote databases using a Web browser is crucial for the emerging cooperative information systems and applications. In this paper, we evaluate all currently available Java-based approaches that support persistent connections between Web clients and database servers. These approaches include Java applets, Java Sockets, Servlets, Remote Method Invocation, CORBA, and mobile agents technology. Our comparison is along the important parameters of *performance* and *programmability*.

1 Introduction

Providing efficient and secure access to remote databases using a Web browser [2,6] is crucial for the emerging cooperative information systems, such as Virtual Enterprises. A number of methods for Web database connectivity and integration have been proposed such as CGI scripts, active pages, databases speaking http, external viewers or plug-ins, and HyperWave [9]. These methods enhance the Web server capabilities with dynamic functionality for interactive and cooperative applications to create database connections, execute queries and transactions, and generate dynamic Web pages. However, there is an increasing interest in those that are Java-based due to the inherent advantages of Java, namely, platform independence support, highly secure program execution, and small size of compiled code.

Several Java-based methods are currently available that can be used for the development of Web cooperative information systems but in the best of our knowledge, there is no quantitative comparison of them in a database context. Existing studies either primarily focused on the various server side scripting mechanisms to support database connectivity (e.g., [8, 12]), or evaluated the Java client/server

* This work was partially supported by NSF IRI-9502091 and IIS-9812532, and AFOSR F49620-98-1-043 awards.

communication paradigm without any database connectivity or lengthy computations (e.g., [14]). This experimental paper contributes a comparison of the six Java-based approaches that support persistent database connections, specifically, Java applets using JDBC (Applet JDBC), Java Sockets, Java Servlets, Remote Method Invocation (RMI), CORBA, and Java Mobile Agents (JMA). We focus on these methods because of their support for *persistent* database connections, which are essential for cooperative environments with long, and repeated data retrievals and updates.

For our evaluation, we used each approach to implement a Web client accessing and querying a remote database. Each approach differs in the way the client establishes connection with remote database servers with the help of a middleware and the implementation of the middleware. Depending on the way the client establishes connection with the middleware, the approaches can be classified as (1) *non-RPC* ones, that do not provide for remote method invocation mechanism, (2) *RPC* ones with clear remote method invocation semantics, and (3) *RPC-like* ones involving mobile agent technology.

We compared the behavior of the different approaches along the following two important parameters: (1) *performance* expressed in terms of response time under different loads, and (2) *programmability* expressed in terms of the number of system calls at the client and the server site. The two salient results of our study are: (1) Best performance is not always achievable with high programmability and low resource requirements, and (2) the mobile agent technology needs to improve its programmability while giving particular emphasis in its infrastructure.

In the next section, we provide a brief review of Java and Java database connectivity. In Section 3, we first discuss our experimental testbed and then elaborate on the implementation details of the six approaches under evaluation. In Section 4, we discuss our performance evaluation results whereas in Section 5, we compare the different approaches from programmability point of view.

2 Background: Java and Java database connectivity

Java [17,1] is an object-oriented programming language designed to support the development of distributed, secure, and portable applications. The uniqueness of Java lies on the fact that its compiled code can run on any platform, which supports a Java runtime environment. Further, Java programs can run in Java-enabled Web browsers in the form of applets, which are downloaded as part of an html page. Security is achieved by restricting the execution of applets within the context of the client's web browser, and by permitting the communication of applets only with their originating web server. That is, Java applets are not allowed to access any system resources or communicate with any arbitrary site. Java's portability is further enhanced by other safety features, such as the absence of pointers, and automatic array bound check.

Two features of Java, important for building cooperative database applications, are: (a) its graphical interface library that supports the development of sophisticated

interfaces, and (2) its database connectivity interface (JDBC API) that facilitates application access to relational databases over the Web at different URLs [11].

The JDBC API is implemented by various drivers, executing under the control of a JDBC manager [19]. A JDBC driver can be implemented in four different ways, as shown in Figure 1. These drives differ in two significant ways: (1) the size of their downloadable code, and (2) in the way they support multiple database connections.

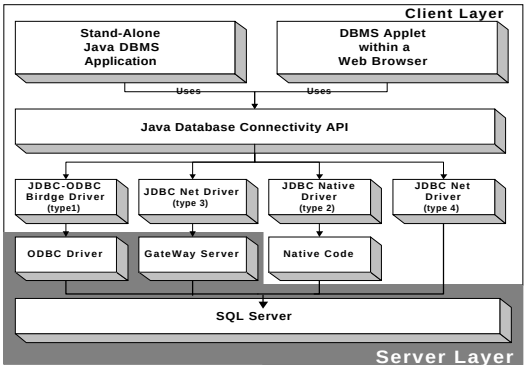


Figure 1: Standard JDBC Methodologies

The type 1 JDBC driver, namely, the *JDBC-ODBC Bridge* driver, translates JDBC calls to ODBC ones and is suitable to access databases with only ODBC interface. A type 1 driver requires remote clients to pre-install some ODBC binary code and is not designed to be downloadable by Java applets.

The type 3 JDBC driver, the *net-protocol fully Java technology-enabled* driver, is the most flexible with Java applets. It is written entirely in Java and can be fully downloaded at run time, requiring no code pre-installation. A type 3 driver translates a client's query into an intermediate language that is converted into a vendor-specific protocol by a middle-tier gateway. The more vendor protocols the gateway supports, the more databases a client can access, without downloading additional drivers.

3 The Experimental Testbed

We use each Java method to implement a Web client querying a remote database. Our testbed is structured along a three-tier *client/middleware/database* model. Two design principles were adopted in the selection of the various components during the development of the testbed. First, our Web clients should be lean for allowing fast downloads, and therefore increasing support for wireless clients. Second, no a-priori configuration of the Web client should be necessary to run the experiments in order to maintain portability, and therefore, support arbitrary clients.

Our Web client program is a Java applet, installed on a Web server along with an html page. Every experiment was initiated by pointing to the html page from a remote computer. After the Java applet was downloaded and initialized at the client computer, database connectivity was established, and queries were issued through the applet's GUI to be executed on the remote database server. Our remote database system, a 3-table Microsoft Access, was on the same machine with the Web server.

The role of the middleware is to accept client requests, execute them on the database server on behalf of the client, and return the results back to it. Due to security/communication restrictions of Java applets, part of the middleware in all approaches has to execute on the Web server machine. In the experiments reported here, because the database server co-resides with the Web server, the entire middleware in all approaches executes on the same machine. Given that an Access database can only be accessed using ODBC, the middleware of all approaches except Applet JDBC, use a JDBC-ODBC (type 1) driver to connect to the database. In the Applet JDBC approach, a type 3 JDBC driver is used whose gateway converts the JDBC calls into ODBC ones. To improve performance, the middleware attempts to connect to the database server when it is activated and before any client request is submitted.

In the rest of this section, we elaborate on the implementation of each approach. *Initialization phase* is the procedure for establishing database connectivity, and *execution phase* is the procedure for querying the database after database connection is established.

3.1 Non-RPC Approaches: Java Socket and Java Servlet

Both the *Java Socket* and *Java Servlet* approaches use sockets to connect a client and the middleware program. In the Java Socket approach, sockets are created by the clients, whereas in the Servlet approach, are created by the run-time environment.

3.1.1 The Java socket approach. In this first approach, the middleware is a standalone Java application server running on the Web server machine. The client collaborates with the application server by establishing an explicit socket connection [18]. Figure 2 illustrates the steps involved for the first query. The applet submits the query through the socket connection to the application server, which decodes the incoming stream of data, and executes the query on the database server. The result table is then passed to the client applet again by the means of data streams.

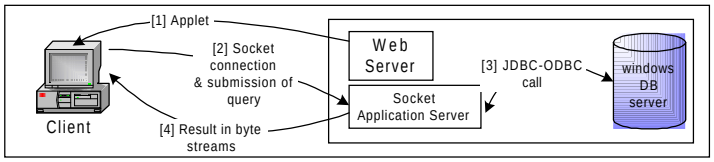


Figure 2: The Socket approach

The cost of the first query in this approach is

1. Initialization phase:
   A. The time for the client to open a socket connection with the application server.
2. Execution phase:
   A. The time for the client to pass to the application server the data stream containing the SQL statement.
   B. The time for the application server to execute the query, obtain the results and return them to the client.

All subsequent queries require only the execution phase.

3.1.2 Java Servlets Approach. In the Java Servlet approach, the middleware program is a Java servlet [5], which is a Java program that runs as a child process within the context of a Web server program. The Web server is responsible for loading, maintaining, and terminating servlets. In our case, servlets were loaded during the Web Server start-up time.

Client's queries are routed by the Web server to a servlet, which submits them to the database server for processing. The results are returned to the client again through the Web server. All queries involve both an initialization and an execution phase. Thus, the cost of any query in this approach is

1. Initialization phase:
   A. The time for the client to open a URL connection with the Web server.
2. Execution phase:
   A. The time for the applet to invoke, through the Web server, the corresponding servlet passing the SQL statement as a parameter (stating explicitly the servlet name and type of operation).
   B. The time for the servlet to execute the request, obtain and return the entire result table to the client.

3.2 RPC approaches: Java RMI, CORBA, and Applet JDBC

The RPC approaches can be classified based on whether or not the client directly maintains the database connection. In the RMI and CORBA approaches, the connection is maintained by the middleware whereas in the Applet JDBC approach, by the web client.

3.2.1 The RMI approach. Java Remote Method Invocation (RMI) [4] is a Java application interface for implementing remote procedure calls between distributed Java objects. In RMI, the middleware consists of two objects: The first object is the *application server* which is responsible for handling requests by allowing clients to remotely invoke methods on it. The second object is the *installer object*, which is used to start up the application server, and register it under a unique service name with the Java virtual machine running on the Web server.

Figure 3 shows the steps for the first query. The client calls a bind method of the RMI to obtain a reference of the application server. The parameters of this bind method are the URL of the machine on which the application server object was registered, and the unique service name with which it was registered. Using this reference, the client calls a method on the remote application server passing the query as a parameter. The application server executes the query at the database server, and returns the result table to the client as the return value of the method called.

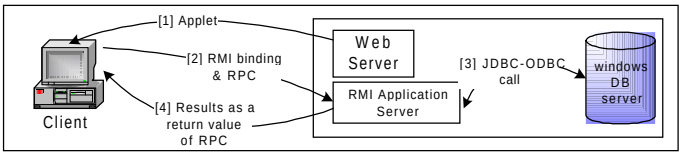


Figure 3. RMI approach

The cost of the initial query is

1. Initialization phase:
   A. The time for the applet to obtain a reference to the remote application server (bind to it).
2. Execution phase:
   A. The time for the client to invoke a method on the application server passing the SQL statement as a parameter.
   B. The time for the application server to execute the SQL statement, obtain and return the results.

The time required for a subsequent query is the execution phase.

3.2.2 The CORBA approach. CORBA, the Common Object Gateway Request Broker Architecture [13], is an emerging distributed object standard that defines client/server relationships between objects in a common interface language. Unlike

RMI, CORBA objects can be implemented in any programming language. In order for a CORBA client object to utilize a CORBA server object, an implementation of CORBA's basic functionality, called the Object Request Broker (ORB), has to be loaded at both the client and the server sites. In our testbed we use Visigenic's Visibroker for Java [20], which is also included in Netscape Navigator and hence, the client does not download the ORB classes from the Web server which would have been the alternative. For security purposes, CORBA allows an applet to bind to a remote CORBA server object only through a firewall called the IIOP (Internet Inter-ORB Protocol) Gatekeeper [21], installed at the Web server machine from which the applet is downloaded. That is, the IIOP Gatekeeper is responsible for routing the client's calls on the loaded ORB, and to the application server.

Except from the IIOP Gatekeeper, the middleware in the CORBA approach is similar to the one in the RMI approach. There is an application server object and an installer object. The installer object in this case is also used to load the ORB, and register the application server with a unique service name with the ORB.

The steps required for the first query are shown in Figure 4. After the client loads the ORB, it bounds to the application server via the Gatekeeper by calling a special bind method and passing as parameter *only* the unique service name of the application server. The client then calls the appropriate method on the application server to carry out the request. The application server will execute the client's request on the database, and return the result table as the return value of the method called.

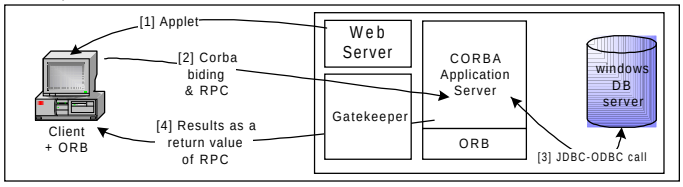


Figure 4: CORBA approach

The cost of the first query is

1. Initialization phase:
   A. The time for the client to initialize core ORB classes.
   B. The time for the client to bind to the application server.
2. Execution phase:
   A. The time for the client to invoke a method on the application server passing the SQL statement as a parameter.
   B. The time for the application server to execute the SQL, obtain and return the results to the client.

Execution phase is only required for any subsequent query.

3.2.3 The Applet JDBC approach: Applets that use directly the JDBC API. In this approach, the client applet downloads a type 3 JDBC driver and uses directly the JDBC API to connect to the database. The Gateway of the type 3 driver plays the role of the middleware. We used a type 3 driver because it is the only JDBC

driver that satisfies our two design principles discussed at the beginning of this section.

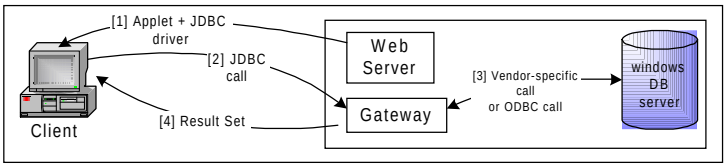


Figure 5: The Applet JDBC approach using a type 3 JDBC driver

After the client downloads the JDBC driver, it establishes database connectivity issuing JDBC calls on the Gateway, which are subsequently mapped on the database server. Figure 5 illustrates the four steps involved in the first query. Their cost is

1. Initialization phase:
   A. The time for JDBC driver to be downloaded from the Web server and initiated by the applet.
   B. The time for the applet to establish connection to the database though the gateway program.
2. Execution phase:
   A. The time for the applet to issue an SQL statement to the database and obtain the results.

All subsequent queries require only the execution phase.

3.3 RPC-like approach: Java Mobile Agents (JMA).

Finally, in this subsection, we describe the approach of using mobile agents to achieve Web database connectivity, and specifically, the best of the three variants proposed in [15]. Mobile agents [3, 7] are processes capable of pausing their execution on one machine, dispatching themselves on another machine and resuming their execution on the new machine. The idea in the JMA approach is to use one or more mobile agents to implement the middleware and carry out the requests of the client. In the best variant, the results as well as subsequent queries are sent to and from the client using a message. This message passing is implemented implicitly as an RPC invocation from the client applet on the dispatched mobile agent.

For our experiments, we used *Aglets* [10], for two reasons: (a) availability of code, and (b) support for hosting mobile agents within applets without significant overhead based on our prior experience with their use. Aglets can be fired from within a special applet, called the *FijiApplet* that provides an aglet execution environment similar to the general stand-alone aglet runtime environment called the *Tahiti Server*.

In the JMA approach, the middleware consists of three components: The *DBMS-aglet*, the (*Stationary) Assistant-aglet* and the *Aglet Router*. The DBMS-aglet can connect to a database and submit queries. Each database server is associated with

an Assistant-aglet identified by a unique aglet ID and the URL of its Tahiti server. An Assistant-aglet provides the information necessary for a DBMS-agent to load the appropriate JDBC driver and connect to the database server. An *Aglet Router* is required to route aglets and messages, dispatched from a FijiApplet to any destination, and vice versa, because of the Java security restrictions. An aglet created within a FijiApplet is neither allowed to dispatch, nor to send a message directly to any URL other than the Web server URL.

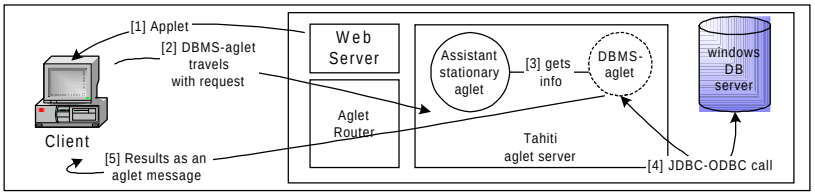


Figure 6: Mobile agents approach configuration (message variation)

When the user enters his first query (Figure 6), the client applet (an extension of the FijiApplet) creates a DBMS-aglet with a specific URL-based itinerary (travel plan) and the specified SQL statement. The DBMS-aglet travels through the aglet router to the database server machine. Upon its arrival, the DBMS-aglet communicates with the Assistant-aglet to retrieve information on the database and the available JDBC driver. It then loads the JDBC-ODBC driver, connects to the database server and executes the client's request. After sending the query result in a message to the client, the DBMS-aglet remains connected to the database server, waiting for a message with new requests from the client. The cost of the initial query is

1. Initialization phase:
   A. The time for the client to create the DBMS-aglet
   B. The time for the client to initialize the DBMS-aglet (SQL statement, itinerary, etc.)
   C. The time for the DBMS-aglet to travel to the remote database server
   D. The time for the DBMS-aglet to negotiate with the assistant aglet
   E. The time for the DBMS-aglet to establish connection with the database
2. Execution phase:
   A. The time for the DBMS-aglet to query the database and send the results to the client using a message.

All subsequent requests required only one message from the client to DBMS-aglet, which includes the new SQL statement, plus the execution phase.

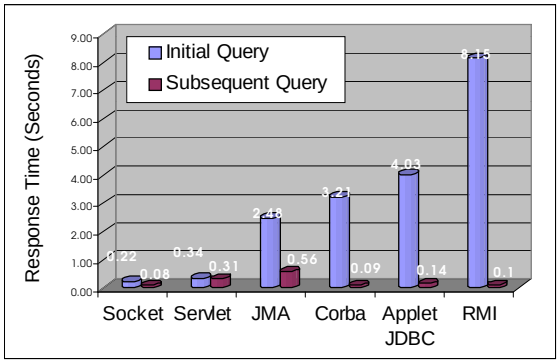
4 Performance Evaluation

We contacted two sets of experiments to evaluate the ability of each approach to support (1) *small interactions* that typically involve a small size of query results (128 bytes), and (2) *heavy cooperation* that involve a wide range of query results.

Given our interest to support both mobile clients and clients over a wide-area network with relatively slow communication links (limited bandwidth), we contacted our experiments on a wireless 1.2Mbps LAN of Pentium PCs. We used Netscape Navigator v4.6 as the Web client's Java-enabled browser. For each approach, a sufficient number of runs were performed to obtain statistically significant results.

4.1 Small Interactions

We measured the response time (a) of the first query and (b) of subsequent queries (Graph 1). Short-duration interactions consist of a single query as opposed to long-duration ones. The first query differs from the subsequent ones because it incurs the overhead of establishing the connection between the client and the remote database.



Graph 1. Performance of all approaches for 128 bytes result size

For the first query (short-duration interactions), the non-RPC approaches have by far the lowest response time. This can be explained by the fact that their initialization phase does not engage any special package loading by the client. Compared to the Socket approach, the Servlet approach performs slightly worse because (a) the communication between the client and the servlet is marshaled by the Web server, and (b) by executing as a Web server thread, the servlet receives less CPU time than the socket application server. Thus, servlets respond slower to requests and take more time to assemble the query results.

From the other approaches, the JMA approach offers the best performance for a single query. Significant part of its cost (around 2 sec) is due to the process of dispatching the DBMS-aglet from the client applet to the aglet router on the Web server and from there to the database server. In the case of the CORBA approach, the first query is slightly more expensive than the one in the JMA approach because of the overhead of initializing the necessary ORB classes and the binding to the application server. This overhead is quite significant (around 3.20 sec). Following the CORBA approach is the Java JDBC approach in which the response time of the

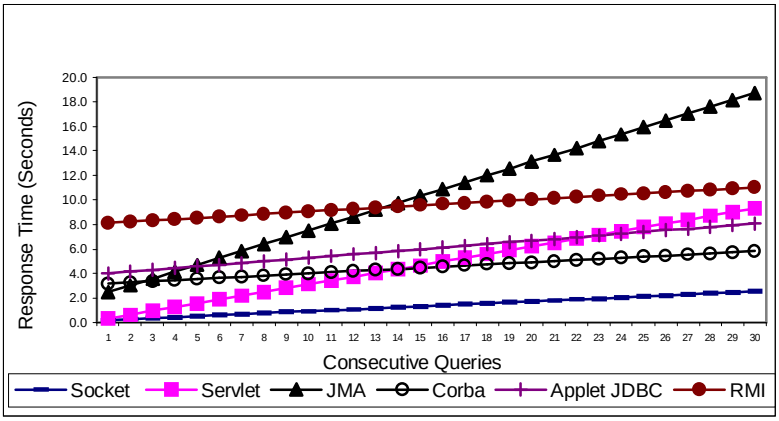
first query is increased by a considerable amount of time by the downloading of the JDBC driver.

To our surprise, the RMI approach performs by far the worst for the first query. We expected the RMI approach to exhibit better performance because, as opposed to the other RPC approaches, it does not involve the loading of any specific package. The only way to explain this is to attribute the increased response time to the interpreted method of RMI calls when binding the client applet to the application server. CORBA compilers create hard-coded encoding/decoding routines for marshaling of objects used as RPC parameters, whereas RMI uses object serialization in an introspective manner. This means that (a) RMI encodes additional class information for each object passed as a RPC parameter, and (b) marshaling is done in an interpreted fashion. Consequently, RMI remote calls are more demanding in terms of CPU time and size of code transmitted, a fact that we observed in all our experiments.

For subsequent queries (long-duration interactions), the performance of the CORBA and RMI approaches dramatically improves, and becomes close to the best performance exhibited by the Socket approach. The reason is that the client applet is already bound to the application server and only a remote procedure call is required to query the database. For a similar reason, the JDBC applet approach also exhibits a significant performance improvement for subsequent queries.

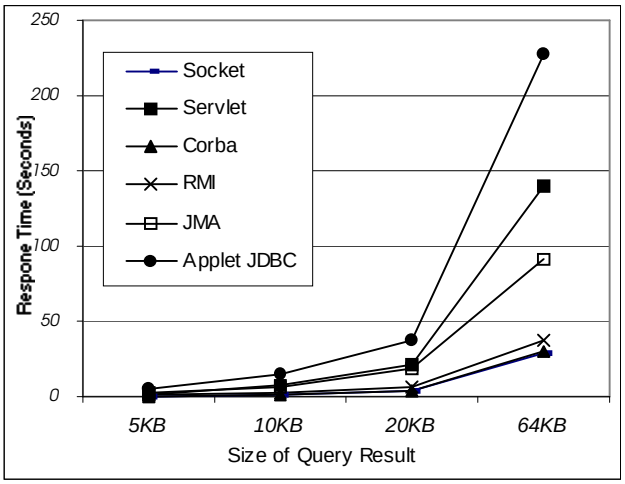
Having the DBMS-aglet already connected to the remote database and ready to process a new query, the JMA approach also improves its response time for subsequent queries. However, this response time is the worst of all other approaches. We attribute this to two reasons: (1) the two required messages to implement subsequent queries have to be routed through the aglet router, and (2) a mobile agent is not a stand-alone process and it does not receive full CPU time.

Finally, the Servlet approach improves slightly its performance although the steps for executing any query are the same. This improvement is due to the fact that any subsequent connection between the client and the Web server require less time because the URL of the Web server has already been resolved in the initial query.



Graph 2: Average performance for up to 30 consecutive queries (128 bytes of result size)

In order to better illustrate the *overall performance* of each approach, we plotted in Graph 2 the average time required by each approach for a number of consecutive queries. It is clear that the socket approach is the most efficient for both short and long interactions. This is not a surprise since all other approaches are built on top of sockets. Both the Servlet and JMA approaches scale very badly. The CORBA, JDBC applet, and RMI approaches appear to scale well, however, the RMI approach appears less attractive due to its worst performance for initial queries.



Graph 3. Subsequent Query

4.2 Heavy Cooperation

In order to evaluate heavy cooperation we adjusted the size of the query result from 5 kilobytes (95 tuples) to 64 kilobytes (1000 tuples) by changing the complexity of the SQL statement issued through the client applet.Query result size directly affects the response time in two ways: (1) in the amount of time spent for the query to execute, and (2) in the transport time for the results to reach the client. For these experiments, we also measured response times of first and subsequent queries. In both cases, each approach exhibited the similar sensitivity which is shown in Graphs 3.

The first observation is that the average response times of Java JDBC applet and JMA approaches increase exponentially with query result sizes larger than 20KB. The JDBC applet approach performs by far the worst for increased result size. This can be explained by the fact that in JDBC rows from a query result are retrieved one at a time. Specifically, to retrieve one row from the query result, the client must call

a method on a Java ResultSet object, which is mapped on the remote database server through the Gateway. Consequently, for a large size of query result, a large number of those remote calls have to take place. In that case, large query results not only increase dramatically the response time but they also increase the Internet traffic.

The bad scaling of the JMA approach can be explained in the same way as the bad performance of the Servlet approach. Both mobile agents and servlets do not execute as stand-alone processes, and therefore, they do not receive full CPU time and heavily depend on the supporting execution environment. The other RPC approaches exhibit acceptable performances (close to linear for sizes above 20KB) with the CORBA approach being slightly better. As indicated above, the implementation of RPC calls in CORBA is much faster compared to RMI's one.

5. Programmability Comparison

In this section, we compare the different approaches in terms of development effort. Our goal is to understand if there is any correlation or trade-off between performance and programming complexity. To quantify the development effort, we use the number of required system calls. The number of systems calls used in each approach is, in some sense, analogous to the number of code lines implementing each approach.

Table 1 shows the total number of system calls required for each approach. Table 1 also distinguishes between the number of system calls required to establish communication between the Web client and the middleware, and the number of calls required to submit a query and get back the results.

A first observation is that the development effort of the client is related to the level of abstraction of communication between the client and the middleware, in general, and the naming scheme used to identify the database services to establish communication, in particular. Not surprisingly, the RPC approaches involve less complex APIs, more transparent client/server communication and hence exhibit high programmability. All non-RPC approaches, including the JMA approach (the RPC-like one), require more development effort and hence have low programmability.

A second observation is that despite the fact that the JMA approach supports RPC-like communication, it exhibits the *lowest* programmability as indicated by the largest number of system calls required. Most of these system calls are used to construct, maintain and execute the URL-based itinerary.

		Socket	**Servlet**	**CORBA**	**RMI**	**Applet JDBC**	**JMA**
System Calls		29	25	15	12	6	29
Total Number							
Establish Connection	At the Client	7	11	2	1	3	11
	At the Middleware	11	3	8	6	0	11

Submit Query and Get Results	At the Client	3	3	1	1	3	2
	At the Middleware	8	8	4	4	0	5
Client Execution Code		6K	6K	23K	9K	50K	27K
Programmability		Low	Low	High	High	High	Low

Table 1: Programmability of the approaches

Finally, the level of programmability does not correspond to the size of the client executable code. Interestingly, the Non-RPC approaches, namely, Java Socket and Servlet, support the smallest client size (6K). On the other hand, the Applet JDBC has the largest client size of 50K: the Java applet is 6K and the JDBC driver is 46K. The JMA approach is the second most resource demanding approach after Applet JDBC with 27K: Java applet 10K, FijiApplet 10K and DBMS-Aglet 7K.

6 Conclusions and Future Work

In this experimental paper, we have implemented, evaluated, and compared all currently available Java-based approaches that support persistent Web database connectivity. Our comparison proceeded along the lines of the performance of query processing and of the programmability of each approach.

The results of our comparison showed that the CORBA approach offers high programmability and hence, is easy to develop, while its performance is comparable to the best performing approach that employs sockets. Therefore, the CORBA approach offers the best promise for the development of large Web applications, in particular, in those with long cooperative interactions involving multiple queries of varying result sizes. For short interactions, typically involving a single query, and environments with resource-starved clients, the socket and servlet approaches should be considered. These approaches yield Web client with the smallest footprint, just 6 Kbytes. Clearly, the best performance is not always achievable with high programmability and low resource requirements.

The recent advancements of the Web technology and mobile computing led to a renew interest on mobile agents technology. Given this renewed interest, our study provided an insight to potential scalability problems with the currently available mobile agent implementations. The JMA approach cannot support interactions that require movement or exchange of large amounts of data such as large number of consecutive queries with increased size of query result. Hence, it is necessary to develop more efficient mobile agent infrastructures, if the full potential of mobile agents is to be explored. As part of our future work, we investigate the possibility of merging mobile agents and the CORBA technology in order to facilitate a scalable and efficient JMA-based Web database connectivity.

References

1. E. Anuff. *Java Sourcebook*. Whiley Publishing, 1996.
2. T. Berners-Lee and D. Connolly. Hypertext Markup Language Specification 2.0, Internet Draft, *Internet Engineering Task Force (IETF), HTML Working Group*. Available at <www.ics.uci.edu/ietf/html/html2spec.ps.gz>, June 1995.
3. D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, Vol. 2, No. 5, October 1993.
4. T. B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, 1998.
5. J. Goodwill. *Developing Java Servlets*. Sams Publishing, 1999.
6. S. P. Hadjiefthymiades and D. I. Martakos. A Generic Framework for the Development of Structured Databases on the WWW. *Fifth Int'l WWW Conference*, May 1996.
7. C. G. Harrison, D. M. Chessm, A. Kershenbaum. Mobile Agents: Are they a good idea? Research Report, *IBM Research Division*, 1994.
8. G. Helmayer, G. Kappel, and S. Reich. Connecting Databases on the Web: A Taxonomy of Gateways. *Eighth Int'l DEXA Conference*, Sept. 1997.
9. H. Maurer. *Hyperwave: The Next Generation Web Solution*, Addison-Wesley, 1996.
10. IBM Japan Research Group. *Aglets Workbench*. Web site: <http:/www.trl.ibm.co.jp/aglets>.
11. B. Jepson. *Java Database Programming*. Wiley Computer Publishing, 1997.
12. A. Lambrinidis and N. Rousopoulos. Generating dynamic content at database-backed web server: cgi-bin vs mod_perl. *Sigmod Record*, March 2000.
13. Object Management Group. The Common Object Request Broker: Architecture and specification. February 1998.
14. R. Orfali, D. Harkley. *Client Server Programming with Java and CORBA*. Second Edition. Whiley Publishing, 1998.
15. S. Papastavrou, G. Samaras, and E. Pitoura. Mobile Agents for WWW Distributed Database Access. *Fourteenth IEEE Int'l Conference on Data Engineering*, Feb. 1999.
16. Sun Microsystems Inc., Java Development Kit, <http://java.sun.com/jdk>.
17. Sun Microsystems Inc. Java Sockets Documentation, <http://java.sun.com/docs>.
18. Sun Microsystems Inc., JDBC drivers, <http://java.sun.com/products/jdbc/drivers.html>.
19. Visibroker for Java: Programmer's Guide, V.3.0. Borland, <http://www.inprise.com/visibroker>.