

**NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA**



19980619 171

THESIS

**ANALYSIS OF
JAVA DISTRIBUTED ARCHITECTURES
IN
DESIGNING AND IMPLEMENTING A
CLIENT/SERVER DATABASE SYSTEM**

by

Ramis Akin
Frederick P. O'Brien

June 1998

Thesis Advisor:

C. Thomas Wu

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE June 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis
----------------------------------	-----------------------------	---

4. TITLE AND SUBTITLE: ANALYSIS OF JAVA DISTRIBUTED ARCHITECTURES IN DESIGNING AND IMPLEMENTING A CLIENT/SERVER DATABASE SYSTEM	5. FUNDING NUMBERS
---	--------------------

6. AUTHOR(S) Akin, Ramis, O'Brien, Frederick P.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
--	--

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
---	--

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT: Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE:
--	-------------------------

13. ABSTRACT (maximum 200 words)

Having timely and accurate information is essential for effective management practices and optimization of limited resources. Information is scattered throughout organizations and must be easily accessible. A new solution is needed for effective and efficient management of data in today's distributed client/server environment.

Java is destined to become a language for distributed computing. Java Development Kit (JDK) comes with a broad range of classes for network and database programming. Java Database Connectivity (JDBC) is one such class for providing client/server database access. There are many different approaches in using JDBC, ranging from low level socket programming, to a more abstract middleware approach. This thesis will analyze three different approaches: Sockets, Remote Method Invocation (RMI) and Commercial Middleware servers.

Among the three approaches this thesis examined, database access through RMI is the most viable approach because it uses an effective distributed object model. RMI abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, programmers can invoke a remote procedure as if it resided locally.

14. SUBJECT TERMS Database, JDBC, Java, RMI, Sockets	15. NUMBER OF PAGES 248
--	-------------------------

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL
---	--	---	----------------------------------

Approved for public release; distribution is unlimited.

**ANALYSIS OF
JAVA DISTRIBUTED ARCHITECTURES IN DESIGNING AND IMPLEMENTING
A CLIENT/SERVER DATABASE SYSTEM**

Ramis Akin

Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1992

Frederick P. O'Brien

Captain, United States Army
B.A., University of Massachusetts, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

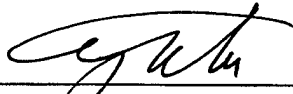
June 1998

Authors:




Ramis Akin / Frederick P. O'Brien

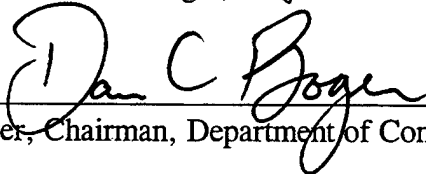
Approved by:



C. Thomas Wu, Thesis Advisor



Chris Eagle, Second Reader



Dan Boger, Chairman, Department of Computer Science

ABSTRACT

Having timely and accurate information is essential for effective management practices and optimization of limited resources. Information is scattered throughout organizations and must be easily accessible. A new solution is needed for effective and efficient management of data in today's distributed client/server environment.

Java is destined to become a language for distributed computing. Java Development Kit (JDK) comes with a broad range of classes for network and database programming. Java Database Connectivity (JDBC) is one such class for providing client/server database access. There are many different approaches in using JDBC, ranging from low level socket programming, to a more abstract middleware approach. This thesis will analyze three different approaches: Sockets, Remote Method Invocation (RMI) and Commercial Middleware servers.

Among the three approaches this thesis examined, database access through RMI is the most viable approach because it uses an effective distributed object model. RMI abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, programmers can invoke a remote procedure as if it resided locally.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
A. POTENTIAL SOLUTION	1
B. RESEARCH OBJECTIVES	3
C. SCOPE.....	3
D. THESIS ORGANIZATION.....	4
II. BACKGROUND	7
A. RELATIONAL DATABASE MODEL	7
B. STRUCTURED QUERY LANGUAGE	7
1. Basic SQL Statements.....	8
a. Create Table.....	8
b. Insert	8
c. Update.....	8
d. Delete	9
e. Select.....	9
2. Processing a SQL Statement.....	9
a. Parse the SQL Statement	9
b. Validate the SQL Statement.....	9
c. Generate an Access Plan.....	10
d. Execute the Access Plan	10
3. SQL Techniques.....	10
a. Embedded SQL.....	10
b. Stored Procedures	11
c. Call Level Interface.....	12
C. DATABASE APPLICATION PROGRAMMING INTERFACES	12
1. Proprietary Database Management System Interface	13
2. Open Database Connectivity.....	14
3. Java Database Connectivity	16
4. Middleware	17
5. Common Gateway Interface	19
D. CLIENT/SERVER MODEL	21
1. Two-Tier Client/Server Architecture.....	23
2. N- Tier Client Server System.....	24
III. JAVA DATABASE CONNECTIVITY (JDBC)	27
A. DRIVERMANAGER CLASS	29
B. CONNECTION INTERFACE.....	30
1. Statement.....	31
2. PreparedStatement	31
3. Callable statement.....	32
4. DatabaseMetaData	33
C. STATEMENT INTERFACE.....	34
1. Execute.....	34
2. Execute Query.....	35

3.	Execute Update	35
a.	Create Table	35
b.	Insert	36
c.	Update	36
d.	Delete	36
D.	RESULTSET INTERFACE	36
1.	Get Meta Data	37
2.	Get Type.....	37
3.	Next.....	37
E.	JDBC DRIVERS	38
1.	JDBC/ODBC Bridge (Type I)	39
2.	Native-API, Partly Java Driver (Type II)	40
3.	Network Protocol, All-Java Driver (Type III)	41
4.	Native-Protocol, All-Java Driver (Type IV).....	42
5.	Driver Selection	43
F.	CONCLUSION.....	43
IV.	SOCKETS AND JDBC.....	45
A.	INTRODUCTION.....	45
B.	SOCKETS	45
1.	Stream Socket Communication.....	47
2.	Datagram Socket Communication	49
3.	Multicast Socket Communication.....	51
C.	MULTICAST SOCKET JDBC MODEL	51
1.	Communication Protocol	53
2.	Model Implementation.....	56
D.	CONCLUSION	59
V.	REMOTE METHOD INVOCATION AND JDBC.....	61
A.	INTRODUCTION	61
B.	REMOTE METHOD INVOCATION (RMI)	62
1.	RMI System Architecture	63
a.	Stub/Skeleton Layer.....	64
b.	Remote Reference Layer.....	65
c.	Transport Layer.....	65
d.	Application Layer	66
2.	RMI Development Process	66
a.	Agree Upon the Interface	66
a.	Implement the Interface	67
b.	Object Server	68
c.	Java Client Application.....	68
d.	Run the System	69
C.	RMI JDBC MODEL	69
1.	Interface	70
2.	Interface Implementation.....	73
3.	Object Server	73
4.	Client Application.....	73

D. CONCLUSION.....	75
VI. MIDDLEWARE APPROACH.....	77
A. SYMANTEC VISUAL CAFÉ	78
1. Server Configuration.....	78
a. Connecting to Microsoft SQL 6.5 Server Database.....	79
b. Connecting to Sybase SQL Anywhere Server Database	79
2. Database Aware Components	80
3. Model Implementation.....	82
4. Visual Café Summary	83
B. BORLAND JBUILDER CLIENT/SERVER SUITE	84
1. Server Configuration.....	84
a. Connecting to Microsoft SQL 6.5 Server Database.....	85
2. Database Aware Components	85
3. Model Implementation.....	87
4. Summary	88
C. CONCLUSION.....	89
VII. CONCLUSION	91
A. SYNOPSIS.....	91
B. AREAS FOR FURTHER RESEARCH.....	92
1. Security	92
2. Application Server	93
3. Multicast Remote Objects.....	94
4. Object Oriented Database	94
5. Common Object Request Broker Architecture (CORBA).....	94
C. CONCLUSION.....	94
LIST OF REFERENCES.....	97
APPENDIX A. JDBC REFERENCE TABLE	101
APPENDIX B. MULTICAST SOCKET MODEL	103
APPENDIX C. RMI JDBC MODEL	161
APPENDIX D. DEPLOYMENT	225
INITIAL DISTRIBUTION LIST	231

LIST OF FIGURES

Figure 1: General Database Access	14
Figure 2: ODBC Implementation [Ref. 26]	16
Figure 3: Middleware Solution	18
Figure 4: Common Gateway Interface.....	20
Figure 5: Two-Tier Client/Server	23
Figure 6: N-Tier Client/Server Model	24
Figure 7: JDBC Object Relations [Ref. 13].....	28
Figure 8: JDBC/ODBC Bridge	40
Figure 9: Net Protocol All Java Driver	41
Figure 10: Native Protocol JDBC Driver.....	43
Figure 11: Socket Communications.....	47
Figure 12: Stream Socket Communication	49
Figure 13: Socket Evaluation Environment.....	52
Figure 14: Message Format	53
Figure 15: Message Format	55
Figure 16: Database Datagram Socket Server	57
Figure 17: Multicast Parts Request	59
Figure 18: RMI Architecture [Ref. 15]	64
Figure 19: Marshalling.....	65
Figure 20: RMI Design Process.....	67
Figure 21: RMI Implementation	70
Figure 22: Commercial Tools Evaluation Environment.....	78
Figure 23: Multicast Model	103
Figure 24 : RMI Object Model	161

LIST OF TABLES

Table 1: JDK1.1 SQL Package	27
Table 2: MiddleWare Database Access	77

ACKNOWLEDGEMENT/DEDICATIONS

One of the great pleasures of finishing up this thesis is acknowledging the support of many people whose names may not appear any where in the thesis, but whose cooperation, friendship, understanding and patience were crucial for us to prepare this thesis and successfully publish it.

We would like to thank our thesis advisor Dr. Thomas Wu for assisting us in exploring various Java technologies and making it a beneficial experience. We would also like to thank our second reader, Lieutenant Commander Chris Eagle for his guidance and support and helping us to see the new horizons in distributed computing. Our special thanks to Dr. Debra Hensgen, whose Distributed Computing class changed the entire focus of our thesis.

Throughout the thesis, we used the word "we" to indicate our two man team, Fred and Ramis. In fact there were behind-the-scene members of our team - our dearest wives Christine and Mine and also Fred's lovely children Patrick, Kelsey, Marisa and Courtney. Without their patience, support and encouragement we would not be able to create a thesis like this. Thank you all!

We would like to thank each other. Working together on this thesis was an outstanding learning experience for both of us. Being able to share periods of frustration and excitement with each other was unique. The friendship developed will last a lifetime.

I. INTRODUCTION

Organizations rely on information to make effective business decisions and corporate intranets are changing the way organizations conduct business. As networking technologies continue to improve, with increasing bandwidth and reliability, effective distributed computing is becoming a reality. Organizations are relying on internet technologies to be the conduit for employees to access and manipulate corporate information.

Having timely and accurate information is essential for effective management practices and optimization of limited resources. Information can be stored effectively and efficiently in Database Management Systems (DBMS), a software system that manages the data integrity, storage and access of data in a database.

The goal of a database is to reduce redundant storage of information throughout an organization. Data is stored in a central location and multiple clients are allowed to access the data from various locations throughout the organization's network or via the internet. In this client/server environment, client processes need to be able to effectively locate the database server, and communicate with the remote server process.

A database aware client/server system is a system that allows client process to access a server process, which in turn communicates with a database management system. The client can manipulate the data, but the location of the database and the type of DBMS used is transparent. There are a number of challenges in implementing a database aware client/server system. Information needs to be accessible from various client operating systems and possibly via the internet and a web browser. The networking protocol, how information is transported from a client machine to a server machine, must be agreed by both the client and server process. System designers must be able to communicate with the DBMS, via an agreed upon interface. Optimization of resources, a high return on investment, is essential in justifying implementing a database aware client/server solution to various corporate managers. Organizations want to keep system development and maintenance costs as low as possible. As system designers perform their problem analysis these issues need to be addressed.

A. POTENTIAL SOLUTION

The programming language selected must facilitate a designers ability to meet the above challenges. Java, developed by Sun Microsystems is an object-oriented network

centric programming language that is poised to enter the Enterprise Client/Server environment.

Java provides the platform independence that may be demanded. A Java application can run on any platform that has a Java Virtual Machine. This reduces the costs and time associated with generating multiple versions of an application to run on various platforms. Applications can easily be modified to create Java applets, which are hosted by a Java-enabled web browser. This added flexibility allows an organization to re-use programs and make them available via their intranet or the internet. System administrators are not required to configure client machines, since all required class files will be downloaded from the server. Any changes to the applet will result in the user getting the most recent version, so will ultimately reduce software distribution expenses.

The `java.net` package provides a powerful and flexible infrastructure for networking. The designer can use datagram sockets or stream sockets to send information between two processes. The package also allows the designer to create a multicast group, which allows a process to join a group. A message sent to the group will be received by all group members. This implementation requires the designer to implement a message passing protocol that will be used between processes. The message passing protocol provides a means for the recipient process to understand what to do with the message.

Java Remote Method Invocation is a distributed model that can be found in the `java.rmi` package. It encapsulates the low level socket requirements and message passing. A Java client application will establish a reference to a remote Java server object and issue method calls as if the remote object resided locally. This higher level of abstraction allows a Java client to talk with a Java remote object.

The `java.sql` package provides a means for Java applications to interface with Relational Database Management Systems. The functionality provided by this package is referred to as Java Database Connectivity (JDBC) and was introduced by Sun with Java Development Kit version 1.1. Relational database access accounts for the majority of client/server programs being employed by organizations. This package passes SQL statements to a Relational DBMS via a Java Database Driver.

By using Java technologies, organizations can maximize resource utilization. Proficient Java Developers can provide total solutions to include: developing a user interface, networking, and database access requirements. Conventional database interface developers are required to understand vendor specific database interface tools, and if web based access is required, then they must understand Common Gateway Interface (CGI).

Version control is simplified with Java due to its platform independence. A Java database aware application is designed to run on a Java Virtual Machine, which has been implemented on many hardware platforms and integrated into numerous web browsers. This relieves the designers from being overly concerned about the target platform, allowing them to focus on application functionality and efficiency. System administrators will be satisfied because application deployment requires minimal client configuration, and can be downloaded via an application installation applet. The client downloads the application or applet, and depending upon how the database system was designed, may not be required to install any database specific drivers.

B. RESEARCH OBJECTIVES

Java technologies are promising but are still immature. Enterprises are just beginning to exploit the potential of using Java in distributed applications. This thesis will study three approaches to using Java technologies to interface with a relational DBMS. We will first examine how JDBC can be utilized to access various relational DBMS. The thesis will then focus on how JDBC can be used with various networking technologies to establish a distributed database aware system. Part of the analysis of each approach will be to develop a working prototype which can be used as a basis for an actual implementation and for further evaluation of the potential for Enterprise Java technologies.

The primary objective of this thesis is to explore how Java technologies are used to access relational databases. Other concerns are to assess the functionality and the ease of using the JDBC interface. In order to develop a client/server system we are going to assess how a designer can implement a Java solution in a distributed environment using Java sockets, Remote Method Invocation and through a middleware solution. We plan on using and assessing two middleware JDBC drivers: Borland's DataGateway and Symantec's dbAnywhere, to determine how effective they are in connecting Java client applications to relational DBMS's.

C. SCOPE

The scope of the thesis will be to determine the effectiveness of Java technology to meet interface designers, system administrators, and users requirements in accessing a relational data base management system. The first phase of this thesis is to develop an understanding of the JDBC classes and methods provided in the java.sql package. The

next phase is to create an evaluation client/server environment for testing various implementations of connecting to a DBMS. This phase will include installing and configuring both local and remote database management systems such as Microsoft Access, Microsoft SQL 6.5 and Sybase SQLAnywhere. The third phase will focus on network connectivity issues such as using sockets, or remote method invocation to access a remote database server. The fourth phase will be to use two existing middleware solutions: Borland's DataGateway and Symantec's dbAnywhere and to evaluate their effectiveness in providing a Java database aware client/server solution. Finally, a summary of the lessons learned and recommended future work are provided.

D. THESIS ORGANIZATION

This thesis is organized into the following chapters:

- Chapter I: Introduction. This chapter gives an introduction to the problem, motivation, purpose, and general outline of the work.
- Chapter II: Background. This chapter is intended to provide an overview of the concepts used throughout the thesis. An explanation of relational database model, Structured Query Language, database application programming interfaces, and client/server model will be provided.
- Chapter III: Java Database Connectivity. This chapter describes the JDBC API and its key classes and methods and how they interact with a client application or database management system to provide database functionality. The four classes of JDBC drivers are described along with the advantages and disadvantages of using each to link a Java application to a relational database. Once a solid understanding of how JDBC can be used to interface with relational databases, the next phase is to assess various ways of connecting to the database server.
- Chapter IV: Socket and JDBC. This chapter will provide an overview of using Java sockets to provide a low level communication link between a client and a JDBC aware server. A model is implemented using multicast sockets replicating a distributed Corporate parts databases, where each department database server is part of a multicast group.

- Chapter V: Remote Method Invocation and JDBC. This chapter will employ RMI which encapsulates the low level details of socket programming for network communication. All JDBC drivers will reside on the server, which will interface with three different databases: a MS Access accounting database, a MS SQL 6.5 Navy database, and a Sybase SQL Anywhere demo database.
- Chapter VI: Middleware and JDBC. This chapter will look at the benefits and limitations of using two middleware solutions: Borland's DataGateway and Symantec's dbAnywhere. These two competing JDBC drivers allow a Java client to talk through the middleware driver to various SQL databases.
- Chapter VII: Conclusion. An overall assessment will be made on using Java technologies to develop a connection to a relational database. The strengths of sockets, RMI, and the use of a middleware solution will be analyzed. Recommendations for future research will also be made.

II. BACKGROUND

In this chapter, we provide the background information necessary for readers to understand this thesis. We will describe: relational database model, structured query language, application programming interfaces to databases, and distributed computing in a client server environment.

A. RELATIONAL DATABASE MODEL

A database is defined as a collection of related data. A relational model is a database model in which all data is held in tables, which are made up of rows and columns. Each table has one or more columns and each column is assigned a specific data type such as integer, characters, or date. Each column has a label which identifies the column, for example "employee id".

Each table in a database has a unique column value, known as a primary key. Primary keys are used to uniquely identify a particular row of data. For example, a persons social security number may be used as the primary key for an employee table. A row of data in a table is known as a tuple or a record. So each table will contain numerous records.

Foreign keys are used to define relationships between tables. A foreign key is a reference to a particular row in a different table that carries the corresponding primary key. For example, a department table may have a manager column, which contains the social security number of the manager. This column represent a foreign key to the primary key, social security number, of the employee table. A table can be related to one or more tables based upon the relationships between primary keys and foreign keys.

B. STRUCTURED QUERY LANGUAGE

Structured Query Language is a declarative language that is used to manipulate relational databases. SQL is a standard for relational database operations, not a communication protocol. It has no knowledge about how the database engine retrieves or processes the SQL statement. This results in a natural separation between the database management system that manages the data, and the client which determines what data to manipulate.

The ANSI standard for SQL (ANSI X3.135.1) defines level 1 and level 2 compliance. Level 1 consists of a Data Definition Language (DDL) and Data Manipulation Language (DML). The DDL includes functions such as CREATE,

ALTER, PRIMARY KEY and FOREIGN KEY. DML extends that functionality to include SELECT, INSERT, DELETE, which operate on rows and columns within a table. Level 1 also defines a cursor capability that can be used by a program for processing rows one at a time. Level 2 compliance consists of Level 1 and the Data Command Language (DCL) commands. DCL includes COMMIT, ROLLBACK, and GRANT, which control security, integrity, concurrency, and recovery of the database [Ref. 22].

1. Basic SQL Statements

This section provides a sample of basic SQL commands that are used to manipulate a database.

a. Create Table

A two dimensional table is used to abstractly view data. Each table has a name and attributes of various data types, such as character, date or integer. The attributes represent the column labels associated with the table. The following SQL statement creates a table whose name is myTable, which contains two attributes: name, of type character with a maximum length of 40 and dept, of type integer.

```
CREATE TABLE myTable (name CHAR(40), dept INT)
```

b. Insert

The SQL insert operation inserts data into an existing table. The user must specify the table name, the column names, and the corresponding values to be inserted into the table.

```
INSERT INTO myTable (name, dept) VALUES ('Fred', 23)
```

c. Update

Update is used to modify existing information in a table. The following example updates the dept field for any tuple with a name of Fred in the table, myTable. If multiple tuples with the name of Fred exist, they all will be modified.

```
UPDATE myTable SET dept = 28 WHERE name = 'Fred'
```

d. Delete

This command removes an entire row (tuple) from the specified table. It can also be used to remove an entire table. The following sample would delete any tuples that have a dept identifier of 23 from myTable.

```
DELETE FROM myTable WHERE dept = 23
```

e. Select

Select statements are used to extract data from various tables based upon search criteria, returning the result set.

```
SELECT dept FROM myTable WHERE name = 'Fred'
```

2. Processing a SQL Statement

SQL statements can result in computationally expensive function calls. For example, a complex join operation between two or more large tables. It is important that system designers have a general understanding of how a database management system processes an SQL statement. To process an SQL statement, a DBMS performs four basic steps: Parse the SQL statement, Validate the statement, Generate an Access Plan, and Execute the Plan.

a. Parse the SQL Statement

The DBMS first parses the SQL statement. It breaks the statement up into individual words, called tokens, makes sure that the statement has a valid verb and valid clauses, and so on. Syntax errors and misspellings can be detected in this step. Parsing a SQL statement does not require access to the database and typically can be done very quickly. It ensures the SQL statement is in the database specific format. This phase ensures that the statement is syntactically correct. It is not concerned with the semantics of the statement, the parameters such as table names or types. It is only ensuring the statement is in the correct SQL format.

b. Validate the SQL Statement

The DBMS validates the statement. It checks the statement against the system catalog. The system catalog contains database metadata, including table names,

attributes and types. This phase ensures the statement parameters are semantically correct. Do all the tables named in the SQL statement exist in the database? Do all of the columns exist and are the column names unambiguous? Does the user have the required privileges to execute the statement? Certain semantic errors can be detected in this step.

c. Generate an Access Plan

The DBMS is responsible for managing the data stored in the database. In this phase, based upon the statement, the DBMS generates an access plan. The access plan is a binary representation of the steps that are required to execute the statement. The DBMS optimizes the access plan. It explores various ways to carry out the access plan. Can an index be used to speed a search? Should the DBMS first apply a search condition to Table A and then join it to Table B, or should it begin with the join and use the search condition afterward? Can a sequential search through a table be avoided or reduced to a subset of the table? After exploring the alternatives, the DBMS chooses one of them.

Optimization is a very CPU-intensive process and requires access to the system catalog. For a complex, multi-table query, the optimizer may explore thousands of different ways of carrying out the same query. However, the cost of executing the query inefficiently is usually so high that the time spent in optimization is more than regained in increased query execution speed. This is even more significant if the same optimized access plan can be reused to perform repetitive queries. [Ref. 19]

d. Execute the Access Plan

The DBMS will execute the access plan, producing a result set that can be passed to the user.

3. SQL Techniques

There are various techniques that can employ SQL statements to process a users request. This section will briefly present embedded SQL, stored procedures, and call level interface. The implementation details of the following are usually specific to each DBMS.

a. Embedded SQL

Embedded SQL allows the use of SQL statements within a host language, such as C or C++. The SQL Statement can be static or dynamic. Static SQL is

effective if the data access can be determined at program design time and is used when speed is important.

Each SQL statement starts with an introducer and ends with a terminator, which serves as a flag. The code is processed by a SQL pre-compiler (provided by the DBMS vendor), which separates the source code and the SQL request. The pre-compiler substitutes calls to proprietary DBMS routines that provide the run-time link between the program and the DBMS. The revised source code is then compiled and ultimately linked with the proprietary DBMS library producing the executable.

The SQL requests that were extracted from the program form a database request module, which is processed by a binding utility. This utility examines the SQL statements, parses, validates, and optimizes them, and produces an access plan for each statement. Because the SQL Statement is hard coded, this processing only needs to occur at compiler time, not at run time, resulting in faster run time query execution.

Dynamic SQL is effective when the data access cannot be determined in advance, such as allowing a user to enter a SQL Statement in which the results will be displayed in a grid object. The application uses a flag, such as a question mark as a place holder for parameters that will be supplied later. The SQL statement with the embedded flags is then sent to the DBMS, via a *PREPARE* (string name) method. This allows the DBMS to parse the string, and prepare an access plan.

When the user enters the input parameters in the client program the application will call *EXECUTE* (string name), passing the DBMS the valid parameters. The DBMS can then execute the query and provide the result set back to the user. This technique is not as fast as Static Embedded SQL, because of the need to bind the input parameters.

b. Stored Procedures

Another SQL API is to use stored procedures. A stored procedure is pre-compiled SQL code that resides on the database server. Stored procedures take input parameters and return a result. A number of procedures can be packaged to form an SQL Module which can be stored in the DBMS or linked to the application. A module provides logical separation of SQL statements and the programming language/statements.

Stored procedures are a form of query optimization. They are used for efficiency, for SQL statements that are frequently executed and are computationally expensive. The database administrator (DBA) creates and stores the procedures in the DBMS. Once they are stored, those procedures can be invoked by a client. So instead of

submitting a SQL statement, the client will simply invoke the stored procedure the DBA has already defined. The database management system can develop, optimize and store an access plan for executing the stored procedure, therefore decreasing response time when the client invokes the procedure because the access plan will not have to be regenerated. Stored procedures can also reduce network congestion, by returning only the result set of an operation rather than entire tables that the client may further process.

The server must be capable of servicing the procedures without limiting transaction throughput and ultimately performance. It may be beneficial if a procedure is computationally complex and network congestion is light, to have the client, which has its own processor, process the request. The server is responsible for serving many requests and cannot over commit to one client at the expense of not serving others.

Triggers are another form of stored procedures. Triggers are special, user defined actions in the form of a stored procedure, that are automatically invoked by the server based upon data related events. An example of a trigger, might be the automatic generation of a parts order if the inventory level of widgets falls below a certain level.

c. Call Level Interface

The last SQL technique is to define a Call Level Interface, providing the application with a library of DBMS functions that can be called by the application program. The database aware application calls CLI functions on the local system, and the calls are sent across the network and processed by the DBMS. The initial call may be to establish a connection with the remote database. The application builds its SQL Statements, places the statement in a buffer then makes a call to send the statement to the DBMS for processing. Then the application makes a CLI call to disconnect from the DBMS.

The CLI can be implemented through dynamically linked libraries (DLL). Each database vendor is responsible for creating a DLL, also known as a driver to reside on the local machine. This allows an application to access information from multiple databases simultaneously. By employing a CLI, the DBMS implementation details can remain hidden from the application programmer.

C. DATABASE APPLICATION PROGRAMMING INTERFACES

There are a number of ways to access a database, unfortunately there is no standard due to the wide range of vendor specific API's and various network messaging protocols employed. The database engines/drivers and various API's that are employed

are usually vendor specific, as each attempts to gain a competitive market advantage by extending its SQL API or modifying its version of remote procedure call (RPC).

1. Proprietary Database Management System Interface

Each database vendor provides its own version of a database interface in the form of a vendor specific API. The DBMS usually has a client utilities application which encapsulates this API and can communicate back to the database engine. Vendors usually support or supply some type of development tool which can be used to create a graphical user interface (GUI) frontend to interact with the database backend. Each database vendor provides its version of increased efficiency and functionality in an attempt to differentiate their relational database management system. In order to take advantage of the increased functionality offered, it is usually easier to use their version of the database server and GUI tools.

The benefit of using a proprietary solution is faster transaction processing. This is due to the fact that the client uses the database specific language, so no additional mapping or translations are required. The disadvantages are that this solution usually costs more than other solutions, and greatly reduces flexibility, tying the organization to a single vendor solution. The organization is limited to only those features available from the vendor and may force the user to compromise on functionality.

Each relational database vendor implements the actual management and manipulation of the database and is referred to as a database management system (DBMS). To allow a client application to communicate with and access the database, a DBMS specific driver or interface must be published. The driver is usually in the form of a dynamically linked library (DLL) and resides on the client machine.

A database aware client graphical user interface (GUI) is created either by using DBMS specific tools or a standard programming language. Rapid Application Development (RAD) tools such as Delphi, Visual Basic, Visual C++, or Visual Café may be used to design the GUI. All client applications must ultimately use the DBMS specific driver to interact or communicate with the database engine, which resides on the server.

The following figure represents how a client would access a remote database. The database aware client application will load a database specific driver. The client will use Structured Query Language (SQL) interface to communicate with the local DBMS driver. Different relational databases provide different types of SQL APIs, or extend existing standards to provide market differentiation, so the SQL statement entered by the client must be understood by the DBMS specific driver or the driver will be responsible for

mapping the SQL statement into a format understandable by the server database engine. The driver passes the request down the stack to the DBMS specific transport layer which transmits the request to the remote database engine across the network. The database transport layer is responsible for establishing and managing the network connection with the remote database engine. The DBMS engine then processes the SQL statement and returns a result.

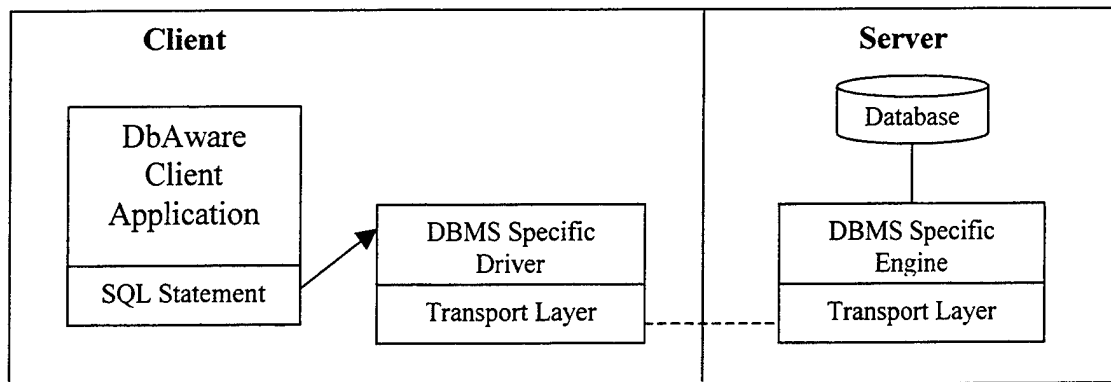


Figure 1: General Database Access

Vendors usually provide multiple protocol stacks with the client driver. The stacks are responsible for formatting the SQL request into the vendor specific format, and transmitting request across the network. The transport layer may implement a vendor specific transport protocol or support a common transport interface (TCP/IP or Named Pipes). By encapsulating a communication layer, the DBMS driver that resides on the client provides the mechanism to communicate with the server over a network. The communication layer will implement some form of Remote Procedure Call (RPC) to send the SQL statement to the database engine and return the results.

2. Open Database Connectivity

"Open Database Connectivity (ODBC) is a specification for a database API. This API is independent of any one DBMS or operating system, the ODBC API is language-independent but is usually implemented in C or C++. The ODBC API is based on the CLI specifications from X/Open and ISO/IEC. ODBC 3.0 fully implements both of these specifications and adds features commonly needed by developers of screen-based database applications, such as scrollable cursors[Ref. 19]."

ODBC is designed and controlled by the Microsoft Corporation. It is the current Windows standard SQL Application Program Interface (API). Every windows operating system installs the ODBC32 administrator in the Windows Control Panel, to register

ODBC compliant drivers as they are loaded on the system. A ODBC compliant DBMS specific driver that is installed on a system allows a client application to establish a database connection and process SQL statements.

To allow an ODBC enabled application access to a relational database, the relational DBMS provides a specific driver in the form of a dynamically linked library (DLL) which implements the ODBC API. For example, when a user installs Sybase SQL Anywhere on his/her system, a Sybase specific ODBC driver is also installed. Users will use the ODBC Administrator Utility to configure a data source. An ODBC datasource is a name association between a database and the vendor supplied driver. When an ODBC enabled application connects to a specific data source, the ODBC driver manager will map the data source name to one it maintains in its `odbc.ini` file. This mapping will yield a reference to a vendor supplied driver, which will be dynamically loaded. The driver will perform the translation of SQL statements into a vendor specific format, and pass the statements down to its implementation of the transport layer, to transmit the request across the network to the database engine to be processed.

The ODBC Driver Manager provides the interface to the DBMS specific driver, which implements the ODBC API. The drivers supplied can extend the ODBC API to offer additional functionality.

The vendor-supplied driver is responsible for connecting and disconnecting from the data source, submitting SQL statements to the data source and correctly formatting the request in a DBMS specific protocol. To access multiple DBMSs simultaneously, the ODBC Drive Manager may load multiple drivers, as required by the application.

Figure 2 captures the process. Assume a client application is attempting to establish a connection to a Sybase database called `heteroSADEMO` via ODBC. The first thing that will happen is the ODBC Drive manager (`odbc32.dll`) will get loaded. The Driver Manager then looks to the system `odbcinst.ini` file to see if a Sybase driver is available. This file holds information about all of the ODBC drivers installed on the system. The file will contain the following entry:

```
[Sybase SQL Anywhere 5.0]
Driver=c:\sqlany50\win\wod50w.dll
```

The driver is available and it contains a reference to the location of the vendor specific driver (DLL) which the ODBC Driver Manager will load. It then looks to the `odbc.ini` file to find an ODBC alias that matches the database name the client is attempting to establish a connection with. The file contains all ODBC data source names

that have been configured on the system, via the ODBC32 administrator utility function. In our example the file contains: *heteroSADEMO=Sybase SQL Anywhere 5.0 (32 bit)* where heteroSADEMO is the data source name to the Sybase SQL database. The name of the specific database file associated with the aliase is also maintained.

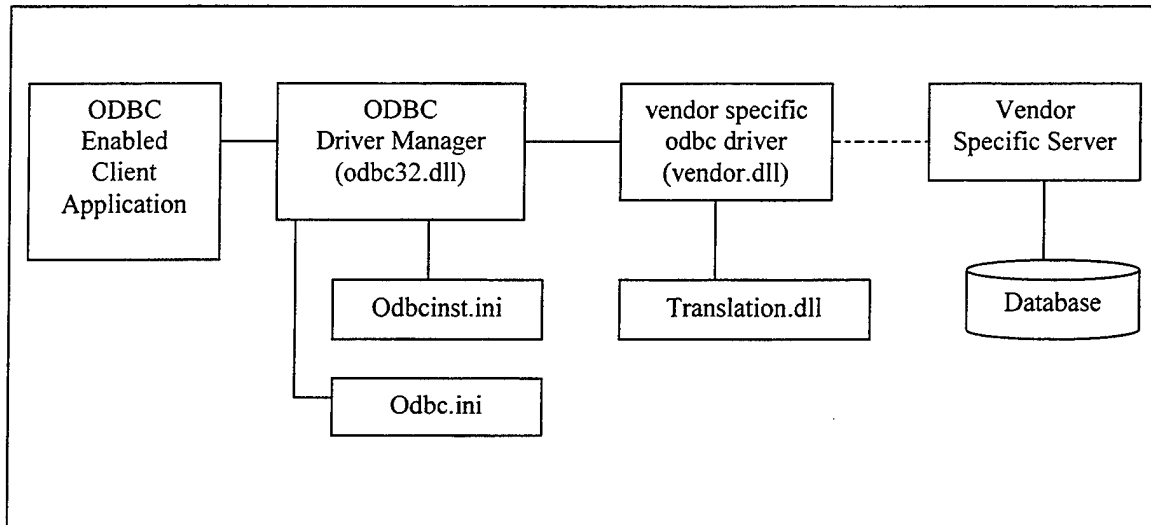


Figure 2: ODBC Implementation [Ref. 26]

Once the ODBC Drive Manager has confirmed the driver and database exists, it will pass all future references to the appropriate vendor specific driver to handle the SQL statement.

The vendor specific driver may use a translation DLL to convert the ASCII character set used by a Windows application, to the default character set used by the specific database. It is also responsible for communicating with the database engine, and passing results back to the ODBC driver, which passes the results to the ODBC client application.

By using ODBC to provide access to a vendor specific database, the application can only employ the SQL functions defined by the ODBC standard. As a result, the client application may not be able to take advantage of additional functionality that the database may be able to provide.

3. Java Database Connectivity

The JDBC API is based upon the X/Open SQL Call Level Interface. It is a Java database access API. It allows Java client applications the ability to communicate with relational databases and is a Java based alternative to Microsoft's ODBC API.

To provide Java developers immediate access to existing databases, the creators of the JDBC API created a Java to ODBC interface called the JDBC-ODBC Bridge. This provided immediate relational database access allowing developers to begin exploring the API while relational database vendors created proprietary JDBC drivers. This was necessary to gain support from the relational database community. As the technology matured, additional Java based DBMS specific drivers appeared. Sun categorizes the drivers into four types:

- *Type 1: JDBC-ODBC Bridge:* short term temporary driver.
- *Type 2. Native API, Partly Java:* driver is written in C, maps JDBC calls into vendor specific language, passed to the vendor driver and processed
- *Type 3. Network Protocol, All-Java Driver:* a middleware solution
- *Type 4. Native Protocol, All-Java Driver:* JDBC calls get translated directly into a DBMS specific network protocol, example Oracles OCI.

Chapter III of the thesis is devoted to further explanation of the JDBC API.

4. Middleware

A middleware server provides access to a number of databases via proprietary drivers or the use of other database API's such as ODBC. Transparency is one of the primary objectives of distributed computing and middleware database servers satisfy this objective. Through the use of a middleware server a client will have no knowledge about where the database information is coming from or even what DBMS is being used to store the data. The client will only understand the middleware API and will only be able to interact with the middleware object.

Middleware solutions provides flexibility allowing the client application to use one protocol or interface to communicate with the middleware server yet have access to many different relational databases. This simplifies the implementation and installation of client software. The client only needs to know how to communicate with the middleware object , which is handled by the middleware driver.

The server is responsible for accessing various relational databases. This means that the middleware server must contain the database specific drivers, which can communicate to specified databases. The overhead associated with providing this broad support (access to multiple databases), is possible decreased performance and loss of some DBMS functionality enhancements that a vendor proprietary solution may provide.

The characteristics of connecting middleware between a client and a server vary depending on the types of server (file servers, database servers, web servers, transaction servers), platform (software and hardware), and client requirements. According to Orfali “Middleware is a vague term that covers all the distributed software needed to support interactions between clients and servers. Middleware does not include the software that provides the actual service, that is in the server’s domain. It also does not include the user interface or the application’s logic –that is in the client’s domain.” [Ref. 10].

Figure 3 depicts a middleware server, that is capable of providing access to two remote database management systems. In order to provide this functionality, the middleware product must know how to communicate with each database, so it will contain a database specific driver for each DBMS to which it must connect. The client can only interface with the databases via a middleware driver, installed on the client machine and must use the middleware API to send requests.

The important concept here is that the client does not ever need to know how many or what types of databases the middleware actually communicates with. All of the details are hidden. Databases can be merged, split, added or deleted as long as the client/middleware interface remains unchanged. This is a very powerful capability. The databases can be remote or local. They can be moved, or upgraded without impacting the client application.

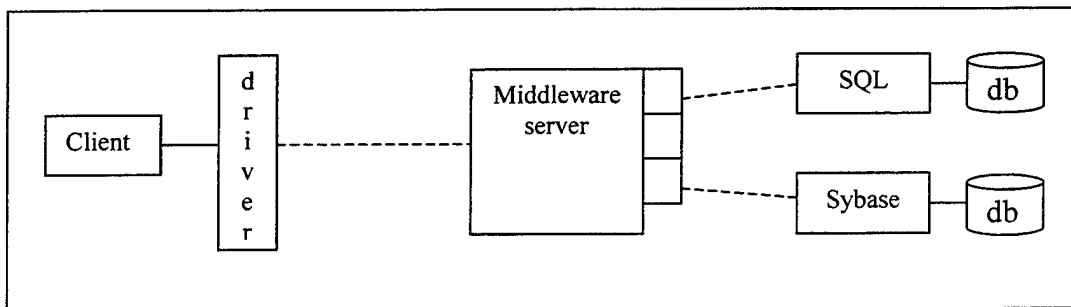


Figure 3: Middleware Solution

Middleware can be referred to as a gateway. Gateway is a piece of software that causes one DBMS to look like another, accepting the programming interface, SQL grammar, and data stream protocol of a single DBMS, and translating it to the programming interface, SQL grammar, and data stream protocol of the hidden (target) DBMS. Multiple drivers can reside on a gateway, or middleware server. Each database requires a proprietary driver to access the database. If multiple clients require access to multiple databases, client configuration would be an administrative challenge. By

employing a middleware server, all drivers reside on one machine instead of on every client machine and can be easily controlled by the system administrator.

Middleware can be divided into two functional layers, the service specific functions, and the underlying utility functions. The service specific functions are the services that will be invoked by the client. The utility functions are the communication stacks, RPC's, and queuing services that provide the base for the service-specific functions.

5. Common Gateway Interface

Hyper Text Transfer Protocol (HTTP) is a stateless protocol that operates on top of TCP/IP. An HTTP client establishes a connection with a web server and issues a request. The server returns a response and closes the connection. HTML forms, web pages with text fields and a submit button, are common on the internet yet web servers are not capable of interpreting their contents. Web servers act as a conduit and pass the content of web forms to a back end process that manipulates the form data.

Common Gateway Interface (CGI) is a protocol, that provides for the manipulation of web forms. It allows clients to pass information to server-based programs that generate responses in the form of dynamically created web pages. An HTTP server listens to incoming connection requests and either returns a file or passes the request to a CGI application. The CGI program resides on the HTTP server. Each form page has a form tag and a submit button. The submit button will cause the browser to gather all user input fields, package them and submit them to the URL specified in the action command. The URL contains a host name and the directory and file name of the CGI script to invoke. In Figure 4, the user input gets sent to the HTTP server which identifies the CGI tag and starts the CGI process, passing it the input information. The CGI process, parses the user input, processes it, generates a response HTML page (using HTML tags) and passes the page to the HTTP server that forwards it to the clients web browser.

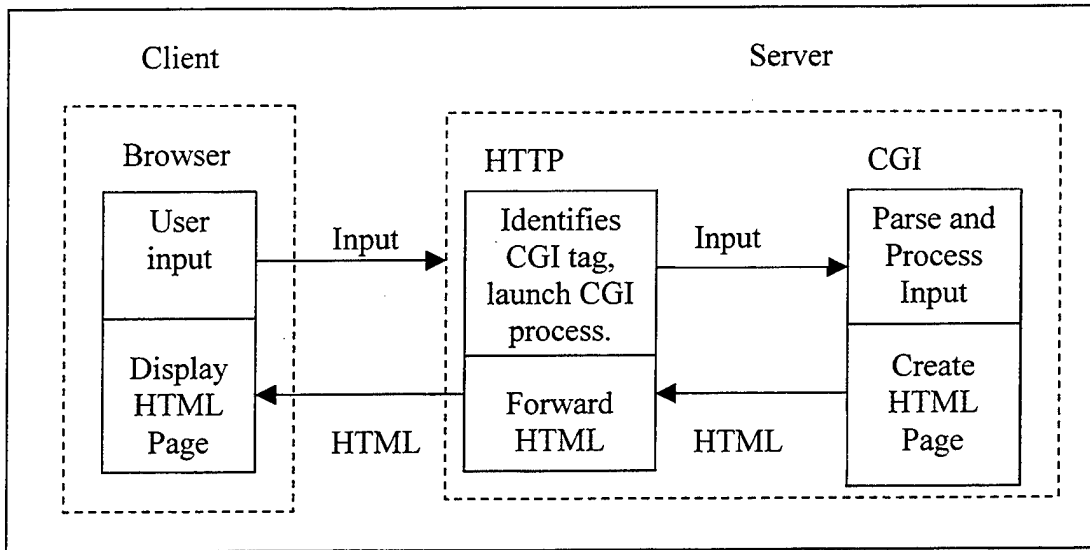


Figure 4: Common Gateway Interface

To handle multiple CGI executions, the HTTP server forks and runs the CGI process. CGI is single threaded, meaning servers must create a separate process for each request received. One of the primary limitations of CGI is that HTTP is a stateless protocol, which means it maintains no knowledge about previous requests. As stated earlier, a client will download a form, input the requested data, submit the form to the server. The server will fork the CGI process, which will service the request, generated the dynamic HTML page, and return it to the client. At this point the transaction is complete. In order to service another request, the client must go through the entire cycle once more, and the system overhead associated with starting another process.

Since all CGI processes are executed on the server, limited error checking is available on the client side and a lot of overhead is associated with executing the script. CGI programs are generally interpreted at run-time and are slower than compiled programs. A CGI script must exist for each web form that exists. Additionally when Perl is chosen as a scripting language, designers must interface a scripted language with the database vendors supplied libraries. Designers must understand the CGI host scripting language and must also understand HTML tags to format a result. If executing a database aware CGI script, since HTTP is a stateless protocol, the database will have to be opened and closed continuously to service the requests, which adds increased overhead and decreased response time.

D. CLIENT/SERVER MODEL

Client/server is an overloaded word and can be interpreted a number of ways. Some uses of the term represent physical partitioning of applications across computers. For example the client computer gets services from a separate file server/computer. According to Orfali: "The client/server model, entails two autonomous processes working together usually over a network; client processes request specific services which server processes respond to and process [Ref. 10]".

In this thesis we define client/server as two separate logical entities that work together over a network to accomplish a task. The server process can reside on the same machine as the client, or be located on a physically different machine. A client/server implementation is different than a stand alone application. A stand alone application does not require the services of another process, it is self sufficient. An example of a stand alone process would be a basic text editor. As network bandwidth continues to increase, the ability to invoke a method on a remote server is becoming transparent to the end user, so client/server architectures are becoming more attractive. When designers use multi-threading, the benefits of threads making remote method calls may drastically improve the performance of an application

In the client/server architecture it is common for servers to become the clients of other server entities. Client/server systems seeks to optimally distribute processing activities via many-to-many relationships over different computer platforms. Each server process provides a unique set of functions, a client process can interact with many server processes. Each server process must be capable of simultaneously handling multiple client requests for service. The anticipated benefit of client/server is the ability to abstract hardware and software concerns and focus on developing and building user-friendly, cost effective systems [Ref. 5].

One of the key requirements for a client/server model is the ability for the client process and the server process to be able to communicate. Since it is possible for client and server processes to be running on separate machines a common network and communication protocol must be employed. A client process must be capable of finding a server process, via some type of addressing protocol. For example, if TCP/IP is being used as the network protocol, the server process may have a thread that is constantly listening to a pre-specified port number, say 5000 for client requests.

Each server has a unique host internet protocol (IP) address for example: 131.120.1.91. The IP address plus the port number specify a TCP/IP communication socket. A client process must know the server socket information to establish a

connection with the server. Once connected, the two processes must be capable of effectively communicating through those sockets using an agreed upon communication protocol, to pass messages back and fourth across a network.

The communication protocol for distributed client/server computing across a network is extremely important. The client and server process must be capable of effectively communicating: establishing network connections, passing parameters, and various objects across a network in support of the services the server is going to provide.

All client/server systems have the following distinguishing characteristics [Ref. 10]:

- *Service*: Client/server is primarily a relationship between processes running on separate or possibly the same machine. The server process is a provider of services. The client is a consumer of services. In essence, client/server provides a clean separation of function based on the idea of service.
- *Shared resources*: A server can service many clients at the same time and regulate their access to shared resources.
- *Asymmetrical protocols*: Clients always initiate the dialog by requesting a service. Servers are passively awaiting requests from the clients.
- *Transparency of location*: The server is a process that can reside on the same machine as the client or on a different machine across a network. As pointed out earlier, a program can be a client, a server, or both.
- *Message-based exchanges*: Clients and servers are loosely coupled systems that interact through a message-passing mechanism. The message is the delivery mechanism for the service requests and replies.
- *Encapsulation of services*: The server is a "specialist". A message tells a server what service is requested; it is then up to the server to determine how to get the job done. Servers can be upgraded without affecting the clients as long as the published message interface is not changed.
- *Scalability*: Client/server systems can be scaled horizontally by adding new clients, and vertically by migrating to a larger, faster server machines or multi-servers.
- *Integrity*: The server code and server data is centrally maintained, which results in cheaper maintenance and the guarding of shared data integrity. At the same time, the clients remain personal and independent.

1. Two-Tier Client/Server Architecture

The functional units of the client/server model consist of a client process and a server process. An interface is declared that specifies what services the server process can provide to the client and what the parameters, and return types will be. The services could be various methods or functions such as:

```
public boolean setConnection(String name, String pass,  
                             String database);  
  
public Vector executeQuery(String query);
```

In a two-tier environment the server process is capable of providing the service and does not require the services of another server. Figure 5 shows a two tier client/server model. The client uses the interface to interact with the server process. The server process can be running locally or on a remote system. The gray part depicts the interface, that declares what methods or services the server process can provide. The interface may also handle the communication details to submit a request across a network. The black boxes depict business logic. Business logic is a set of rules, that enforce or implement an organization's policies. The rules may be located on the server, client or both.

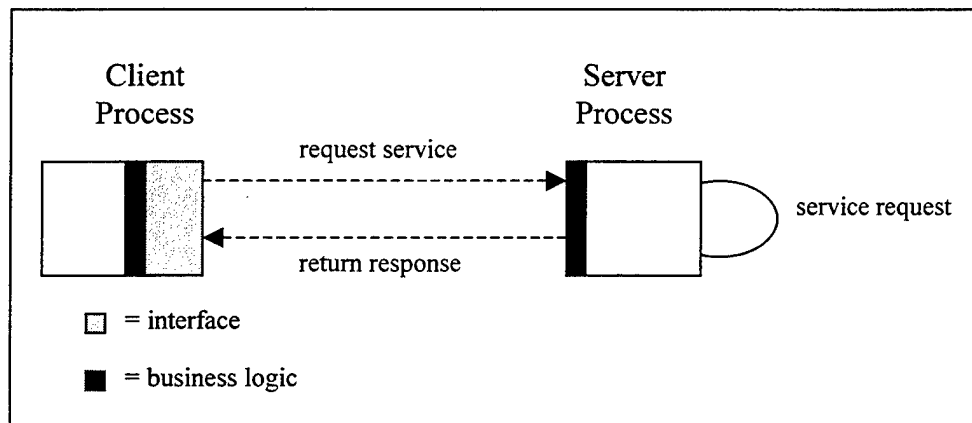


Figure 5: Two-Tier Client/Server

One issue that makes the client/server model attractive is that an organization can store its business logic on a central server. Business logic is composed of various rules that enforce the company's policies. For example, if an organization requires an expense report for any expenditures over \$20. The logic can be enforced on the server or on the client. As with all corporate policies, the rules may change. A new manager decides that

\$20 is to low, and that an expense report will only be required for expenditures over \$100.

If the logic was enforced in the client application, the organization will have to generate a new client application and distribute it throughout the organization. Since there are numerous client applications, which may be dispersed throughout the world, this can be an expensive endeavor. Also since the business logic can be fairly complex, and lengthy, enforcing it in the client will make the client code larger than it otherwise could be. So by encapsulating all business logic on the server, organizations can change the logic at one location, once again decreasing administrative costs.

A simple example of a two tier client/server application is MS SQL 6.5 Client Configuration Utility. This application is used to configure a client process to communicate with a server process.

2. N- Tier Client Server System

Since servers can become clients of other servers, ultimately what exists is the potential for multiple tiers. Each server provides an interface to the services it can provide. Since the implementation details of those services are transparent to the client application, the server may use the services of another server process to actually execute a method or perform a function.

The following figure depicts a three tier implementation of a client server model. A database aware client is making a database request via a middleware server that uses the database management system to actually process the request.

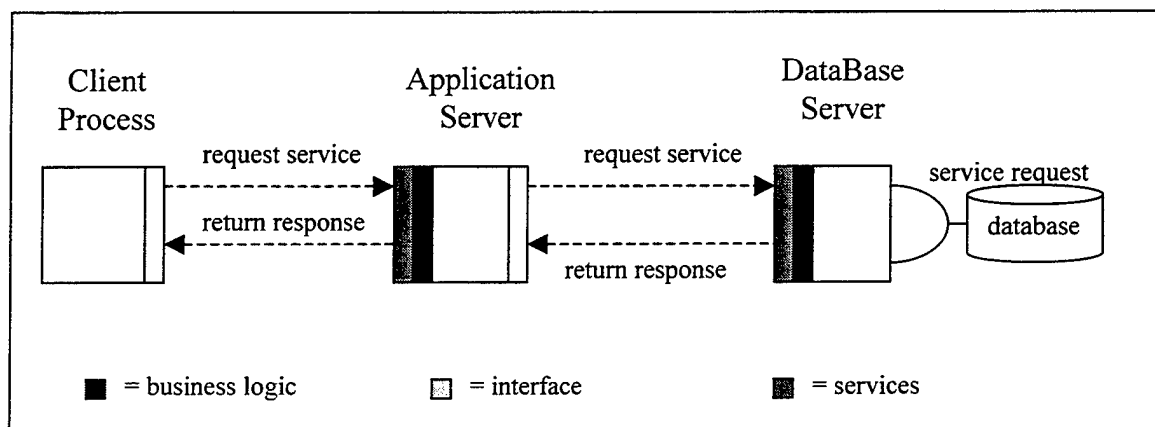


Figure 6: N-Tier Client/Server Model

Tier one of this system is a client process that can only interface with the middleware server. It contains no business logic, so it can be considered a thin client. A thin client is a client application that is as small as possible, containing minimal business logic.

The second tier, consists of an application server. The application server implements the interface depicted by the dark gray bar. The middleware server, also contains the organization's business logic, which is used to enforce policy. In this example, the application server, will utilize the interface of the database server. This interface may include a database driver, which can process requests into a format understandable by the database server (engine), and is responsible for the transport layer, or the network protocols to effectively communicate with the database engine.

The database server, is the third tier in this example. It provides the implementation details of the methods made available in its interface, and is once again depicted by a dark gray bar. It also is responsible for the implementation of some business logic. For example, the DBMS maintains and enforces user access to various database operations. This logic is usually defined by the Data Base Administrator. The Database Server will service the request and transmit the results back to the application server, which may perform some additional manipulation before sending a result back to the client. Spreading the functions among multiple tiers provides scalability, faster performance, robustness and flexibility. By implementing a middle tier, data can be integrated from multiple resources [Ref. 10].

III. JAVA DATABASE CONNECTIVITY (JDBC)

In this chapter, we provide the details of Java Database Connectivity which is implemented in the `java.sql` package of JDK 1.1. The JDBC API will be covered, providing an explanation of key classes, and methods used. For each method a short example will be presented. The examples presented represent only a small subset of the available methods found in the `java.sql` package. Consult the JDBC API reference for additional information.

Java Database Connectivity (JDBC) is an application program interface developed by Sun Microsystems, that allows a Java program to communicate with a database server using Structured Query Language (SQL) commands. It provides Java programs the ability to communicate with relational database management systems similar to Microsoft's Open Database Connectivity (ODBC) API [Ref. 13]. The JDBC Application Programming Interface is found in the `java.sql` interface, of JDK 1.1. and contains eight interfaces and six classes as can be seen in the following table.

Interfaces	Classes	Exceptions
CallableStatement	Date	DataTruncation
Connection	DriverManager	SQLException
DatabaseMetaData	DriverPropertyInfo	SQLWarning
Driver	Time	
PreparedStatement	Timestamp	
ResultSet	Types	
ResultSetMetaData		
Statement		

Table 1: JDK1.1 SQL Package

An explanation of all the classes and interfaces is beyond the scope of this thesis, so this chapter will explain and demonstrate the core classes and interfaces which are: `DriverManager`, `Connection`, `PreparedStatement`, and `ResultSet`. Each of the core classes corresponds to a critical phase of database access[Ref. 13]. The client cannot continue if one of the phases fails by throwing an exception. For example if an appropriate JDBC driver cannot be loaded, then a client cannot establish a database connection, because the

Java application will not be able to talk to the backend database engine. If the client cannot establish a connection then it cannot create a statement object, which uses the connection to transmit a query for processing.

Figure 7 provides an abstract view at the key interfaces included in the java.sql package. The DriverManager, loads the JDBC driver that can communicate with the DBMS, for example: the JDBC/ODBC bridge "sun.jdbc.odbc.JdbcOdbcDriver". Once the driver is properly loaded the Java client application can establish a connection by using: driver.getConnection, which returns a connection object. The client then uses the connection object to create a statement object. A Statement object moves the SQL string to the database engine for preprocessing and eventually execution, which may or may not return a result set object.

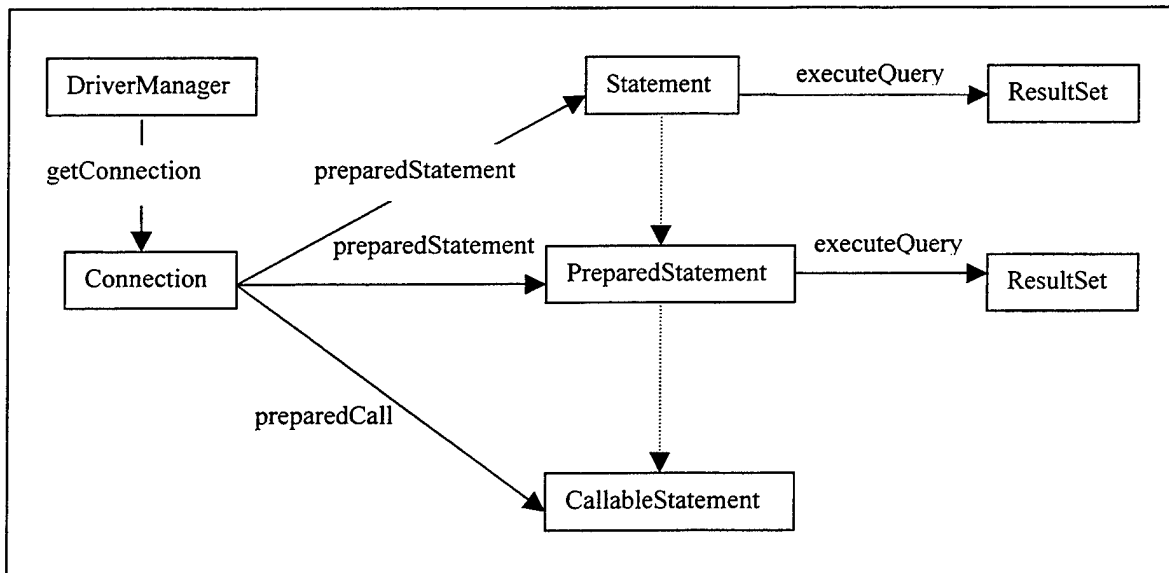


Figure 7: JDBC Object Relations [Ref. 13]

Before explaining the interfaces and class methods used in this research the following code demonstrates the entire process. A Jdbc:Odbc driver is explicitly loaded followed by establishing a connection, submitting a SQL Statement and processing a result set.

```

try {
    //set the connection
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    //connection paramters
    String user      = "dba";
    String password  = "sql";
    String dbase     = "jdbc:odbc:acctsDatabase";

    //get connection object
    Connection con = DriverManager.getConnection(dbase,
        user, password);

    //get statement object
    Statement stmt = con.createStatement();

    //execute the SQL statement
    ResultSet result = stmt.execute("select partType from
        parts");

    //display the result
    while( result.next() ){
        String part = result.getString("partType");
        System.out.println( "Part Type: " + part );
    }

    //close the connection
    con.close();
}

```

A. DRIVERMANAGER CLASS

The DriverManager class contains methods that are used to manage a set of JDBC drivers. Each JDBC driver must provide a class that implements the Driver interface, which is used by the DriverManager. As part of initialization, a program can explicitly tell the DriverManager what driver to load, by using the Class.forName(<driver name>) call. If the user does not use this call and attempts to create a connection object the DriverManager class will check with each registered driver to see if it can connect to the given URL. If more than one driver can connect to the URL, the DriverManager will invoke the first compatible driver encountered.

Connection objects are generated from the class DriverManager. When getConnection is called, the DriverManager will attempt to locate a suitable driver from those loaded at initialization and those loaded explicitly using the same classloader as the

current applet or application. The URL provided to the getConnection function names the driver to be used to establish the connection. The connection protocol supported by Sun is:

```
jdbc:<subprotocol>://<host>:<portnumber>/<datasource>
```

For example: String url = "jdbc:dbaw://131.120.1.91:8899/acctsDatabase" uses jdbc protocol with a dbaw (dbAnywhere) sub protocol to connect to port 8899, on host 131.120.1.91, and then presents the data source name acctsDatabase to the port to locate the specific database. The DriverManager uses this URL to find a registered Driver which can connect to the source. The driver manager is used to establish a connection to a datasource.

```
public void setConnection()  
{  
    //explicitly load the driver  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
  
    //connection paramters  
    String user = "dba";  
    String password = "sql";  
    String dbase = "jdbc:odbc:acctsDatabase";  
  
    //get connection object  
    Connection con = DriverManager.getConnection(dbase,  
                                                user, password);  
  
} //end setConnection
```

B. CONNECTION INTERFACE

A connection object represents a connection of your application to a database and is used to execute the next phase of database access, creating a statement object which will allow the user to execute a SQL command. It can also be used to commit a change to the database, as well as rollback.

A transaction is a set of statements that have been executed and committed or rolled back. By default all SQL statements are set for automatic commitment, so each individual statement is committed to the database upon it's completion. This can be disabled by using the method setAutoCommitment(false), which requires the designer to

explicitly invoke `connection.commit()`, to commit the transaction to the database. This is effective if the user wants to make a number of changes and to commit them all at once. To rollback, (remove all the changes since the last commit), invoke `connection.rollback()`. The following section provides an overview of some of the key methods that exist in this class.

1. Statement

There are three statement objects of which the inheritance hierarchy is: `Statement`, `PreparedStatement` and `CallableStatement`. To obtain a statement object the user can call `connection.createStatement()`. The statement object can be used to execute a SQL statement. This type of statement object is useful for SQL statements that will only be generated once. For example:

```
Statement stmt = con.createStatement();
stmt.execute("CREATE TABLE parts(" +
            "partID    SMALLINT NOT NULL, " +
            "partType CHAR(20) NOT NULL) ");
con.commit();
stmt.close();
```

2. PreparedStatement

To pass in-parameters or to increase efficiency for a SQL statement that will be executed a number of times, use a `PreparedStatement()`. A `PreparedStatement` takes in-parameters which can be passed into a SQL statement after the statement has been prepared by the DBMS server. The server formulates and optimizes an access plan, that can be re-used multiple times. The access plan is executed when the client uses the `execute()` command. This increases performance, because the DBMS does not need to incur the overhead associated with generating an access plan for every statement. Not all DBMS's can support this option, in which case the code executes as a statement object. For example prepared statements cannot be used with the Text File ODBC Driver.

To bind input parameters, `setXXX` is used where `XXX` can be any primitive type or a `String`. The `setXXX` methods for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type `Integer` then `setInt` should be used. Columns can be referenced by column index, which begin with one, for greater efficiency, or by column name for

convenience. The following example demonstrates how a `PreparedStatement` can be effectively used to populate a table with 100 items, each with a unique `partID`.

```
//set auto commit to false
con.setAutoCommit(false);

// insert 100 parts into parts table
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO parts (partType, partID, quantity) " +
    "VALUES ( ?, ?, ?)");

pstmt.setString(1, "Tire");
pstmt.setInt(3, 4);

for (partID = 1; partID <= 100; partID++){
    pstmt.setShort(2, partID);
    pstmt.executeUpdate();
}

con.commit();
pstmt.close();
```

3. Callable statement

`CallableStatement` extends `PreparedStatement` and is used to execute stored procedures. Stored procedures are blocks of SQL code that are stored in the database and executed on the server. This increases efficiency for SQL Statements that are executed often, by reducing the overhead of regenerating an access plan. The DBMS generates and stores the access plan once, and other applications can use the procedure. Stored procedures need to be supported by the database, and the exact syntax may differ between vendors.

JDBC provides a stored procedure SQL escape that allows stored procedures to be called in a standard way for all RDBMS's. This escape syntax has one form that includes a result parameter and one that does not. If used, the result parameter must be registered as an OUT parameter. The other parameters may be used for input, output or both. Parameters are referred to sequentially, by number. The first parameter is 1. The following example of creating a stored procedure is from JavaSoft's Java DataBase Programming Tutorial [Ref. 27].

```

Stmt.execute("CREATE PROCEDURE getDailyTotal " +
            "@day char(3), @dayTotal int output " +
            "AS " +
            "BEGIN " +
            "    SELECT @dayTotal = sum (cups) " +
            "    FROM coffeeData " +
            "    WHERE day = @day " +
            "END" );

```

If the designer knows there are no output parameters, but input parameters are required use `PreparedStatement`. If there are no input and output parameters `Statement` can be used. Once the stored procedure is created, it maybe executed with the following:

```

CallableStatement cstmt = con.prepareCall("{call
            getDailyTotal (?, ?) }");
cstmt.setString(1, "Mon");
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.executeUpdate();
System.out.println( "Total for Mon is " +
            cstmt.getInt(2) );

```

IN parameter values are set using the set methods inherited from `PreparedStatement`. So in this case an in parameter is "Mon". The type of all OUT parameters must be registered prior to executing the stored procedure so `registerOutParameter(2, java.sql.Types.INTEGER)`, then execute the statement. The get method must be used to retrieve the OUT parameters. Columns are referenced by index for efficiency in this example. A Callable statement may return a `ResultSet` or multiple `ResultSets`.

4. DatabaseMetaData

Another key method provided by the connection class is `getMetaData()`, which returns a `DatabaseMetaData` object. Metadata is data about the database and its contents can be discovered at run time. Each database maintains a dynamic catalog that describes itself to include the tables, views, privileges and capabilities of the database. This object

is able to provide information describing the database tables, its supported SQL grammar, its stored procedures, and the capabilities of this connection. The DatabaseMetaData interface has a total of 133 methods [Ref. 10]. For example, to get the names of all the tables in a database the following method could be used:

```
getTables(String catalog,  
          String schemaPattern,  
          String tableNamePattern,  
          String types[]) throws SQLException
```

A schema describes a group of related tables and access permissions. A catalog describes a group of related schemas. There are a number of types to include "TABLE" and "VIEW". To use the method to provide a list of available tables from a database:

```
DatabaseMetaData dmd = con.getMetaData();  
String[] types = { "TABLE" };  
ResultSet rs = dmd.getTables(null, null, "%", types)
```

The ResultSet contains information about all tables in the database, for example to get the table name, which is contained in the third argument, use `String name = rs.getString(3)`. This information is important for code portability. To populate a drop down menu, this information can provide the table choices dynamically.

C. STATEMENT INTERFACE

A statement object is a container for executing SQL statements. This represents the next phase of database access, the execution of a SQL statement. One statement object can be reused many times to execute a number of SQL Statements. There are three functions that can be invoked to execute the SQL statement depending upon the type of SQL operation.

1. Execute

The method *boolean execute(String sql)* is used to execute dynamic SQL statement when the designer does not know if the statement will be an update, or query operation. The method returns true if a ResultSet was generated via a query. This indicates to the designer that the operation was a select, now the calling method may

process the result set. If the method returns false, this indicates that the statement was an update, and returns the number of rows affected by the statement. In both cases the results can be obtained as follows:

```
ResultSet result = statement.getResultSet()
int rowsUpdated = stmt.getUpdateCount().
```

When the designer knows what type of SQL Statement is being executed it is more appropriate to use `executeQuery` or `executeUpdate`.

2. Execute Query

The method *resultSet executeQuery(String sql)* should be used when the client is executing a SELECT SQL Statement and is prepared to process a ResultSet. Select statements are used to extract data from various tables based upon search criteria. The ResultSet object returned is never null. The rows in the result are accessed in order, but the elements in the various columns can be accessed in an order. It is recommended that columns be accessed from left to right and that each row be read only once.

```
ResultSet result = stmt.executeQuery("SELECT * FROM
                                      Parts");
```

3. Execute Update

The method *int executeUpdate(String sql)* is used to execute a SQL INSERT, UPDATE or DELETE statement. It returns a row count representing the number of rows affected by the SQL statement or zero for SQL statements that return nothing. The following code demonstrates the various update functions (Create, Insert, Update, Delete). For each of the following examples assume that "stmt" is a valid Statement object.

a. Create Table

A two dimensional table is used to abstractly view tables. Each table has a table name and has various attributes of various types which represent the column headers. The following example depicts the SQL string for creating a table called invoices, that contains two attributes: id and supplier.

```
stmt.executeUpdate("CREATE TABLE invoices( id int,  
supplier char(20))");
```

b. Insert

Inserts data into a pre-existing table. User must specify the table name, the column names, and the corresponding values to be inserted into the table.

```
stmt.executeUpdate("INSERT INTO invoices (id,  
supplier) VALUES (2033, 'DOL') ");
```

c. Update

This SQL command is used to update a specific field of a table.

```
stmt.executeUpdate("UPDATE parts SET qnty = 5  
WHERE part = 'brake pad' ");
```

d. Delete

This command removes one or more rows, or tuples from the specified table. The following example would delete all tuples where part is muffler from the parts table.

```
stmt.executeUpdate("DELETE FROM parts WHERE part =  
'muffler' ");
```

D. RESULTSET INTERFACE

A `ResultSet` object captures the information returned from a `SELECT` Statement. The object maintains a cursor that points to the current row of data, which can be traversed via the `next()` method. This object provides methods that allow access to the results of a query. A `ResultSet` is automatically closed by the Statement that generated it when that Statement is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results.

1. Get Meta Data

The `ResultSetMetaData` `getMetaData()` method returns a `ResultSetMetaData` object which can provide detailed information about the `ResultSet`, to include column information. This information is useful in presenting the `ResultSet` in an interface. The following lists only a few of the `ResultSetMetaData` methods, see Interface `java.sql.ResultSetMetaData` API for additional methods.

```
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount      = rsmd.getColumnCount();
int columnWidth      = rsmd.getColumnDisplaySize(ix);
String columnName    = rsmd.getColumnName(ix);
int javaSQLType      = rsmd.getColumnType(ix)
```

2. Get Type

The `XXX` `getXXX` method is used to get results from a `ResultSet`. If `getString` is used, then it will return a `String`. So based upon the column index, it will convert the type into the Java specified type by the `getXXX`. Not all conversions are legal so the correct conversion should be used. The method is overloaded to also accept a `String`, the column name, as an in parameter. Column names used as input to `getXXX` methods are case insensitive.

3. Next

There is a "row cursor" in the result that points to the current row. The method `result.next()`, moves the cursor to the next row, and returns true if the row exists. The `ResultSet` object controls access to the row results of a given statement. The method must be called to initialize the cursor to point to the first row of data. For example to step through a `ResultSet` result, the `next()` method would be used as follows:

```
String part;
int quantity;
while(result.next() ){
    part = result.getString("part");
    quantity = result.getInt("qnty");
    System.out.println( part + " | " + quantity );
```

}

The application designer needs to know what data type is included in the result set, and the name of the column to access the results. Once again, the designer can reference the column containing the data by name or by index. To get the results a data conversion from the SQL return type to the desired Java data type is required. The getXXX method must be used to get various SQL data types.

For each getXXX method, the JDBC driver must convert between the database type and a Java equivalent. An invalid data conversion will result in an exception. Some SQL types have a direct Java equivalent, such as SQL Integer to a Java int. Several SQL types can be converted to a Java equivalent, such as SQL char, varchar, and longvarchar to Java string type.

E. JDBC DRIVERS

Database drivers provide the implementation of the abstract classes provided by the JDBC API. The driver resides on the Java client machine and is used to establish a connection to a relational database. The JDBC driver can be a JDBC/ODBC bridge, a middleware protocol library, or a native database driver. The driver provides the "black box" interface, that accepts JDBC input from the Java application, and understands the vendor specific relational database language and network protocols. It accepts the JDBC input from the client application, translates it to a vendor specific protocol, and uses a vendor supporting networking protocol to transmit the request across the network.

In order to pass the JDBC compliance tests, drivers are required to support at least ANSI SQL 92 Entry Level (SQL2). There are four categories of JDBC drivers as designated by JavaSoft.[Ref. 12] The drivers are:

- *Type I. JDBC/ODBC bridge:* Available with JDK 1.1, uses existing ODBC vendor drivers. The slowest driver to translating JDBC calls to ODBC calls to vendor specific calls.
- *Type II. Native-API, partly Java driver:* Translates JDBC call into language understood by the specific DMBS client driver, then invokes the client driver for further processing. Faster than Type I, slower than Type IV. Can only interface with specific DBMS.

- *Type III. Network-protocol, all-Java driver:* Provides flexibility, may use ODBC/JDBC bridge or native protocol drivers to access DBMS's. Not as fast as Type II or IV drivers. Simplifies administration, all DBMS specific drivers are loaded on the middleware server machine. Clients only requires the network protocol interface and middleware API to communicate with the TYPE III driver.
- *Type IV. Native-protocol, all-Java driver:* Provides the fastest access, mapping the JDBC calls directly into a DMBS protocol. Can only interface with vendor specific DBMS and can be expensive. Requires each client machine to have a driver, so increases administrative burden of managing licenses.

1. JDBC/ODBC Bridge (Type I)

The JDBC/ODBC Bridge (Figure 8) was designed to take advantage of the large number of ODBC enabled drivers. The bridge was intended to provide an initial solution until database vendors could produce their own vendor specific JDBC drivers. Basically the bridge converts the JDBC calls into ODBC calls. The ODBC driver manager, will invoke the database vendor specific ODBC driver, and pass the calls to database driver for further processing.

The client side application or applet uses the JDBC API to load the "sun.jdbc.odbc.JdbcOdbcDriver". This driver translates the Java SQL statements into and ODBC format, then invokes the ODBC Driver Manager (odbc32.dll) which refers to the odbc.ini file that contains a data source name and vendor specific driver it is associated with. The vendor specific driver or dll then translates the ODBC call into a vendor specific call, and sends the request across the network to the database manager. The process is reversed when a response is sent from the database manager back to the client application.

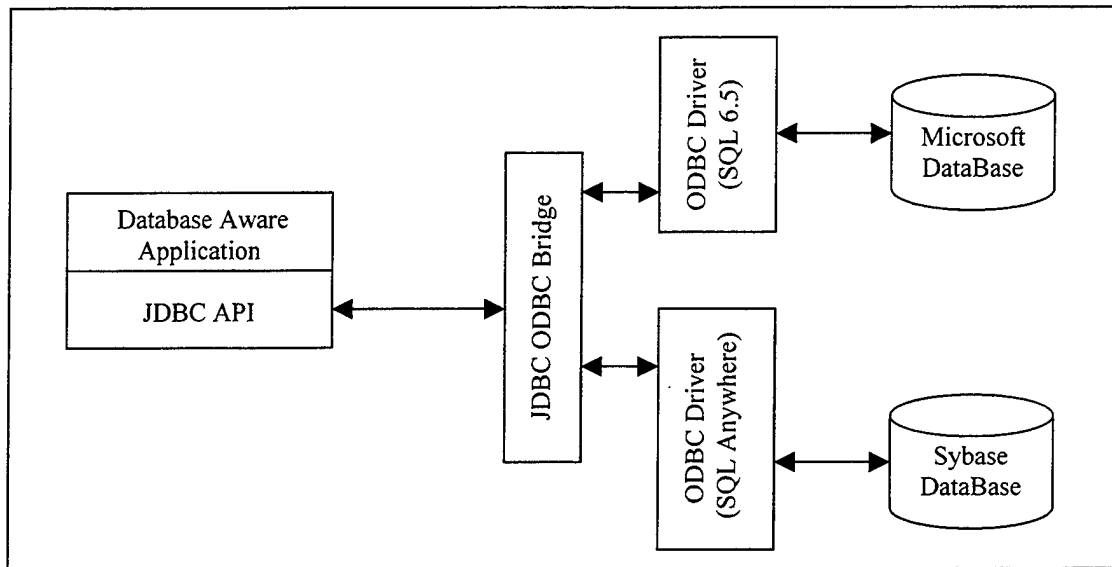


Figure 8: JDBC/ODBC Bridge

One of the disadvantages of using the JDBC/ODBC Bridge is the overhead associated with the call mappings. Java applications can be written using the JDBC API but all calls must be filtered through the bridge and translated into ODBC calls that are understood by the respective relational databases. The call must go from JDBC through the bridge to the ODBC driver and finally from ODBC to the native client-API, say Sybase SQL Anywhere in this example, then to the database.

A JDBC/ODBC bridge is effective for an application server. A middleware server provides the database access, so all ODBC drivers reside on that machine. The client makes a call to the application server, which establishes the database connection and returns a string or data stream to the client. The application server used the JDBC API to interact with the database. The client does not use the JDBC API at all. This was the implementation used in our socket and remote method invocation models that follow in Chapters IV, and V respectively.

2. Native-API, Partly Java Driver (Type II)

Type II drivers convert Java calls into vendor specific calls. The drivers are usually written in C, accept the Java calls then map them to vendor specific calls. The call then gets processed by the vendor specific driver, translating it into the DBMS's specific query language and communication protocol. This is a partly Java driver, that requires a vendor supplied library to translate JDBC functions into the DBMS's specific

query language, such as Oracle's OCI [Ref.23]. Type II Drivers bypass the ODBC translation, so are faster than Type I drivers.

These drivers usually cost more than other drivers and reduce flexibility. The performance increase may warrant employing native drivers, if the organization is committed to a single DBMS.

3. Network Protocol, All-Java Driver (Type III)

The network Type III driver (Figure 9) translates JDBC calls into a database independent network protocol, and passes this request to a middle tier server which then translates the request into a DBMS specific protocol. Type III drivers are attractive for Internet/Intranet based multi-user data intensive applications (requiring access to multiple databases). A Type III driver can be written in Java. The cost associated with writing the driver in Java is decreased performance time since Java is slower than compiled languages like C/C++. The advantage of being written in Java is that the server can use any JDBC driver to connect to the target database(s). In Chapter VI, we use a type III Middleware driver to connect our Java client application to various relational databases.

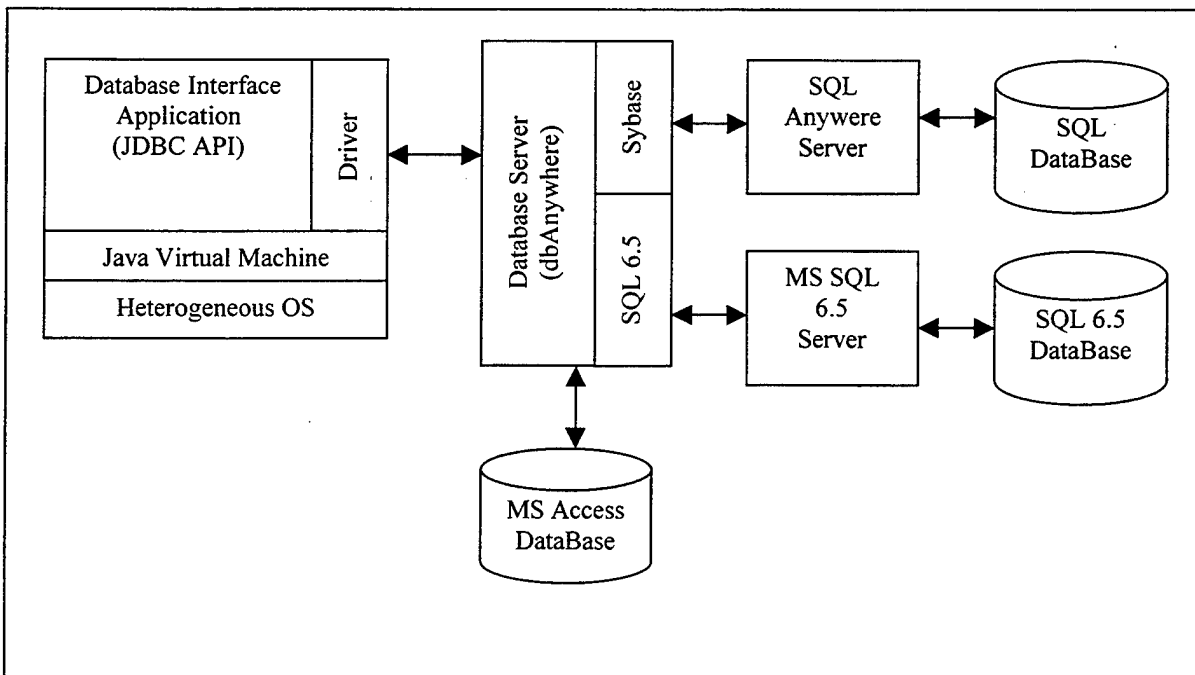


Figure 9: Net Protocol All Java Driver

To communicate with the Type 3 server a client class file must be installed on the client machine. This class file can be downloaded with the applet or application. For

example, when dbAnywhere is used as the middleware server, the client needs to have the dbaw.zip archive to communicate with the dbAnywhere Server. The Database Server manages the transfer of information and calls between the Client and the DBMS. The bridge transforms Java calls to DBMS protocol and the DBMS responses back into Java.

The Type III driver contains a number of vendor specific drivers, or can use the ODBC/JDBC driver to provide database access. The bridge can connect the client to local databases, which must reside on the same machine as the middleware server, such as MS Access, or to remote databases such as Oracle, MS SQL Server, SyBase, InterBase, or IBM DB/2, stored on another machine.

As can be expected, a Type III driver is slower than other JDBC drivers. In an evaluation by Mukal Sood [Ref. 23], the average connection time for a Type IV driver was approximately 1.1seconds, and the fastest for a type III driver was 8.4 seconds. The benefit of using a Type III driver is flexibility, installation, and database administrator maintenance. Type III drivers, can encapsulate Type IV drivers. For example, dbAnywhere provides proprietary database access for Oracle 7.x.

4. Native-Protocol, All-Java Driver (Type IV)

Type IV JDBC drivers convert the JDBC calls directly into the network protocol used by the specific DBMS. These drivers can be written entirely in Java, and can provide just in time delivery of applets. Type IV drivers provide for the best database access because of the direct translation, unfortunately they can only be supplied by the vendor and can only interface with the vendor specific database. For example, Sybase jConnect is a Type IV JDBC driver written entirely in Java and communicates directly to Sybase data sources such as Sybase SQL Anywhere. It uses Sybase's Tabular Data Stream (TDS) as the communication protocol between Sybase clients and servers.

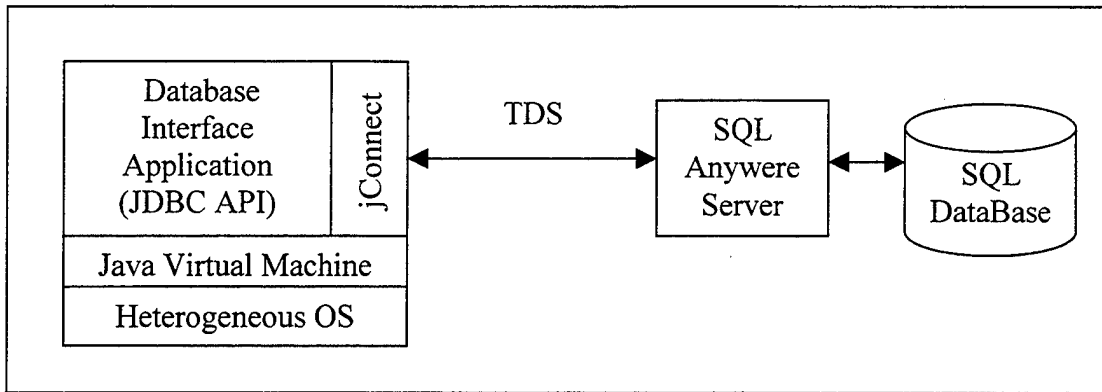


Figure 10: Native Protocol JDBC Driver

The advantage of a Type IV driver is increased performance. The number of translations is reduced to one, which directly translates the call into a DBMS specific call. The disadvantage of such a solution is the driver is supplied by the vendor and is proprietary and therefore comes with a higher cost.

5. Driver Selection

The selection of which type of driver to employ depends upon a number of factors: number of databases requiring access, performance requirements, financial, and system administration requirements.

F. CONCLUSION

Java Data Base Connectivity is a competing standard for database access. It provides the interface for application designers to be able access a database and is easy to understand, implement and use. Throughout the rest of this thesis JDBC will be used in various client-server configurations to provide heterogeneous database access.

In order to fully demonstrate the potential for Java database access in the commercial environment, the networking abilities of the language need to be explored. The next chapter will look at socket programming to provide the communication link between the client and the database aware server, and Chapter V will use Java's Remote Method Invocation to provide the communication link. For both these implementations the JDBC-ODBC bridge will be used as the driver and we will use the java.sql package to provide the database manipulation and access. In Chapter VI, two type III JDBC drivers will be used: Borland's DataGateway and Symantec's dbAnywhere. Their respective

database development integrated development environment tools will be used to facilitate database access.

IV. SOCKETS AND JDBC

A. INTRODUCTION

Connecting to a database and receiving information from it requires a lot of underlying communication. The client and server application have an established communication protocol, to effectively communicate and handle requests and responses. As pointed out earlier in a client/server environment, this communication may take place in a two-tier architecture (client database application talks directly to the database server over a network) or in a three-tier architecture (client application talks to the database server via a middleware server). Regardless of these architectures, the underlying communication between client and server side can be handled in a variety of ways.

Java provides an effective communications API to assist developers in developing distributed applications. Java offers socket-based communications that enable applications to handle network communications as if it were file I/O.

Communicating through sockets involves low-level programming. Associated with this requirement is the added flexibility of implementing a solution that meets a specific need. The next level of abstraction is Java Remote Method Invocation (RMI), which encapsulates low level socket programming, allowing local objects to communicate with remote objects via standard method invocations.

In this chapter, we provide an overview of Java socket programming, followed by a multi-cast distributed socket database model. We will evaluate the database connectivity issues using RMI in Chapter V.

B. SOCKETS

A socket encapsulates the information about both the host computer identification and the port number on which the socket is created and acts as a conduit between processes. Sockets are the software abstraction of the address space of an Internet host and one of its ports. An internet address is a unique 32-bit (IPv6 is 128-bit) number for a host. A TCP/IP host can may have as many internet addresses as it has network interfaces. A port is an entry point to an application that resides on a host and it is represented by a 16-bit integer. When a socket is created, the process that owns it can communicate with another process by going through that socket and a corresponding socket on the other side.

A socket is either a stream socket or a datagram socket. Stream sockets interface to Transfer Control Protocol (TCP) and are connection-oriented where datagram sockets interface to User Datagram Protocol (UDP) and are connection-less.

In connection-oriented communication, a connection between processes is established before data can be exchanged. Establishing the connection involves creating sockets, sending a connection request and accepting the request. This may take place as a two-way (request, reply) or three-way (request, request-reply, reply) handshake. Once the connection is established and while it is in place, data flows between processes through the sockets as streams of bytes.

The datagram sockets handle connectionless communication. There is no need to establish a connection in order to send a message for connectionless socket-based communication. Data sent between processes is called a packet and contains the destination IP address, destination port and data. Datagram socket communication is fast, but unreliable. As Stallings pointed out "In the datagram approach, each packet is treated independently, with no reference to packets that have gone before" [Ref. 24]. As local area networks become more reliable through the use of fiber, datagram socket communication is attractive due to increased performance over TCP sockets.

In order to implement socket-based communication, Java provides a socket API contained in the `java.net` package. This package allows programmers to do both low-level networking with `DatagramPacket` objects which use datagram sockets, and connection-oriented networking with `ServerSocket` and `socket` objects which implement stream sockets.

Java also provides for multicast socket communication. Multicast socket communication is a communication method which is implemented using datagram sockets to send UDP packets to a group of processes that may be servers or clients. The `java.net` package contains a `MulticastSocket` class which lets programmers implement multicast communication.

Regardless of what type of socket communication takes place, the client and the server programs must agree upon the communication protocol that is to be used. The protocol must include what port numbers to send data to, what message format to use. The client or recipient of the datagram packet must parse the data and determine what action to take. So for example, the protocol may establish that a "|", will separate tokens, and if the first token is the word "net", then the client knows it is a network request and can invoke the appropriate handler. Regardless of what type of information is contained in the message, both server and client side applications must invoke appropriate methods to process the data. Figure 11 depicts a general socket-based

client/server communication diagram. The client machine is located at IP address 131.120.1.91 and is using port number 6500 to communicate with a server object located at 131.120.1.226, listening to port 7500. The server object may then access a database via a JDBC driver to service the clients request. We are going to modify this diagram later on based on the type of sockets being used between client and server:

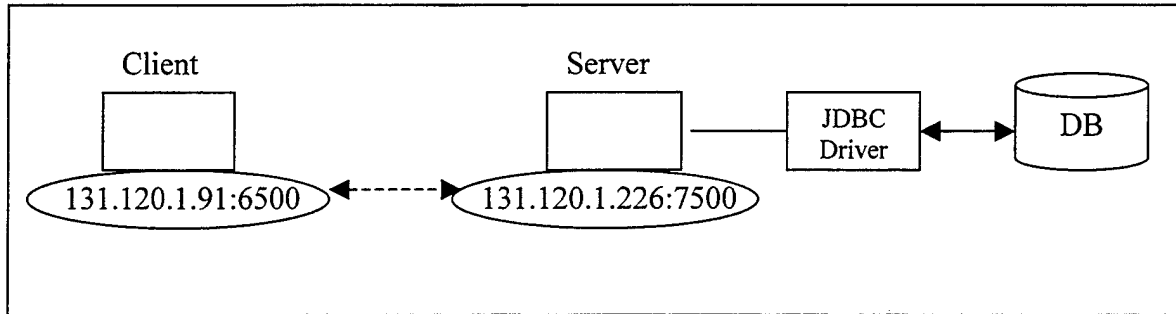


Figure 11: Socket Communications

1. Stream Socket Communication

A server may have more than one client, so it should have a dedicated listening socket for clients that wish to create a session. This socket cannot be blocked so, a communication channel is established for each client. The server socket is the first socket that receives the clients' initial connection request message and assigns the actual communication socket by accepting the request, depending on how many connections it can handle in total. This type of communication involves the following steps:

- Create a server socket whose sole purpose is to wait for connection requests from clients (by using an infinite “while” or “for” loop)


```

int port = 5000; //port number that the server will be listening on (this port
//should not be used simultaneously by any other processes in the system)
ServerSocket ss = new ServerSocket(port);
      
```
- Clients make connection requests to the server socket by creating a stream socket to server's IP address and port number


```

int serverPort = 5000; //client must know server's port number
//convert string representation of server's IP address to InetAddress object
InetAddress host = InetAddress.getByName("131.120.1.91");
//make the connection request
      
```



```
Socket clientSideSocket = new Socket(host, port); //stream socket
```

- Server accepts the request (blocking until a request comes) and assigns a new stream socket to be used for additional communication

```
Socket serverSideSocket = ss.accept(); //wait for a client to connect and  
//generate a stream socket to communicate with the calling client
```

- Data communication (stream of data)

An Output stream object is used to send the stream of bytes (data) and an input stream object for receiving the data. If both the client and the server want to send and receive the data, then they must create their output and input stream object pairs.

- Closing the connection

```
serverSideSocket.close(); //server process code  
clientSideSocket.close(); //client process code
```

Figure 12 depicts the general idea behind having one server socket which then assigns sockets for clients to continue to communicate with the server. The important thing in this communication cycle is that the `ServerSocket` `ss` is not used for the actual communication between the server and its client. It is only used to establish a connection to a client by assigning a new stream socket for follow-up communication. In order to setup a connection to any number of clients, the same `ServerSocket` may be used repeatedly.

After a communication channel has been established between the client and the server, each needs to create its own input and output stream objects in order to pass messages back and forth. A sample implementation can be seen in Appendix B which demonstrates how `ServerSocket` and `Socket` classes can be used to serve clients for database access and data manipulation requests. The client in this implementation sends queries as stream of bytes through its output stream object. The server receives requests via input stream object and handles it by calling our `ExecuteSQL` function. The `executeQuery` method executes the query by invoking the appropriate JDBC API methods. The server places the results into a vector of hash tables which can be easily converted into stream of bytes to be sent to the client via the server's output stream socket. When the client receives the data from its input stream, it just prints it to the

screen without worrying about its format, because the stream has been pre-formatted by the server (i.e., each line contains one tuple).

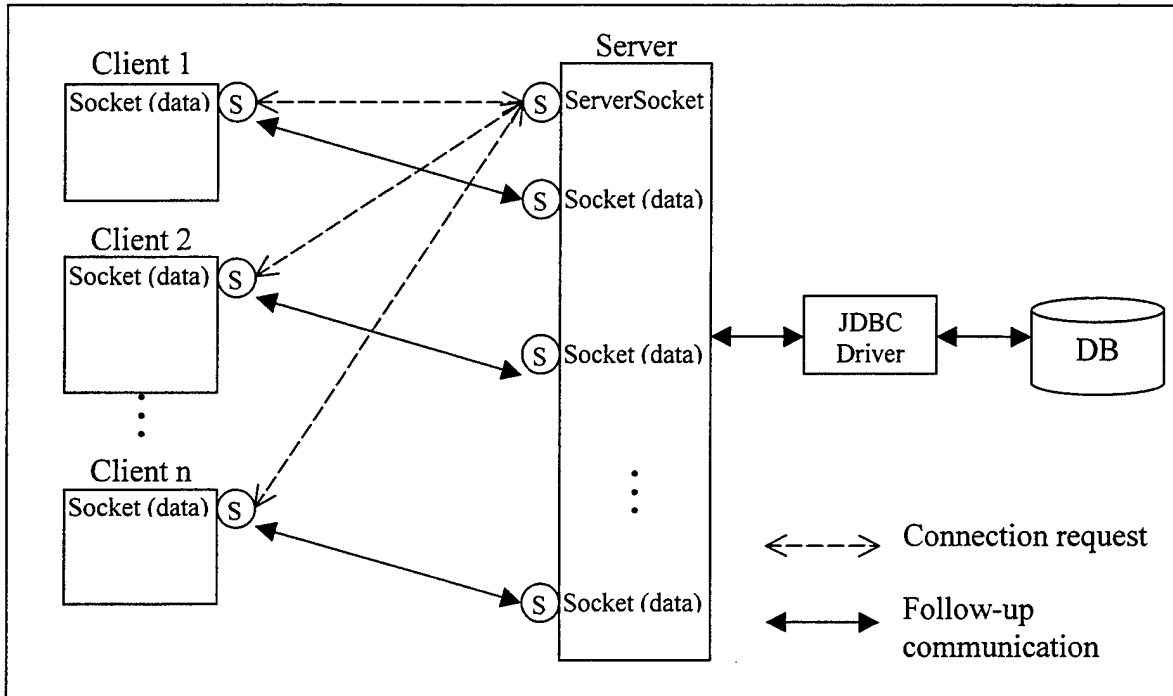


Figure 12: Stream Socket Communication

2. Datagram Socket Communication

Datagram socket communication is based on User Datagram Protocol (UDP) which is a packet-switching protocol and uses unreliable datagram packets. UDP does not attempt to ensure that each packet reaches its destination. According to David Flanagan, "A datagram is a very low-level networking interface: it is simply an array of bytes sent over the network. A datagram does not implement any kind of stream-based communication protocol, and there is no connection established between the sender and the receiver. Datagram packets are called 'unreliable' because the protocol does not make any attempt to ensure that they arrived or to resend them if they did not. Thus, packets sent through a datagram socket are not guaranteed to arrive in the order sent, or to arrive at all. On the other hand, this low-overhead protocol makes datagram transmission very fast"[Ref. 24].

The `java.net` package has a `DatagramSocket` class to implement datagram transmission. Datagram sockets are created by specifying a port number. If left

unspecified, the operating system will assign a port number automatically. The `java.net` package also offers a `DatagramPacket` class which programmers can use to create datagram packets that contain data, a destination address and a destination port. The following is an example of how to create and send a datagram packet:

```
//create a byte array that will contain the data
int dataLength = 100;
byte[] dataArray = new byte[dataLength];

String host = "230.0.1.222"; //destination address
int port = 4446; //destination port number
//get the Internet address of desired host
InetAddress address = InetAddress.getByName(host);

//create a datagram packet
datagramPacket packet = new DatagramPacket(dataArray,
                                           dataLength, address, port);

//create a datagram socket in order to send the packet
DatagramSocket datagramSocket = new DatagramSocket();

//send the packet
datagramSocket.send(packet);

//close the socket
datagramSocket.close();
```

The following is an example of how to receive a datagram packet:

```
//create a datagram socket to listen on a desired port
int port = 4446;
DatagramSocket datagramSocket = new
    DatagramSocket(port);

// create a buffer (a byte array) to read datagrams
//into. If the incoming packet
```

```

int maxBuffer = 1024;
byte[] buffer = new Byte[maxBuffer];

//create an empty buffered datagram packet
DatagramPacket packet = new DatagramPacket(buffer,
                                           maxBuffer);

//wait to receive a datagram (blocking)
datagramSocket.receive(packet);

```

Section C of this chapter provides a more detailed example of the use of datagram sockets. The model also demonstrates the power of multicast sockets which will be explained next.

3. Multicast Socket Communication

The exchange of single messages is not the best model for communication from one client to a group of servers which provide either common or different sets of services. This raises the issue of group communication which can be implemented using Java's MulticastSocket class. Multicasting means sending a single message to a group of processes. Multicast messages may be multicast UDP packets or stream objects.

The Java MulticastSocket class offers joinGroup() and leaveGroup() methods for group management. A process can be a member of multiple groups. Each group must have a valid IP address in the range of 224.0.0.1 to 239.255.255.255. In order to receive a multicast message a process must be a member of a group, but it is not necessary to be a member of a group in order to send a multicast message to a group.

In a client/server environment, server processes may be clients of other server processes. If the members of a multicast group are database servers, then coordination and control facilities between different databases can be implemented in an efficient way. The model explained in the following section demonstrates how multicast socket communication is used effectively between the companies of an Army Battalion. Each of which is maintaining a repair parts database.

C. MULTICAST SOCKET JDBC MODEL

This model has been created to demonstrate Java Database Connectivity using both unreliable, yet fast datagram sockets for point-to-point communication between

client and server applications and multicast sockets for handling coordination and control facilities between a group of database servers and client applications. It has been implemented using the Java Programming Language and the source code can be found in Appendix B. The actual evaluation environment is shown in the following figure. The objects a_CoParts, b_CoParts and c_CoParts act as database servers whose only privileges are querying (select only) and updating their own databases. The a_CoParts and b_CoParts servers and their databases (Microsoft Access Database) reside on a machine named "roxanne". The c_CoParts server resides on a machine called "fido", but its database (Microsoft SQL 6.5 Database) resides on a remote machine named "cryptologist" which is running Microsoft SQL 6.5 Server. The 1_BnParts object is a high-privileged database server which can manipulate all three of the databases. 1_BnParts resides on machine "roxanne".

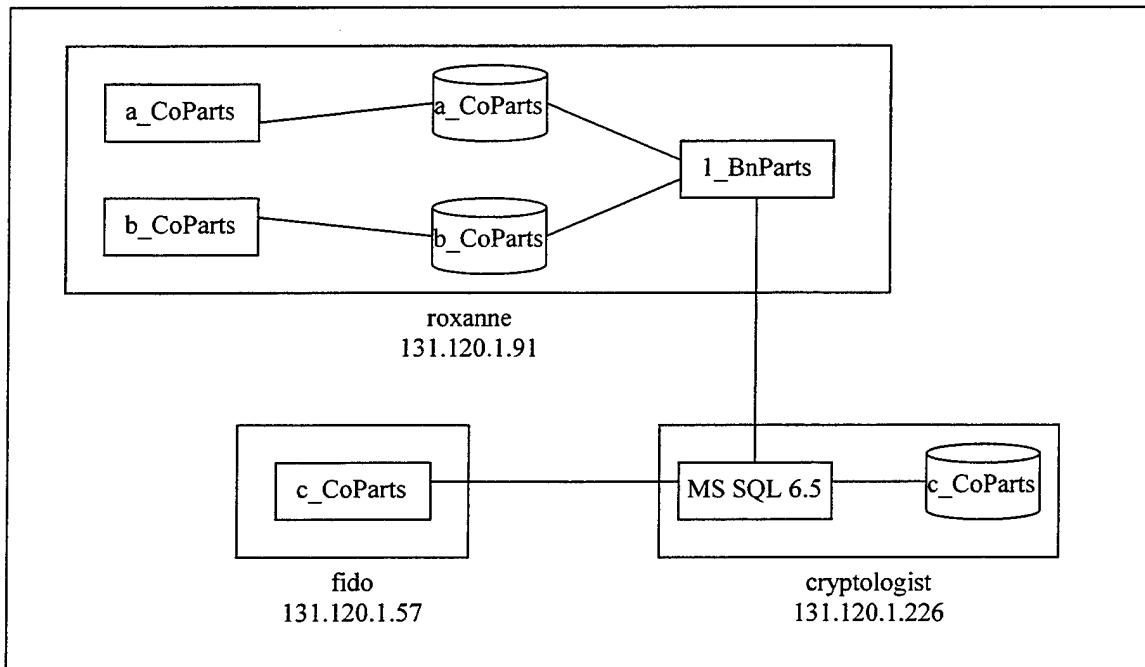


Figure 13: Socket Evaluation Environment

In our model each database server has a different method of handling client requests based on what level of authority it has. For example some database servers have search only capabilities and these can only return the result sets of database queries. Other database servers may have the capability of updating the database and others may have more detailed database management capabilities. You can picture this as a chain of

command with each individual level in the chain having different responsibilities and based on those responsibilities having different functionality.

Database servers may form a multicast group that handles client requests. Clients can ask database servers to join or leave a particular multicast group by sending “join” or “leave” messages. Clients can also communicate directly with database servers through datagram sockets.

The most significant advantage this model is giving clients access to heterogeneous databases in a heterogeneous environment. By doing so, each client can easily find what it is looking for by sending a multicast request to the group. So for example, if a company was looking for a certain part, it would check it's local database. If the part did not exist, it would send a multicast request to the group to find if another company has the required part. If no company has the required part, the request gets forwarded to the Battalion object which will attempt to find the part or ultimately order it.

1. Communication Protocol

This model allows client applications to access multiple databases and manipulate (query, update, insert etc.) them by multicasting or communicating with them directly in a point-to-point manner. So each database object has a private listening port, used for point-point communication and a multicast port used for group communication.

For this model we created a communications protocol that client applications must use in order to establish communication with servers. This communications protocol is a message format that must be followed in order to send a multicast or a point-to-point message to any database server.

A message contains three parts as seen in Figure 14. The first two parts constitute the message header. And the third part contains the actual message.

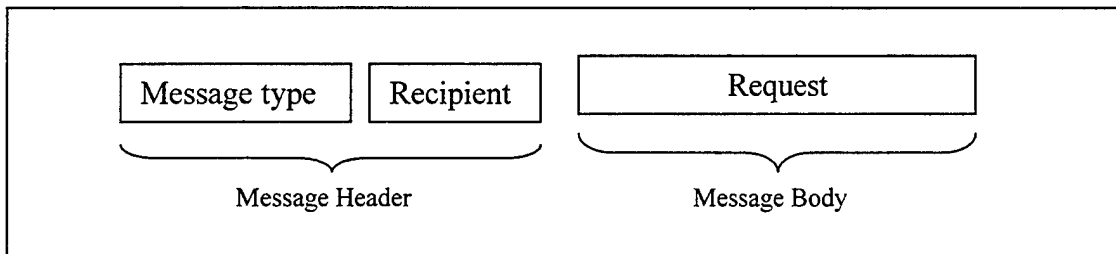


Figure 14: Message Format

A detailed explanation of the communication protocol used in the socket implementation will highlight the level of detail required to effectively communicate between two processes. Figure 15 depicts the simple communication protocol used in the model. To implement the details, various logic modules were created to include db, net, and respond modules. Each logic module is an object that can provide a requested service. The first token of the message (db, net, response) indicated which logic module to instantiate to service the message. So as the message is received by a server, it parses the message. Based upon the first token, a logic module is instantiated and the remainder of the message is passed to the logic module.

Since a message can be directed to an individual server, or group of servers, the message recipient looks at the next token. The second token is used to differentiate a request for a specific server from a request for a group of servers. If the message is a multicast message, then the "all" key word is used, otherwise the second token must be the name of the database server to make a point-to-point communication. The following examples demonstrate the protocol:

“db all <request>” is an example of a database related multicast message.

“net a_CoParts <request>” is an example of a communication related point-to-point message. The recipient of this message is the Alfa Company (a_CoParts) database server.

“response db <request> “ is an example of a response message to the client after a database query. So the client knows it has received a result set that needs to be displayed.

By parsing the message and implementing various logic modules, functionality could be added to the objects. For example all database objects (a_CoParts, b_CoParts, and c_CoParts) used the same class files. So the application code was reused by multiple objects. When an instance was created, the constructor accepted a name, such as "a_CoParts" which would be used to differentiate the object. This was effective in handling general networking logic, and client display logic, but was not effective for the database logic module.

So each database object, would have it's own specific database logic module. This module would be used to set a connection to the database, and manipulate the database. If a company was given more database manipulation capabilities, such as being authorized to order parts, then it would be given the appropriate database logic module.

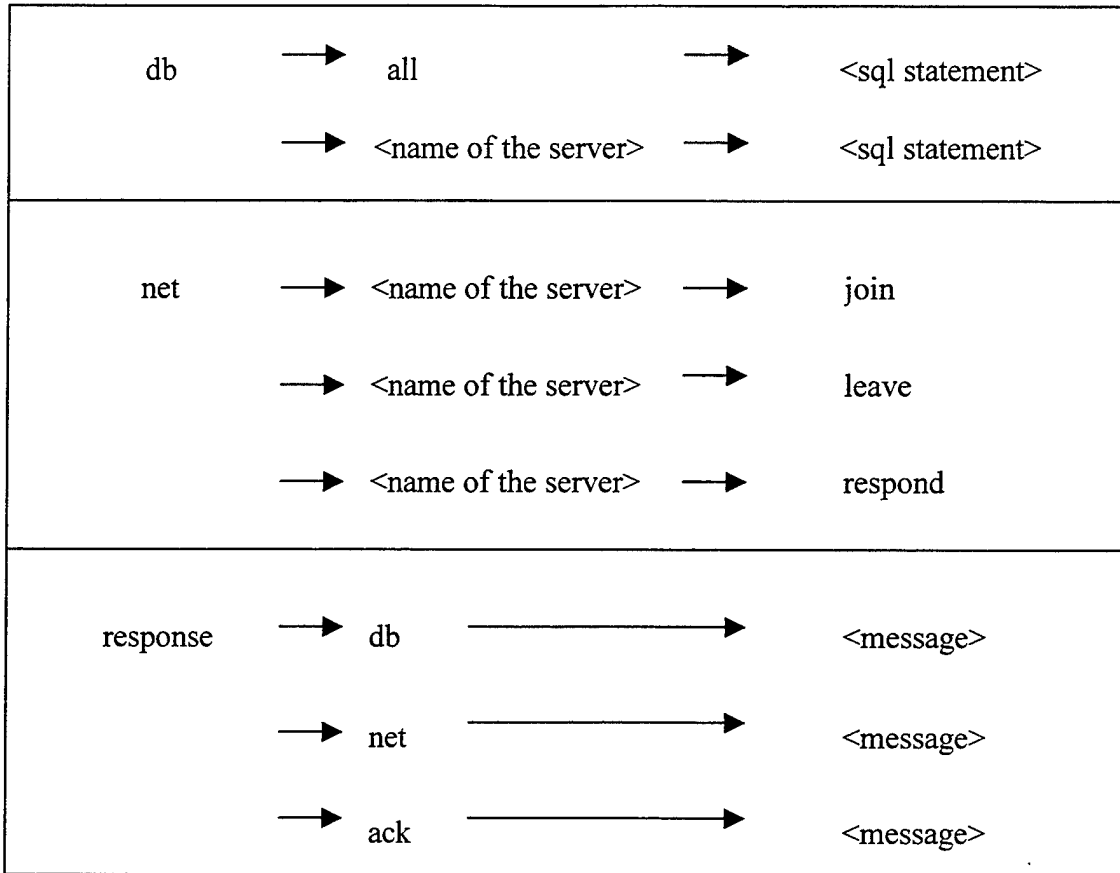


Figure 15: Message Format

The message body of a database related messages must be SQL statements specific to the databases maintained by database servers. For example:

“ db all SELECT part FROM PartsTable WHERE part = ‘tire-5443’ ”

could be a multicast message looking for a specific tire from a database server group. Any database server that finds the tire in its database responds to the client. If the server does not have the part, a negative acknowledgement is sent.

The body of multicast communication related messages should be either a “join” or a “leave” key word. A “join” message is a request to a database server to enter a multicast group and remain a part of that group until a “leave” message is sent. For example, “net aCoParts join ” instructs the Alfa Company database server to join the multicast group. The network logic module implemented by "a_CoParts" would start a java thread. The thread would be a group multicast listening thread, that would listen to

multicast communications, and spawn a message handler thread in response to any group messages received.

A Java StringTokenizer object is used to parse these messages with a delimiter of white space. Tokens are evaluated by middleware servers and an appropriate logic module is called to handle the requests. The following section explains how these messages are sent between processes and how multicast and datagram sockets are used effectively in this environment.

2. Model Implementation

To effectively communicate an established protocol must be followed by the sending and receiving processes. The figure 16 depicts a client/server socket model used to make database requests. The client sends a datagram packet containing the following message:

```
db a_CoParts select tire from parts where partid = 5433
```

The database middleware server receives the datagram packet and spawns a messageHandlerThread to service the request. The messageHandlerThread parses the message into tokens. If the first token is db, the handler knows it is a database request and will instantiate a database object to further process the message. The database object extracts the next token, which specifies the database that is to process the SQL request. In this case the second token is a_CoParts, so the server knows the request is specifically for A Company, and generates an a_CoPartsLogic object.

The third token is the actual SQL statement to be processed by a_CoParts. The database logic object uses the JDBC API to communicate with the remote or local database management system. In our model the JDBC-ODBC bridge is used to translate the Java calls into ODBC calls, which invoke the database specific ODBC driver. The request is then translated from ODBC to a vendor specific protocol and sent across the network to the DBMS Driver manager. The driver manager executes the SQL statement and returns a result set, if applicable.

If a result set is generated, it is the responsibility of the a_CoPartsLogic object to create a socket and datagram packet, and send the results to the client process. The result set must be packaged using the agreed upon delimiters so the client can parse and utilize the response.

The client process follows the same communication protocol as the database server, and generates a `messageHandlerThread` to process the message. The `messageHandlerThread` removes the first token, which identifies the message as a net logic response. The net logic further processes the request and realizes it is database response whose result set is then displayed on the screen.

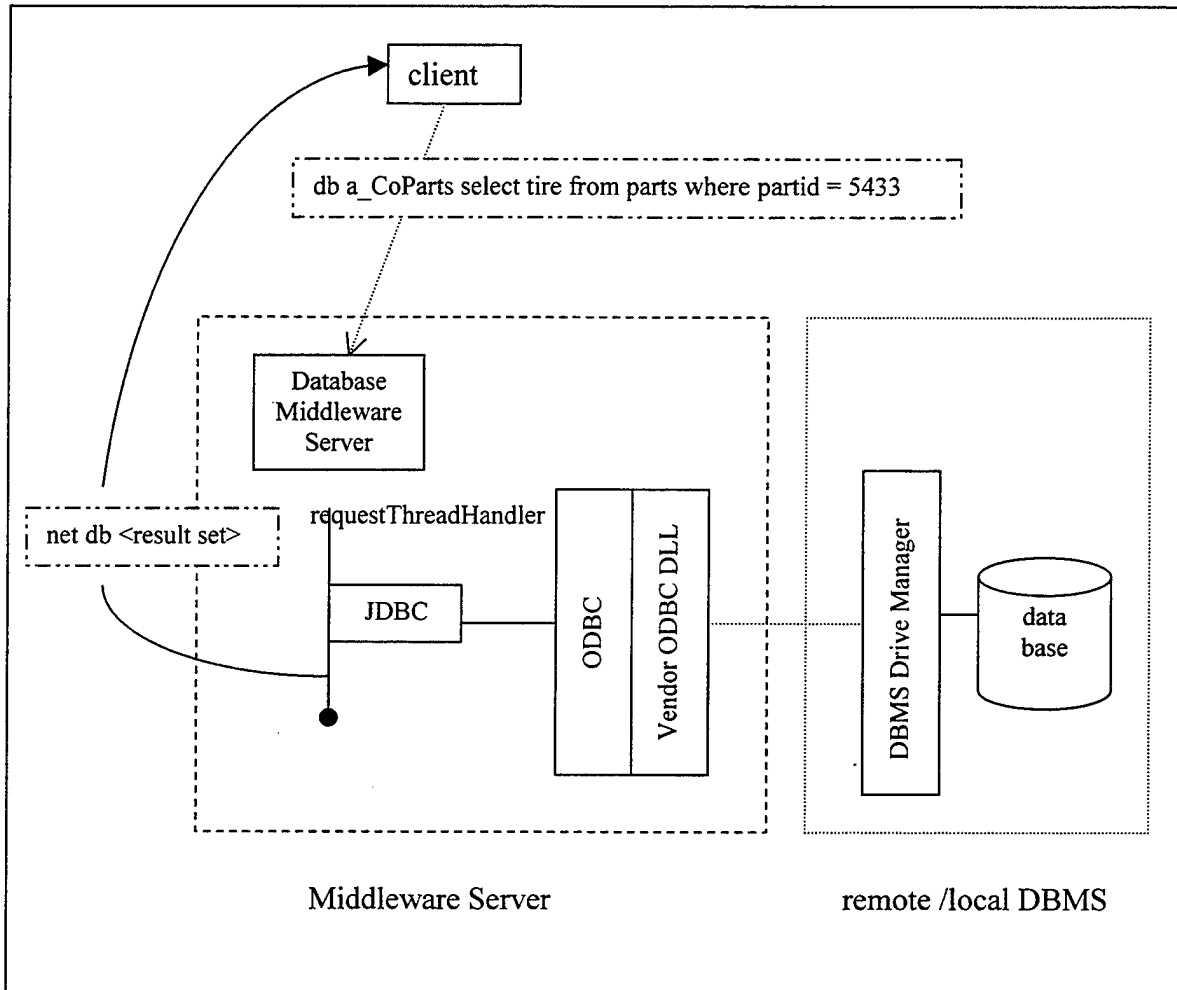


Figure 16: Database Datagram Socket Server

To increase the functionality of datagram sockets, database servers can join groups to provide common sets of services. The model we implemented portrayed an Infantry Battalion, consisting of companies. Each company maintains a repair parts database, consisting of a standard prescribed load list (PLL). The PLL consists of communications, weapons, and vehicle parts the unit is authorized to maintain for routine maintenance. The database should accurately reflect what spare parts the organization has on hand. A unit is authorized certain types and quantities of parts to be maintained, based

upon the units' mission. The quantity on hand cannot fall below or exceed a specified level.

Company PLL clerks can query their database for parts. As parts are consumed, an update is generated and the quantity is decreased. If a part is received, then the part quantity is increased. PLL clerks cannot delete parts from the parts table, or insert new parts into the table. For example, a Company PLL clerk cannot arbitrarily add part X to the database, or decide to stop stocking part Y. To change the prescribed load list the unit must make a request to Battalion. Units are authorized to laterally transfer parts, from one company to another to prevent systems from becoming non-mission capable.

In the hierarchy, a Battalion oversees Companies and also maintains its own prescribed load list. The organization is responsible for monitoring the maintenance programs of its subordinate units. In our model, Battalion is authorized to coordinate with peer Battalions. Battalion is also authorized to add and delete items from the Company's PLL based upon unit mission. To assist company clerks in locating parts, all company database servers are part of a Battalion multicast group. Each Battalion is part of a Battalion level multicast group.

In the following example the A Company PLL clerk cannot find a specific tire, with a part id of 5433, so the clerk generates a request to all members of the group. Each database server will respond to the requestor if it can provide the part. The organizations can then coordinate to transfer the part to the requester. If the part cannot be found within the Battalion, a request is generated to the Battalion PLL clerk who multicasts a request to peer Battalion parts databases. If the part exists, through the hierarchy of multicast groups it will be found or ultimately ordered.

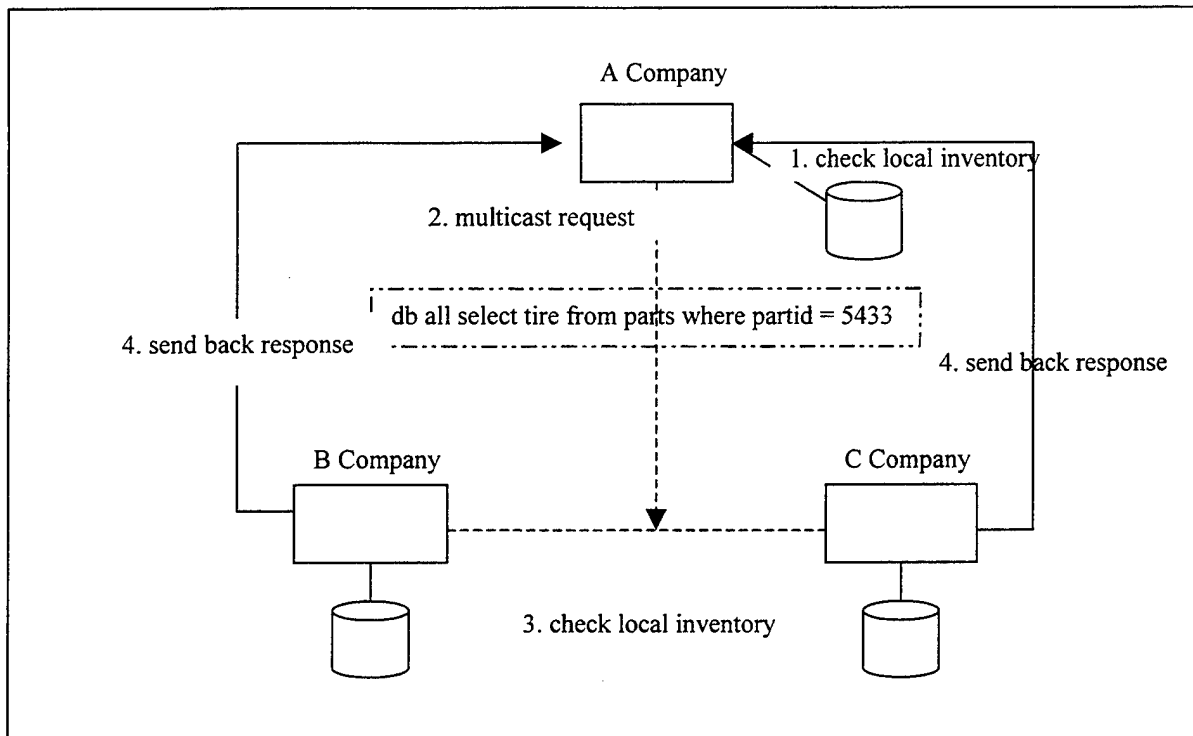


Figure 17: Multicast Parts Request

The example above depicted the client process as being a database server group member, but the client could be a stand alone client application. For example, if a Company Commander wanted a parts inventory report, the application would send a SQL query statement to the database to collect information. Depending upon where in the hierarchy the commander was, he would dictate how many organization's PLL he would be able to view.

By employing a middleware server the Commander may access a number of distributed heterogeneous databases. In our model, one database was a MS SQL 6.5 database and the remaining databases were MS Access databases.

By parsing the message, the specific business logic can reside at various levels. For example, if a_CoParts inventory of widgets drops below a certain level, logic residing on the server may trigger a message or report to be sent to the parts manager.

D. CONCLUSION

Communication between computers ultimately goes through sockets. Regardless what types of messages are being passed, sockets will be required to make the communication happen. Implementing socket-based communication requires a

substantial amount of low-level programming. When it comes to database connectivity and database manipulation paradigm, writing all that low-level code becomes tedious.

Commercial middleware solutions to the problem of database connectivity and database manipulation contain these low-level routines. But socket implementation allows developers to freely customize their applications depending on the users and/or administrators' need. The overall performance of the application can also be increased by this customization.

One other thing that we experienced while implementing the model explained in section C was the power of using Java interfaces. Interfaces allowed us to separate the implementation logic from the GUI implementation. Through the use of Java interfaces, the logic modules were able to invoke methods declared in the interface. GUI objects such as frames, or dialog boxes would implement the interface. Parent threads and objects passed themselves via the "this" operator would allow subordinate threads and objects the ability to invoke methods defined in the interface and implemented by the GUI. This allowed those objects to display result sets, connection information and various messages passed and received.

V. REMOTE METHOD INVOCATION AND JDBC

A. INTRODUCTION

The low level details of socket programming hinders a designers ability to focus on interface development or efficient database access and results in a longer development cycle. An alternative to sockets is Remote Method Invocation (RMI), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to a remote object to process the call.

Remote Method Invocation (RMI) is a powerful distributed computing technology that allows a designer to develop a networked application without having to worry about the low-level networking details. It is a Java distributed object solution that allows a Java client the ability to access a Java server, and invoke methods that the server makes available in its interface. RMI encapsulates the underlying mechanisms for transporting method arguments and return values across a network.

Coupled with JDBC, RMI can be used to create a middle tier heterogeneous database server. A Java client will be able to invoke methods on a remote Java server. The server can employ JDBC to communicate with various relational database management systems. This implementation can reduce the system administrators burden of managing database access and client application configuration. All database management system drivers will reside on the server, resulting in zero client configuration. The client will not have direct access to the database, and will only be able to use the methods specified in the interface. Business Logic can also be stored on the server, so when a user attempts to access a database, the most recent organizational policies will be enforced. This chapter will provide an explanation of RMI and discuss a model that was implemented using RMI and JDBC to provide heterogeneous database access.

RMI is a distributed object model. According to Coulouris "A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system software enables computers to coordinate their activities and to share the resources of the system – hardware, software and data"[Ref. 4].

A distributed system can be visualized as a collection of server processes and their client processes. These processes do not necessarily reside on a same computer. In order for a client process to be served, it needs to find the server that offers the desired service, then send its request and wait for a reply. The server process may reside on the same computer as the client or on a remote system. A server may employ the services provided by another server, so it may also be considered a client process. Each service program has an interface defining the services it provides. These services may be invoked by clients. Clients invoke service operations by sending request messages to the servers, which perform the requested operation and send a reply back to the client.

The Remote Procedure Calling mechanism integrates this client-server arrangement with a conventional local procedure call of a standard object oriented programming language. Since, the server and client processes may reside on different machines communication between them must be handled. The parameters being passed to the remote method and its return types must be packaged for transmission across the network. Data structures must be flattened before transmission and rebuilt upon arrival on the other side. To flatten a data structure implies taking a complex object and breaking it down into a stream of bytes on the client side, then transmitting the data to the server object which will reconstruct the complex object and forward it up to the server object.

B. REMOTE METHOD INVOCATION (RMI)

RMI is Sun's version of RPC and translates well into distributed object systems, where communication between program level objects residing in different address spaces is needed. The following highlights key features of RMI [Ref. 16].

- *Object Oriented:* RMI can pass full objects as arguments and return values. The objects must be serializable, such as Java primitives or objects that implement the `java.io.Serializable` interface. For example, Java Vectors and Hashtables implement `Serializable`, so these complex objects can be passed to or from a client using RMI, with no additional processing. With RPC the user would have to decompose the object, send it and recompose it on the client. RMI encapsulates this process for the programmer in the form of client stubs and server skeletons that are responsible to marshal and unmarshal the data.
- *Implementation Changes:* RMI can encapsulate the class implementation details from a client. Clients only have access to the interface, the

implementation resides on the server. So as business logic, or implementation details change, they only need to be changed on the server. As long as the interface signature does not change, the implementation details are totally encapsulated.

- *Rapid Development:* RMI makes it easy to write servers for a full scale distributed object system. A server object implements the interface, declares a security manager, and binds the instantiated object to the rmi registry. Writing reliable distributed applications is simple.
- *General Goals:* RMI supports seamless remote invocation of objects in different virtual machines, including call backs from applets. Remote Method Invocation (RMI) was designed to seamlessly support remote method invocations on objects across Java Virtual Machines. Because RMI centers around Java, it brings the power the of Java safety and portability to distributed computing.
- *Garbage Collection:* RMI uses a reference counting garbage collection scheme. It keeps track of the number of references to an object.

One of the limitations of Java RMI is that Java must be run on both ends of the network connection. A Java client can only communicate with a Java server. But the Java server can communicate with non-Java applications such as a database management system.

The following section will provide a more detailed explanation of the RMI system architecture, and how RMI encapsulates interprocess communication.

1. RMI System Architecture

Figure 18 highlights the three layers of the RMI system: the stub/skeleton layer, the reference layer, and the transport layer. An independent interface and protocol is defined for each layer allowing alternate implementations to be replaced without affecting other layers of the system. A remote method invocation from a client to a remote server object travels down the client stack, then up through the server stack, where the method gets invoked and the results are passed down the server stack through the network to the client and up the client stack.

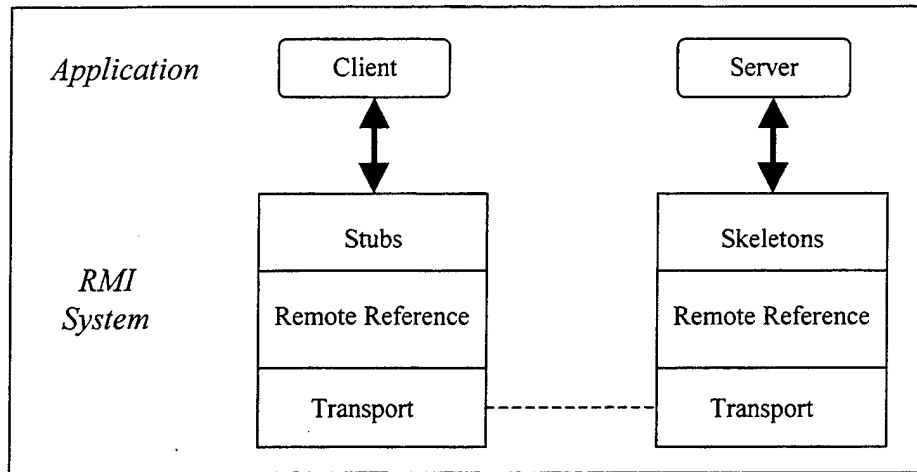


Figure 18: RMI Architecture [Ref. 15]

a. Stub/Skeleton Layer

This layer is the interface between the application (client and server applications) layer and the remote reference layer of the RMI system. Stubs and skeletons are used by the client and server processes respectively to communicate. Sun's `rmic` tool (rmi compiler) is used to generate a stub and skeleton class file from the object implementation class file. A stub/skeleton is generated for every remote implementation object that is made available to client applications via an interface.

When the client application makes a remote method call, the parameters and call are passed to the stub layer. The stub layer is responsible for marshalling (flattening) the call. It then passes the call down the RMI protocol stack, which ultimately transmits the call across the network to the remote server. The call moves up the server stack to the skeleton layer. The skeleton layer is responsible for unmarshalling the call. The skeleton reformats the method call, then invokes the appropriate method of the instantiated implementation object. The method is executed and its return values are passed back through the skeleton, where they are marshalled and returned to the client. The process is depicted in Figure 19.

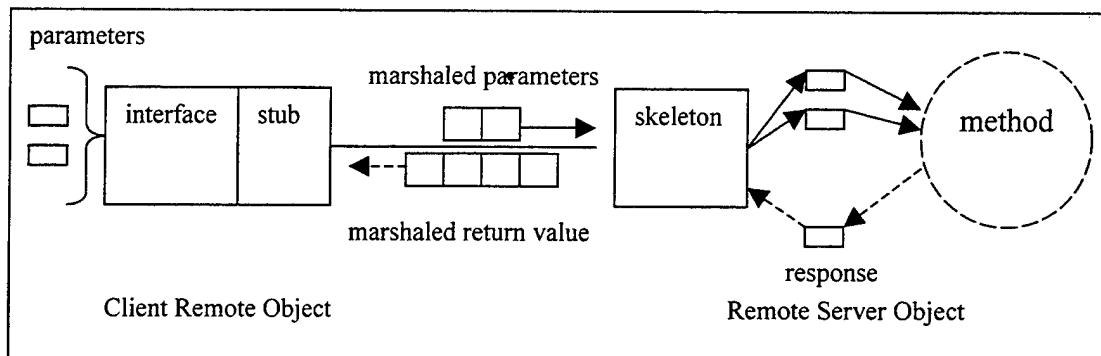


Figure 19: Marshalling

b. Remote Reference Layer

The remote reference layer deals with the lower-level transport interface. This layer is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communications with multiple locations. The remote reference layer transmits data to the transport layer via an abstraction of a stream-oriented connection. When a server is exported its reference type is defined. If the server is a `UnicastRemoteObject`, then it is a point to point unreplicated server. This was designed to allow for modification and future expansion. For example, a `MulticastRemoteObject`, would have reference semantics that allowed for a replicated service.

c. Transport Layer

The transport layer provides the communication implementation between the client process and server process using TCP-based Java Sockets. It listens for incoming requests, establishes a connection (channel) to process the request, and manages the connection. A channel is a conduit between two address spaces, the transport layer manages the connection between the local and remote address spaces.

When the client first establishes a remote reference to an object, the client must find that object on the network. To do so, the client performs a lookup. On the server an `rmi registry` process runs and continuously listens for messages on a

predetermined port. The clients lookup request moves down the RMI stack, the transport layer gets a port from the local operating system and sends the request to the remote registry process. The registry consults a table to confirm that the object exists, if it does, a communication channel (TCP/IP stream socket) is established between the client and the remote object.

d. Application Layer

The application layer sits on top of the RMI system. The client and server applications interface with the stubs and skeletons respectively. The server application is responsible for instantiating implementation objects. Once the object is instantiated it is bound to the registry making the object available to client applications.

The client application is responsible for performing the lookup, to find the remote object and establish a communication channel. Once the channel has been established the client can invoke remote methods just as if the object resided locally. The only difference is that each method call must catch a `RemoteException`. RMI is synchronous, so the calling process will block until the remote method completes.

2. RMI Development Process

An RMI object is a remote Java object whose methods can be invoked from another Java Virtual Machine across a network. The following section outlines the steps in using RMI for distributed Java computing. Figure 20 depicts the process graphically, demonstrating the key steps.

a. Agree Upon the Interface

The first step in employing RMI is to agree upon the interface of the remote object. The object interface declares what methods are visible to the client, and what services the server must provide. The interface is the contract between the client and the server. The client must understand what parameters and return types each method requires so they can be handled properly. RMI allows complex objects to be passed back and forth between the client and server process. For example, if the method returns a vector of hashtables containing various primitive types, the client needs to know this. The object that implements the interface must extend `java.rmi.Remote` and each method which is to be visible to clients must throw a `RemoteException`.

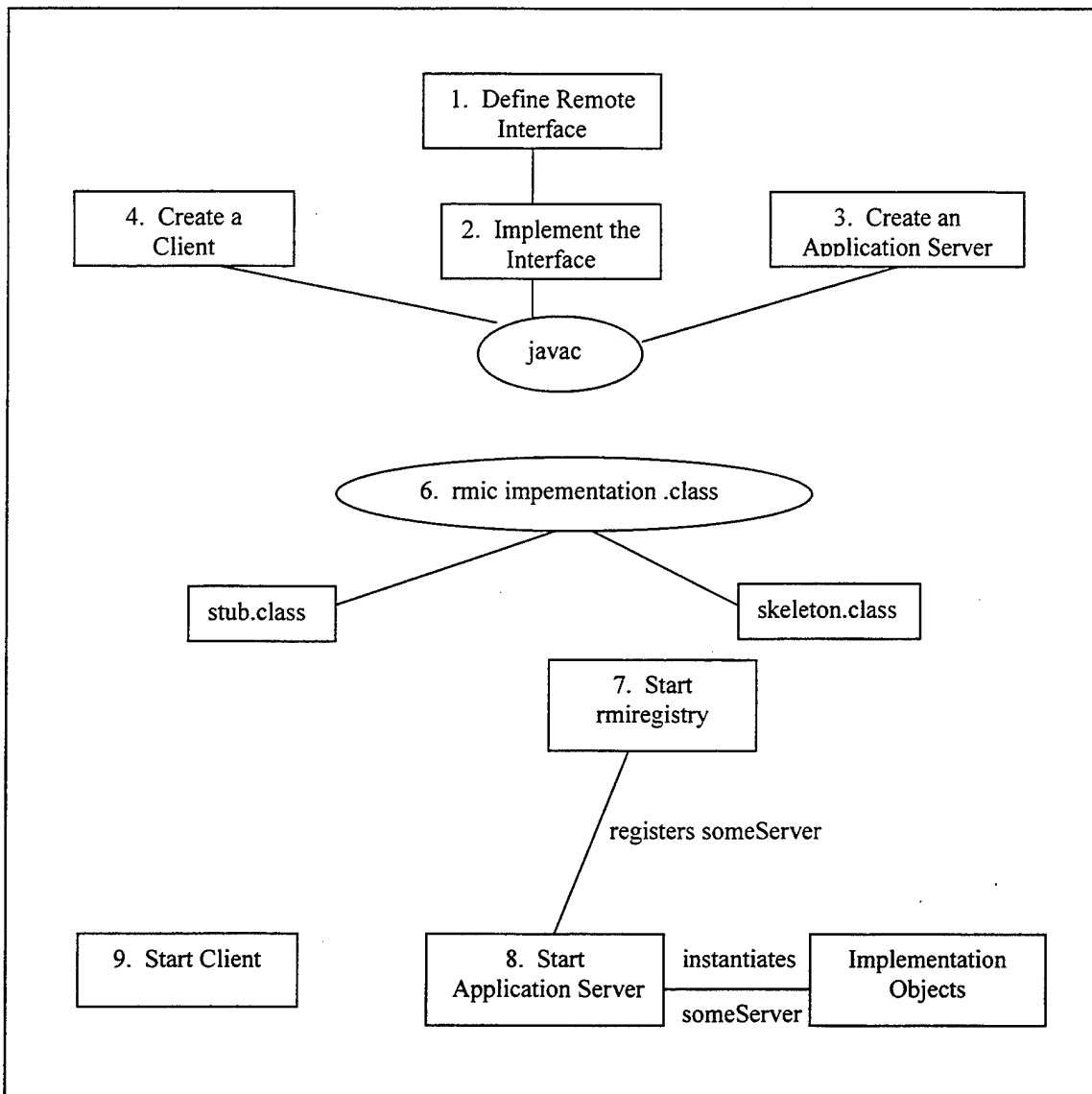


Figure 20: RMI Design Process

a. Implement the Interface

The next step is to implement the interface. The object that implements the interface must extend `java.rmi.UnicastRemoteObject`. Each method that is declared in the interface must be implemented by this object and must throw a `RemoteException`. The implementation can be simple or complex. For example, the implementation may use JDBC to establish a database connection, to a Microsoft SQL 6.5 server, which resides on another remote machine. The implementation details are hidden from the client. The only methods the client have access to are those specified in the interface.

The implementation can be modified or changed, as long as the signature does not change. This can be effective in storing business policies or logic on the server. The client makes a policy request, and will always get the current policy. If the policy changes, the changes are made on the server, and all future users will get the updated version. In the case of database access, if the back end database manage system changes from a legacy implementation to a newer implementation, the changes remain transparent to the client.

RMI provides a convenient means of partitioning a distributed system design between the logic and the application design. As end users continue to rely on graphical user interfaces, the application designers job is becoming more challenging. RMI allows the logic, or back end designers and application designers the ability to focus on their respective areas of responsibility. In most cases, the client, using the interface and it's declared methods will be the application designer.

b. Object Server

The object server is a file that contains a main function. The process instantiates and binds implementation objects. When the object server binds an instantiated object, the name is registered with the registry service. For example:

```
Naming.rebind("navyServer", new navydbImpl());
```

This call instantiates a navydbImpl() object which implements navydbInt, an interface. The function call Naming.rebind comes from the RMI Naming class. This call is used to obtain the services of the rmi registry, to register the new object with the name "navyServer". Rebind updates the mapping between the name, navyServer and the remote object instantiated by new. The registry will discard any previous binding to the name.

One object server can instantiate and bind as many different objects as the system designer wishes. The application server may also be responsible for logging all requests that were received and processed by remote objects.

c. Java Client Application

The fourth step is to create the Java client application. The application will usually consist of the GUI for the end user. To utilize the services provided by a remote object, the client must establish a reference to the desired remote object. A URL based

naming scheme is used to perform a lookup for the instantiated implementation object. The request gets passed down the rmi stack to the transport layer, which makes a TCP/IP connection to the remote object. The server machine must have an rmi registry active, which consists of a listening thread. Once a connection is established the rmi registry performs a lookup in its local table for an object entry name that matches the name provided by the client. For example:

```
navydbInt server =  
(navydbInt)Naming.lookup("rmi://131.120.1.91/navyServer");
```

The lookup method returns a remote object for the URL name: "rmi://131.120.1.91/navyServer". The call returns a reference object to the client, which is type cast into a navyInt object. Now the client can invoke the methods specified in the interface by using `server.<methodName>`.

d. Run the System

In order to run the system, the rmi registry must be started on the server machine. The rmi registry is a utility program provided by Sun, that listens for rmi requests. The registry registers rmi objects that have been instantiated and made available on that machine. Each server process can support its own registry, or one registry can support all the virtual machines on the server node, such as an application server.

Once the rmi registry is active the object server can be started. As stated before, the object server, instantiates and binds implementation objects to the registry. For example if the server instantiates a navydbImp object and registers it with the registry under the name of navyServer, that object is now available to service requests.

The final step is to start the client process. The client process can reside on the same, or different machine as the server. As stated before, in order for the client to interface with a remote object, it must instantiate the interface object and performs a URL lookup. It must reference the remote object by the name, to which it was bound on the server side registry.

C. RMI JDBC MODEL

To demonstrate the capabilities of using RMI with JDBC we developed an object server that provided relational database access to a MS Access database, and a MS SQL

6.5 database. As depicted in Figure 21, all database drivers were stored on the object server. The Microsoft Access database resided on the same machine as the object server, and the Microsoft SQL 6.5 database resided on a remote system.

The accounts database represented a financial database, while the navy database represented an organizational database. The object server instantiated an implementation of both the navy and the accounts database interfaces. The instantiations were bound to the rmi registry which made the interface implementation methods available for use by client applications.

The client GUI was designed using Symantec Visual Café and provided various views of the database, and various database manipulation methods via the remote implementation objects.

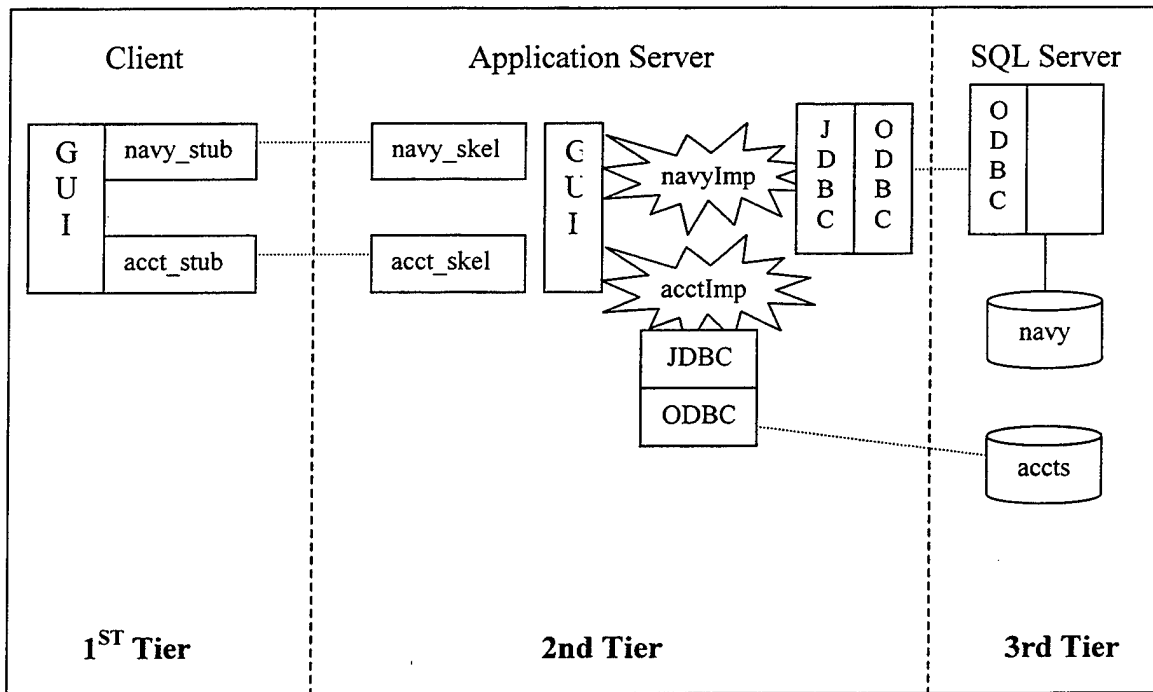


Figure 21: RMI Implementation

To demonstrate the potential of RMI coupled with JDBC the remainder of this section will explain the details of the system. The explanation will attempt to follow the design steps as outlined in the preceding RMI Development Process above.

1. Interface

An interface to each database was designed to provide a module design, and was called `acctsdBInt` and `navydbInt`. One interface could have been used to access multiple

databases. The data base methods made available to the client consisted of specific methods for the basic or naïve user and general, powerful methods for super users. The client was the application designer who was designing the front end interface. In creating the system the designers must agree on where the business logic will reside. In our model the logic was present on both the client and the server.

For example, do the system designers want to give the application designer complete access to the database, to create tables, delete tables, and insert tuples. These are powerful options that can be made available in the interface, yet may only be needed by database or system administrators. Or does the system designer want to provide limited database access and manipulation, providing only high level methods.

The following functions allow the user to directly manipulate the database which may be appropriate for a database or system administrator. In order to use these methods the user must have a general understanding of SQL and the structure of the databases. These methods were used by the GUI designer. For example, the client may enter a string in a text box and click on the submit button, which would call the executeSQLGetString remote method.

```
//for database administrator GUI
public abstract String executeSQLGetString(String
                                           sql)throws RemoteException;
public abstract Vector getTableName()throws
                       RemoteException;
public abstract Vector getTableMetaData(String
                                           sql)throws RemoteException;
```

The first method allows a user to issue a basic SQL statement (CREATE a table, Delete a table or tuples, Insert data, or execute a Select operation). The two supporting functions are used to provide meta data about the databases to assist the user in writing a SQL statement. This type of interface may be appropriate for a database administrator, but most likely not for a basic user. It provides a dynamic means to view the contents of a database and its structure, without having any prior knowledge about the database.

To demonstrate restricted database access the following methods were provided for the basic user. The user can add a new employee or view the current employees from the accounts database.

```
public abstract String viewEmployees() throws
                       RemoteException;
```



```
public abstract void insertEmployee(String name,  
                                     boolean faculty) throws  
                                     RemoteException;
```

This restricts database access and manipulation to only the methods specified in the interface. Various views and database manipulation methods can be specified in various remote objects such as: a database administrator object, a clerk object, or a manager object. Each object would encapsulate the business logic, and views the client will have access to and can be easily modified on the server to reflect changing rules. The end user does not enter any SQL statements, but is limited to executing the methods provided. This provides limited database views and manipulation.

RMI allows a means to pass behavior from the server to the client. This allows the client machine to evaluate and implement policy or business logic. The following function allows a client to get the current database access policy from the server. It returns an `accessPolicy_Int` object. The implementation object implements `serializable`, so it can be transmitted via RMI.

```
//returns an object that implements the accessPolicy  
public abstract accessPolicy_Int getAccessPolicy()  
    throws RemoteException;  
public abstract void addUser(String uid, String pass)  
    throws RemoteException;
```

The policy object contains various methods including one called `getAccessRights`. To demonstrate how a policy can be maintained and modified on the server, yet implemented or enforced on the client a simple example was developed. To provide database access, the administrator enters the users name, password, and access code. The information is stored on the remote server, via the `addUser` method declared in the interface. The access code is used to determine what views and database manipulation functions are appropriate for the user, based upon the current policy.

When a user logs onto a system, the current policy object is downloaded from the server. The object contains a function called `getAccessRights`, which takes the users name and password, performs a lookup to get the users access code. The access code is compared to the current policy to determine what view the user is authorized. Based upon this logic, the GUI frame that appears may be a super user frame, providing full database access or manipulation, or a simple user frame, providing limited access.

If the policy changes, and all personal that have an access code of X are authorized a different view, the policy is changed on the server, and the next time that users logs in he or she will get the new view. This implementation can be extended to provide more computationally complex data processing or error checking on the client machine to reduce network traffic.

2. Interface Implementation

The implementation details are completely hidden from the client. This allows the implementation to be modified, provided the method signatures do not change. In our model, the implementation utilizes JDBC to provide the database access and manipulation.

3. Object Server

The object server consists of an active process which instantiates various remote objects and registers them with the registry. In the model, the object server instantiated and bound two remote objects as depicted in the following code:

```
public static void main(String args[])
{
try{
    System.setSecurityManager(new RMISecurityManager());
    Naming.rebind("acctsdbServer", new acctsdbImpl());
    Naming.rebind("navydbServer", new navydbImpl());
}catch(RemoteException e){
    System.out.println("Remote Exception " + e);
}
} //end main
```

The names used by the server, acctsdbServer and navydbServer are the names that get registered by the registry. When a client performs a lookup, it must specify these names in order to create a communication channel.

4. Client Application

The client application is the graphical user interface designed for the end user. To the end user, how or where the data is stored is irrelevant. The end user is only

concerned that the application provides the services he or she desires. It is the responsibility of the interface designer to meet the end users requirements.

The following briefly discusses some of the key methods used in the client application by the application or interface designers. When the application is initialized, it establishes a security manager and performs a lookup to connect to an instantiated object, for example with the name of "navydbServer".

```
//Navy Data Base Object  
navydbServer =  
(navydbInt)Naming.lookup("rmi://131.120.1.91/navydbServer")
```

If the lookup is successful, then a communication channel is established and managed by the transport layer of the RMI stack. The client now has established a reference to the remote object, navydbServer. If the user enters his password and is granted super user access, then when the GUI frame is created the remote method getNames() is invoked.

When the client invokes navydbServer.getNames() a vector containing the table names of the database is returned. The names are displayed in a drop down menu for the client to use while writing SQL Statements. The database metadata function, used by the server implementation object, provides a dynamic view of the current state of the database. If new tables are added to the database, the next time an authorized super user frame is generated, the new table will be displayed. The remote object that implements this function only needs to know how to establish a connection with the database. This implementation separates the backend database implementation from the front end GUI implementation.

To provide the user with additional database metadata, the client may select a table name from the drop down menu to invoke the navydbServer.getTableMetaData(String sql) method. The SQL statement that gets passed to the function is "select * from <selected table>". The server side implementation of the function creates a JDBC statement object, executes the query, and gets the ResultSet meta data. Based upon the metadata, the column names are extracted, placed in a vector and returned to the remote client. The implementation code is as follows (a valid connection has already been established to the database):

```
Statement stmt = con.createStatement();  
ResultSet results = stmt.executeQuery(sql);
```

```

ResultSetMetaData  rsmd = results.getMetaData();
Vector resultVector = new Vector();
int cols = rsmd.getColumnCount();

for(int ix = 1; ix < cols; ix++)
{
    String colName = rsmd.getColumnName(ix);
    resultVector.addElement(colName);
} //end for
stmt.close();
return resultVector;

```

The client application displays the column names in a drop down to assist the end user in generating a SQL Statement. The column display width and the column type could also be returned in the Vector.

The user can then enter a SQL Statement, using the table meta data provided to assist in writing the statement. All super user SQL statements are processed using the executeSQLGetString method, which processes all basic SQL statements. This provides the super user the ability to create a new table, delete a table or execute a select statement. If a result set is returned, the remote object, places the contents of the result set in a string, inserting a delimiter between tuples and returns the string to the client. The client application can parse the string and display the results.

D. CONCLUSION

Remote Method Invocation provides a powerful means of creating a distributed database aware system. Some of the strengths of RMI are:

- *SetUp*: All database client drivers and configuration remains on a central server node. This simplifies administration, requiring no database drivers to reside on client machines.
- *Scalability*: The database being used can be replaced transparently to the client, provided the agreed upon interfaces do not change. The physical location of the database can be moved, and only the implementation details need to be modified, again, transparently to the client.

- *Logic:* All business logic or policies can reside on the server, and be downloaded and enforced on the client. This ensures all end users are complying with the most recent policies.
- *Communication:* The RMI model encapsulates low level communication requirements. System designers are not concerned with socket programming, and communication protocols. They only need to agree upon the interface. Methods are used to communicate between a client and a server.
- *Modularity:* By employing RMI the division of system design is simplified. Once the interface is agreed upon, the application designer can begin implementing the GUI. The database administrator manages the database. The business logic, regardless of how complex, can be designed and stored on the server. Implementation details can change at all levels of system design, provided the agreed interfaces do not change, with minimal impact on other system components.

RMI is a Java client to Java server implementation of distributed object model. Coupled with JDBC the two packages provide an easy means of developing distributed applications.

VI. MIDDLEWARE APPROACH

Remote Method Invocation is a powerful technology which allows designers to customize their object brokers, embed logic on the server, and provide client applications with dynamic database access. The cost associated with this flexibility is the amount of programming required to provide a total solution (GUI, Logic, and Database Server), which can slow the development cycle and increase cost. The next layer of development abstraction is to use Rapid Application Development (RAD) Tools to design database aware applications.

As Java technology matures, RAD tools are appearing that assist commercial designers in creating Java solutions. Tools that assist in building a graphical user interface allow designers to focus on the total system, and not overly commit resources to GUI design. For Java RAD vendors, a key technique to differentiate a tool, and gain market share is to extend the functionality offered by the tools to include database aware components. Database aware components are Java beans capable of providing database functionality. By adding this functionality, these tools can provide a total solution to Java designers, decreasing the cost associated in providing a database aware solution.

This chapter will explore the capabilities of two Java RAD tools: Borland's JBuilder Client/Server Suite and Symantec's Visual Café. Each tool comes bundled with a proprietary Type III Net-Protocol/All Java Driver, to provide heterogeneous distributed databases access. These drivers are middleware servers, which process JDBC calls into a DBMS specific network protocol and communicate the result to the DBMS specific server. Table 2 depicts the desktop and client/server databases both middleware servers can connect to. A desktop database implies that the middleware server and the database should reside on the same computer.

DeskTop DBMS	Client/Server DBMS
Dbase	IBM DB/2
Paradox	Informix
MS Access*	Oracle
FoxPro	Sybase*
	MS SQL Server*

Table 2: MiddleWare Database Access

In attempting to use a reasonable mix of relational database access, one desktop and two client/server DBMS's were selected. Microsoft Access was chosen as the

desktop database due to its availability. Microsoft SQL 6.5 server was selected due to its increasing popularity and market share as a client/server database. Sybase SQL server was chosen as a non-Microsoft solution.

The middleware servers (Symantec dbAnywhere and Borland DataGateway) were connected to the databases as depicted in the figure 22. These middleware servers were the conduit between Java client applications and remote databases. A Java client would use the middlewares API to communicate requests. The middleware server would use JDBC to forward the request to the appropriate backend database engine.

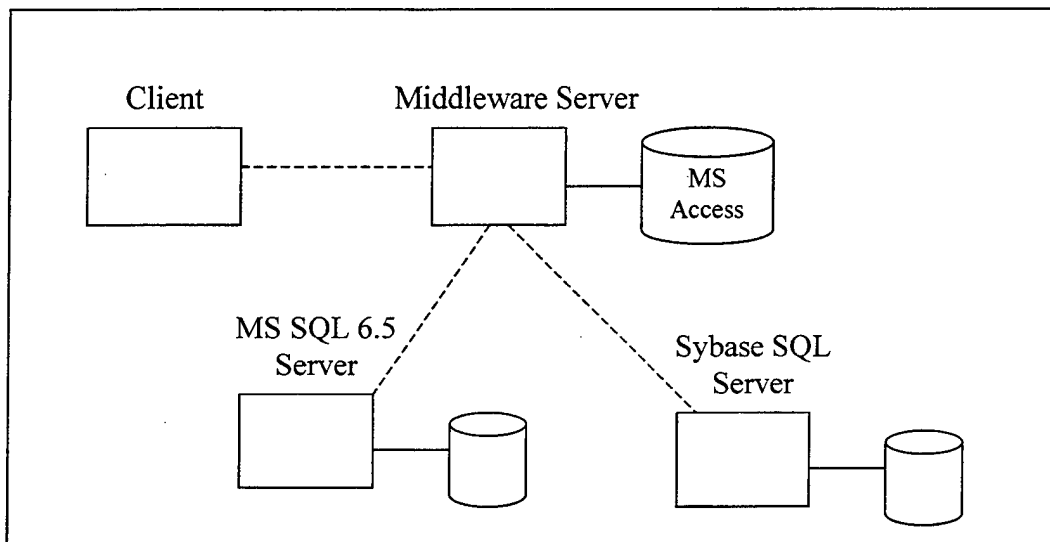


Figure 22: Commercial Tools Evaluation Environment

A. SYMANTEC VISUAL CAFÉ

Visual Café 2.0 Database Development Edition was released in August 1997 and was one of the first commercially available Java Database RAD tools. The Database Development Edition version includes a dbAnywhere Type III JDBC server, as well as database aware components and wizards integrated within the IDE to facilitate relational database interface development. To properly use the database aware components provided by Visual Café, the dbAnywhere server must be properly configured.

1. Server Configuration

The middleware server must be running to enable the database aware component functionality. To provide heterogeneous relational database access, the system administrator needs to be familiar with the client/server DBMS that the dbAnywhere

server will interact with. In our example, the dbAnywhere server was going to interact with a MS SQL6.5 database, a MS Access database and a Sybase database, so the DBMS client utilities of the DBMS must be installed on the dbAnywhere middleware server, since it becomes a client to the SQL 6.5 server or Sybase server.

a. Connecting to Microsoft SQL 6.5 Server Database

To interface with the SQL 6.5 server, the first step is to install the SQL 6.5 client utilities on the dbAnywhere server machine. The SQL 6.5 Configuration Utility is used to provide connection information to the SQL6.5 Server. For example, the network protocol used was TCP/IP. A server alias was provided (cryptologistServer) and the SQL 6.5 server socket address entered as the connection string 131.120.1.226,1433. Port 1433 is the default SQL 6.5 server listening port. dbAnywhere contains a proprietary MS SQL 6.5 driver so the final step was to use the dbAnywhere configuration utility to register the database. The specified data source URL was:

```
jdbc:dbaw://localhost:8899/SQL_SERVER/  
cryptologistServer/NSGDB
```

Where the protocol was jdbc, with a sub-protocol of dbaw, running on port number 8899 of local host, with SQL_Server engine, to access cryptologistServer, specifically the NSGDB database. Since the SQL 6.5 server utilities were used to register the alias cryptologistServer, and its associated TCP/IP socket information, the SQL_Server engine will invoke the appropriate driver, which looks to its registry to get the connection information for the specified cryptologistServer.

An ODBC alias can also be created using the ODBC Administrator by adding a SQL server data source. This is not necessary since dbAnywhere contains a proprietary driver, but was used for our RMI and socket implementation. The administrator will display the SQL server aliases registered by the SQL server client configuration utility.

b. Connecting to Sybase SQL Anywhere Server Database

This set up was similar to the Microsoft SQL 6.5 setup. The first step is to install and configure SQL Anywhere 5.0 Client Utilities on the dbAnywhere server. This installs the appropriate Sybase drivers. Once the client utilities are installed the system database administrator must use the SQL Central utility to establish a network

connection. A connection profile must be created, which specifies the user identification, password, database name, and a server alias. A database start up box is available to specify startup options.

By default the SQL Anywhere client broadcasts a request to identify a server on the network matching the server name specified, in our case heteroServer. Under most networking topologies the broadcast will not leave the sub-net, so if the server is on a different sub-net its IP address needs to be specified through command line parameters, entered in the database startup box. This was not intuitive and required a few days of research to realize we had to specify:

```
dbclient -x tcpip{HOST = 131.120.2.8}heteroServer
```

The Final step is to configure dbAnywhere. To access a SQL Anywhere 5.0 server, dbAnywhere uses the Sybase SQL Anywhere 5.X ODBC Driver. This requires using the 32 bit ODBC Administrator utility to create an alias to the database. The ODBC configuration followed the same procedure as the SQL Anywhere client configuration, including specifying the command line start up switches. Once the ODBC configuration was complete the dbAnywhere configuration utility was used to establish the connection. Even with this configuration, we could not connect to the SQL Anywhere server unless we used SQL Central to establish a connection first. Then the dbAnywhere connection was successful.

Once the dbAnywhere server is configured to connect to the desired databases, when the designer users the Visual Café database aware components, they will refer to dbAnywhere to interact with the appropriate datasource.

2. Database Aware Components

As explained above, the administrator must understand the relational DBMS's that the middleware server is going to provide access to. Once the dbAnywhere server is configured designers can develop database aware graphical applications.

Visual Café provides a database wizard that takes the designer through the process of connecting to a database, generating a query, and displaying the results neatly in a form. All code, including the query is generated by the wizard.

The wizard steps include:

- Select the dbAnywhere server
- Select a Datasource
- Choose a table
- Select columns to display
- Specify column display component, (text area, list, checkbox) and label name
- Select record operators, such as next, previous, and first

The database aware project provides no additional functionality, other than asking the user to create an applet or application. The wizard generates a form view of the selected table. The program can be compiled and executed without requiring the designer to write any code. The wizard does not allow the designer to bind the query to a grid view (matrix of rows and columns) and does not allow the designer to enter a unique query. It only generates a subset of `select * from table name`. The limited record operators allows a user to look at the next or previous record of the result set, or to enter and commit a new record. The wizard works well to provide simple form view database aware application.

Visual Café's Database connectivity is provided through `symantec.itools.db.pro` package, a Symantec extension of Sun's `java.sql` package. Designers cannot use Sun's `java.sql` package to connect to the dbAnywhere server, so therefor cannot use `java.sql` to perform any database manipulation. To display Symantic's version of a result set, the designer must use `symantec.itools.db.awt` package, an extension of `java.awt` package. So, in order to customize the database functionality provided by the database aware components the designer must understand these packages. The key objects used to provide basic database manipulation were: `Session`, `ConnectInfo`, and a `RelationView`.

A `Session` object represents a dbAnywhere Session. A connection to a dbAnywhere server. This object establishes the connection with the dbAnywhere Server via a TCP network connection and provides access to the related dbAnywhere classes.

A `ConnectInfo` object represents a specific data source name, the database the designer is connecting to via the session object. This object includes the database name, username and password.

A `RelationView` object, is used to submit and display the SQL statement. This is the component that defines and maintains the result set. The result set handles record navigation and data manipulation. A `RelationView` object gets bound to a `symantec.itools.db.awt` component, such as a text field, or list box in order to display the information.

To graphically create these objects, the designer can drag and drop them onto the designer form view or the dbNavigator can be used to automatically create them. dbNavigator provides means to navigate across dbAnywhere servers, and the databases they offer to include the column names. The objects(database, column name(s)) can be dragged to the form designer and will create the required components (Session, ConnectInfo, and RelationView).

3. Model Implementation

Generating a form view of all tables was not difficult because of the wizard. The add table utility would launch the wizard and the selected table would be appended to the existing frame object. In most cases a new frame was created to display the table. This solution was not effective, because users would be required to open a new frame to view a table, which cluttered the desktop. We tried to use panels to display the contents which was more effective, but generated an excessive amount of code.

One of the functions we implemented in the RMI model required a user to log onto a system, and a drop down menu would then be populated with data sources the user was authorized access to. To implement the same functionality with Visual Café, was challenging. A StringBuffer was used to format a SQL string, which was submitted and returned a Symantec relation view object. We were not able to bind the relation view to a symantec.itools.db.awt choice box, so the results were placed in a text field.

When the user selected the data source, the goal was to create a connection to that source, and allow the user to get the database metadata which could assist the user in preparing SQL statements. These statements could be submitted and the results displayed in a grid box. Since, dbAnywhere does not support java.sql, to implement this would require a more thorough understanding of the symantic.itool.db.pro package which was beyond the scope of this thesis. We were able to use java.sql to establish a connection via Borland's DataGateway and use the result set to populate the choice component. In this case, a simple java.sql implementation was much cleaner than the Symantec implementation. Unfortunately, the only interface to a dbAnywhere server is via Symantec's API. Borland Datagateway accepted Borlands API and Sun's JDBC API.

One of the shortcomings observed was that the wizard hard codes physical path name to the database. For example, when the add table wizard was used to connection to the Microsoft Access database that resided on the same machine as the dbAnywhere server, the following would be hard coded by the wizard:

```
("SELECT[RQ],[Account] FROM  
[C:\\dataBases\\Project97].[Travel]");
```

In this implementation, dbAnywhere used the JDBC-ODBC bridge to provide access to the data source. The database name was Project97, and resided in the dataBases subdirectory, on the C drive of the development system. Later, when the project was deployed to a separate machine, the physical location of the database was changed and the ODBC alias was updated. This resulted in the client application throwing an exception. The dbAnywhere server failed to encapsulate the physical location of the database. When the database was moved back to its original location, the client application worked fine. This failed to support a true client/server model and was a serious shortcoming. This problem was not apparent for the MS SQL 6.5 database.

To deploy the application a Jar utility function exists within the IDE. It can consolidate all required classes and create a Jar file or a zip file to facilitate distribution. To provide database access via dbAnywhere, the dbaw.zip and dbaw_awt.zip files can also be included in the Jar file.

4. Visual Café Summary

The integrated development environment (IDE) provided by Visual Café does not have a Java RMI compiler. The awt visual design tools are outstanding for creating a graphical user interface and fast compilation time is refreshing. Its database functionality is effective for simple form view data access and manipulation, but lacks the flexibility to be an effective enterprise development tool. One of the biggest limitations is that Visual Café only provides JDBC access through the dbAnywhere server. This requires the designer to learn Symantec's database API. Learning a proprietary API defeats the purpose of using Java. Also, all database aware components and wizards will only work with a dbAnywhere server, making the application reliant upon a proprietary server.

This also constrained how dbAnywhere could be used as a database gateway. For example, it may be effective to use the proprietary JDBC drivers (MS SQL6.5 or Sybase) in conjunction with an RMI implementation. The client could call the RMI object server, which could use java.sql to interface with the dbAnywhere middleware server, but dbAnywhere has a proprietary interface, so the interface will have to be via a Symantec package.

Visual Cafe also does not provide any integration with legacy data and transaction systems and the IDE is not integrated for Java RMI. The next version of Visual Café is

expected to offer integrated JDBC support.[Ref. 3]. For simple form views and rapid design, Visual Café is an effective tool. Java.sql is the JDBC standard interface, so most Java database designers are going to be reluctant to invest time and resources into learning Symantec's version of a database API.

B. BORLAND JBUILDER CLIENT/SERVER SUITE

Borland JBuilder Client/Server Suite version 1.01 is one of the most powerful and complete tools for developing Enterprise, pure Java database aware applications/applets. The JBuilder component palette is decorated with many database aware objects to display and manipulated data.

JBuilder comes with a type three JDBC driver (network-protocol/all-Java driver) called Borland DataGateway. It provides developers a multi-tier, fast and reliable database connectivity solution. Borland DataGateway is a collection of JDBC drivers that allow Java applications and applets on any platform to access both the desktop databases and client/server databases listed in Table 2.

1. Server Configuration

Borland DataGateway consists of four major parts. First the DataGateway Client which may be either LocalDriver.class or RemoteDriver.class. This driver communicates with the DataGateway Server using the TCP/IP protocol. It is downloaded with the Java database aware application/applet. Second is the DataGateway Server, which manages the transfer of information and calls between the Client and the Bridge. The bridge translated the calls that come from a Java application/applet to a protocol understood by a database engine. The final part is the engine which communicates with the vendor specific database management system drivers, or the ODBC API.

Borland DataGateway uses the Borland Database Engine (BDE) and SQL Links, which supply both native DBMS drivers and ODBC drivers, to provide heterogeneous relational database connectivity. In order to connect to the client/server databases, the DBMS specific client utilities, in our case MS SQL6.5 and Sybase, must be installed on the machine where Borland DataGateway Server is running, similar to the dbAnywhere configuration. The BDE Administrator utility, which comes with the DataGateway, lets developers create aliases to the actual databases.

a. Connecting to Microsoft SQL 6.5 Server Database

As we have pointed out earlier, the client utilities of SQL 6.5 must be installed before interfacing with the SQL 6.5 server. The steps for the installation and the configuration of client utilities are exactly the same as explained in dbAnywhere SQL 6.5 configuration.

The important thing here is to bind a server name to a valid IP address and a port number where the actual SQL server is running. This server name will be referenced inside the BDE Administrator while creating the alias to the SQL 6.5 database.

Once aliases have been established and the server is running, connections to the databases can be done by using either the *Remote Driver* (for both local and remote databases) or the *Local Bridge* (for local databases only) in the following ways. To invoke and register the Remote Driver class, call the Java method:

```
Class.forName("borland.jdbc.Broker.RemoteDriver");
```

For example, to the data source URL for the NSGDB data base was:

`jdbc:BorlandBroker://131.120.1.91/NSGDB`, where `jdbc` is the protocol, with a sub-protocol of `BorlandBroker` and `DataGateway` running on the machine whose IP address is 131.120.1.91 and `NSGDB` is the alias to the database which might reside on that machine or on a different machine that has a SQL server running. To invoke and register the *Local Driver* class, call the Java method:

```
Class.forName("borland.jdbc.Bridge.LocalDriver");
```

2. Database Aware Components

Jbuilder defines its own components to provide database connectivity. Borland has created many APIs specific to JBuilder that abstract some of the Java APIs in order to provide database connectivity and data-aware object creation. The package `borland.sql.dataset` has been created to provide data connectivity functionality that is JDBC specific. Its classes are used in conjunction with the classes in the `borland.jbcl.dataset` package, which provide general routines for data connectivity and data management and manipulation.

DataSet is an abstract class in borland.jbcl.dataset package to provide a cursor for accessing and navigating table data. The DataSet abstract class has been extended by additional Borland classes so that data-aware objects can store the data.

StorageDataSet class (extends DataSet) has been created to support easy and flexible manipulation and navigation of data in a common way regardless of how the data was obtained. Data can be obtained from a remote server through the use of a query (or QueryDataSet, which will be explained later). After data is stored in a StorageDataSet, it can be connected to the data-aware objects to display and manipulate.

In order to connect to and retrieve data from a database, even though the straight JDBC calls can be used throughout the application, JBuilder's borland.jdbc.dataset package offers many high level objects. The Database class encapsulates a database connection through JDBC to a data source and provides transaction support using local caching. A Database object that has a connection to a database can be created by the following Java statements:

```
Database db = new Database();
db.setConnection(new
borland.sql.dataset.ConnectionDescriptor
("jdbc:BorlandBroker://131.120.1.91/sqlDB", "userid",
"psswd", false, "borland.jdbc.Broker.RemoteDriver "));
```

A ConnectionDescriptor object requires the data source URL as a first parameter, then user id, password and finally the JDBC driver class name. JBuilder will generate all of the statements above for the developers.

After creating a Database object that will handle the JDBC connection to the SQL database and a QueryDescriptor object to store the query properties, a QueryDataSet component (from borland.sql.dataset package) can be easily created. The QueryDataSet component is an extension of its superclass (StorageDataSet) and provides functionality to run a query against a table in a SQL database. The data contained in a QueryDataSet is the result of the most recent query. The "result set" from the execution of the query is stored in the QueryDataSet, allowing tremendous flexibility in navigation of the resulting data.

DataModule is a powerful interface which developers implement when creating multiple GUIs that will use the same data model in one application/applet. For example, a designer will instantiate a DataModule object, and then will be able to graphically drag and drop database query components into the object. This centralizes all database

requests in one object instead of having it dispersed throughout the client code. According to JBuilder's online help manual "Data modules (often referred to as data models) are specialized containers where data access components and their associated properties are collected into a reusable component....The data module also provides a centralized location where "business logic" can be stored. "Business logic" describes the rules by which data is manipulated before and after the user (or client) sees the data." [Ref. 2]. One of the most important reasons to have a data model concept in the application/applet is to be able to access a single instance of the data model shared across the application/applet instead of allocating memory for multiple instances.

JBuilder offers many data-aware objects which have the ability to display and manipulate data. A Grid object can be used to display data in a tabular format and be modified by updating the appropriate cell. All JBuilder data-aware components are strictly tied to the JBuilder's database APIs. When a developer wishes to use Sun's JDBC calls in order to make objects display data, a conversion must be done from the `java.sql` objects to `borland.sql.dataset` and/or `borland.jbcl.dataset` objects.

3. Model Implementation

An application and applet were created which provided database connectivity through the DataGateway server. The MS Access database resided on the same machine as the DataGateway server, and the MS SQL database resided on a remote machine.

Creating the application was very quick and easy by using the objects and the methods provided in the JBuilder environment and APIs. The Remote Driver of DataGateway was used to handle the connection to both databases. We were able to easily implement a user log on, and display of the data sources the user had access to, as implemented in the RMI. With Jbuilder navigating among the data that was stored inside the DataSet objects was very flexible. When the user selected the data source, a connection to that source was created. A frame that contained the data-aware components was then opened which allowed users to view and manipulate the data.

Two choice control objects were added to the frame to allow users to change the data set view of the grid control object to a different table. One of the choice control objects displayed the table names. This was accomplished using straight JDBC calls. This proved that even though we were using JBuilder objects we could use appropriate JDBC calls to manipulate them. For example `borland.sql.dataset` package provides a Database class method (`getMetaData()`) which returns a JDBC DatabaseMetaData object.

Converting from application to applet is easy as long as certain applet restrictions are met such as, there should be no menu objects. If the application is modular enough to have reusable components, like generic frames and data modules, then using those components in applets will save time in this conversion process.

After finishing the implementation of the application, we used the deployment wizard to create a jar file (or it can create a zip file) which contained all the necessary class files (either Borland JBuilder proprietary or JDK 1.1) to facilitate distribution. Since the application was using Borland DataGateway Client to communicate with the server for providing database access, we shipped broker.zip (contains client class files) to the machines where our application was running. Detailed deployment issues can be found in the Appendix D.

4. Summary

The integrated development environment (IDE) of JBuilder Client/Server Suite provides an effective environment which makes projects easier to manage and organize. DataGateway is an effective type III Java driver that can be used by any client application via java.sql or whose functionality can be extended via Borland's database API. The interface to DataGateway allows visual database components to be used for database access and manipulation. JBuilder also contains a set of SQL tools, including a SQL Builder, which allows developers to create flawless drag-and-drop queries for QueryDataSet objects.

Even though JBuilder's database aware objects are tied to the JBuilder specific APIs, most of the classes provide methods to handle JDBC calls where customization is needed. When it comes to customizing the application/applet, JBuilder specific APIs must be learned in order to reduce the overhead associated with the conversion between JDBC methods and JBuilder database methods. JBuilder IDE offers an effective online help manual where all Java APIs and JBuilder specific APIs have been documented.

As a suite, Borland DataGateway can be used to provide database access, via java.sql calls. Developers can use the middleware driver in an RMI or socket implementation, which may not use Jbuilder's database aware GUI components. The IDE also includes support for Java Remote Method Invocation (RMI), so developers do not need to exit the JBuilder IDE in order to run the rmic compiler. JBuilder also comes with integrated CORBA/IIOP development tool that allow interoperability between objects built in different languages, running on different machines or running in

heterogeneous distributed environments. Borlands Jbuilder is an effective Java enterprise development tool.

C. CONCLUSION

One of the primary limitations of using a Commercial Java Database Solution is that the designer cannot manipulate the "black box" middleware server. The middleware server provides the access to the data source, but does not provide room for the designer to embed any logic, so all logic must be embedded in the client application. This does not provide a solid base for distributed computing solutions. This brings up the flexibility provided via an RMI implementation, where you can install the logic wherever it is convenient for the application. In the Symantec and Borland models, the client application was making database specific calls, to a backend database engine via a middleware server. The client code was directly tied to the backend database, which may be a limitation.

Another limitation, is that designers are forced to learn a proprietary API to take full advantage of database aware components in using the tools to design the user interface. At least with Jbuilder, the java.sql package could also be used, providing the designer the flexibility to use the most appropriate API to perform a task. This also reinforced that Borland Datagateway can be used as a middleware solutions in designing a database aware system. Symantec's dbAnywhere was only effective in implementin a Symantec solution. For example, dbAnywhere could not be used as a type III middleware driver in our RMI model, because it does not understand the java.sql API.

The implementation details are embedded in the client application, making the code harder to understand. This may affect future code maintenance issues. For example, the tools embed database functionality throughout the file. With the RMI implementation all database manipulation functions went through a database logic module, which provided a cleaner separation.

From the GUI designer's point of view, the RAD tools drastically decreased the overhead associated with creating professional looking database aware application. So depending upon the clients specifications, these tools are appropriate for simple form or grid database aware applications.

VII. CONCLUSION

A. SYNOPSIS

Java is prepared to provide Enterprises relational database solutions. Sun's Java Development Kit provides standardized packages which can be used for distributed computing solutions. As demonstrated in the models, Java's sql package provides a robust means of relational database access and manipulation. The communication link in distributed computing can be resolved by using Java's net package for a customized socket solution, or Java's RMI package for a more powerful object solution. The three models we implemented demonstrated some of the capabilities and limitations of combining the communication and the database requirements in providing a client/server relational database aware system. The technology selected by a designer will depend upon the customer's system specifications.

The multicast model, provided a customized way to allow clients to be servers and to join multicast groups. This allowed group members the ability to multicast a query to other group members who would service the request and respond. The capabilities of the server objects were implemented in various logic modules. This technology could also be used to analyze the database's of an organization to perform statistical analysis. The price associated with this flexibility was the time it took to implement the low level network protocol and define the communication protocols. In this implementation there was no single dedicated database server. Each application embedded database calls in its database logic module to connect to its local database server. When a client application joined a multicast group, it then became a server.

Remote Method Invocation was the next layer of abstraction. This technology encapsulated the low level network programming concerns, and allowed servers to pass objects to client applications. Multiple remote objects provided various services to client applications. In our model, a client would log onto the system via the RMI administrative object which used JDBC to access an authorization database. It would ensure the client was authorized access to the system and return a vector of databases the user was authorized access to, which would populate a choice component. When the user made a selection, the request would go to the RMI database object, which would provide database functionality. In this implementation all database access and coordination was encapsulated in the remote objects (servers). This implementation required system designers to develop interfaces, implementation details and client applications.

The commercial tools provided a one stop solution for simple database aware systems. dbAnywhere and DataGateway JDBC drivers provided the link to various databases. The client application would contain that appropriate classes that would allow it to talk with the middleware driver. Database aware components facilitated designing the client interface and in providing database manipulation. To take advantage of the database aware components, we had to use the vendors version of database API instead of java.sql. These tools provided a professional view, which is important for IT managers to convince leery Enterprise managers to install Java applications on their Windows desktop. The cost associated with the RAD tools was the decreased flexibility to customize system solutions, and the requirement to use a proprietary database API. For example, with RMI business logic could reside on the server, and be enforced on the server or dynamically downloaded to the client to enforce. With the Java RAD tools, the server is an untouchable black box, requiring the policy to be embedded in client applications. This requires system administrators to ensure clients are using the most recent version of each application.

B. AREAS FOR FURTHER RESEARCH

The objective of this thesis was to demonstrate how Java Database Connectivity can be used to access heterogeneous relational database's in a client/server environment. The models implemented demonstrate various configurations that can be used to provide a database aware solution. There are a number of areas that warrant future research to confirm the technology is prepared for commercial implementation. It is our assessment that Java Remote Method Invocation provides the most robust area for distributed computing solutions, so the majority of our recommendation for further research are in regards to RMI.

1. Security

As distributed computing becomes more and more widely accepted and employed, the assorted security risks associated continue to increase. This thesis did not explore the security issues associated with distributed Java computing. According to Sun, Java was designed for network based computing and security measures are integrated into Java's design [Ref. 8]. In order to convince managers to accept a Java distributed solution, it would be essential to understand the security strengths and weaknesses of the technology.

With the introduction of JDK 1.2, RMI enhancements makes it possible to use Secure Socket Layer (SSL). SSL is a network protocol that encrypts data sent and returned from the remote object. If the information is intercepted it is unreadable. It also ensures that client applications are dealing with legitimate servers and that only authorized clients are able to connect to the server. This reinforces RMI as a solid distributed computing technology.

2. Application Server

One of the attractive features of Java RMI is the ability to transport objects that are serializable across the network. Java's `awt` package provides the objects to implement a graphical user interface and `java.awt.Component` class implements serializable. `Container` extends the component class, and `Window` extends the `Container` class, so in concept a graphical user interface is serializable. This means that a user can log onto a system, and make a method call to the RMI implemented application server, to get user interface X. The interface will then be downloaded to the client. This will ensure the client gets the most recent version of the interface, and streamline and organizations version control. This is important in organizations that constantly change their database aware applications, or have a dynamic workforce.

For example, if a user is promoted, the database administrator grants the user access to the new database, say the maintenance database. The database administrator also updates the users access authorization via the remote RMI data admin object. When the user logs on, a list of databases the user is authorized to access is returned, and populates a choice box. Since this may be the first time the user has seen the maintenance database as a choice, he may not have the interface residing on his local machine. When the user selects the maintenance database it invokes a method to get maintenance interface version X. Since different clients will be authorized different views and privileges, the version number depicts what the user is authorized.

The interface gets returned to the client application to be displayed. When the user makes a selection, a request goes to the RMI database broker, that implements the access. Such a dynamic use of RMI provides a solid means for organizations to control which version of an application a client is using. If the application gets modified, the next time the user logs on, he or she will get the most recent version of the interface.

3. Multicast Remote Objects

Our RMI implementation extended `UnicastRemoteObject` which provides support for point-to-point active object references using TCP-based streams. As organizations depend on information for critical information, fault tolerance becomes an issue. With a single remote object implementation, if that remote object is not available, the system will fail. A remote object may not be available due to the remote system going down or a network failure. RMI also supports a `MulticastRemoteObject`. In this circumstance the request will go to multiple remote objects. This provides the fault tolerance mechanism that Enterprise organizations require for distributed computing. Quality of service factors may also be considered with multiple remote objects. The client makes a remote method call, and transparently the request goes out to N remote objects. If one remote server is overloaded, it may not be able to process the request in a timely fashion and depending upon how the client application was designed may block the client process from doing any additional work. With multiple remote objects, there is a higher probability that one of the remote objects will be capable of responding.

4. Object Oriented Database

As network bandwidth continues to increase, more complex objects such as video segments, audio segments and pictures continue to be sent. Object oriented databases provide a more suitable means to store the objects. It would be interesting to demonstrate how a heterogeneous RMI database server could be developed using RMI. The remote object may be able to provide access to relational or object oriented databases, transparently to the client.

5. Common Object Request Broker Architecture (CORBA)

Java RMI only allows Java clients to talk with Java servers. In the ubiquitous world of computing, the next level is to implement an object broker that is not language dependent. A CORBA implementation could provide the next level of generality. A performance comparison between our implementation and a Java/CORBA implementation would be interesting.

C. CONCLUSION

Java's platform independence is an attractive feature for many Enterprise organizations. Developer's enjoy the language because it is robust, fairly easy to learn,

and provides enormous flexibility. One of the strengths of Java technologies is that it allows designers to provide a total system solution in one programming language. Java technologies can be used to provide internet functionality via applets, for creating complex graphical user interfaces or for creating a distributed computing solution.

As the technology matures, so will development tools and JVM performance. Initially these tools encapsulated and automated designing and implementing a graphical user interface. This allows designers to focus on providing an effective solution, not on the details of implementing an interface. The functionality of these tools continues to be extended to encompass and automate database manipulation, and in Jbuilder case, to interact with CORBA.

LIST OF REFERENCES

1. Borland International, Inc., *Borland DataGateway for Java, Users Guide*, 1997.
2. Borland International, Inc., *Client/Server Suite Online Help Manual*, 1997.
3. Biggs, M., *Java IDEs Differ in Strengths*, InfoWorld, September 29, 1997.
4. Coulouris, G., Dollimore, J., Kindberg, T., *Distributed Systems Concept and Design*, Addison Wesley, 1996.
5. Dean, Andrew, *Database Access From The Web*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1997.
6. Flanagan, D., *Java Examples in a Nutshell, Second Edition*, O'Reilly & Associates, Inc., May 1997.
7. Flanagan, D., *Java In a Nutshell, a Desktop Quick Reference*, O'Reilly & Associates, May 1997.
8. Gosling, J., McGilton, H., *The Java™ Language Environment, A White Paper*, <http://java.sun.com/docs/white/langenv/>, May 1996.
9. Orfali, R., Harkey, D., *Client Server Programming with Java and Corba*, Wiley, 1996.
10. Orfali, R., Harkey, D., Edwards, J., *The Essential Client/Server Survival Guide*, Wiley, 1996.
11. Papageorg, J., *Getting Started with JDBC*, Sun Microsystems, <http://developer.javasoft.com/developer/>, August 1997.
12. JavaSoft, *JDBC Drivers*, Sun Microsystems, <http://java.sun.com/products/jdbc/jdbc.drivers.html>, September 1997.

13. JavaSoft, *JDBC: A Java SQL API*, Sun Microsystems, January 1997.
14. JavaSoft, *The Java Language: An Overview*, Sun Microsystems,
<http://java.sun.com/docs/overviews/java/java-overview-1.html>, 1995.
15. JavaSoft, *Java Remote Method Invocation Specification-JDK 1.1 FCS*,
<http://java.sun.com/docs/white/javarmi.html>, February 1997.
16. *Java Remote Method Invocation- Distributed Computing for Java*,
<http://java.sun.com/marketing/collateral/javarmi.html>, November 1997.
17. Kramer, D., *The Java Platform, A White Paper*,
<http://java.sun.com/docs/white/platform/CreditsPage.doc.html>, May 1996
18. Lindholm, T., Yellin, F., *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
19. Microsoft Corporation, *ODBC 3.0 Reference Manual*,
<http://www.microsoft.com/data/odbc/download.htm>
20. Naughton, P., *The Java™ Handbook, The Authoritative Guide to the Java Revolution*, Osborne McGraw-Hill, 1996.
21. Reese, G., *Database Programming with JDBC and Java*, O'Reilly & Associates, May 1997.
22. Renauld, Paul, *Introduction to Client/Server Systems*, Wiley, 1993.
23. Sood, M., *Examining JDBC Drivers*, Dr. Dobb's Journal, January 1998, p82-87.
24. Stallings, W., *Data and Computer Communications*, Prentice-Hall, Inc., 1997.
25. Stevenson, J., *An Enterprise Information System For The Naval Security Group*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1998.

26. Sybase Inc., *Sybase SQL Anywhere Users Guide Volume I*, Sybase Inc, 1995.
27. JavaSoft, *Java Database Programming*, <http://www.javasoft.com/.../whitepapers>, February 1997.

APPENDIX A. JDBC REFERENCE TABLE

<i>DriverManager</i>	
Methods	Purpose
getConnection(String url, String id, String password)	Returns a connection object to the database located at the url.
<i>Connection</i>	
Methods	Purpose
close()	Closes the connection to the database
commit()	Update the table with the changes. Required if user has disabled autoCommit
setAutoCommit(boolean)	Allows user to explicitly set to autoCommit to false
getMetaData()	A Connection's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, etc.
createStatement()	Returns a statement object. A Statement object is used for executing a static SQL statement and obtaining the results produced by it.
prepareCall(String)	A SQL stored procedure

APPENDIX A. JDBC Reference Table (cont.)

<i>Statement</i>	
Methods	Purpose
execute(String sql)	Executes the SQL statement, returns true if the first result is a ResultSet and false if it is an int. Useful if designer does not know if the SQL statement was a result producing SQL statement.
ExecuteQuery(String sql)	Executes the SQL statement, returns the result set containing the query results. Useful for SELECT statements.
ExecuteUpdate(String sql)	Executes the SQL UPDATE, INSERT, DELETE, CREATE statements that do not produce a ResultSet. Returns an int of the number of rows affected or zero
GetMoreResults()	Moves to a Statement's next result. It returns true if this result is a ResultSet. getMoreResults also implicitly closes any current ResultSet obtained with getResultSet.
GetResultSet()	Returns a ResultSet if there are results, else returns null
<i>ResultSet</i>	
Methods	Purpose
GetMetaData()	Returns a ResultSetMetaData object which contains the number of
getXXX(int columnIndex)	Used with CallableStatement Objects. Returns a data type represented by the XXX (long, string, int, object) containing the value in the current row based upon the column index.
getXXX(String columnName)	Same as above, based upon column name

APPENDIX B. MULTICAST SOCKET MODEL

This appendix provides the source code for the multicast socket implementation. It is organized from the graphical user interface, down through the logic modules as seen in the following figure.

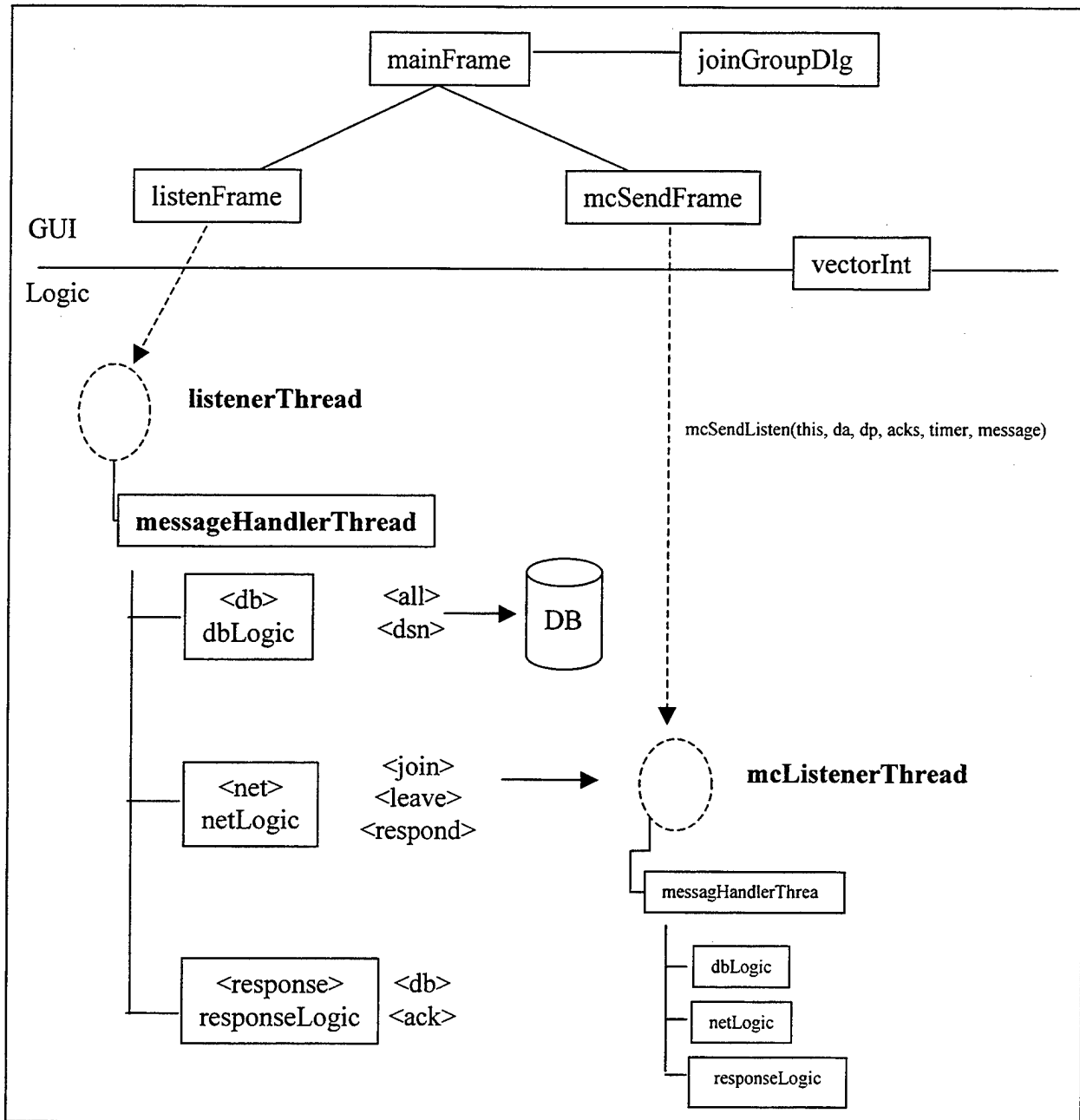


Figure 23: Multicast Model

The evaluation environment must be set up exactly as described in Chapter IV in order not to have problems during the execution of the program. The model is implemented as follows: The object server (objectServer.class) must be started on a machine where the 1_BnParts database is (in our environment, it was on "roxanne"). The object server will create the 1_BnParts object, which will handle the requests that can not be handled by the other database server objects and forwarded to it. Once the object server is up and running, the main frame object can be started on the remote machines. This frame object is the initial frame on which users have accesses to create the database servers (in this case, a_CoParts, b_CoParts and c_CoParts), initialize multicast groups and send messages either directly to a server or to a group.

The a_CoParts and the b_CoParts database server objects need to be created on the same machine where object server is running. In our evaluation environment we created the c_CoParts database server on a remote machine, namely "fido", which had a data source name defined to the c_CoParts database on machine "cryptologist".

Database server objects have to be alive, in order to process a request. If a message is sent before the database server is created, then that request will be timed out. From the initial frame (we called it as mainFrame), users can open up the listenFrame of database server objects (by clicking the appropriate name of the database server from the *DataSource* menu item) in order to activate their listening threads for receiving requests. Clicking the "start listening" button on listenFrame initiates the listening thread of that database server. Then created database server starts to wait for a request from its listening port and each received request will be displayed in the list box of the listenFrame.

Sending a direct request to a server can only be achieved from the main frame by selecting the request type (db, net, and response) and the recipient and by typing the message. Message types and their usage have been explained in Chapter IV in detail.

Multicast groups are formed by making the database server object join a the group. From Group | InitializeGroup menu item (on the main frame) the joinGroupDlg dialog box can be opened. The multicast group IP and port number and the name of the database server object must be specified on that dialog box in order to send a "join" or a "leave" message. When the object either joins or leaves the group successfully, it returns an acknowledgement message which will be displayed inside the responses window in joinGroupDlg dialog box. When a server object is a member of a multicast group, that means, its multicast listening thread is running. When it leaves the group, its multicast listening thread is destroyed.

A database server object can be a member of multiple multicast groups. In that case, it will have that many multicast listening threads, and each one will be listening on different port numbers.

From the main frame, by going through Group | Multicast Send menu items, one can send a multicast message by supplying the necessary information (Multicast Group IP address, port number and message) on the Multicast Request frame that pops up. Multicast Request frame (mcSendFrame) allows to specify the request type, multicast group IP and port number and the number of replies being waited after sending the request (normally, it will be the total number of group members) and the actual message body. The list box inside the Multicast Request Frame displays the replies in the order they are received.

```

//*****
//File: mainFrame.java
//Purpose: Initial frame, that allows user to create the db server
//         objects, open up the the multicast frame GUI implemented
//         with Visual Cafe
//*****
import java.awt.*;
import java.util.*; //for vector

public class mainFrame extends Frame implements vectorInt
{
    networkUtil network = new networkUtil();
    Vector resultVector = new Vector();

    public mainFrame()
    {
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 550,insets().top +
insets().bottom + 400);
        openFileDialog1 = new java.awt.FileDialog(this);
        openFileDialog1.setMode(FileDialog.LOAD);
        openFileDialog1.setTitle("Open");
        //$$ openFileDialog1.move(40,277);
        panell = new java.awt.Panel();
        panell.setLayout(null);
        panell.setBounds(insets().left + 12,insets().top + 12,528,144);
        add(panell);
        requestTypeChoice = new java.awt.Choice();
        requestTypeChoice.addItem("db");
        requestTypeChoice.addItem("net");
        requestTypeChoice.addItem("response");
        panell.add(requestTypeChoice);
        requestTypeChoice.setBounds(96,0,118,24);
        sendToChoice = new java.awt.Choice();
        sendToChoice.addItem("a_CoParts");
        sendToChoice.addItem("b_CoParts");
        sendToChoice.addItem("c_CoParts");
        panell.add(sendToChoice);
        sendToChoice.setBounds(96,48,117,18);
        label1 = new java.awt.Label("Request Type:");
        label1.setBounds(12,0,108,24);
        panell.add(label1);
        label2 = new java.awt.Label("Send To:");
        label2.setBounds(12,48,72,12);
        panell.add(label2);
        label3 = new java.awt.Label("Message or Request");
        label3.setBounds(300,60,163,16);
        panell.add(label3);
        messageTextField = new java.awt.TextField();
        messageTextField.setBounds(228,84,288,25);
        panell.add(messageTextField);
        sqlChoices = new java.awt.Choice();
        sqlChoices.addItem("INSERT INTO parts VALUES(12, 'trailer', 50, 1)");
        sqlChoices.addItem("DELETE FROM parts WHERE part = 'muffler'");
        sqlChoices.addItem("CREATE TABLE invoices(id int, supplier char(20))");
        sqlChoices.addItem("SELECT * FROM parts");
        sqlChoices.addItem("UPDATE parts SET quantity = 10 WHERE part = 'brake
pad'");
        sqlChoices.addItem("DROP TABLE testTable");
        sqlChoices.addItem("SELECT part FROM parts WHERE part = 'tank'");
    }
}

```

```

try {
    sqlChoices.select(-1);
} catch (IllegalArgumentException e) { }
panell.add(sqlChoices);
sqlChoices.setBounds(228,24,276,24);
label4 = new java.awt.Label("Sample SQL:");
label4.setBounds(324,0,78,21);
panell.add(label4);
sendButton = new java.awt.Button();
sendButton.setActionCommand("button");
sendButton.setLabel("Send");
sendButton.setBounds(396,120,80,25);
sendButton.setBackground(new Color(12632256));
panell.add(sendButton);
clearTextFieldBttn = new java.awt.Button();
clearTextFieldBttn.setActionCommand("button");
clearTextFieldBttn.setLabel("Clear");
clearTextFieldBttn.setBounds(288,120,80,25);
clearTextFieldBttn.setBackground(new Color(12632256));
panell.add(clearTextFieldBttn);
timeOut = new java.awt.TextField();
timeOut.setText("15000");
timeOut.setBounds(108,96,60,22);
panell.add(timeOut);
timeOutLable = new java.awt.Label("Set TimeOut (ms):");
timeOutLable.setBounds(0,96,96,20);
panell.add(timeOutLable);
panel2 = new java.awt.Panel();
panel2.setLayout(null);
panel2.setBounds(insets().left + 12,insets().top + 168,528,200);
add(panel2);
resultSetList = new java.awt.List(0,false);
panel2.add(resultSetList);
resultSetList.setBounds(12,12,504,180);
clearListBttn = new java.awt.Button();
clearListBttn.setActionCommand("button");
clearListBttn.setLabel("Clear ResultSet");
clearListBttn.setBounds(insets().left + 216,insets().top + 372,96,28);
clearListBttn.setBackground(new Color(12632256));
add(clearListBttn);
setTitle("Simulation Manager");
    //}}

    //{{(INIT_MENUS
mainMenuBar = new java.awt.MenuBar();
menul = new java.awt.Menu("File");
miExit = new java.awt.MenuItem("Exit");
menul.add(miExit);
mainMenuBar.add(menul);
group = new java.awt.Menu("Group");
joinGroup = new java.awt.MenuItem("Initilize Group");
group.add(joinGroup);
mcSend = new java.awt.MenuItem("Multicast Send");
group.add(mcSend);
mainMenuBar.add(group);
becomeMember = new java.awt.Menu("DataSource");
a_CoParts = new java.awt.MenuItem("a_CoParts");
becomeMember.add(a_CoParts);
b_CoParts = new java.awt.MenuItem("b_CoParts");
becomeMember.add(b_CoParts);
c_CoParts = new java.awt.MenuItem("c_CoParts");
becomeMember.add(c_CoParts);
mainMenuBar.add(becomeMember);

```

```

menu3 = new java.awt.Menu("Help");
mainMenuBar.setHelpMenu(menu3);
miAbout = new java.awt.MenuItem("About..");
menu3.add(miAbout);
mainMenuBar.add(menu3);
setMenuBar(mainMenuBar);
//$$ mainMenuBar.move(4,277);
//}}

//{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
miAbout.addActionListener(lSymAction);
miExit.addActionListener(lSymAction);
joinGroup.addActionListener(lSymAction);
sendButton.addActionListener(lSymAction);
clearTextFieldBtn.addActionListener(lSymAction);
clearListBtn.addActionListener(lSymAction);
mcSend.addActionListener(lSymAction);
SymItem lSymItem = new SymItem();
sqlChoices.addItemListener(lSymItem);
a_CoParts.addActionListener(lSymAction);
b_CoParts.addActionListener(lSymAction);
c_CoParts.addActionListener(lSymAction);
//}}
}

public mainFrame(String title)
{
    this();
    setTitle(title);
}

public synchronized void show()
{
    move(50, 50);
    super.show();
}

static public void main(String args[])
{
    mainFrame mainWindow = new mainFrame();
    mainWindow.show();
}

public void addNotify()
{
    // Record the size of the window prior to calling parents
addNotify. Dimension d = getSize();

    super.addNotify();

    if (fComponentsAdjusted)
        return;

    // Adjust components according to the insets
    setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
    Component components[] = getComponents();
    for (int i = 0; i < components.length; i++)
    {

```

```

        Point p = components[i].getLocation();
        p.translate(insets().left, insets().top);
        components[i].setLocation(p);
    }
    fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//{{DECLARE_CONTROLS
java.awt.FileDialog openFileDialog;
java.awt.Panel panel1;
java.awt.Choice requestTypeChoice;
java.awt.Choice sendToChoice;
java.awt.Label label1;
java.awt.Label label2;
java.awt.Label label3;
java.awt.TextField messageTextField;
java.awt.Choice sqlChoices;
java.awt.Label label4;
java.awt.Button sendButton;
java.awt.Button clearTextFieldBtn;
java.awt.TextField timeOut;
java.awt.Label timeOutLabel;
java.awt.Panel panel2;
java.awt.List resultSetList;
java.awt.Button clearListBtn;
//}}

//{{DECLARE_MENUS
java.awt.MenuBar mainMenuBar;
java.awt.Menu menu1;
java.awt.MenuItem miExit;
java.awt.Menu group;
java.awt.MenuItem joinGroup;
java.awt.MenuItem mcSend;
java.awt.Menu becomeMember;
java.awt.MenuItem a_CoParts;
java.awt.MenuItem b_CoParts;
java.awt.MenuItem c_CoParts;
java.awt.Menu menu3;
java.awt.MenuItem miAbout;
//}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == mainFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide();           // hide the Frame
    dispose();       // free the system resources
    System.exit(0); // close the application
}

class SymAction implements java.awt.event.ActionListener

```

```

    {
        public void actionPerformed(java.awt.event.ActionEvent event)
        {
            Object object = event.getSource();
            if (object == miAbout)
                miAbout_Action(event);
            else if (object == miExit)
                miExit_Action(event);
            else if (object == joinGroup)
                joinGroup_Action(event);
            else if (object == sendButton)
                sendButton_Action(event);
            else if (object == clearTextFieldBttn)
                clearTextFieldBttn_Action(event);
            else if (object == clearListBttn)
                clearListBttn_Action(event);
            else if (object == mcSend)
                mcSend_Action(event);
            else if (object == a_CoParts)
                aCoParts_Action(event);
            else if (object == b_CoParts)
                bCoParts_Action(event);
            else if (object == c_CoParts)
                cCoParts_Action(event);
        }
    }

    void miAbout_Action(java.awt.event.ActionEvent event)
    {
        //{{CONNECTION
        // Action from About Create and show as modal
        (new AboutDialog(this, true)).show();
        //}}
    }

    void miExit_Action(java.awt.event.ActionEvent event)
    {
        //{{CONNECTION
        // Action from Exit Create and show as modal
        (new QuitDialog(this, true)).show();
        //}}
    }

    void miOpen_Action(java.awt.event.ActionEvent event)
    {
        //{{CONNECTION
        // Action from Open... Show the OpenFileDialog
        openFileDialog1.show();
        //}}
    }

    void joinGroup_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.

        //{{CONNECTION
        // Create and show as non-modal
        (new joinGroupDlg(this, false)).show();
        //}}
    }

    void sendButton_Action(java.awt.event.ActionEvent event)
    {

```

```

        StringBuffer buff = new StringBuffer();
        String type = requestTypeChoice.getSelectedItemAt();
        String sendTo = sendToChoice.getSelectedItemAt();
        String message = messageTextField.getText();
        int timer = Integer.parseInt(timeOut.getText());

        if(type.equals("net")){
            buff.append(type);
            buff.append(" ");
            buff.append(message);
            buff.append(" ");
        }
        else if(type.equals("db")){
            String sql = message.toLowerCase();
            buff.append(type);
            buff.append(" ");
            buff.append(sendTo);
            buff.append(" ");
            buff.append(sql);
        }
        else if(type.equals("response") ){
            buff.append(type);
            buff.append(" ");
            buff.append("net");
            buff.append(" ");
            buff.append(message);
            buff.append(" ");
        }

        String request = new String(buff);
        //send the request
        network.sendListenForResponses(this, sendTo, request, timer);
    }

    void clearTextFieldBttn_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.
        //{{CONNECTION
        // Clear the text for TextField
        messageTextField.setText("");
        //}}
    }

    public void updateVector(String message)
    {
        resultVector.addElement(message);
        resultSetList.addItem(message);
    }

    void clearListBttn_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.

        //{{CONNECTION
        // Clear the List
        resultSetList.removeAll();
        //}}
    }
}

```



```

public void updateResultList(String message)
{
    resultVector.addElement(message);

    //post to response List
    StringTokenizer tok = new StringTokenizer(message, ",");

    if(message == null){
        resultSetList.addItem("No results were found ");
    }else{
        int count = tok.countTokens();
        for (int ix = 1; ix <= count; ix++){
            String tuple = tok.nextToken();
            resultSetList.addItem(tuple);
        } //end for
    } //end else
}

//for interface
public boolean mcListen()
{
    return true;
}

public void setMCListen(boolean flag){}

void mcSend_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Create and show the Frame
    (new mcSendFrame()).show();
    //}}
}

class SymItem implements java.awt.event.ItemListener
{
    public void itemStateChanged(java.awt.event.ItemEvent event)
    {
        Object object = event.getSource();
        if (object == sqlChoices)
            sqlChoices_ItemStateChanged(event);
    }
}

void sqlChoices_ItemStateChanged(java.awt.event.ItemEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Set the text for TextField... Get the current item text
    messageTextField.setText(sqlChoices.getSelectedItem());
    //}}
}

void aCoParts_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION

```

```

        // Create and show the Frame with a title... Get the MenuItem's
label      (new listenFrame(a_CoParts.getLabel())).show();
           //}}

           a_CoParts.setEnabled(false);
        }

void bCoParts_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
label      // Create and show the Frame with a title... Get the MenuItem's
           (new listenFrame(b_CoParts.getLabel())).show();
           b_CoParts.setEnabled(false);
           //}}
        }

void cCoParts_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
label      // Create and show the Frame with a title... Get the MenuItem's
           (new listenFrame(c_CoParts.getLabel())).show();
           c_CoParts.setEnabled(false);
           //}}
        }
    }
}
//*****
// End File:    mainFrame.java
//*****

```

```

//*****
// File:    listenFrame.java
// Purpose: A simple frame, that allows user to initiate the listen
//          thread of the database servers and allows to collect
//          messges received so far.
//          GUI implemented with Visual Cafe
//*****
import java.awt.*;
import symantec.itools.awt.util.ToolBarPanel;

public class listenFrame extends Frame implements vectorInt
{
    //flag checked by the mcListenerThread, set by netLogic
    boolean mcListen = false;

    public listenFrame()
    {
        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
    }
}

```

```

        setSize(insets().left + insets().right + 400,insets().top +
insets().bottom + 300);
        toolBarPanell = new symantec.itools.awt.util.ToolBarPanel();
        toolBarPanell.setLayout(new FlowLayout(FlowLayout.LEFT,0,0));
        toolBarPanell.setBounds(insets().left + 0,insets().top + 0,395,36);
        add(toolBarPanell);
        privateLine = new java.awt.Button();
        privateLine.setActionCommand("button");
        privateLine.setLabel("Start Listening");
        privateLine.setBounds(0,0,93,23);
        privateLine.setBackground(new Color(12632256));
        toolBarPanell.add(privateLine);
        clearList = new java.awt.Button();
        clearList.setActionCommand("button");
        clearList.setLabel("Clear List");
        clearList.setBounds(93,0,67,23);
        clearList.setBackground(new Color(12632256));
        toolBarPanell.add(clearList);
        hideWindow = new java.awt.Button();
        hideWindow.setActionCommand("button");
        hideWindow.setLabel("Hide Window");
        hideWindow.setBounds(160,0,87,23);
        hideWindow.setBackground(new Color(16776960));
        toolBarPanell.add(hideWindow);
        responseList = new java.awt.List(0,false);
        add(responseList);
        responseList.setBounds(insets().left + 12,insets().top + 96,382,193);
        mcCheckBox = new java.awt.Checkbox("mc Listener is Active");
        mcCheckBox.setBounds(insets().left + 252,insets().top + 36,144,17);
        add(mcCheckBox);
        setTitle("Untitled");
        //}}

        //{{INIT_MENU
        //}}

        //{{REGISTER_LISTENERS
        SymWindow aSymWindow = new SymWindow();
        this.addWindowListener(aSymWindow);
        SymAction lSymAction = new SymAction();
        privateLine.addActionListener(lSymAction);
        clearList.addActionListener(lSymAction);
        hideWindow.addActionListener(lSymAction);
        //}}
    }

    public listenFrame(String title)
    {
        this();
        setTitle(title);
    }

    public synchronized void show()
    {
        move(50, 50);
        super.show();
    }

    public void addNotify()
    {
        // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();

```

```

super.addNotify();

if (fComponentsAdjusted)
    return;

// Adjust components according to the insets
setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
Component components[] = getComponents();
for (int i = 0; i < components.length; i++)
{
    Point p = components[i].getLocation();
    p.translate(insets().left, insets().top);
    components[i].setLocation(p);
}
fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//{{{DECLARE_CONTROLS
    symantec.itools.awt.util.ToolBarPanel toolBarPanel1;
    java.awt.Button privateLine;
    java.awt.Button clearList;
    java.awt.Button hideWindow;
    java.awt.List responseList;
    java.awt.Checkbox mcCheckBox;
//}}}

//{{{DECLARE_MENUS
//}}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == listenFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide();        // hide the Frame
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == privateLine)
            privateLine_Action(event);
        else if (object == clearList)
            clearList_Action(event);
        else if (object == hideWindow)
            hideWindow_Action(event);
    }
}

void privateLine_Action(java.awt.event.ActionEvent event)

```

```

    {

        String dataSource = this.getTitle();
        int listenPort = portNumbers.getListenPort(dataSource);

        listenerThread listen = new listenerThread(this, dataSource,
listenPort);
        //listenerThread listen = new listenerThread(dataSource, listenPort);
        listen.start();

        //{{CONNECTION
        // Disable the Button
        privateLine.setEnabled(false);
        //}}
    }

    //vectorInt Implementation
    public void updateVector(String message)
    {
        //post to response List
        responseList.addItem(message);
    }

    //vectorInt Implementation
    public void updateResultList(String message)
    {
        //do nothing
    }

    //allow mcast listner to check if still in the group
    public boolean mcListen()
    {
        return mcListen;
    }

    //sets the flag, done by netLogic
    public void setMCListen(boolean flag)
    {
        mcListen = flag;
        if(flag){
            mcCheckBox.setState(true);
        }else
            mcCheckBox.setState(false);
    }

    void clearList_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.

        //{{CONNECTION
        // Clear the List
        responseList.removeAll();
        //}}
    }

    void hideWindow_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.

        //{{CONNECTION

```

```

// Move Frame to the back
toBack();
//}}
}

void closeButton_Action(java.awt.event.ActionEvent event)
{
/*      // to do: code goes here.
String myName = this.getTitle();
if(myName.equals("a_CoParts" )){
    mainWindow.a_CoParts.setEnabled(true);
}else if(myName.equals("b_CoParts" )){
    mainWindow.b_CoParts.setEnabled(true);
}else if(myName.equals("c_CoParts" )){
    mainWindow.c_CoParts.setEnabled(true);
}
*/
}

}

}
//*****
// End File:    listenFrame.java
//*****

//*****
// File:    mcSendFrame.java
// Purpose: A simple frame, that allows user to create and send
//          multicast requests based on the communication protocol
//          we implemented
//          GUI implemented with Visual Cafe
//*****
import java.awt.*;
import java.util.*;

import symantec.itools.awt.shape.HorizontalLine;
public class mcSendFrame extends Frame implements vectorInt
{
    Vector resultVector = new Vector();
    networkUtil network = new networkUtil();
    public mcSendFrame()
    {
        // This code is automatically generated by Visual Cafe when you
        // add components to the visual environment. It instantiates
        // and initializes the components. To modify the code, only use
        // code syntax that matches what Visual Cafe can generate, or
        // Visual Cafe may be unable to back parse your Java file into
        // its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 500,insets().top +
insets().bottom + 425);
        panell = new java.awt.Panel();
        panell.setLayout(null);
        panell.setBounds(insets().left + 12,insets().top + 12,481,168);
        add(panell);
        requestTypeChoice = new java.awt.Choice();
        requestTypeChoice.addItem("db");
        requestTypeChoice.addItem("net");
        requestTypeChoice.addItem("response");
        panell.add(requestTypeChoice);

```

```

requestTypeChoice.setBounds(96,0,118,24);
label1 = new java.awt.Label("Request Type:");
label1.setBounds(12,0,108,24);
panell.add(label1);
label3 = new java.awt.Label("Message or Request");
label3.setBounds(264,60,163,16);
panell.add(label3);
messageTextField = new java.awt.TextField();
messageTextField.setBounds(216,84,264,25);
panell.add(messageTextField);
sendButton = new java.awt.Button();
sendButton.setActionCommand("button");
sendButton.setLabel("Send");
sendButton.setBounds(348,120,80,25);
sendButton.setBackground(new Color(12632256));
panell.add(sendButton);
clearTextFieldBtn = new java.awt.Button();
clearTextFieldBtn.setActionCommand("button");
clearTextFieldBtn.setLabel("Clear");
clearTextFieldBtn.setBounds(240,120,80,25);
clearTextFieldBtn.setBackground(new Color(12632256));
panell.add(clearTextFieldBtn);
mcIP = new java.awt.TextField();
mcIP.setText("230.0.0.1");
mcIP.setBounds(108,36,72,19);
panell.add(mcIP);
mcPort = new java.awt.TextField();
mcPort.setText("4446");
mcPort.setBounds(108,72,48,20);
panell.add(mcPort);
label2 = new java.awt.Label("Group IP:");
label2.setBounds(12,36,60,18);
panell.add(label2);
label4 = new java.awt.Label("Group Port:");
label4.setBounds(12,72,72,20);
panell.add(label4);
acksTextField = new java.awt.TextField();
acksTextField.setText("1");
acksTextField.setBounds(108,108,42,19);
panell.add(acksTextField);
label5 = new java.awt.Label("# of Replies:");
label5.setBounds(12,108,70,19);
panell.add(label5);
sqlChoices = new java.awt.Choice();
sqlChoices.addItem("INSERT INTO parts VALUES(12, 'trailer', 50,
1)");
sqlChoices.addItem("DELETE FROM parts WHERE part = 'muffler'");
sqlChoices.addItem("CREATE TABLE invoices(id int, supplier
char(20))");
sqlChoices.addItem("SELECT * FROM parts");
sqlChoices.addItem("UPDATE parts SET quantity = 10 WHERE part =
'brake pad'");
sqlChoices.addItem("DROP TABLE testTable");
sqlChoices.addItem("SELECT part FROM parts WHERE part =
'tank'");
try {
    sqlChoices.select(-1);
} catch (IllegalArgumentException e) { }
panell.add(sqlChoices);
sqlChoices.setBounds(228,24,240,24);
timeOut = new java.awt.TextField();
timeOut.setText("15000");
timeOut.setBounds(108,144,48,19);

```

```

        panell1.add(timeOut);
        label6 = new java.awt.Label("Set Timer (ms):");
        label6.setBounds(12,144,89,17);
        panell1.add(label6);
        label7 = new java.awt.Label("Sample SQL");
        label7.setBounds(276,0,131,16);
        panell1.add(label7);
        horizontalLine1 = new
symantec.itools.awt.shape.HorizontalLine();
        horizontalLine1.setBounds(-12,179,492,1);
        panell1.add(horizontalLine1);
        resultSetList = new java.awt.List(0,false);
        add(resultSetList);
        resultSetList.setBounds(insets().left + 12,insets().top +
204,466,165);
        clearListBtnn = new java.awt.Button();
        clearListBtnn.setActionCommand("button");
        clearListBtnn.setLabel("Clear ResultSet");
        clearListBtnn.setBounds(insets().left + 252,insets().top +
384,96,28);
        clearListBtnn.setBackground(new Color(12632256));
        add(clearListBtnn);
        closeButton = new java.awt.Button();
        closeButton.setActionCommand("button");
        closeButton.setLabel("Close");
        closeButton.setBounds(insets().left + 132,insets().top +
384,71,29);
        closeButton.setBackground(new Color(16711680));
        add(closeButton);
        setTitle("Multicast Request");
        //}}

        //{{INIT_MENUS
        //}}

        //{{REGISTER_LISTENERS
        SymWindow aSymWindow = new SymWindow();
        this.addWindowListener(aSymWindow);
        SymAction lSymAction = new SymAction();
        clearListBtnn.addActionListener(lSymAction);
        closeButton.addActionListener(lSymAction);
        clearTextFieldBtnn.addActionListener(lSymAction);
        sendButton.addActionListener(lSymAction);
        SymItem lSymItem = new SymItem();
        sqlChoices.addItemListener(lSymItem);
        //}}
    }

    public mcSendFrame(String title)
    {
        this();
        setTitle(title);
    }

    public synchronized void show()
    {
        move(50, 50);
        super.show();
    }

    public void addNotify()
    {

```



```

addNotify.        // Record the size of the window prior to calling parents
                  Dimension d = getSize();

                  super.addNotify();

                  if (fComponentsAdjusted)
                      return;

                  // Adjust components according to the insets
                  setSize(insets().left + insets().right + d.width, insets().top
+ insets().bottom + d.height);
                  Component components[] = getComponents();
                  for (int i = 0; i < components.length; i++)
                  {
                      Point p = components[i].getLocation();
                      p.translate(insets().left, insets().top);
                      components[i].setLocation(p);
                  }
                  fComponentsAdjusted = true;
                }

                // Used for addNotify check.
                boolean fComponentsAdjusted = false;

                {{{DECLARE_CONTROLS
                java.awt.Panel panel1;
                java.awt.Choice requestTypeChoice;
                java.awt.Label label1;
                java.awt.Label label3;
                java.awt.TextField messageTextField;
                java.awt.Button sendButton;
                java.awt.Button clearTextFieldBttn;
                java.awt.TextField mcIP;
                java.awt.TextField mcPort;
                java.awt.Label label2;
                java.awt.Label label4;
                java.awt.TextField acksTextField;
                java.awt.Label label5;
                java.awt.Choice sqlChoices;
                java.awt.TextField timeOut;
                java.awt.Label label6;
                java.awt.Label label7;
                symantec.itools.awt.shape.HorizontalLine horizontalLine1;
                java.awt.List resultSetList;
                java.awt.Button clearListBttn;
                java.awt.Button closeButton;
                }}}

                {{{DECLARE_MENUS
                }}}

                class SymWindow extends java.awt.event.WindowAdapter
                {
                    public void windowClosing(java.awt.event.WindowEvent event)
                    {
                        Object object = event.getSource();
                        if (object == mcSendFrame.this)
                            Frame1_WindowClosing(event);
                    }
                }

                void Frame1_WindowClosing(java.awt.event.WindowEvent event)

```

```

{
    hide();          // hide the Frame
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == clearListBtn)
            clearListBtn_Action(event);
        else if (object == closeButton)
            closeButton_Action(event);
        else if (object == clearTextFieldBtn)
            clearTextFieldBtn_Action(event);
        else if (object == sendButton)
            sendButton_Action(event);
    }
}

void clearListBtn_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Clear the List
    resultSetList.removeAll();
    //}}
}

void closeButton_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Hide the Frame
    setVisible(false);
    //}}
}

void clearTextFieldBtn_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Clear the text for TextField
    messageTextField.setText("");
    //}}
}

void sendButton_Action(java.awt.event.ActionEvent event)
{
    StringBuffer buff = new StringBuffer();
    String type      = requestTypeChoice.getSelectedItem();
    String groupIP   = mcIP.getText();
    int groupPort    = Integer.parseInt(mcPort.getText() );
    int acks         = Integer.parseInt(acksTextField.getText() );
    String message   = messageTextField.getText();

    if(type.equals("net")){
        buff.append(type);
        buff.append(" ");
        buff.append(message);
    }
}

```

```

        buff.append(" ");
    }
    else if(type.equals("db")){
        String sql = message.toLowerCase();
        buff.append(type);
        buff.append(" all ");
        buff.append(sql);
    }
    else if(type.equals("response") ){
        buff.append(type);
        buff.append(" ");
        buff.append(message);
        buff.append(" ");
    }

    String request = new String(buff);

    int timer = Integer.parseInt(timeOut.getText() );

    //send the request
    network.mcSendListenForResponses(this, groupIP, groupPort, acks,
timer, request);
    }

    public void updateVector(String message)
    {
        resultVector.addElement(message);
        resultSetList.addItem(message);
    }

    private synchronized void updateResultSet(String message)
    {
        resultVector.addElement(message);

        //post to response List
        StringTokenizer tok = new StringTokenizer(message, ",");

        if(message == null){
            resultSetList.addItem("No results were found ");
        }else{
            int count = tok.countTokens();
            for (int ix = 1; ix <= count; ix++){
                String tuple = tok.nextToken();
                resultSetList.addItem(tuple);
            }//end for
        }//end else
    }

    public void updateResultList(String message)
    {
        updateResultSet(message);

    }//end function

    //for interface
    public boolean mcListen()
    {
        return true;
    }

    public void setMCListen(boolean flag){}

```

```

class SymItem implements java.awt.event.ItemListener
{
    public void itemStateChanged(java.awt.event.ItemEvent event)
    {
        Object object = event.getSource();
        if (object == sqlChoices)
            sqlChoices_ItemStateChanged(event);
    }
}

void sqlChoices_ItemStateChanged(java.awt.event.ItemEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Set the text for TextField... Get the current item text
    messageTextField.setText(sqlChoices.getSelectedItem());
    //}}
}
} //end class
//*****
// End File:    mSendFrame.java
//*****

//*****
// File:    joinGroupDlg.java
// Purpose: A dialog object, that allows database servers to join
//          to thje specified multicast group (based on the
//          IP address and port number)
//          GUI implemented with Visual Cafe
//*****
import java.awt.*;
import java.util.*; //for vector

public class joinGroupDlg extends Dialog implements vectorInt
{
    networkUtil network = new networkUtil();

    Vector resultVector = new Vector();

    public joinGroupDlg(Frame parent, boolean modal)
    {
        super(parent, modal);

        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 451, insets().top +
insets().bottom + 270);
        panell = new java.awt.Panel();
        panell.setLayout(null);
        panell.setBounds(insets().left + 12, insets().top + 12, 408, 80);
        add(panell);
        labell = new java.awt.Label("Group IP:");
        labell.setBounds(12, 12, 103, 22);

```

```

    panell.add(label1);
    mcPort = new java.awt.TextField();
    mcPort.setText("4446");
    mcPort.setBounds(132,48,48,20);
    panell.add(mcPort);
    label2 = new java.awt.Label("Group Port #:");
    label2.setBounds(0,48,110,22);
    panell.add(label2);
    recipiant = new java.awt.TextField();
    recipiant.setBounds(252,12,136,20);
    panell.add(recipiant);
    joinGroupButton = new java.awt.Button();
    joinGroupButton.setActionCommand("button");
    joinGroupButton.setLabel("Join Group");
    joinGroupButton.setBounds(276,48,95,25);
    joinGroupButton.setBackground(new Color(12632256));
    panell.add(joinGroupButton);
    mcIP = new java.awt.TextField();
    mcIP.setText("230.0.0.1");
    mcIP.setBounds(120,12,72,19);
    panell.add(mcIP);
    panel2 = new java.awt.Panel();
    panel2.setLayout(null);
    panel2.setBounds(insets().left + 12,insets().top + 108,204,132);
    add(panel2);
    label3 = new java.awt.Label("Responses:");
    label3.setBounds(48,0,72,12);
    panel2.add(label3);
    responseList = new java.awt.List(0,false);
    panel2.add(responseList);
    responseList.setBounds(0,24,180,84);
    closeButton = new java.awt.Button();
    closeButton.setActionCommand("button");
    closeButton.setLabel("Close");
    closeButton.setBounds(insets().left + 156,insets().top + 240,71,29);
    closeButton.setBackground(new Color(12632256));
    add(closeButton);
    leaveGroupButton = new java.awt.Button();
    leaveGroupButton.setActionCommand("button");
    leaveGroupButton.setLabel("Leave Group");
    leaveGroupButton.setBounds(insets().left + 288,insets().top + 108,95,25);
    leaveGroupButton.setBackground(new Color(16711680));
    add(leaveGroupButton);
    setTitle("Join Group");
    //}}

    //{{REGISTER_LISTENERS
    SymWindow aSymWindow = new SymWindow();
    this.addWindowListener(aSymWindow);
    SymAction lSymAction = new SymAction();
    joinGroupButton.addActionListener(lSymAction);
    closeButton.addActionListener(lSymAction);
    leaveGroupButton.addActionListener(lSymAction);
    //}}
}

public void addNotify()
{
    // Record the size of the window prior to calling parents addNotify.
    Dimension d = getSize();

    super.addNotify();
}

```

```

        if (fComponentsAdjusted)
            return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();
            p.translate(insets().left, insets().top);
            components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
    }

    // Used for addNotify check.
    boolean fComponentsAdjusted = false;

    public joinGroupDlg(Frame parent, String title, boolean modal)
    {
        this(parent, modal);
        setTitle(title);
    }

    public synchronized void show()
    {
        Rectangle bounds = getParent().bounds();
        Rectangle abounds = bounds();

        move(bounds.x + (bounds.width - abounds.width) / 2,
            bounds.y + (bounds.height - abounds.height) / 2);

        super.show();
    }

    //{{DECLARE_CONTROLS
        java.awt.Panel panel1;
        java.awt.Label label1;
        java.awt.TextField mcPort;
        java.awt.Label label2;
        java.awt.TextField recipiant;
        java.awt.Button joinGroupButton;
        java.awt.TextField mcIP;
        java.awt.Panel panel2;
        java.awt.Label label3;
        java.awt.List responseList;
        java.awt.Button closeButton;
        java.awt.Button leaveGroupButton;
    //}}

    class SymWindow extends java.awt.event.WindowAdapter
    {
        public void windowClosing(java.awt.event.WindowEvent event)
        {
            Object object = event.getSource();
            if (object == joinGroupDlg.this)
                Dialog1_WindowClosing(event);
        }
    }

    void Dialog1_WindowClosing(java.awt.event.WindowEvent event)

```

```

{
    hide();
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == joinGroupButton)
            joinGroupButton_Action(event);
        else if (object == closeButton)
            closeButton_Action(event);
        else if (object == leaveGroupButton)
            leaveGroupButton_Action(event);
    }
}

void leaveGroupButton_Action(java.awt.event.ActionEvent event)
{
    String IP = mcIP.getText();

    String mailTo = recipiant.getText();

    String message = leaveGroup(IP);

    network.sendListenForResponses(this, mailTo, message, 15000);

    // Clear the text for TextField
    recipiant.setText("");
}

void joinGroupButton_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    //get the mcIP
    String IP = "230.0.0.1";
    IP = mcIP.getText();
    int port = 4446;
    port = Integer.parseInt(mcPort.getText() );
    String mailTo = null;
    mailTo = recipiant.getText();
    String message = null;
    message = joinGroup(IP, port);

    network.sendListenForResponses(this, mailTo, message, 15000);

    // Clear the text for TextField
    recipiant.setText("");
}

public String joinGroup(String da, int dp)
{
    //createm message
    StringBuffer mBuff = new StringBuffer( "net join" );
    mBuff = mBuff.append( " " );
    mBuff = mBuff.append(da); //230.0.0.1
    mBuff = mBuff.append( " " );
    mBuff = mBuff.append( dp); //4446
    mBuff = mBuff.append( " " );
    String message = new String( mBuff );

    return message;
}

```

```

    }

    public String leaveGroup(String da)
    {
        //createm message
        StringBuffer mBuff = new StringBuffer( "net leave" );
        mBuff = mBuff.append( " " );
        mBuff = mBuff.append(da); //230.0.0.1
        mBuff = mBuff.append( " " );
        String message = new String( mBuff );

        return message;
    }

    public void updateVector(String message)
    {
        resultVector.addElement(message);

        //post to response List
        responseList.addItem(message);
    }

    public void updateResultList(String message)
    {
        //do nothing
    }

    //for interface
    public boolean mcListen()
    {
        return true;
    }
    public void setMCListen(boolean flag){}

    void closeButton_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.

        //{{CONNECTION
        // Hide the Dialog
        setVisible(false);
        //}}
    }
} //end class
//*****
// End File:    joinGroupDlg.java
//*****

//*****
// File:    vectorInt.java
// Purpose: The interface between logic objects and the GUI
//          that displays the results
//*****
import java.util.*; //for vecor

public interface vectorInt
{
    //sends a message to mother
    public abstract void updateVector(String message);
}

```



```

//tells mom its a resultSet
public abstract void updateResultList(String message);

// checks mothers mcListen flag, used by mcListenThread
// to evaluate if still in group or not
public abstract boolean mcListen();
public abstract void setMCListen(boolean flag);
}
//*****
// End File:    vectorInt.java
//*****

//*****
// File:       listenerThread.java
// Purpose: Implementation of a generic listener thread for database.
//           server objects. This thread is spawned when the database
//           objects are created and listens on a specified port.
//           This listener thread spawns a message handler thread and
//           sends the message to it.
//*****
import java.io.*;
import java.net.*;
import java.util.*;

public class listenerThread extends Thread{

    static int threadNumber = 1;           //threadID
    protected InetAddress    address = null; //who sent me the packet
    protected DatagramPacket inPacket = null; //response packets

    int listenPort = 0;
    int sendPort = 0;
    DatagramSocket listenerSocket = null; //socket to listen
    String name = null;
    vectorInt mother = null;
    boolean haveMom = false;

    private static final boolean debug = true;

    //*****
    // Function: clientListenThread(int listenPort)
    // Purpose: Constructor, user specifies port to listen to
    //           for a message
    //*****
    public listenerThread(String name, int listenPort)
    {
        super(name);
        System.out.println("Created listenerThread " + name + " | " +
listenPort);
        this.listenPort = listenPort;
        try{
            listenerSocket = new DatagramSocket(listenPort);
        }catch(SocketException e){
            System.err.println("Client Thread Error " + e);
        }catch(IOException e){
            System.err.println("Client Thread Error " + e);
        }
    }
}
//end constructor

```

```

//*****
// Function: clientListenThread(int listenPort)
// Purpose: Constructor, user specifies port to listen to
//           for a message
//*****
public listenerThread(vector<int> mother, String name, int listenPort)
{
    super(name);
    this.mother = mother;
    this.name = name;
    this.haveMom = true;

    System.out.println("Created listenerThread " + name + " | " + listenPort
+ " mother " + mother);
    this.listenPort = listenPort;
    try{
        listenerSocket = new DatagramSocket(listenPort);
    }catch(SocketException e){
        System.err.println("Client Thread Error " + e);
    }catch(IOException e){
        System.err.println("Client Thread Error " + e);
    }
}
//end constructor

//*****
// Function: void run()
// Purpose: Starts the listener thread
//*****
public void run()
{
    //forever loop
    while(true){
        try{
            //creat a inPacket for incoming messages
            byte [] buff = new byte[512];
            inPacket = new DatagramPacket(buff, buff.length);

            //wait to recieve a message
            listenerSocket.receive(inPacket);
            System.out.println(this.getName() + " listnerThread has recived a
packet");

            if(haveMom){
                System.out.println("creating a mother message HandlerThread");
                messageHandlerThread handler = new messageHandlerThread(mother,
inPacket, this.getName() );
                handler.start();
            }else {
                System.out.println("creating a non--mother message
HandlerThread");
                messageHandlerThread handler = new
messageHandlerThread(inPacket, this.getName());
                handler.start();
            }

        }catch (IOException e) {
            System.out.println(e);
        }
    }
}
//end while
//end run

}
//end class
//*****

```

```

// End File:      listenerThread.java
//*****

//*****
// File:      mcListenerThread.java
// Purpose: When a client joins a multicast group, this
//           (mcListenerThread)thread is spawned to monitor and
//           display incoming messages. Each object in the group has
//           one multicast listener and one private listener thread.
//*****
import java.io.*;
import java.net.*;
import java.util.*;

public class mcListenerThread extends Thread{

    boolean    inGroup = true;
    protected MulticastSocket socket = null;
    protected InetAddress    address = null;
    protected DatagramPacket packet = null;
    static int  threadNumber = 1;
    vectorInt  mother = null;
    boolean    haveMom = false;

    public mcListenerThread(String groupID, int groupSocket, String owner)
    {
        this.setName(owner);
        try{
            //join the Multicast group
            socket = new MulticastSocket(groupSocket);
            address = InetAddress.getByName(groupID);
            socket.joinGroup(address);

        }catch(UnknownHostException e){
            System.err.println("Client Thread Error " + e);
        }catch(IOException e){
            System.err.println("Client Thread Error " + e);
        }
    }
    //end constructor

    public mcListenerThread(vectorInt mother, String groupID, int groupSocket,
String owner)
    {
        this.setName(owner);
        this.mother = mother;
        this.haveMom = true;

        try{

            //join the Multicast group
            socket = new MulticastSocket(groupSocket);
            address = InetAddress.getByName(groupID);
            socket.joinGroup(address);

        }catch(UnknownHostException e){
            System.err.println("Client Thread Error " + e);
        }catch(IOException e){
            System.err.println("Client Thread Error " + e);
        }
    }
    //end constructor

```

```

//start thread
public void run()
{
    if(haveMom){
        System.out.println("mcListenerThread with mom " + mother + " " + "
status " + mother.mcListen());

        while( mother.mcListen() ){

            byte [] buff = new byte[512];
            packet = new DatagramPacket(buff, buff.length);

            try{
                //recieve a message
                socket.receive(packet);

                //pass root name
                StringTokenizer tok = new StringTokenizer(this.getName());
                String rootName = tok.nextToken();
                //spawn a request handler thread
                messageHandlerThread handler = new messageHandlerThread(mother,
packet, rootName);
                handler.start();
                System.out.println("spawning a messageHandlerThread for " +
mother);

            }catch (IOException e) {
                System.out.println(e);
                System.exit(-1);
            }
        }
    }

    try{
        socket.leaveGroup(address);
        socket.close();
    }catch(IOException e){
        System.err.println("Client Thread Error " + e);
    }

}

//for the orphans
else{
    //forever loop
    while(inGroup){

        //creat a inPacket for incoming messages
        byte [] buff = new byte[256];
        packet = new DatagramPacket(buff, buff.length);

        try{
            //recieve a message
            socket.receive(packet);

            //pass root name
            StringTokenizer tok = new StringTokenizer(this.getName());
            String rootName = tok.nextToken();
            //spawn a request handler thread
            if(haveMom){
                messageHandlerThread handler = new
messageHandlerThread(mother, packet, rootName);
                handler.start();
            }
        }
    }
}
}

```



```

//keep track of mcgroups thread has joined
Vector mcVector = null;
String message = "";
//declare a dbServer
dbUtil dbServer = new dbUtil();
networkUtil network = new networkUtil();

//*****
// Function:messageHandlerThread(DatagramPacket inPacket, int threadNumber)
// Purpose: Thread constructor, takes the datagram packet and it's assigned
// thread number
//*****
public messageHandlerThread(DatagramPacket inPacket, String owner)
{
    this.setName(owner);
    this.threadName = owner;
    //get the request message
    message = new String(inPacket.getData() );

    //get the address and port of the requestor
    senderAddress = inPacket.getAddress();
    senderPort = inPacket.getPort();

} //end constructor

//*****
// Function: messageHandlerThread(DatagramPacket inPacket, int threadNumber)
// Purpose: Thread constructor, takes the datagram packet and it's assigned
// thread number
//*****
public messageHandlerThread(vectorInt mother, DatagramPacket inPacket,
String owner)
{
    this.setName(owner);
    this.threadName = owner;
    this.mother = mother;
    this.haveMom = true;

    //get the request message
    message = new String(inPacket.getData() );

    //get the address and port of the requestor
    senderAddress = inPacket.getAddress();
    senderPort = inPacket.getPort();

} //end constructor

//*****
// Function: void run()
// Purpose: Starts the thread, called by start();
//*****
public void run()
{

    //parse the message recieved
    StringTokenizer tok = new StringTokenizer(message);
    //who needs to process this message
    String logicModule = tok.nextToken();

    //reformat the rest of the message
    String restOfMessage = restOfMessage(tok);

```

```

        //evaluate first token to determine which logicModule to call and pass
        // that module the rest of the message
        if (logicModule.equals( "net" ) ){
            if(haveMom){
                networkLogic nl = new networkLogic(mother, threadName,
restOfMessage, senderAddress, senderPort);
                nl.executeRequest();
            }else {
                networkLogic nl = new networkLogic(threadName, restOfMessage,
senderAddress, senderPort);
                nl.executeRequest();
            }
        }
        else if(logicModule.equals( "db" ) ){
            if (haveMom){
                dbLogic dl = new dbLogic(mother, threadName, restOfMessage,
senderAddress, senderPort);
                dl.executeRequest();
            }else{
                dbLogic dl = new dbLogic(threadName, restOfMessage, senderAddress,
senderPort);
                dl.executeRequest();
            }
        }
        else if(logicModule.equals( "response" ) ){
            if (haveMom){
                responseLogic rl = new responseLogic(mother, threadName,
restOfMessage, senderAddress, senderPort);
                rl.executeRequest();
            }else{
                responseLogic rl = new responseLogic(threadName, restOfMessage,
senderAddress, senderPort);
                rl.executeRequest();
            }
        }
        else{
            //dont process this request
            System.out.println("Message Handler cannot process this request ...");
            if (haveMom){
                mother.updateVector("Message Handler cannot process this request");
            }
        }
    } //end if

} //end run

private String restOfMessage(StringTokenizer tok)
{
    StringBuffer buff = new StringBuffer( );

    //get the rest of the request
    while( tok.hasMoreTokens() ){

        buff = buff.append( tok.nextToken() ); //appends the sql statement
        buff = buff.append( " " );
    }

    String result = new String( buff );

    return result;
}

} //end class

```

```

//*****
// End File:    messageHandlerThread.java
//*****

//*****
// File:       dbLogic.java
// Purpose:    This class contains the business logic to handle the
//            database requests based on the given privileges of the
//            database server.
//*****
import java.net.*;
import java.sql.*;
import java.util.*;

class dbLogic
{
    String threadName = null;
    String message = null;
    InetAddress senderAddress = null;
    int senderPort = 0;

    static int mcID = 1; //for join mcgroup command

    networkUtil network = new networkUtil();
    dbUtil      dbServer= new dbUtil();

    boolean debug = true;
    boolean ack = false;

    vectorInt mother = null;
    boolean haveMom = false;

    public dbLogic(String threadName, String message, InetAddress sa, int sp)
    {
        this.threadName = threadName;
        this.senderAddress = sa;
        this.senderPort = sp;
        this.message = message;

    } //end constructor

    public dbLogic(vectorInt mother, String threadName, String message,
    InetAddress sa, int sp)
    {
        this.threadName = threadName;
        this.senderAddress = sa;
        this.senderPort = sp;
        this.message = message;
        this.mother = mother;
        this.haveMom = true;

    } //end constructor

    public void executeRequest()
    {
        dbRequest(message);
    }
}

```



```

//*****
// Function: void dbRequest(String message)
// Purpose: This is where all the messag logic will reside
//          - service database requests based upon listenerThread name
//*****
private void dbRequest(String message)
{
    System.out.println("db request " + message);
    String mess = message.trim();
    if (haveMom)
        mother.updateVector(mess);
    //use a string tokenizer to parse the request, the delimiter is
    //blankspace
    StringTokenizer tok = new StringTokenizer(mess);

    //get the data source
    String dsn = tok.nextToken();

    //repackage the rest of the message
    String sql = restOfMessage(tok);
    String whoAmI = threadName;

    // FOR MULTICAST COMMS
    if( dsn.equals( "all" ) ){
        //bn_dbRequest("l_BnParts", sql);
        if (whoAmI.equals("a_CoParts" ) )

            co_dbRequest("a_CoParts", sql);
        else if (whoAmI.equals("b_CoParts" ) ){
            try{
                Thread.currentThread().sleep(2000);
            }catch(InterruptedException e){}
            co_dbRequest("b_CoParts", sql);
        }
        else if (whoAmI.equals("c_CoParts" ) ){
            dbLogic_C cl = new dbLogic_C(mother, "c_CoParts", sql,
senderAddress, senderPort);
            cl.executeRequest();
        }
        else if (whoAmI.equals("l_BnParts" ) ) {
            bn_dbRequest(sql);
        }
    }

    //end if

    //FOR POINT TO POINT COMMUNICATION
    //can handle select and update
    else if ( dsn.equals( "a_CoParts" ) && whoAmI.equals("a_CoParts" ) ) {
        co_dbRequest(dsn, sql);
    }
    else if ( dsn.equals( "b_CoParts" ) && whoAmI.equals("b_CoParts" ) ) {
        //test to demonstrate why synch is needed
        try{
            Thread.currentThread().sleep(2000);
        }catch(Exception e){}

        co_dbRequest(dsn, sql);
    }
    else if ( dsn.equals( "c_CoParts" ) && whoAmI.equals("c_CoParts" ) ) {
        dbLogic_C cl = new dbLogic_C(mother, "c_CoParts", sql,
senderAddress, senderPort);
        cl.executeRequest();
    }
}

```

```

    }
    else if ( dsn.equals( "1_BnParts" ) && whoAmI.equals("1_BnParts") ) {
        System.out.println("making bn_dbCall ");

        bn_dbRequest(sql);
    }

    else{
        System.out.println("Database logic Module cannot process " + message);
    }

} //end function

//*****
// Function: co_dbRequest(String dsn, String sql)
// Purpose:  Companys are authorizes to perfrom database queries and to
//           perform updates.
// Logic:   establishes a connection to the database to process the request
//*****
public void co_dbRequest(String dsn, String sql)
{
    //use a string tokenizer to parse the request, the delimiter is
    //blankspace
    StringTokenizer tok = new StringTokenizer(sql);

    //evaluate the first token <CREATE, INSERT, SELECT, DELETE>
    String command = tok.nextToken();

    if( command.equals( "select" ) ) {

        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String userID = null;
        String password = null;

        //set the connection, jdbc:odbc:a_CoParts,
        dbServer.setConnection(driver, "jdbc:odbc:" + dsn, userID, password);

        //process the query
        String resultSet = dbServer.executeSQLGetString(sql);

        //close the connection
        dbServer.closeConnection();

        StringBuffer buff = new StringBuffer();
        buff.append("response db ");
        buff.append(threadName);
        buff.append(" ");
        buff.append(resultSet);

        String message1 = new String( buff );

        //send the string to the user
        network.sendMessage(senderAddress, senderPort, message1);
        print.printResultSet( threadName, resultSet);

        if (haveMom){
            StringBuffer buff_1 = new StringBuffer();
            buff_1.append("send resultSet to ");
            buff_1.append(senderAddress);
            String r = new String(buff_1);
            String q = r.trim();

```

```

        mother.updateVector(q);
    }
} //end if
else if (command.equals( "update" ) ){

    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    String userID = null;
    String password = null;

    //set the connection, jdbc:odbc:a_CoParts,
    dbServer.setConnection(driver, "jdbc:odbc:" + dsn, userID, password);

    //process request returning a vector
    Vector resultVector = new Vector();
    resultVector = dbServer.executeSQL(sql);

    //close the connection
    dbServer.closeConnection();

    StringBuffer buff = new StringBuffer();
    buff.append("response db");
    buff.append(" ");
    buff.append(" update was successful ");

    String message1 = new String( buff );

    //send the string to the user
    network.sendMessage(senderAddress, senderPort, message1);

    if (haveMom){
        String r = "update complete";
        String q = r.trim();
        mother.updateVector(q);
    }
    dbServer.printConsole(resultVector);
}
else
{
    System.out.println(threadName + " forwarding to Battalion " +
sql);
    if (haveMom){
        StringBuffer buff = new StringBuffer();
        buff.append("forwarding request to Battalion ");
        buff.append(sql);
        String r = new String(buff);
        String q = r.trim();
        mother.updateVector(q);
    }

    //get bnPort number
    int bnPort = portNumbers.getListenPort("1_BnParts");
    int myPort = portNumbers.getSendPort(threadName);
    InetAddress bnHost = portNumbers.getHost();

    StringBuffer buff2 = new StringBuffer();
    buff2.append("db 1_BnParts ");
    buff2.append(threadName);
    buff2.append(" ");
    buff2.append(sql);

    String message2 = new String( buff2 );

```

```

String sql2 = message2.trim();
//send it to battalion to process
network.sendMessage(bnHost, bnPort, myPort, sql2);

//tell client cannot process request
StringBuffer buff = new StringBuffer();
buff.append("response net ");
buff.append(threadName);
buff.append(" Request was Forwarded to Battalion");

String message1 = new String( buff );

//send the string to the user
network.sendMessage(senderAddress, senderPort, message1);

if(haveMom)
    mother.updateVector(message2);

} //end else

} //end function

//company objects can only perform SELECT AND UPDATE
public void bn_dbRequest(String inMessage)
{
    //peel the next token off to get dsn
    StringTokenizer tok = new StringTokenizer(inMessage);

    //evaluate the first token "select part from parts"
    String dsn = tok.nextToken();
    String sql = restOfMessage(tok);

    //who sent the request
    //String dsn = portNumbers.getName(senderPort);
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    String userID = null;

    if(dsn.equals("c_CoParts")){
        //assign admin userid for c_CoParts db
        userID = "sa";
    }
    String password = null;

    //set the connection, jdbc:odbc:a_CoParts,
    dbServer.setConnection(driver, "jdbc:odbc:" + dsn, userID, password);

    //process the query
    String resultSet = dbServer.executeSQLGetString(sql);
    //close the connection
    dbServer.closeConnection();

    StringBuffer buff = new StringBuffer();
    buff.append("response db "); //select //message ! null
    buff.append(threadName);
    buff.append(" ");
    buff.append(resultSet);

    String message1 = new String( buff );

    //send the string to the user
    network.sendMessage(senderAddress, senderPort, message1);
} //end function

```

```

private String restOfMessage(StringTokenizer tok)
{
    StringBuffer buff = new StringBuffer( );

    //get the rest of the request
    while( tok.hasMoreTokens() ){

        buff = buff.append( tok.nextToken() ); //appends the sql statement
        buff = buff.append( " " );
    }

    String result = new String( buff );

    return result;
}

} //end class
//*****
// End File:    dbLogic.java
//*****

//*****
// File:    dbLogic_C.java
// Purpose: This class contains the business logic to handle the
//          database requests specific to c_CoParts database server.
//*****
import java.util.*;
import java.sql.*;
import java.net.*;

class dbLogic_C
{
    String threadName = null;
    String message = null;
    InetAddress senderAddress = null;
    int senderPort = 0;

    static int mcID = 1; //for join mcgroup command

    networkUtil network = new networkUtil();
    dbUtil dbServer= new dbUtil();
    vectorInt mother = null;
    boolean haveMom = false;

    boolean debug = true;

    public dbLogic_C(vectorInt mother, String threadName, String message,
InetAddress sa, int sp)
    {
        this.threadName = threadName;
        this.message = message;
        this.senderAddress = sa;
        this.senderPort = sp;
        this.mother = mother;
        this.haveMom = true;
    }

    public void executeRequest()
    {

```

```

        dbRequest(threadName, message);
    }

    //company objects can only perform SELECT AND UPDATE
    private void dbRequest(String dsn, String sql)
    {
        //use a string tokenizer to parse the request, the delimiter is
        //blankspace
        StringTokenizer tok = new StringTokenizer(sql);

        //evaluate the first token "select part from parts"
        String command = tok.nextToken();

        if( command.equals( "select" ) ) {

            String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
            String userID = "sa";
            String password = null;

            //set the connection, jdbc:odbc:a_CoParts,
            dbServer.setConnection(driver, "jdbc:odbc:" + dsn, userID, password);

            //process the query
            String resultSet = dbServer.executeSQLGetString(sql);
            //close the connection
            dbServer.closeConnection();

            StringBuffer buff = new StringBuffer();
            buff.append("response db "); //select //message ! null
            buff.append(threadName);
            buff.append(" ");
            buff.append(resultSet);

            String message1 = new String( buff );

            //send the string to the user
            network.sendMessage(senderAddress, senderPort, message1);
            print.printResultSet(threadName, resultSet);

        } //end if
        else if( command.equals( "update" ) ) {

            String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
            String userID = null;
            String password = null;

            //set the connection, jdbc:odbc:a_CoParts,
            dbServer.setConnection(driver, "jdbc:odbc:" + dsn, userID, password);

            //process request returning a vector
            Vector resultVector = new Vector();
            resultVector = dbServer.executeSQL(sql);
            //close the connection
            dbServer.closeConnection();

            StringBuffer buff = new StringBuffer();
            buff.append("response db "); //select //message ! null
            buff.append(threadName);
            buff.append(" ");
            buff.append(" update was successful ");

            String message1 = new String( buff );

```

```

        //send the string to the user
        network.sendMessage(senderAddress, senderPort, message1);

        //to server screen
        if (debug) System.out.println("SQL = " + sql + " processed by " +
threadName);
        //print it to the server screen
        dbServer.printConsole(resultVector);

    }
    else
    {
        System.out.println(threadName + "cannot process forwarding it to
Battalion" + sql);

        //get bnPort number
        int bnPort = portNumbers.getListenPort("1_BnParts");

        int myPort = portNumbers.getSendPort(threadName);
        InetAddress bnHost = portNumbers.getHost();

        StringBuffer buff2 = new StringBuffer();
        buff2.append("db 1_BnParts ");
        buff2.append(threadName);
        buff2.append(" ");
        buff2.append(sql);

        String message2 = new String( buff2 );
        //send it to battalion to process
        network.sendMessage(bnHost, bnPort, myPort, message2);

        //tell client cannot process request
        StringBuffer buff = new StringBuffer();
        buff.append("response net ");
        buff.append(threadName);
        buff.append(" Request was Forwarded to Battalion");

        String message1 = new String( buff );

        //send the string to the user
        network.sendMessage(senderAddress, senderPort, message1);

    } //end else

} //end function

} //end class
//*****
// End File:    dbLogic_C.java
//*****

//*****
// File:    responseLogic.java
// Purpose: This class contains the logic for handling the responses
//          from objcets. Responses might be either query resultset
//          or an acknowledgement or a network response.
//*****
import java.net.*;
import java.sql.*;
import java.util.*;

```

```

class responseLogic
{
    String threadName = null;
    String message = null;
    InetAddress senderAddress = null;
    int senderPort = 0;

    static int mcID = 1; //for join mcgroup command

    networkUtil network = new networkUtil();
    dbUtil      dbServer= new dbUtil();

    vectorInt mother = null;
    boolean debug = true;

    public responseLogic(String threadName, String message, InetAddress sa, int
sp)
    {
        this.threadName = threadName;
        this.senderAddress = sa;
        this.senderPort = sp;
        this.message = message;

    } //end constructor

    public responseLogic(vectorInt mother, String threadName, String message,
InetAddress sa, int sp)
    {
        this.threadName = threadName;
        this.senderAddress = sa;
        this.senderPort = sp;
        this.message = message;
        this.mother = mother;

    } //end constructor

    public void executeRequest()
    {
        responseRequest(message);

    }

    public void responseRequest(String message)
    {
        //use a string tokenizer to parse the request, the delimiter is
blankspace
        StringTokenizer tok = new StringTokenizer(message);

        //type of response <db, ack,
        String command = tok.nextToken();
        String from     = tok.nextToken();
        String restOfMsg = restOfMessage(tok);
        if (command.equals( "db" ) ){
            //a synchronized print function
            print.printResultSet(from, restOfMsg);

            //send response to mother
            mother.updateVector(from);
            mother.updateResultList(restOfMsg);
        }
    }
}

```



```

        } //end if
        else if ( command.equals( "ack" ) ){
            System.out.println("\nack" + "from " + senderAddress + " | " +
senderPort);
        }
        else if ( command.equals( "net" ) ){
            print.printMessage(from, restOfMsg);
            mother.updateVector(from);
            mother.updateVector(restOfMsg);
        } //end if
    } //end function

private String restOfMessage(StringTokenizer tok)
{
    StringBuffer buff = new StringBuffer( );

    //get the rest of the request
    while( tok.hasMoreTokens() ){

        buff = buff.append( tok.nextToken() ); //appends the sql statement
        buff = buff.append( " " );
    }

    String result = new String( buff );
    String res = result.trim();

    return res;
}
}
//*****
// End File:    responseLogic.java
//*****

//*****
// File:    networkLogic.java
// Purpose: This class contains the logic for handling the network
//          protocol messages from objects. Network request messages
//          might be either a join command (to a multicast group),
//          or leave command (from a multicast group), or an ack,
//          or a query about the aliveness of the object.
//*****
import java.net.*;
import java.util.*;

class networkLogic
{

    String threadName = null;
    String message = null;
    InetAddress senderAddress = null;
    int senderPort = 0;

    static int mcID = 1; //for join mcgroup command

    networkUtil network = new networkUtil();

    vectorInt mother = null;
    boolean haveMom = false;

```

```

//*****
// Function: networkLogic(String threadName, String message, InetAddress sa,
// int sp)
// Purpose:  handlet net protocol
//*****
public networkLogic(String threadName, String message, InetAddress sa, int
sp)
{
    this.threadName = threadName;
    this.senderAddress = sa;
    this.senderPort = sp;
    this.message = message;
} //end constructor

//*****
// Function: networkLogic(String threadName, String message, InetAddress sa,
//int sp)
// Purpose:  handlet net protocol
//*****
public networkLogic(vectorInt mother, String threadName, String message,
InetAddress sa, int sp)
{
    this.threadName = threadName;
    this.senderAddress = sa;
    this.senderPort = sp;
    this.message = message;
    this.mother = mother;
    this.haveMom = true;
} //end constructor

public boolean executeRequest()
{
    networkRequest(message, threadName);
    return true;
}

//*****
// Function: networkRequest(String message)
// Purpose:
//*****
private void networkRequest(String message, String threadName)
{
    StringTokenizer tok = new StringTokenizer(message);
    String command = tok.nextToken(); //get the first token

    if ( command.equals( "join" ) ){

        //get the next two tokens
        String mcIP = tok.nextToken();
        String a = tok.nextToken();
        int mcPort = Integer.parseInt(a);

        //display result in gui
        if(haveMom){
            mother.updateVector("Join " + mcIP + " | " + mcPort);
            //set the mcListen flag
            mother.setMCListen(true);
        }

        StringBuffer buff = new StringBuffer( threadName );
        buff.append(" mc");
        buff.append(mcID);
    }
}

```

```

        mcID++;
        String mcThreadName = new String(buff);

        //create a new mcLister thread
        if(haveMom){
            mcListenerThread mclistener = new mcListenerThread(mother, mcIP,
mcPort, mcThreadName );
            mclistener.start();
        }else{
            mcListenerThread mclistener = new mcListenerThread(mcIP, mcPort,
mcThreadName );
            mclistener.start();
        }

        //send a an ack back
        int myPort = portNumbers.getSendPort(threadName);
        //format message
        StringBuffer buff1 = new StringBuffer();
        buff1.append("net ack ");
        buff1.append(threadName);
        buff1.append(" ");

        String response = new String( buff1 );
        System.out.println("Sending response: " + response);
        //use private line so receiver can get your addrss and port
        network.sendMessage(senderAddress, senderPort, myPort, response);
    }
    else if ( command.equals( "leave" ) ){

        String ip = tok.nextToken();

        //display result in gui
        if(haveMom){
            mother.updateVector("leave " + ip);
            //set the mcListen flag
            mother.setMCListen(false);
        }
    }
    //incoming message
    else if ( command.equals( "ack" ) ){

        String rom = restOfMessage(tok);

        //print the ack to the screen
        System.out.println("ack from " + rom);
        mother.updateVector("ack from " + rom);
    }
    //outgoing message
    else if ( command.equals( "respond" ) ){

        int myPort = portNumbers.getSendPort(threadName);
        //format message
        StringBuffer buff1 = new StringBuffer();
        Buff1.append("net message ");
        buff1.append(threadName);
        buff1.append(" is alive ");

        String response = new String( buff1 );
        //use private line so receiver can get your addrss and port
        network.sendMessage(senderAddress, senderPort, myPort, response);

        if(haveMom)

```

```

        mother.updateVector(response);
    }
    //outgoing
    else if (command.equals( "message" )){
        String contents = restOfMessage(tok);
        System.out.println(contents);
        if (haveMom) mother.updateVector(contents);
    }
    else{
        System.out.println("Network Module cannot process request " +
message);
    }
} //end processPackage(message)

private String restOfMessage(StringTokenizer tok)
{
    StringBuffer buff = new StringBuffer( );

    //get the rest of the request
    while( tok.hasMoreTokens() ){

        buff = buff.append( tok.nextToken() ); //appends the sql statement
        buff = buff.append( " " );
    }

    String result = new String( buff );
    String res = result.trim();

    return res;
}

} //end networkLogic
//*****
// End File:    networkLogic.java
//*****

//*****
// File:    dbUtil.java
// Purpose: A utility class with functions to establish a connection,
//          submit a SQL Statement, execute a query, get database
//          meta data and so on.
//*****
import java.sql.*;
import java.util.*; //for vector and hash

public class dbUtil {

    private Connection con = null;
    private Statement stmt = null;

    private static final boolean debug = false; //for debugging

    //*****
    // Function: bool setConnection(String driver, String url,
    //                               String name, String password)
    // Purpose : constructor which allows user to specify connection
    //*****
    public boolean setConnection(String driver, String url, String name, String
password)
    {

```

```

    try{
        Class.forName(driver);
        con = DriverManager.getConnection(url, name, password);

    }catch(SQLException e){
        System.out.println("Failed to connect to database: " + url + " " +
e.getMessage());
        return false;
    }catch(ClassNotFoundException e){
        System.out.println("Unable to find driver class.");
        return false;
    }
    System.out.println("Connected to Database :" + url);
    return true;
} //end setConnection

//*****
// Function: getConnection(String driver, String url,
//                          String name, String password)
// Purpose : returns a connection object to client
//*****
public Connection getConnection(String driver, String url, String name,
String password)
{
    try{
        Class.forName(driver);
        con = DriverManager.getConnection(url, name, password);
    }catch(SQLException e){
        System.out.println("Failed to connect to database: " + url + " " +
e.getMessage());
        return null;
    }catch(ClassNotFoundException e){
        System.out.println("Unable to find driver class.");
        return null;
    }
    System.out.println("Connected to Database :" + url);
    return con;
} //end setConnection

//*****
// Function: closeConnection(
// Purpose : closes the Connection to the datasource
//*****
public void closeConnection()
{
    try{
        con.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
} //end closeConnection

//*****
// Function: executeSQL(String sql)
// Purpose : dynamically get table data, based upon a sql statment
//          returns a vector with resultSet
// Ref: DataBase Programming with JDBC and JAVA
//*****
public Vector executeSQL(String sql)
{

```

```

Vector resultVector = new Vector(); //to store resultSet by hashTables

int cols;
try{
    stmt = con.createStatement();

    if(stmt.execute(sql)){ //returns true if sql produces a resultSet

        //get the SQL results
        ResultSet result = stmt.getResultSet();

        //get the resultSet metadata
        ResultSetMetaData meta = result.getMetaData();

        //how many columns
        cols = meta.getColumnCount();
        if (debug) System.out.println("Number of columns: " + cols);

        int xx =0;
        //increment through the rows (tuples) of the result set
        while(result.next() ){

            //each tuple gets a hashtable to store information
            Hashtable rowResults = new Hashtable(cols);

            //increment through the tuple <name, ssn, dept>
            for (int ix = 0; ix < cols; ix++){

                //get the object <string, int ect.> stored in column
                //index 1,2,...n and store in an objec
                int index = ix + 1;
                Object obj = result.getObject(index);

                //use the column lable as the hash key and put object
                hashtable
                if(obj == null){
                    rowResults.put(meta.getColumnLabel(index), "");
                } else {
                    rowResults.put(meta.getColumnLabel(index), obj);
                } //end if
            } //end for

            //add the hash object to the vector
            resultVector.addElement(rowResults);
        } //end while

        //System.out.println(resultVector.capacity());

        return resultVector;
    }

    return null; //SQL statement did not produce a ResultSet
}
catch(SQLException e){
    System.err.println("Failed to executeSQL(" + sql + ") function");
    e.printStackTrace();
    return null;
}

} //end executeSQL

//*****

```

```

// Function : printConsole(Vector v)
// Purpose  : Prints the data taken from hash table which
//           is received by vector v
//*****
public void printConsole(Vector v)
{
    if (v == null){
        //do nothing
    }
    else{
        Vector resultVector = v;
        int vectSize = resultVector.size();

        for(int ix = 0; ix < vectSize; ix++) {
            //take the hashtables from the vector one by one
            Hashtable myHashTable = (Hashtable)resultVector.elementAt(ix);
            //enumerate the objects in hashtable
            Enumeration hashEnum = (Enumeration)myHashTable.elements();

            while(hashEnum.hasMoreElements()){
                Object myObj = (Object)hashEnum.nextElement();
                System.out.println(myObj);
            }//end while

        }//end for
    }//end else
}

//*****
// Function: executeSQL(String sql)
// Purpose  : dynamically get table data, based upon a sql statment
//           returns a vector with resultSet
// Ref: DataBase Programming with JDBC and JAVA
//*****
public String executeSQLGetString(String sql)
{
    //System.out.println(" executeSQLGetString Request # " + sqlRequest++);
    String resultString = null;

    try{
        stmt = con.createStatement();

        if(stmt.execute(sql)){ //returns true if sql produces a resultSet

            ResultSet result_set = stmt.getResultSet();
            //put resultSet in string format
            resultString = processResults(result_set);
        }//end if

        return resultString;
    }
    catch(SQLException e){
        System.err.println("Failed to executeSQL(" + sql + ") function");
        e.printStackTrace();
        return null;
    }
}

//*****
// Function: String processResults(ResultSet results)
// Purpose  : processes resultSet into a string
//           : ',' is being used as a delimitator for String Tokenizer

```

```

//*****
public String processResults(ResultSet results) throws SQLException
{
    try {
        ResultSetMetaData meta = results.getMetaData();
        StringBuffer bar = new StringBuffer();
        String buffer = "";
        int cols = meta.getColumnCount();
        int row_count = 0;
        int i, width = 0;

        // Prepare headers for each of the columns
        // The display should look like:
        // -----
        // |   Column One   |   Column Two   |
        // -----
        // |   Row 1 Value  |   Row 1 Value  |
        // -----

        // create the bar that is as long as the total of all columns
        for(i=1; i<=cols; i++) {
            width += meta.getColumnDisplaySize(i);
        } //end for
        width += 1 + cols;
        for(i=0; i<width; i++) {
            bar.append('-');
        } //end for
        bar.append('\n');
        bar.append(","); //my delimitator
        buffer += bar + "|";
        // After the first bar goes the column labels
        for(i=1; i<=cols; i++) {
            StringBuffer filler = new StringBuffer();
            String label = meta.getColumnLabel(i);
            int size = meta.getColumnDisplaySize(i);
            int x;

            // If the label is long than the column is wide,
            // then we truncate the column label
            if( label.length() > size ) {
                label = label.substring(0, size);
            } //end if
            // If the label is shorter than the column, pad it with spaces
            if( label.length() < size ) {
                int j;
                x = (size-label.length())/2;
                for(j=0; j<x; j++) {
                    filler.append(' ');
                } //end for
                label = filler + label + filler;
                if( label.length() > size ) {
                    label = label.substring(0, size);
                }
            } else {
                while( label.length() < size ) {
                    label += " ";
                } //end while
            } //end if
        } //end for
        // Add the column header to the buffer
        buffer = buffer + label + "|";
    } //end for
}

```



```

// Add the lower bar
buffer = buffer + "\n" + "," + bar;
// Format each row in the result set and add it on
while( results.next() ) {
    row_count++;
    buffer += "|";
    // Format each column of the row
    for(i=1; i<=cols; i++) {
        StringBuffer filler = new StringBuffer();
        Object value = results.getObject(i);
        int size = meta.getColumnDisplaySize(i);
        String str = value.toString();
        if( str.length() > size ) {
            str = str.substring(0, size);
        } //end if
        if( str.length() < size ) {
            int j, x;
            x = (size-str.length())/2;
            for(j=0; j<x; j++) {
                filler.append(' ');
            } //end for
            str = filler + str + filler;
            if( str.length() > size ) {
                str = str.substring(0, size);
            }
            else {
                while( str.length() < size ) {
                    str += " ";
                } //end while
            }
        }
        buffer = buffer + str + "|";
    } //end for
    buffer = buffer + "\n" + ",";
} //end while
// Stick a row count up at the top
if( row_count == 0 ) {
    buffer = "No rows selected.\n" + "," + buffer;
}
else if( row_count == 1 ) {
    buffer = "1 row selected.\n" + "," + buffer;
}
else {
    buffer = row_count + " rows selected.\n" + "," + buffer;
}
return buffer;

} catch( SQLException e ) {
    throw e;
} finally {
    try { results.close(); }
    catch( SQLException e ) { }
}
} //end processResults(ResultSet results)

} //end class
//*****
// End File:    dbUtil.java
//*****

```

```

//*****
// File:      networkUtil.java
// Purpose:   Utility package to send requests, listen for requests,
//           send multicast requests and so on.
//*****
import java.util.*; //for Vector
import java.io.*;  //for sleep function
import java.net.*;

public class networkUtil
{
    //*****
    // Function: mcSendListenForResponses(vectorInt mother, String mcAddress,
    //                                     int mcPort, int expReplies, int
    // timer, String message)
    // Purpose:  vectorInt = gui interface, send message to mulicast group,
    //           The function will block waiting for N reponses or until it Times Out
    //*****
    public void mcSendListenForResponses(vectorInt mother, String mcAddress, int
    mcPort, int expReplies, int timer, String message)
    {
        int numReplies = 0; //for acks
        int mcSL_MH = 1;

        //for message
        DatagramSocket socket = null;
        int assignedPort = 0;

        InetAddress mcDA = null;

        byte[] buff = new byte[1024];
        buff = message.getBytes();

        try{
            //os dynamically assigned port
            socket = new DatagramSocket();
            assignedPort = socket.getLocalPort();
            socket.setSoTimeout(2000); //set so socket times out every 2 seconds

            //get group address
            mcDA = InetAddress.getByName(mcAddress);

            //create packet
            DatagramPacket packet = new DatagramPacket(buff, buff.length, mcDA,
    mcPort);

            //send the packet
            socket.send(packet);
        }catch(Exception e){
            System.out.println(e);
        }

        /*** listen for n responses or timeOut ***/
        long startTime = System.currentTimeMillis();
        long elapsedTime = 0;
        long timeOut = timer;

        System.out.println(message + " startTime " + startTime);

        while( (elapsedTime < timeOut) && (numReplies < expReplies) ){

            //creat a inPacket for incoming messages

```

```

byte [] buff_1 = new byte[1024];
DatagramPacket inPacket = new DatagramPacket(buff_1, buff_1.length);
try{
    //blocks for 2 seconds then throwInteruptIOException
    elapsedTime = System.currentTimeMillis() - startTime;
    socket.receive(inPacket); //throws IO Exception

    //create a messageHanlder object using the 2nd constructor that
    //does not increment off the sendPort synchListen
    messageHandlerThread handler = new messageHandlerThread(mother,
inPacket, "mcSL_MH # " + mcSL_MH);
    handler.start();
    mcSL_MH++;
    numReplies++;

}catch (SocketException e){
    System.err.println(e);

}catch (IOException e) {
    elapsedTime = System.currentTimeMillis() - startTime;
    System.out.println("timeOutCheck, elapsedTime | " + elapsedTime);
}
} //end while

if (elapsedTime > timeOut)
    System.out.println(message + " TimedOut");
    mother.updateVector(message + " TimedOut");

    socket.close();
} //end function

//*****
// Function: sendListenForResponses(vectorInt mother, Vector recipVector,
// String message)
// Purpose: user can send a message to N recipiants
// The function will block waiting for N repsonses or until it Times Out
// Object recieves an interface object to mother object
//*****
public void sendListenForResponses(vectorInt mother, String recipiant,
String message, long timeOut)
{

    //for acks
    int numberAcks = 0;
    int expectedAcks = 1;
    //for times
    long startTime = 0;
    long elapsedTime = 0;

    //for this message
    DatagramSocket socket = null;
    int myPort = 0;

    //recipiants socket info
    InetAddress addr = null;
    int port = 0;

    //*** get a socket ***
    try{
        //os dynamically assigned port
        socket = new DatagramSocket();
        myPort = socket.getLocalPort();
    }
}

```

```

        //set socket timeout parameter
        socket.setSoTimeout(2000);
    }catch(SocketException e){
        System.out.println(e);
    }

    //get recipients address and port
    addr = portNumbers.getIP(recipient);
    port      = portNumbers.getListenPort(recipient);

    /*** prep the message ***/
    byte[] buff3 = new byte[1024];
    buff3 = message.getBytes();
    DatagramPacket packet = new DatagramPacket(buff3, buff3.length, addr,
port);

    //send it
    try{
        socket.send(packet);
    }catch(IOException e){
        System.err.println(e);
    }

    /*** listen for n responses or timeOut ***/
    startTime = System.currentTimeMillis();
    System.out.println(message + " startTime " + startTime);

    while( (elapsedTime < timeOut) && (numberAcks < expectedAcks) ){

        //creat a inPacket for incoming messages
        byte [] buff = new byte[1024];
        DatagramPacket inPacket = new DatagramPacket(buff, buff.length);
        try{
            //blocks for 2 seconds then throwInterruptIOException
            elapsedTime = System.currentTimeMillis() - startTime;
            socket.receive(inPacket); //throws IO Exception

            //create a messageHandler object using the 2nd constructor that
            //does not increment off the sendPort synchListen
            messageHandlerThread = new messageHandlerThread(mother,
inPacket, recipient + " Response");
            handler.start();
            numberAcks++;

        }catch (SocketException e){
            System.err.println(e);

        }catch (IOException e) {
            elapsedTime = System.currentTimeMillis() - startTime;
            System.out.println("check for timeOut, elapsedTime | " +
elapsedTime);
        }
    } //end while

    if (elapsedTime > timeOut){
        System.out.println(message + " TimedOut");
        mother.updateVector(message + " TimedOut");
    }
    socket.close();
} //end function

//*****
// Function: sendListenForResponses(Vector recipVector, String message)

```

```

// Purpose: user can send a message to N recipiants
// The function will block waiting for N repsonses or until it Times Out
// NON GUI VERSION
//*****
public void sendListenForResponses(String recipiant, String message, long
timeOut)
{
    //for acks
    int numberAcks = 0;
    int expectedAcks = 1;
    //for times
    long startTime = 0;
    long elapsedTime = 0;

    //for this message
    DatagramSocket socket = null;
    int myPort = 0;

    //recipiants socket info
    InetAddress addr = null;
    int port = 0;

    //*** get a socket ***
    try{
        //os dynamically assigned port
        socket = new DatagramSocket();
        myPort = socket.getLocalPort();
        //set socket timeout parameter
        socket.setSoTimeout(2000);
    }catch(SocketException e){
        System.out.println(e);
    }

    //get recipiants address and port
    addr = portNumbers.getIP(recipiant);
    port = portNumbers.getListenPort(recipiant);

    //*** prep the message ***
    byte[] buff3 = new byte[1024];
    buff3 = message.getBytes();
    DatagramPacket packet = new DatagramPacket(buff3, buff3.length, addr,
port);

    //send it
    try{
        socket.send(packet);
    }catch(IOException e){
        System.err.println(e);
    }

    //*** listen for n responses or timeOut ***
    startTime = System.currentTimeMillis();
    System.out.println(message + " startTime " + startTime);

    while( (elapsedTime < timeOut) && (numberAcks < expectedAcks) ){

        //creat a inPacket for incoming messages
        byte [] buff = new byte[1024];
        DatagramPacket inPacket = new DatagramPacket(buff, buff.length);
        try{
            //blocks for 2 seconds then throwInteruptIOException
            elapsedTime = System.currentTimeMillis() - startTime;
            socket.receive(inPacket); //throws IO Exception

```

```

        //create a messageHanlder object using the 2nd constructor that
        does not increment off the sendPort synchListen
        messageHandlerThread handler = new messageHandlerThread(inPacket,
        recipiant + " Response");
        handler.start();
        numberAcks++;

    }catch (SocketException e){
        System.err.println(e);

    }catch (IOException e) {
        elapsedTime = System.currentTimeMillis() - startTime;
        System.out.println("check for timeOut, elapsedTime | " +
        elapsedTime);
    }
} //end while

if (elapsedTime > timeOut)
    System.out.println(message + " TimedOut");
socket.close();
} //end function

//*****
// Function: sendMessage(String ip, int port, String message)
// Purpose: sends a mutlicast message to specified group
//*****
public void sendMulticastMessage(String ip, int port, int myPort, String
message)
{
    try{

        DatagramSocket socket = new DatagramSocket(myPort); //port 5000

        byte[] buf = new byte[512];
        buf = message.getBytes();

        // send it
        //the group identifier is 230.0.0.1, monitoring port 4446
        InetAddress group = InetAddress.getByName(ip);
        DatagramPacket packet = new DatagramPacket(buf, buf.length, group,
port);
        socket.send(packet);

        //close the socket
        socket.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
} //sendMulticastMessage(String ip, int port, String message)

//*****
// Function: void sendMessage(InetAddress address, int port, String answer)
// Purpose: creates a DatagramSocket, sends a message to the address and
//          port supplied.
//*****
public void sendMessage(InetAddress da, int dp, String message)
{
    try{
        DatagramSocket socket = new DatagramSocket();

```

```

        byte[] buff = new byte[1024];
        buff = message.getBytes();

        //prep the packet
        DatagramPacket packet = new DatagramPacket(buff, buff.length, da, dp);

        //send the packet
        socket.send(packet);

        socket.close();

    }catch(SocketException e){
        System.out.print(e);
    }catch(IOException e){
        System.out.print(e);
    }
}

public void sendMessage(String sDA, int dp, String message)
{
    try{
        InetAddress da = InetAddress.getByName(sDA);
        DatagramSocket socket = new DatagramSocket();

        byte[] buff = new byte[1024];
        buff = message.getBytes();

        //prep the packet
        DatagramPacket packet = new DatagramPacket(buff, buff.length, da, dp);

        //send the packet
        socket.send(packet);

        socket.close();

    }catch(SocketException e){
        System.out.print(e);
    }catch(IOException e){
        System.out.print(e);
    }
}

//*****
// Function: void sendMessage(InetAddress address, int port, String message)
// Purpose: creates a DatagramSocket and send a message
// Parameters: da = destination address, dPort = destination port
//              sPort = port to use to send
//*****
public void sendMessage(InetAddress da, int dPort, int sPort, String
message)
{
    try{
        //create a socket to use
        DatagramSocket responseSocket = new DatagramSocket(sPort);

        byte[] buff = new byte[256];
        buff = message.getBytes();

        //prep the packet

```



```

        else if (unit.equals("l_BnParts") )
            result = InetAddress.getByName(BnIP);
        else
            result = null;
    return result;
}catch(Exception e){
    System.err.println("Could not find ip for " + unit);
    return null;
}
}
}

public static String getName(int senderPort)
{
    if (senderPort == ( a_CoPortL ))
        return "a_CoParts";
    else if (senderPort == ( b_CoPortL ))
        return "b_CoParts";
    else if (senderPort == ( c_CoPortL ))
        return "c_CoParts";

    else return null;

}

public static int getListenPort(String unit)
{
    if (unit.equals("a_CoParts"))
        return a_CoPortL;
    else if ( unit.equals("b_CoParts"))
        return b_CoPortL;
    else if ( unit.equals("c_CoParts"))
        return c_CoPortL;
    else if ( unit.equals("l_BnParts") )
        return BnPortL;
    else return -1;
}

public static int getSendPort(String unit)
{
    if (unit.equals("a_CoParts") )
        return a_CoPortS;
    else if ( unit.equals("b_CoParts"))
        return b_CoPortS;
    else if ( unit.equals("c_CoParts"))
        return c_CoPortS;
    else if (unit.equals("l_BnParts") )
        return BnPortS;
    else return -1;
}

}
//*****
// End File:    portNumbers.java
//*****

```

APPENDIX C. RMI JDBC MODEL

This appendix provides the source code for the RMI implementation. It is organized from the graphical user interface, down through the logic modules as depicted in the following figure.

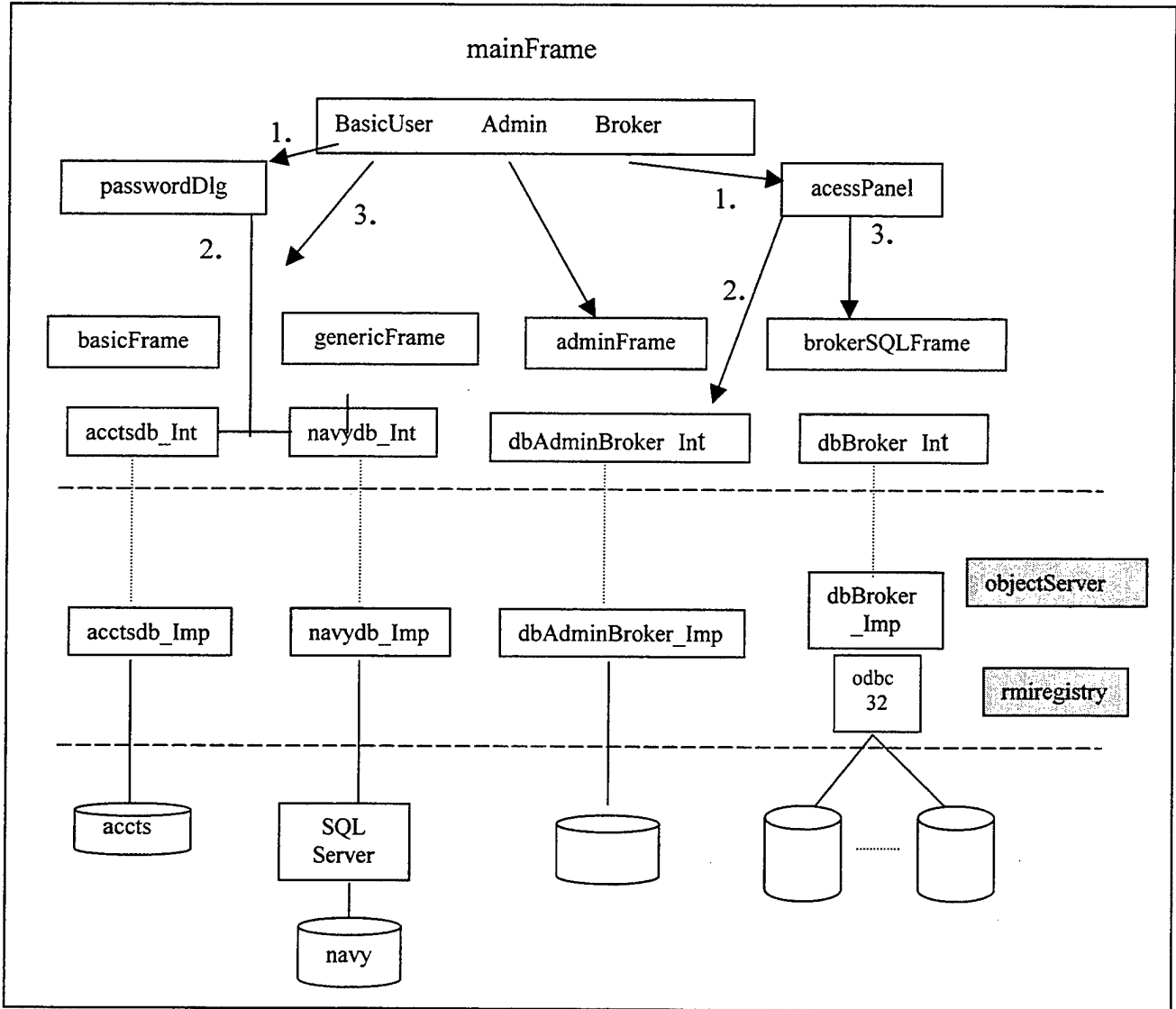


Figure 24 : RMI Object Model

The model is implemented as follows: The `rmregistry` and object server must be started on the remote machine. The object server instantiates and registers the remote objects. The client application consists of a `mainFrame`, and can be started on any machine. The `mainFrame` is an application that has a menu bar with the following options: Basic User, Admin, and Broker.

Under the Basic User option the user can select accounts or navy. A password dialog box is displayed (1). When the user enters his name and information, it calls the remote object, either navy or accounts method `getAccessPolicy(2)` which returns an access policy object. The access policy object defines an access code, which allows the user to get a certain database view (different GUI basic or generic). The intent here is to show that policy can reside on the server, yet be enforced on the client. Based upon the access code the user may get a `genericFrame`, or a `basicFrame(3)`.

The basic frame provides a few high level methods that invoke the remote `acctsdB_Impl` object. A `basicFrame` was not implemented for the navy database. The generic frame allow the user to enter a SQL statement and is more appropriate for a database administrator. Both frames make remote calls through the `accts` or `navy db` implementation objects.

The admin menu option displays the admin Utility Frame. The admin frame invokes remote methods provided by `dbAdminBroker_Impl`. This implementation uses JDBC to maintain a database of user names, passwords and available data sources. When a `datasource` is going to be made available to the organization, the DBA would create an ODBC aliase to the `datasource` (We used the JDBC-ODBC bridge to provide a generic implementation). Then the administrator would enter the `datasource` into the available `databases` table via the admin utility tool/frame. The admin Frame consists of three options: user, access, and admin. The admin panel is used to add `datasources` to the `datasource` list. The user panel allows the administrator to enter a user name and password, and allow that person access, or delete his/her access from the system. The access panel is used to authorize users access to specific `datasources`.

The broker option from the main application, opens an `accessPanel`. After the user enters his name and password, the panel uses the remote `dbAdminBroker_Int` to ensure the client has access to the system and populates a drop down menu with the `datasources` the user has access to. When the user selectes a `datasource`, it creates a `brokerSQLFrame`, which uses the `datasource` selected to invoke the `setConnection` method implemented by `dbBroker_Impl`. The user can then perfrom basic database manipulation functions.

```

//*****
// File:    mainFrame.java
// Purpose: The main window for the application, GUI created
//          by Visual Cafe
//*****

import java.awt.*;
import java.rmi.*; //for RMI
import java.util.*; //for vector

public class mainFrame extends Frame
{
    //remote objects
    static navydbInt navydbServer = null;
    static acctldbInt acctldbServer = null;
    static accessPolicy_Int policy = null;

    //remoteInterface flags
    static boolean remoteAccts = false;
    static boolean remoteNavy = false;
    static int accessCode = -1;

    public mainFrame()
    {
        //only need one security manager,
        System.setSecurityManager(new RMISecurityManager());

        // This code is automatically generated by Visual Cafe

        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 600,insets().top +
insets().bottom + 400);
        openFileDialog = new java.awt.FileDialog(this);
        openFileDialog1.setMode(FileDialog.LOAD);
        openFileDialog1.setTitle("Open");
        //$$ openFileDialog1.move(40,277);
        broker_Panel = new java.awt.Panel();
        broker_Panel.setLayout(null);
        broker_Panel.setVisible(false);
        broker_Panel.setBounds(insets().left + 12,insets().top + 36,578,213);
        add(broker_Panel);
        label1 = new java.awt.Label("User ID:");
        label1.setBounds(144,24,99,27);
        broker_Panel.add(label1);
        name_TF = new java.awt.TextField();
        name_TF.setBounds(252,24,150,25);
        broker_Panel.add(name_TF);
        label2 = new java.awt.Label("Authorized Access To:");
        label2.setBounds(96,156,132,27);
        broker_Panel.add(label2);
        dsn_Choices = new java.awt.Choice();
        broker_Panel.add(dsn_Choices);
        dsn_Choices.setBounds(252,156,135,26);
        password_TF = new java.awt.TextField();
        password_TF.setEchoChar('*');
        password_TF.setBounds(252,60,150,25);

```

```

broker_Panel.add(password_TF);
label3 = new java.awt.Label("User Password:");
label3.setBounds(144,60,96,19);
broker_Panel.add(label3);
submit_Button = new java.awt.Button();
submit_Button.setActionCommand("button");
submit_Button.setLabel("Submit");
submit_Button.setBounds(204,96,75,25);
submit_Button.setBackground(new Color(12632256));
broker_Panel.add(submit_Button);
cancel_Button = new java.awt.Button();
cancel_Button.setActionCommand("button");
cancel_Button.setLabel("Cancel");
cancel_Button.setBounds(324,96,75,25);
cancel_Button.setBackground(new Color(12632256));
broker_Panel.add(cancel_Button);
go_Button = new java.awt.Button();
go_Button.setActionCommand("button");
go_Button.setLabel("GO");
go_Button.setBounds(408,156,40,22);
go_Button.setBackground(new Color(12632256));
broker_Panel.add(go_Button);
statusBar = new java.awt.TextField();
statusBar.setBounds(insets().left + 0,insets().top + 372,600,29);
add(statusBar);
setTitle("JDBC Remote Method Invocation");
//}}

//{{INIT_MENUS
mainMenuBar = new java.awt.MenuBar();
menu1 = new java.awt.Menu("File");
miExit = new java.awt.MenuItem("Exit");
menu1.add(miExit);
mainMenuBar.add(menu1);
DataSources = new java.awt.Menu("Administrator");
admi_AdminTool = new java.awt.MenuItem("Admin Tool");
DataSources.add(admi_AdminTool);
mainMenuBar.add(DataSources);
menu2 = new java.awt.Menu("Basic User");
basicAccts = new java.awt.MenuItem("Accounts");
menu2.add(basicAccts);
basicUserNavy = new java.awt.MenuItem("Naval Group");
menu2.add(basicUserNavy);
mainMenuBar.add(menu2);
m_Broker = new java.awt.Menu("Broker");
broker_DataAccess = new java.awt.MenuItem("Data Access");
m_Broker.add(broker_DataAccess);
mainMenuBar.add(m_Broker);
menu3 = new java.awt.Menu("Help");
mainMenuBar.setHelpMenu(menu3);
miAbout = new java.awt.MenuItem("About..");
menu3.add(miAbout);
mainMenuBar.add(menu3);
setMenuBar(mainMenuBar);
//$$ mainMenuBar.move(4,277);
//}}

//{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
miAbout.addActionListener(lSymAction);
miExit.addActionListener(lSymAction);

```

```

        basicUserNavy.addActionListener(lSymAction);
        basicAccts.addActionListener(lSymAction);
        broker_DataAccess.addActionListener(lSymAction);
        cancel_Button.addActionListener(lSymAction);
        submit_Button.addActionListener(lSymAction);
        go_Button.addActionListener(lSymAction);
        admi_AdminTool.addActionListener(lSymAction);
        //}}
    }

    public mainFrame(String title)
    {
        this();
        setTitle(title);
    }

    public synchronized void show()
    {
        move(50, 50);
        super.show();
    }

    static public void main(String args[])
    {
        (new mainFrame()).show();
    }

    public void addNotify()
    {
        // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();

        super.addNotify();

        if (fComponentsAdjusted)
            return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();
            p.translate(insets().left, insets().top);
            components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
    }

    // Used for addNotify check.
    boolean fComponentsAdjusted = false;

    //{{DECLARE_CONTROLS
    java.awt.FileDialog openFileDialog1;
    java.awt.Panel broker_Panel;
    java.awt.Label label1;
    java.awt.TextField name_TF;
    java.awt.Label label2;
    java.awt.Choice dsn_Choices;
    java.awt.TextField password_TF;

```

```

java.awt.Label label3;
java.awt.Button submit_Button;
java.awt.Button cancel_Button;
java.awt.Button go_Button;
java.awt.TextField statusBar;
//}}

//{{DECLARE_MENUS
java.awt.MenuBar mainMenuBar;
java.awt.Menu menu1;
java.awt.MenuItem miExit;
java.awt.Menu DataSources;
java.awt.MenuItem admi_AdminTool;
java.awt.Menu menu2;
java.awt.MenuItem basicAccts;
java.awt.MenuItem basicUserNavy;
java.awt.Menu m_Broker;
java.awt.MenuItem broker_DataAccess;
java.awt.Menu menu3;
java.awt.MenuItem miAbout;
//}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == mainFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide(); // hide the Frame
    dispose(); // free the system resources
    System.exit(0); // close the application
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == miAbout)
            miAbout_Action(event);
        else if (object == miExit)
            miExit_Action(event);

        else if (object == basicUserNavy)
            navy_Action(event);
        else if (object == basicAccts)
            menuItem1_Action(event);
        else if (object == broker_DataAccess)
            brokerDataAccess_Action(event);
        else if (object == cancel_Button)
            cancelButton_Action(event);
        else if (object == submit_Button)
            submitButton_Action(event);
        else if (object == go_Button)
            goButton_Action(event);
        else if (object == admi_AdminTool)
            admiAdminTool_Action(event);
    }
}

```

```

    }
}

void miAbout_Action(java.awt.event.ActionEvent event)
{
    //{{CONNECTION
    // Action from About Create and show as modal
    (new AboutDialog(this, true)).show();
    //}}
}

void miExit_Action(java.awt.event.ActionEvent event)
{
    //{{CONNECTION
    // Action from Exit Create and show as modal
    (new QuitDialog(this, true)).show();
    //}}
}

void miOpen_Action(java.awt.event.ActionEvent event)
{
    //{{CONNECTION
    // Action from Open... Show the OpenFileDialog
    openFileDialog1.show();
    //}}
}

//dba selected accounts
void acctsDB_Action(java.awt.event.ActionEvent event)
{
    // Create and show the Frame
    (new genericSQLFrame("accounts")).show();
}

//dba selected navy
void navyDB_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Create and show the Frame
    (new genericSQLFrame("navy")).show();
    //}}
}

//basic user navy
void navy_Action(java.awt.event.ActionEvent event)
{
    //bound to navy server
    remoteNavy = true;
    bindRMIObject();

    //{{CONNECTION
    // Create and show as modal
    (new PasswordDialog(this, true)).show();
    //}}
}

//basic user accounts
void menuItem1_Action(java.awt.event.ActionEvent event)
{
    remoteAccts = true;
    bindRMIObject();
    PasswordDialog passDlg = new PasswordDialog(this, true);
}

```



```

        passDlg.show();

        //open a new basic userframe
        // Create and show the Frame
        (new basicAcctFrame(acctsdbServer)).show();
        //}}
    }

    public static void lookup(String user, String pw)
    {
        //get the policy
        if(remoteNavy){
            try{
                policy = navydbServer.getAccessPolicy();
            }catch(RemoteException e){
                System.out.println("Remote Exception: " + e);
            }
        }
        else if(remoteAccts){
            try{
                policy = acctsdbServer.getAccessPolicy();
            }catch(RemoteException e){
                System.out.println("Remote Exception: " + e);
            }
        }

        //get the access code
        accessCode = policy.getAccessCode(user);
        //test the policy
        System.out.println(user + " Access Code = " + accessCode);

        if (accessCode == 1){
            if(remoteNavy)
                (new genericSQLFrame("navy")).show();
            else if(remoteAccts)
                (new genericSQLFrame("accounts")).show();

        }else if(accessCode == 2){

            //show a dialog box stating not authorized.
            System.out.println("To Do");
        }else
            System.out.println("Not a valid Access Code");
    }

    //*****
    // Function : bindRMIObject()
    // Purpose:   Binds client to remote object, allowing client
    //            to execute methods defined in the objects interface
    //*****
    public void bindRMIObject()
    {
        try{

            if(remoteNavy){
                System.out.println("Binding to Remote Navy Data Base...");
                navydbServer =
                (navydbInt)Naming.lookup("rmi://131.120.1.91/navydbServer");
            }else if(remoteAccts){
                System.out.println("Binding to Remote Accounts Data Base...");
            }
        }
    }

```



```

        name_TF.setText("");
        password_TF.setText("");
        //}}
    }

void submitButton_Action(java.awt.event.ActionEvent event)
{
    //get user input
    String name = name_TF.getText();
    String password = password_TF.getText();

    //call adminBrokerServer to get database access list
    Vector ans = new Vector();

    try{
        ans = adminBrokerServer.getDBAccess(name);
    }catch(RemoteException e){
        System.out.println(e);
    }
    if(!ans.isEmpty() ){
        Enumeration enum = ans.elements();
        while(enum.hasMoreElements() ){
            String dsn = (String)enum.nextElement();
            dsn_Choices.addItem(dsn);
        }//end while
    }else
        statusBar.setText("User is not Authorized Access to any DataBase");

    //
    //{{CONNECTION
    // Hide the Panel
    //broker_Panel.setVisible(false);
    //}}
}

//user has selected a database to go to
void goButton_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    String choice = dsn_Choices.getSelectedItem();

    //create a genericFrame which will bind to dbBrokerServer
    //{{CONNECTION
    // Create and show the Frame
    (new brokerSQLFrame(choice)).show();
    //}}

    //clear text fields
    name_TF.setText("");
    password_TF.setText("");
    dsn_Choices.removeAll();
    //{{CONNECTION
    // Hide the Panel
    broker_Panel.setVisible(false);
    //}}

}

void admiAdminTool_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

```

```

        //{{CONNECTION
        // Create and show the Frame
        (new Frame1()).show();
        //}}
    }
}
//*****
//end mainFrame.java
//*****

//*****
// File:    basicAcctFrame.java
// Purpose: A simple frame, that allows client ability to
//          enter a new employee into the accts database,
//          or retrieve a list of existing employees
//          GUI implemented with Visual Cafe
//*****
import java.rmi.*;
import java.awt.*;
import java.util.*; //stringtokenizer

public class basicAcctFrame extends Frame
{
    acctsdbInt server = null;
    public basicAcctFrame()
    {
        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 430,insets().top +
insets().bottom + 343);
        newEmployeePanel = new java.awt.Panel();
        newEmployeePanel.setLayout(null);
        newEmployeePanel.setVisible(false);
        newEmployeePanel.setBounds(insets().left + 108,insets().top +
36,228,144);
        add(newEmployeePanel);
        tnameTextField = new java.awt.TextField();
        tnameTextField.setBounds(60,12,120,19);
        newEmployeePanel.add(tnameTextField);
        Group1 = new CheckboxGroup();
        facultyRadioButton = new java.awt.Checkbox("", Group1, false);
        facultyRadioButton.setBounds(72,36,24,16);
        newEmployeePanel.add(facultyRadioButton);
        label1 = new java.awt.Label("Name");
        label1.setBounds(12,12,82,20);
        newEmployeePanel.add(label1);
        label2 = new java.awt.Label("Faculty");
        label2.setBounds(12,36,82,20);
        newEmployeePanel.add(label2);
        insertButton = new java.awt.Button();
        insertButton.setActionCommand("button");
        insertButton.setLabel("Insert");
        insertButton.setBounds(132,84,88,23);
        insertButton.setBackground(new Color(12632256));
        newEmployeePanel.add(insertButton);
    }
}

```

```

cancelButton = new java.awt.Button();
cancelButton.setActionCommand("button");
cancelButton.setLabel("Cancel");
cancelButton.setBounds(36,84,88,23);
cancelButton.setBackground(new Color(12632256));
newEmployeePanel.add(cancelButton);
employeeListPanel = new java.awt.Panel();
employeeListPanel.setLayout(null);
employeeListPanel.setVisible(false);
employeeListPanel.setBounds(insets().left + 0,insets().top + 36,415,216);
add(employeeListPanel);
employeeViewList = new java.awt.List(0,false);
employeeListPanel.add(employeeViewList);
employeeViewList.setBounds(36,0,314,160);
okButton = new java.awt.Button();
okButton.setActionCommand("button");
okButton.setLabel("OK");
okButton.setBounds(156,168,62,27);
okButton.setBackground(new Color(12632256));
employeeListPanel.add(okButton);
setTitle("Basic User Account Frame");
//}}

//{{{INIT_MENUS
menuBar1 = new java.awt.MenuBar();
file = new java.awt.Menu("File");
exit = new java.awt.MenuItem("Exit");
file.add(exit);
menuBar1.add(file);
menu1 = new java.awt.Menu("Views");
employeeView = new java.awt.MenuItem("Employees");
menu1.add(employeeView);
menuBar1.add(menu1);
insert = new java.awt.Menu("Insert");
Employee = new java.awt.MenuItem("Employee");
insert.add(Employee);
menuBar1.add(insert);
setMenuBar(menuBar1);
//$$ menuBar1.move(0,0);
//}}

//{{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
exit.addActionListener(lSymAction);
Employee.addActionListener(lSymAction);
insertButton.addActionListener(lSymAction);
employeeView.addActionListener(lSymAction);
okButton.addActionListener(lSymAction);
cancelButton.addActionListener(lSymAction);
//}}
}

public basicAcctFrame(acctsdbInt server)
{
    this();
    this.server = server;
}

public basicAcctFrame(String title)
{
    this();

```

```

        setTitle(title);
    }

    public synchronized void show()
    {
        move(50, 50);
        super.show();
    }

    public void addNotify()
    {
        // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();

        super.addNotify();

        if (fComponentsAdjusted)
            return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();
            p.translate(insets().left, insets().top);
            components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
    }

    // Used for addNotify check.
    boolean fComponentsAdjusted = false;

    /**{DECLARE_CONTROLS
    java.awt.Panel newEmployeePanel;
    java.awt.TextField tnameTextField;
    java.awt.Checkbox facultyRadioButton;
    CheckboxGroup Group1;
    java.awt.Label label1;
    java.awt.Label label2;
    java.awt.Button insertButton;
    java.awt.Button cancelButton;
    java.awt.Panel employeeListPanel;
    java.awt.List employeeViewList;
    java.awt.Button okButton;
    //}}

    /**{DECLARE_MENUS
    java.awt.MenuBar menuBar1;
    java.awt.Menu file;
    java.awt.MenuItem exit;
    java.awt.Menu menu1;
    java.awt.MenuItem employeeView;
    java.awt.Menu insert;
    java.awt.MenuItem Employee;
    //}}

    class SymWindow extends java.awt.event.WindowAdapter
    {
        public void windowClosing(java.awt.event.WindowEvent event)
        {

```

```

        Object object = event.getSource();
        if (object == basicAcctFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide();        // hide the Frame
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == exit)
            exit_Action(event);
        else if (object == Employee)
            Employee_Action(event);
        else if (object == insertButton)
            insertButton_Action(event);
        else if (object == employeeView)
            employeeView_Action(event);
        else if (object == okButton)
            okButton_Action(event);
        else if (object == cancelButton)
            cancelButton_Action(event);
    }
}

void exit_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Hide the Frame
    setVisible(false);
    //}}
}

void Employee_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    //{{CONNECTION
    // Show the Panel
    newEmployeePanel.setVisible(true);
    //}}
}

void insertButton_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    String name = tnameTextField.getText();
    boolean fac = facultyRadioButton.getState();
    System.out.println(name + " " + fac);
    if(name.equals("")){
        return;
    }
    try{
        server.insertEmployee(name, fac);
    }catch(RemoteException e){

```

```

        System.out.println(e);
    }
    //{{CONNECTION
    // Hide the Panel
    newEmployeePanel.setVisible(false);
    //}}
}

void employeeView_Action(java.awt.event.ActionEvent event)
{
    String result = null;
    //get results
    try{
        result = server.viewEmployees();
    }catch(RemoteException e){
        System.out.println(e);
    }

    //process the results
    //print one tuple per line using new line as delimiter
    StringTokenizer tok = new StringTokenizer(result, "\n");

    if(result == null){
        System.out.println("No results found");
        employeeViewList.addItem("No results were generated");
    }else{
        int count = tok.countTokens();
        for (int ix = 1; ix <= count; ix++){
            String tuple = tok.nextToken();
            employeeViewList.addItem(tuple);
        }//end for
    }//end else

    //{{CONNECTION
    // Show the Panel
    employeeListPanel.setVisible(true);
    //}}
}

void okButton_Action(java.awt.event.ActionEvent event)
{
    // Hide the Panel
    employeeListPanel.setVisible(false);
}

void cancelButton_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    newEmployeePanel.setVisible(false);
}
}
//*****
// End File:    basicAcctFrame.java
//*****

//*****
// File:    genericSQLFrame.java
// Purpose: A generic frame that displays the database table
//          names, and column Names. User enters a SQL query
//          and frame displays result set. Constructor accepts
//          datasource name, to bind to.
//          GUI created by Visual Cafe.

```



```

//*****

import java.awt.*;
import java.rmi.*; //for remote object binding
import java.sql.*; //for resultSet manipulation
import java.util.*; //for Vector
import java.io.*; //for sleep function

import symantec.itools.awt.BorderPanel;
public class genericSQLFrame extends Frame
{
    //class scope usage
    dbUtil dbUtilities = null; //for print functions
    //remote objects
    navydbInt navydbServer = null;
    accessPolicy_Int currentAccessPolicy = null;
    acctsdInt acctsdServer = null;

    //remoteInterface flags
    boolean accts = false;
    boolean navy = false;

    static StringBuffer buff = null;

    public genericSQLFrame()
    {
        //MY CODE
        //create a global dbUtilities tool
        dbUtilities = new dbUtil();
        buff = new StringBuffer();

        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 650,insets().top +
insets().bottom + 500);
        panell = new java.awt.Panel();
        panell.setLayout(null);
        panell.setBounds(insets().left + 0,insets().top + 36,360,96);
        add(panell);
        tableNames = new java.awt.Choice();
        panell.add(tableNames);
        tableNames.setBounds(25,40,150,25);
        db = new java.awt.Label("Data Base Tables");
        db.setBounds(48,0,106,20);
        panell.add(db);
        label2 = new java.awt.Label("Table Column Names");
        label2.setBounds(192,0,132,23);
        panell.add(label2);
        columnChoices = new java.awt.Choice();
        panell.add(columnChoices);
        columnChoices.setBounds(192,40,150,25);
        resultSet = new java.awt.List(0,false);
        add(resultSet);
        resultSet.setBounds(insets().left + 36,insets().top + 168,588,169);
        clearList = new java.awt.Button();
        clearList.setActionCommand("button");
    }
}

```

```

clearList.setLabel("Clear List");
clearList.setBounds(insets().left + 312,insets().top + 360,77,27);
clearList.setBackground(new Color(12632256));
add(clearList);
panel2 = new java.awt.Panel();
panel2.setLayout(null);
panel2.setBounds(insets().left + 360,insets().top + 36,273,108);
add(panel2);
label1 = new java.awt.Label("Enter SQL Statement Below");
label1.setBounds(24,0,168,32);
label1.setForeground(new Color(255));
panel2.add(label1);
sqlStmt = new java.awt.TextField();
sqlStmt.setBounds(0,36,264,25);
panel2.add(sqlStmt);
sendSQL = new java.awt.Button();
sendSQL.setActionCommand("button");
sendSQL.setLabel("Submit");
sendSQL.setBounds(168,72,76,28);
sendSQL.setBackground(new Color(12632256));
panel2.add(sendSQL);
clearSQL = new java.awt.Button();
clearSQL.setActionCommand("button");
clearSQL.setLabel("Clear");
clearSQL.setBounds(36,72,85,27);
clearSQL.setBackground(new Color(12632256));
panel2.add(clearSQL);
label3 = new java.awt.Label("Result Set");
label3.setBounds(insets().left + 24,insets().top + 132,96,24);
add(label3);
exitButton = new java.awt.Button();
exitButton.setActionCommand("button");
exitButton.setLabel("Exit");
exitButton.setBounds(insets().left + 312,insets().top + 408,72,23);
exitButton.setBackground(new Color(16711680));
add(exitButton);
accessCodeTextField = new java.awt.TextField();
accessCodeTextField.setBounds(insets().left + 48,insets().top +
384,68,26);
add(accessCodeTextField);
label4 = new java.awt.Label("Access Code");
label4.setBounds(insets().left + 48,insets().top + 360,84,20);
add(label4);
setTitle("Navy Data Source");
//}}

//{{INIT_MENUS
menuBar1 = new java.awt.MenuBar();
menu1 = new java.awt.Menu("File");
fileExit = new java.awt.MenuItem("Exit");
menu1.add(fileExit);
menuBar1.add(menu1);
menu2 = new java.awt.Menu("Policy");
menuItem1 = new java.awt.MenuItem("Get Current");
menu2.add(menuItem1);
menuBar1.add(menu2);
menu3 = new java.awt.Menu("Users");
userAdd = new java.awt.MenuItem("Add User");
menu3.add(userAdd);
menuBar1.add(menu3);
setMenuBar(menuBar1);
//$$ menuBar1.move(0,0);
//}}

```

```

//{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
sendSQL.addActionListener(lSymAction);
clearList.addActionListener(lSymAction);
SymItem lSymItem = new SymItem();
tableNames.addItemListener(lSymItem);
clearSQL.addActionListener(lSymAction);
exitButton.addActionListener(lSymAction);
fileExit.addActionListener(lSymAction);
menuItem1.addActionListener(lSymAction);
userAdd.addActionListener(lSymAction);
//}}

}

//*****
// Function: genericSQLFrame(String dsn)
// Purpose: Constructor, uses dsn to bind to remote object,
//          get the database table names, and update frame title
//*****
public genericSQLFrame(String dsn)
{
    this();

    if( dsn.equals("navy")){
        setTitle("Naval Support Group Data Base");
        this.navy = true;
    }
    else if( dsn.equals("accounts")){
        setTitle("Accounts Data Base");
        this.accts = true;
    }
    //bind to remote object
    bindRMIObject();

    //update database table drop down
    getDataBaseTables();
}

//*****
// Function: getDataBaseTables(String remoteObj)
// Purpose: Based upon remoteObj communicating with
//*****
public void getDataBaseTables()
{
    Vector ans = new Vector();
    try{
        if(navy){
            ans = navydbServer.getTableNames();
        }
        else if(accts) {
            ans = acctsdbServer.getTableNames();
        }
    }catch(RemoteException e){
        System.err.println(e);
    }
}

```

```

        Enumeration enum = ans.elements();
        while(enum.hasMoreElements() ){
            String name = (String)enum.nextElement();
            //update the tableNames
            tableNames.addItem(name);
        }

    }

}

//end getDataBaseTables

public void updateColumnList(String tableName)
{
    columnChoices.removeAll(); //clear the column choice object

    if(tableName != null){
        StringBuffer buff = new StringBuffer("SELECT * FROM ");
        buff.append(tableName);

        String sql = new String(buff);
        Vector colVect = new Vector();
        try{
            if(navy){
                colVect = navydbServer.getTableMetaData(sql);
            }else if (accts){
                colVect = acctsdbServer.getTableMetaData(sql);
            }
        }catch(RemoteException e){
            System.out.println(e);
        }
        //now update colChoiceBox
        Enumeration enum = colVect.elements();
        while(enum.hasMoreElements() ){
            String name = (String)enum.nextElement();
            //update the tableNames
            columnChoices.addItem(name);
        }
    }

}

}

//end function

/*****
// Function : bindRMIObject()
// Purpose:   Binds client to remote object, allowing client
//            to execute methods defined in the objects interface
*****/
public void bindRMIObject()
{
    try{

        if(navy){
            System.out.println("Binding to Remote Navy Data Base...");
            navydbServer =
(navydbInt)Naming.lookup("rmi://131.120.1.91/navydbServer");
        }else if(accts){
            System.out.println("Binding to Remote Accounts Data Base...");
            acctsdbServer =
(acctsdbInt)Naming.lookup("rmi://131.120.1.91/acctsdbServer");
        }

    } catch(NotBoundException e){
        System.out.println("NotBoundException " + e);
    }

    }catch(RemoteException e){
        System.out.println("Remote Exception " + e);
    }
}

```



```

java.awt.Menu menu2;
java.awt.MenuItem menuItem1;
java.awt.Menu menu3;
java.awt.MenuItem userAdd;
//}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == genericSQLFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide();        // hide the Frame
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == sendSQL)
            sendSQL_Action(event);
        else if (object == clearList)
            clearList_Action(event);
        else if (object == clearSQL)
            clearSQL_Action(event);
        else if (object == exitButton)
            exitButton_Action(event);
        else if (object == fileExit)
            fileExit_Action(event);
        else if (object == menuItem1)
            menuItem1_Action(event);
        else if (object == userAdd)
            userAdd_Action(event);
    }
}

//*****
// Function : sendSQL_Action
// Purpose:  User has entered a SQL statement and hit the send button
//           Gets the result set and displays it in the result list
//*****
void sendSQL_Action(java.awt.event.ActionEvent event)
{
    String sql = sqlStmt.getText();//get the statement

    try{
        String result = new String();

        if(navy){
            result = navydbServer.executeSQLGetString(sql);
        }else if(accts){
            result = acctsdbServer.executeSQLGetString(sql);
        }

        //print one tuple per line using new line as delimiter
        StringTokenizer tok = new StringTokenizer(result, "\n");

```

```

        if(result == null){
            System.out.println("No results found");
        }else{
            int count = tok.countTokens();
            for (int ix = 1; ix <= count; ix++){
                String tuple = tok.nextToken();
                resultSet.addItem(tuple);
            }//end for
        }//end else

    }catch (RemoteException e){
        System.out.println(e);
    }
    //clear the sql statement text box
    sqlStmt.setText("");

} //end func

void clearList_Action(java.awt.event.ActionEvent event)
{
    resultSet.removeAll();
} //end clearList

class SymItem implements java.awt.event.ItemListener
{
    public void itemStateChanged(java.awt.event.ItemEvent event)
    {
        Object object = event.getSource();
        if (object == tableNames)
            tableNames_ItemStateChanged(event);
    }
}

//*****
// Function: tableNames Action Handler
// Gets the item selected from choice box and updates teh column choice box
//*****
void tableNames_ItemStateChanged(java.awt.event.ItemEvent event)
{
    String item = tableNames.getSelectedItem();
    updateColumnList(item);
}

void clearSQL_Action(java.awt.event.ActionEvent event)
{
    // Clear the text for TextField
    sqlStmt.setText("");
}

//*****
// Function: exit_Button_Action
//*****
void exitButton_Action(java.awt.event.ActionEvent event)
{
    if(navy)
        mainFrame.remoteNavy = false;
    else if(accts)
        mainFrame.remoteAccts = false;
}

```



```

//*****
// File:      adminFrame.java
// Purpose: An administrator frame, to enter new users,
//           provide them database access, and to enter
//           new datasources.
//*****

import java.awt.*;

public class adminFrame extends Frame
{
    public adminFrame()
    {
        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setSize(insets().left + insets().right + 430,insets().top +
insets().bottom + 270);
        setTitle("Untitled");
        //}}

        //{{INIT_MENUS
        //}}

        //{{REGISTER_LISTENERS
        SymWindow aSymWindow = new SymWindow();
        this.addWindowListener(aSymWindow);
        //}}
    }

    public adminFrame(String title)
    {
        this();
        setTitle(title);
    }

    public synchronized void show()
    {
        move(50, 50);
        super.show();
    }

    public void addNotify()
    {
        // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();

        super.addNotify();

        if (fComponentsAdjusted)
            return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();

```

```

        p.translate(insets().left, insets().top);
        components[i].setLocation(p);
    }
    fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//{{{DECLARE_CONTROLS
//}}}

//{{{DECLARE_MENUS
//}}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == adminFrame.this)
            Frame1_WindowClosing(event);
    }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide();    // hide the Frame
}
}
//*****
// END:    adminFrame.java
//*****

//*****
// File:    brokersSQLFrame.java
// Purpose: A generic frame that can be bound to any ODBC datasource.
//          The constructor accepts a string which is used to establish
//          JDBC-ODBC connection, via the dbBrokerImplementation
//*****

import java.awt.*;
import java.rmi.*; //for remote object binding
import java.sql.*; //for resultSet manipulation
import java.util.*; //for Vector
import java.io.*; //for sleep function
import symantec.itools.awt.BorderPanel;

public class brokersSQLFrame extends Frame
{
    //class scope usage
    dbUtil dbUtilities = null; //for print functions
    //remote objects
    dbBroker_Int server = null;

    public brokersSQLFrame()
    {
        //MY CODE
        //create a global dbUtilities tool
        dbUtilities = new dbUtil();
    }
}

```

```

// This code is automatically generated by Visual Cafe when you add
// components to the visual environment. It instantiates and initializes
// the components. To modify the code, only use code syntax that matches
// what Visual Cafe can generate, or Visual Cafe may be unable to back
// parse your Java file into its visual environment.
//{{INIT_CONTROLS
setLayout(null);
setVisible(false);
setSize(insets().left + insets().right + 650,insets().top +
insets().bottom + 500);
panell = new java.awt.Panel();
panell.setLayout(null);
panell.setBounds(insets().left + 0,insets().top + 36,360,96);
add(panell);
tableNames = new java.awt.Choice();
panell.add(tableNames);
tableNames.setBounds(25,40,150,25);
db = new java.awt.Label("Data Base Tables");
db.setBounds(48,0,106,20);
panell.add(db);
label2 = new java.awt.Label("Table Column Names");
label2.setBounds(192,0,132,23);
panell.add(label2);
columnChoices = new java.awt.Choice();
panell.add(columnChoices);
columnChoices.setBounds(192,40,150,25);
resultSet = new java.awt.List(0,false);
add(resultSet);
resultSet.setBounds(insets().left + 36,insets().top + 168,588,169);
clearList = new java.awt.Button();
clearList.setActionCommand("button");
clearList.setLabel("Clear List");
clearList.setBounds(insets().left + 312,insets().top + 360,77,27);
clearList.setBackground(new Color(12632256));
add(clearList);
panel2 = new java.awt.Panel();
panel2.setLayout(null);
panel2.setBounds(insets().left + 360,insets().top + 36,273,108);
add(panel2);
label1 = new java.awt.Label("Enter SQL Statement Below");
label1.setBounds(24,0,168,32);
label1.setForeground(new Color(255));
panel2.add(label1);
sqlStmt = new java.awt.TextField();
sqlStmt.setBounds(0,36,264,25);
panel2.add(sqlStmt);
sendSQL = new java.awt.Button();
sendSQL.setActionCommand("button");
sendSQL.setLabel("Submit");
sendSQL.setBounds(168,72,76,28);
sendSQL.setBackground(new Color(12632256));
panel2.add(sendSQL);
clearSQL = new java.awt.Button();
clearSQL.setActionCommand("button");
clearSQL.setLabel("Clear");
clearSQL.setBounds(36,72,85,27);
clearSQL.setBackground(new Color(12632256));
panel2.add(clearSQL);
label3 = new java.awt.Label("Result Set");
label3.setBounds(insets().left + 24,insets().top + 132,96,24);
add(label3);

```

```

exitButton = new java.awt.Button();
exitButton.setActionCommand("button");
exitButton.setLabel("Exit");
exitButton.setBounds(insets().left + 312,insets().top + 408,72,23);
exitButton.setBackground(new Color(16711680));
add(exitButton);
setTitle("Navy Data Source");
//}}

//{{{INIT_MENU
menuBar1 = new java.awt.MenuBar();
menu = new java.awt.Menu("File");
fileExit = new java.awt.MenuItem("Exit");
menu1.add(fileExit);
menuBar1.add(menu1);
setMenuBar(menuBar1);
//$$ menuBar1.move(0,0);
//}}

//{{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
sendSQL.addActionListener(lSymAction);
clearList.addActionListener(lSymAction);
SymItem lSymItem = new SymItem();
tableNames.addItemListener(lSymItem);
clearSQL.addActionListener(lSymAction);
exitButton.addActionListener(lSymAction);
fileExit.addActionListener(lSymAction);
//}}

}

//*****
// Function: genericSQLFrame(String dsn)
// Purpose: Constructor, uses dsn to bind to remote object,
//          get the database table names, and update frame title
//*****
public brokerSQLFrame(String dsn)
{
    this();

    setTitle(dsn);

    //bind to remote object
    try{
        System.out.println("Binding to dbBroker for " + dsn);
        server =
(dbBroker_Int)Naming.lookup("rmi://131.120.1.91/dbBrokerServer");

    } catch (NotBoundException e){
        System.out.println("NotBoundException" + e);

    }catch (RemoteException e){
        System.out.println("Remote Exception " + e);

    }catch (java.net.MalformedURLException e){
        System.out.println(e);
    }

    //create a connection
    try{

```



```

public synchronized void show()
{
    move(50, 50);
    super.show();
}

public void addNotify()
{
    // Record the size of the window prior to calling parents addNotify.
    Dimension d = getSize();

    super.addNotify();

    if (fComponentsAdjusted)
        return;

    // Adjust components according to the insets
    setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
    Component components[] = getComponents();
    for (int i = 0; i < components.length; i++)
    {
        Point p = components[i].getLocation();
        p.translate(insets().left, insets().top);
        components[i].setLocation(p);
    }
    fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//{{{DECLARE_CONTROLS
java.awt.Panel panel1;
java.awt.Choice tableNames;
java.awt.Label db;
java.awt.Label label2;
java.awt.Choice columnChoices;
java.awt.List resultSet;
java.awt.Button clearList;
java.awt.Panel panel2;
java.awt.Label label1;
java.awt.TextField sqlStmt;
java.awt.Button sendSQL;
java.awt.Button clearSQL;
java.awt.Label label3;
java.awt.Button exitButton;
//}}}

//{{{DECLARE_MENUS
java.awt.MenuBar menuBar1;
java.awt.Menu menu1;
java.awt.MenuItem fileExit;
//}}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == brokerSQLFrame.this)
            Frame1_WindowClosing(event);
    }
}

```

```

}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
    hide();        // hide the Frame
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == sendSQL)
            sendSQL_Action(event);
        else if (object == clearList)
            clearList_Action(event);
        else if (object == clearSQL)
            clearSQL_Action(event);
        else if (object == exitButton)
            exitButton_Action(event);
        else if (object == fileExit)
            fileExit_Action(event);
    }
}

//*****
// Function : sendSQL_Action
// Purpose:   User has entered a SQL statement and hit the send button
//           Gets the result set and displays it in the result list
//*****
void sendSQL_Action(java.awt.event.ActionEvent event)
{
    String sql = sqlStmt.getText();//get the statement

    try{
        String result = new String();

        result = server.executeSQLGetString(sql);

        //print one tuple per line using new line as delimiter
        StringTokenizer tok = new StringTokenizer(result, "\n");

        if(result == null){
            System.out.println("No results found");
        }else{
            int count = tok.countTokens();
            for (int ix = 1; ix <= count; ix++){
                String tuple = tok.nextToken();
                resultSet.addItem(tuple);
            }//end for
        }//end else

    }catch(RemoteException e){
        System.out.println(e);
    }
    //clear the sql statement text box
    sqlStmt.setText("");
}

//end func

//*****
// Function: clearList_Action(java.awt.event.ActionEvent event)

```

```

// Purpose:
//*****
void clearList_Action(java.awt.event.ActionEvent event)
{
    resultSet.removeAll();
} //end clearList

class SymItem implements java.awt.event.ItemListener
{
    public void itemStateChanged(java.awt.event.ItemEvent event)
    {
        Object object = event.getSource();
        if (object == tableNames)
            tableNames_ItemStateChanged(event);
    }
}
//*****
// Function: tableNames Action Handler
// Gets the item selected from choice box and updates teh column choice box
//*****
void tableNames_ItemStateChanged(java.awt.event.ItemEvent event)
{
    String item = tableNames.getSelectedItem();
    updateColumnList(item);
}

void clearSQL_Action(java.awt.event.ActionEvent event)
{
    // Clear the text for TextField
    sqlStmt.setText("");
}

//*****
// Function: exit_Button_Action
//*****
void exitButton_Action(java.awt.event.ActionEvent event)
{
    hide(); // hide the Frame
    dispose(); // free the system resources
}

void fileExit_Action(java.awt.event.ActionEvent event)
{
    hide(); // hide the Frame
    dispose(); // free the system resources
}
}
//*****
// END: brokerSQLFrame.java
//*****

//*****
// File: newUser.java
// Purpose: A dialog box to enter a new users name and password,
// Calls back to static function in genericSQLFrame
//*****

```



```

import java.awt.*;
import symantec.itools.awt.util.dialog.ModalDialog;

public class newUser extends ModalDialog
{
    public newUser(Frame parent, String title)
    {
        super(parent, title);

        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.
        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 215,insets().top +
insets().bottom + 159);
        setBackground(new Color(12632256));
        nameLabel = new java.awt.Label("Name:");
        nameLabel.setBounds(insets().left + 12,insets().top + 12,60,15);
        add(nameLabel);
        passwordLabel = new java.awt.Label("Password:");
        passwordLabel.setBounds(insets().left + 12,insets().top + 48,72,15);
        add(passwordLabel);
        userTextField = new java.awt.TextField(1);
        userTextField.setBounds(insets().left + 84,insets().top + 12,100,22);
        add(userTextField);
        passTextField = new java.awt.TextField(1);
        passTextField.setEchoChar('*');
        passTextField.setBounds(insets().left + 84,insets().top + 48,100,22);
        add(passTextField);
        okButton = new java.awt.Button();
        okButton.setLabel("OK");
        okButton.setBounds(insets().left + 84,insets().top + 132,40,20);
        add(okButton);
        dsnTextField = new java.awt.TextField();
        dsnTextField.setBounds(insets().left + 84,insets().top + 84,96,21);
        add(dsnTextField);
        labell = new java.awt.Label("Data Source");
        labell.setBounds(insets().left + 0,insets().top + 84,72,15);
        add(labell);
        setTitle("Add a DataBase User");
        //}}

        //{{REGISTER_LISTENERS
        SymAction lSymAction = new SymAction();
        okButton.addActionListener(lSymAction);
        //}}
    }

    public newUser(Frame parent)
    {
        this(parent, "Username/Password");
    }

    // Add a constructor for Interactions (ignoring modal)
    public newUser(Frame parent, boolean modal)
    {
        this(parent);
    }
}

```

```

}

// Add a constructor for Interactions (ignoring modal)
public newUser(Frame parent, String title, boolean modal)
{
    this(parent, title);
}

public String getUsername()
{
    return userTextField.getText();
}

public String getPassword()
{
    return passTextField.getText();
}

public void setUsername(String name)
{
    userTextField.setText(name);
}

public void setPassword(String pass)
{
    passTextField.setText(pass);
}

public void addNotify()
{
    // Record the size of the window prior to calling parents addNotify.
    Dimension d = getSize();

    super.addNotify();

    if (fComponentsAdjusted)
        return;

    // Adjust components according to the insets
    setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
    Component components[] = getComponents();
    for (int i = 0; i < components.length; i++)
    {
        Point p = components[i].getLocation();
        p.translate(insets().left, insets().top);
        components[i].setLocation(p);
    }
    fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//{{DECLARE_CONTROLS
java.awt.Label nameLabel;
java.awt.Label passwordLabel;
java.awt.TextField userTextField;
java.awt.TextField passTextField;
java.awt.Button okButton;
java.awt.TextField dsnTextField;
java.awt.Label label1;
//}}

```

```

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == okButton)
            okButton_Action(event);
    }
}

void okButton_Action(java.awt.event.ActionEvent event)
{
    // to do: code goes here.
    String uid = getUsername();
    String pass = getPassword();
    String dsn = dsnTextField.getText();

    System.out.println(uid + " " + " " + " " + dsn);

    genericSQLFrame.buff.append(uid);
    genericSQLFrame.buff.append(" ");
    genericSQLFrame.buff.append(pass);
    genericSQLFrame.buff.append(" ");
    genericSQLFrame.buff.append(dsn);
    genericSQLFrame.buff.append(" ");

    //get rid of box
    hide();
    dispose();

    //{{CONNECTION
    // Disable the Button
    //okButton.setEnabled(false);
    //}}
}
}

//*****
// END:    newUser.java
//*****

//*****
// File:    passwordDialog.java
// Purpose: A dialog box to enter a users name and password,
//          Calls back to static function (lookup(uid, pw) in mainFrame
//*****

import java.awt.*;
import symantec.itools.awt.util.dialog.ModalDialog;

public class PasswordDialog extends ModalDialog
{
    public PasswordDialog(Frame parent, String title)
    {
        super(parent, title);

        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.

```

```

        //{{INIT_CONTROLS
        setLayout(null);
        setVisible(false);
        setSize(insets().left + insets().right + 215,insets().top +
insets().bottom + 130);
        setBackground(new Color(12632256));
        nameLabel = new java.awt.Label("Name:");
        nameLabel.setBounds(insets().left + 10,insets().top + 32,75,15);
        add(nameLabel);
        passwordLabel = new java.awt.Label("Password:");
        passwordLabel.setBounds(insets().left + 10,insets().top + 60,75,15);
        add(passwordLabel);
        userTextField = new java.awt.TextField(1);
        userTextField.setBounds(insets().left + 85,insets().top + 28,100,22);
        add(userTextField);
        passTextField = new java.awt.TextField(1);
        passTextField.setEchoChar('*');
        passTextField.setBounds(insets().left + 85,insets().top + 57,100,22);
        add(passTextField);
        okButton = new java.awt.Button();
        okButton.setLabel("OK");
        okButton.setBounds(insets().left + 80,insets().top + 95,40,20);
        add(okButton);
        setTitle("");
        //}}

        //{{REGISTER_LISTENERS
        SymAction lSymAction = new SymAction();
        okButton.addActionListener(lSymAction);
        //}}
    }

    public PasswordDialog(Frame parent)
    {
        this(parent, "Username/Password");
    }

    // Add a constructor for Interactions (ignoring modal)
    public PasswordDialog(Frame parent, boolean modal)
    {
        this(parent);
    }

    // Add a constructor for Interactions (ignoring modal)
    public PasswordDialog(Frame parent, String title, boolean modal)
    {
        this(parent, title);
    }

    public String getUsername()
    {
        return userTextField.getText();
    }

    public String getPassword()
    {
        return passTextField.getText();
    }

    public void setUsername(String name)
    {
        userTextField.setText(name);
    }

```

```

    }

    public void setPassword(String pass)
    {
        passTextField.setText(pass);
    }

    public void addNotify()
    {
        // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();

        super.addNotify();

        if (fComponentsAdjusted)
            return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();
            p.translate(insets().left, insets().top);
            components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
    }

    // Used for addNotify check.
    boolean fComponentsAdjusted = false;

    //{{DECLARE_CONTROLS
    java.awt.Label nameLabel;
    java.awt.Label passwordLabel;
    java.awt.TextField userTextField;
    java.awt.TextField passTextField;
    java.awt.Button okButton;
    //}}

    class SymAction implements java.awt.event.ActionListener
    {
        public void actionPerformed(java.awt.event.ActionEvent event)
        {
            Object object = event.getSource();
            if (object == okButton)
                okButton_Action(event);
        }
    }

    void okButton_Action(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.
        String uid = getUsername();
        String password = getPassword();

        System.out.println("User " + uid + " Password " + password);

        //pass back results to mainFrame
        mainFrame.lookup(uid, password);

        //get rid of the frame

```

```

        hide();
        dispose();
    }
}

//*****
// END:    passwordDialog.java
//*****

//*****
// File:    dbUtil.java
// Purpose: A utility class object with functions to
//          establish a connection, submit a SQL Statment,
//          get database meta data
//*****

import java.sql.*;
import java.util.*; //for vector and hash
import java.io.*;   //for OutputStream

public class dbUtil {

    private Connection con = null;
    private Statement stmt = null;
    static int sqlRequest = 1;

    private static final boolean debug = false; //for debugging

    //*****
    // Function: bool setConnection(String driver, String url,
    //                               String name, String password)
    // Purpose : constructor which allows user to specify connection
    //*****
    public boolean setConnection(String driver, String url, String name, String
password)
    {

        try{
            Class.forName(driver);
            con = DriverManager.getConnection(url, name, password);

        }catch(SQLException e){
            System.out.println("Failed to connect to database: " + url + " " +
e.getMessage());
            return false;
        }catch(ClassNotFoundException e){
            System.out.println("Unable to find driver class.");
            return false;
        }
        System.out.println("Connected to Database :" + url);
        return true;
    } //end setConnection

    //*****
    // Function: Connection getConnection(String driver, String url,
    //                               String name, String password)
    // Purpose : Creates a connection and returns a conneciton object
    //*****
    public Connection getConnection(String driver, String url, String name,
String password)
    {

```

```

    try{
        Class.forName(driver);
        con = DriverManager.getConnection(url, name, password);
    }catch(SQLException e){
        System.out.println("Failed to connect to database: " + url + " " +
e.getMessage());
        return null;
    }catch(ClassNotFoundException e){
        System.out.println("Unable to find driver class.");
        return null;
    }
    System.out.println("Connected to Database : " + url);
    return con;
} //end setConnection

//*****
// Function: closeConnection(
// Purpose : closes the Connection to the datasource
//*****
public void closeConnection()
{
    try{
        con.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
} //end closeConnection

//*****
// Function: executeSQL(String sql)
// Purpose : dynamically get table data, based upon a sql statement
//           returns a vector with resultSet
// Ref: DataBase Programming with JDBC and JAVA
//*****
public Vector executeSQL(String sql)
{
    Vector resultVector = new Vector(); //to store resultSet by hashTables

    int cols;
    try{
        stmt = con.createStatement();

        if(stmt.execute(sql)){ //returns true if sql produces a resultSet

            //get the SQL results
            ResultSet result = stmt.getResultSet();

            //get the resultSet metadata
            ResultSetMetaData meta = result.getMetaData();

            //how many columns
            cols = meta.getColumnCount();

            int xx =0;
            //increment through the rows (tuples) of the result set
            while(result.next() ){

                //each tuple gets a hashtable to store information
                Hashtable rowResults = new Hashtable(cols);

```

```

        //increment through the tuple <name, ssn, dept>
        for (int ix = 1; ix < cols; ix++){

            Object obj = result.getObject(ix);

            //use the column lable as the hash key and put object
hashtable
            if(obj == null){
                rowResults.put(meta.getColumnLabel(ix), "");
            } else {
                rowResults.put(meta.getColumnLabel(ix), obj);
            } //end if

            //add the hash object to the vector
            resultVector.addElement(rowResults);
        } //end while

        return resultVector;
    }

    return null; //SQL statement did not produce a ResultSet
}
catch(SQLException e){
    System.err.println("Failed to executeSQL(" + sql + ") function");
    e.printStackTrace();
    return null;
}

} //end executeSQL

//*****
// Function : printConsole(Vector v)
// Purpose : Prints the data taken from hash table which
//           is received by vector v
//*****
public void printConsole(Vector v)
{
    if (v == null){
        //do nothing
    }
    else{
        Vector resultVector = v;
        int vectSize = resultVector.size();

        for(int ix = 0; ix < vectSize; ix++) {
            //take the hashtables from the vector one by one
            Hashtable myHashTable = (Hashtable)resultVector.elementAt(ix);
            //enumerate the objects in hashtable
            Enumeration hashEnum = (Enumeration)myHashTable.elements();

            while(hashEnum.hasMoreElements()){
                Object myObj = (Object)hashEnum.nextElement();
                System.out.println(myObj);
            } //end while

        } //end for
    } //end else
} //end printHashedVector()

//*****

```



```

// Function : String convertToString(Vector v)
// Purpose  : Converts Vector into a string with the
//           a delimiator |
//*****
public String convertResultSetToString(Vector v)
{
    if (v == null){
        //do nothing
        return null;
    }
    else{
        Vector resultVector = v;
        int vectSize = resultVector.size();
        StringBuffer buff = new StringBuffer();
        buff.append(" ");

        for(int ix = 0; ix < vectSize; ix++) {
            //take the hashtables from the vector one by one
            Hashtable myHashTable = (Hashtable)resultVector.elementAt(ix);
            //enumerate the objects in hashtable
            Enumeration hashEnum = (Enumeration)myHashTable.elements();

            while(hashEnum.hasMoreElements()){
                Object oneItem = (Object)hashEnum.nextElement();
                buff.append(" ");
                buff.append( oneItem.toString() );
                buff.append(" "); //space between each attribute
            }//end while

            //a delimiator to be used for printing
            buff.append("|");
            //append a newline statement at the end of each tuple
            buff.append("\n");

        }//end for

        String result = new String ( buff );

        return result;
    }//end else
} //end convertToString()

//*****
// Function : printVectorOfVectors(Vector v)
// Purpose  : Prints the data taken out of the 2-D vector v
//           First element of the vector is again a vector that contains
//           the name of the attributes
//*****
public static void printVectorOfVectors(Vector v)
{
    Vector resultVector = v;
    int vectSize = resultVector.size();
    System.out.println("in printing cycle...");
    Vector attributeVector = (Vector)resultVector.firstElement();

    System.out.println("First vector received...");

    //print the name of the attributes of the table
    for(int xx = 0 ; xx < attributeVector.size(); xx++) {
        String attributeName = (String)attributeVector.elementAt(xx);
        System.out.print(attributeName + " ");
    }
}

```

```

System.out.println();

for(int ix = 1; ix < vectSize; ix++) {
    //take the hashtable from the vector one by one
    Vector dataVector = (Vector)resultVector.elementAt(ix);

    for(int yy = 0; yy < dataVector.size(); yy++){
        Object myObj = (Object)dataVector.elementAt(yy);
        System.out.println(myObj);
    } //end inner for
} //end outer for
} //end printVectorOfVectors()

//*****
// Function: printResultsTable(ResultSet rs, OutputStream output)
// Purpose : prints a ResultSet
// Call: printResultsTable(rs, System.out)
//*****
public void printResultsTable(ResultSet rs, OutputStream output)
throws SQLException {
    // Set up the output stream
    PrintWriter out = new PrintWriter(new OutputStreamWriter(output));

    // Get some "meta data" (column names, etc.) about the results
    ResultSetMetaData metadata = rs.getMetaData();

    // Variables to hold important data about the table to be displayed
    int numcols = metadata.getColumnCount(); // how many columns
    String[] labels = new String[numcols]; // the column labels
    int[] colwidths = new int[numcols]; // the width of each
    int[] colpos = new int[numcols]; // start position of each
    int linewidth; // total width of table

    // Figure out how wide the columns are, where each one begins,
    // how wide each row of the table will be, etc.
    linewidth = 1; // for the initial '|'.
    for(int i = 0; i < numcols; i++) { // for each column
        colpos[i] = linewidth; // save its position
        labels[i] = metadata.getColumnLabel(i+1); // get its label
        // Get the column width. If the db doesn't report one, guess
        // 30 characters. Then check the length of the label, and use
        // it if it is larger than the column width
        int size = metadata.getColumnDisplaySize(i+1);
        if (size == -1) size = 30; // some drivers return -1...
        int labelsize = labels[i].length();
        if (labelsize > size) size = labelsize;
        colwidths[i] = size + 1; // save the column the size
        linewidth += colwidths[i] + 2; // increment total size
    }

    // Create a horizontal divider line we use in the table.
    // Also create a blank line that is the initial value of each
    // line of the table
    StringBuffer divider = new StringBuffer(linewidth);
    StringBuffer blankline = new StringBuffer(linewidth);
    for(int i = 0; i < linewidth; i++) {
        divider.insert(i, '-');
        blankline.insert(i, " ");
    }
    // Put special marks in the divider line at the column positions
    for(int i=0; i<numcols; i++) divider.setCharAt(colpos[i]-1, '+');
    divider.setCharAt(linewidth-1, '+');
}

```

```

// Begin the table output with a divider line
out.println(divider);

// The next line of the table contains the column labels.
// Begin with a blank line, and put the column names and column
// divider characters "|" into it. overwrite() is defined
//below.
StringBuffer line = new StringBuffer(blankline.toString());
line.setCharAt(0, '|');
for(int i = 0; i < numcols; i++) {
    int pos = colpos[i] + 1 + (colwidths[i]- labels[i].length())/2;
    overwrite(line, pos, labels[i]);
    overwrite(line, colpos[i] + colwidths[i], "|");
}

// Then output the line of column labels and another divider
out.println(line);
out.println(divider);

// Now, output the table data. Loop through the ResultSet, using
// the next() method to get the rows one at a time. Obtain the
// value of each column with getObject(), and output it, much as
// we did for the column labels above.
while(rs.next()) {
    line = new StringBuffer(blankline.toString());
    line.setCharAt(0, '|');
    for(int i = 0; i < numcols; i++) {
        Object value = rs.getObject(i+1);
        overwrite(line, colpos[i] + 1, value.toString().trim());
        overwrite(line, colpos[i] + colwidths[i], "|");
    }
    out.println(line);
}

// Finally, end the table with one last divider line.
out.println(divider);
out.flush();
}

/** This utility method is used when printing the table of results */
static void overwrite(StringBuffer b, int pos, String s)
{
    int len = s.length();
    for(int i = 0; i < len; i++) b.setCharAt(pos+i, s.charAt(i));
}

//*****
// Function: getMetaData(Statement stmt)
// Purpose : outputs the names of the employee
//           who has consumed the most coffee
//*****
public void getMetaData(ResultSet result)
{
    try{

        ResultSetMetaData meta = result.getMetaData();

        int numbers = 0;
        int columns = meta.getColumnCount();
        for (int i=1;i<=columns;i++) {

```

```

        System.out.println (meta.getColumnLabel(i) + "\t"
            + meta.getColumnTypeName(i));
        if (meta.isSigned(i)) { // is it a signed number?
            numbers++;
        }
    }
    System.out.println ("Columns: " + columns + " Numeric: " +
numbers);

    }catch(Exception e){
        e.printStackTrace();
    }
} //end getMetaData()

//*****
// Function: getDataBaseMetaData()
// Purpose : outputs the capabilities of the dbms vendor
// Source:   java.sql.Connection
//*****
public void getDataBaseMetaData()
{
    try{
        DatabaseMetaData md = con.getMetaData();

        //there are many questions you can ask, example:

        if(md==null){
            System.out.println("No DataBase Meta Data");
        }
        else{
            System.out.println("Database Product :" +
                md.getDatabaseProductName());

            System.out.println("Allowable Connection :" +
                md.getMaxConnections());

            System.out.println("Support Stored Procedures :" +
                md.supportsStoredProcedures());

            System.out.println("SQL Support of ODBC Drivers");

            System.out.println("Support Core SQL :" +
                md.supportsCoreSQLGrammar());

            System.out.println("Support Minimum SQL :" +
                md.supportsMinimumSQLGrammar());

            System.out.println("Support Extended SQL :" +
                md.supportsExtendedSQLGrammar());

            System.out.println("SQL Support of JDBC Drivers");

            System.out.println("Supports ANSI 92 Entry:" +
                md.supportsANSI92EntryLevelSQL());

            System.out.println("Supports ANSI 92 Intermediate:" +
                md.supportsANSI92IntermediateSQL());

            System.out.println("Supports ANSI 92 Full:" +
                md.supportsANSI92FullSQL());
        }
    }
}

```

```

    }
        }catch(Exception e){
    }
} //end getDataBaseMetaData()

//*****
// Function : printResultSetString(String message)
// Purpose :Works with above fuction
//*****
public void printResultSetString(String message)
{
    //use a string tokenizer to parse the request, the delimiter is
blankspace
    StringTokenizer tok = new StringTokenizer(message);

    //evaluate the first token "select part from parts"
    String command = tok.nextToken();
    String from = tok.nextToken();
    String restOfMsg = restOfMessage(tok);

    StringTokenizer tok2 = new StringTokenizer(restOfMsg, "|");
    int row = 1;
    int count = 0;

    System.out.println("\n***** Result Set *****");

    if(message == null){
        System.out.println("No results found");
    }else{
        count = tok2.countTokens();
        for (int ix = 1; ix < count; ix++){
            String result = tok2.nextToken();
            System.out.println("row " + row++ + " " + result);
        } //end for

        System.out.println("\n***** End Result Set *****");
    } //end else

} //end function

//*****
// Function : String restOfMessage(StringTokenizer tok)
// Purpose :Utility function used by above function
//*****
private String restOfMessage(StringTokenizer tok)
{
    StringBuffer buff = new StringBuffer( );

    //get the rest of the request
    while( tok.hasMoreTokens() ){

        buff = buff.append( tok.nextToken() ); //appends the sql statement
        buff = buff.append( " " );
    }

    String result = new String( buff );
    return result;
}

//*****
// Function: executeSQL(String sql)
// Purpose : dynamically get table data, based upon a sql statment

```

```

//          returns a vector with resultSet
// Ref: DataBase Programming with JDBC and JAVA
//*****
public String executeSQLGetString(String sql)
{
    System.out.println(" executeSQLGetString Request # " + sqlRequest++);
    String resultString = null;

    try{
        stmt = con.createStatement();

        if(stmt.execute(sql)){ //returns true if sql produces a resultSet

            ResultSet result_set = stmt.getResultSet();
            //put resultSet in string format
            resultString = processResults(result_set);
        }//end if

        return resultString;
    }
    catch(SQLException e){
        System.err.println("Failed to executeSQL(" + sql + ") function");
        e.printStackTrace();
        return null;
    }
}

//end executeSQLGetString(String sql)

//*****
// Function: String processResults(ResultSet results)
// Purpose : Formats the result set for pretty printing
//*****
public String processResults(ResultSet results) throws SQLException
{
    try {
        ResultSetMetaData meta = results.getMetaData();
        StringBuffer bar = new StringBuffer();
        String buffer = "";
        int cols = meta.getColumnCount();
        int row_count = 0;
        int i, width = 0;

        // create the bar that is as long as the total of all columns
        for(i=1; i<=cols; i++) {
            width += meta.getColumnDisplaySize(i);
        }
        width += 1 + cols;
        for(i=0; i<width; i++) {
            bar.append('-');
        }
        bar.append('\n');
        buffer += bar + "|";
        // After the first bar goes the column labels
        for(i=1; i<=cols; i++) {
            StringBuffer filler = new StringBuffer();
            String label = meta.getColumnLabel(i);
            int size = meta.getColumnDisplaySize(i);
            int x;

            // If the label is long than the column is wide,
            // then we truncate the column label
            if( label.length() > size ) {

```

```

        label = label.substring(0, size);
    }
    // If the label is shorter than the column, pad it with spaces
    if( label.length() < size ) {
        int j;
        x = (size-label.length())/2;
        for(j=0; j<x; j++) {
            filler.append(' ');
        }
        label = filler + label + filler;
        if( label.length() > size ) {
            label = label.substring(0, size);
        }
        else {
            while( label.length() < size ) {
                label += " ";
            }
        }
    }
    // Add the column header to the buffer
    buffer = buffer + label + "|";
}
// Add the lower bar
buffer = buffer + "\n" + bar;
// Format each row in the result set and add it on
while( results.next() ) {
    row_count++;

    buffer += "|";
    // Format each column of the row
    for(i=1; i<=cols; i++) {
        StringBuffer filler = new StringBuffer();
        Object value = results.getObject(i);
        int size = meta.getColumnDisplaySize(i);
        String str = value.toString();

        if( str.length() > size ) {
            str = str.substring(0, size);
        }
        if( str.length() < size ) {
            int j, x;

            x = (size-str.length())/2;
            for(j=0; j<x; j++) {
                filler.append(' ');
            }
            str = filler + str + filler;
            if( str.length() > size ) {
                str = str.substring(0, size);
            }
            else {
                while( str.length() < size ) {
                    str += " ";
                }
            }
        }
    }
    //end if
    buffer = buffer + str + "|";
} //end for
buffer = buffer + "\n";
} //end while

// Stick a row count up at the top
if( row_count == 0 ) {

```



```

        System.out.println("Instantiating the database Broker");
        System.err.println("dbBrokerServer...");

        System.err.println("\nObject Server, version 3.1 is ready...");

    }catch(UnknownHostException e){
        System.out.println("Unknown Host Exception " + e);
    }catch(RemoteException e){
        System.out.println("Remote Exception " + e);
    }catch(java.net.MalformedURLException e){
        System.out.println(e);
    }catch(SQLException e){
        System.out.println(e);
    }
}
} //end main

} //end objectServer

//*****
// END:    objectServer.java
//*****

//*****
// File:    dbAdminBroker_Int.java
// Purpose: Interface for database administrator to enter new users,
//          authorize them access
//*****

import java.rmi.*;
import java.sql.*; //for ResultSet
import java.util.*; //for vector

public interface dbAdminBroker_Int extends Remote
{
    //add a user
    public abstract boolean addUser(String name, String password) throws
RemoteException;

    //delete a user
    public abstract boolean deleteUser(String name) throws RemoteException;

    //allow user access to a datasource
    public abstract boolean provideAccess(String name, String dataSource) throws
RemoteException;

    //modify user access privileges
    public abstract void deleteAccess(String name, String dataSource) throws
RemoteException;

    //admin utility
    public abstract Vector getDSNNames()throws RemoteException;
    public abstract boolean addDSN(String name) throws RemoteException;
    public abstract Vector getUsers() throws RemoteException;
    public abstract Vector getDBAccess(String name) throws RemoteException;
}

//*****
// END:    dbAdminBroker_Int.java
//*****

```

```

//*****
// File:      dbAdminBroker_Impl.java
// Purpose: Implementation for database administrator to enter new users,
//           authorize them access
//*****

import java.sql.*; //for JDBC
import java.util.*; //for Vector
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

public class dbAdminBroker_Impl extends UnicastRemoteObject implements
dbAdminBroker_Int
{

    dbUtil dbUtilities = null;
    Statement stmt = null;
    Connection con=null;

    //*****
    // Function: acctsdbImpl()
    // Purpose : object default constructor
    //           must be declared to throw RemoteException
    //*****
    public dbAdminBroker_Impl()throws RemoteException, SQLException
    {
        //create a dbUtil object
        dbUtilities = new dbUtil();

        String driver   = "sun.jdbc.odbc.JdbcOdbcDriver";
        String url      = "jdbc:odbc:brokerAccess";
        String uid      = "";
        String password = "";

        con = dbUtilities.getConnection(driver,url,uid,password);
    }//end constructor

//*****USER METHODS*****
    //add a user
    public boolean addUser(String name, String password) throws RemoteException
    {
        //prep the SQL Statement
        StringBuffer buff = new StringBuffer();
        buff.append("insert into names values ('");
        buff.append(name);
        buff.append(",");
        buff.append(password);
        buff.append(")");
        String sql = new String(buff);

        System.out.println(sql);

        //submit the statement
        try{
            Statement stmt = con.createStatement();
            stmt.executeUpdate(sql);
            stmt.close();
            return true;
        }catch(SQLException e){
            System.out.println(e);
        }
    }
}

```

```

    }

    return false;
} //end func

//delete a user
public boolean deleteUser(String name) throws RemoteException
{
    //prep the SQL Statement
    StringBuffer buff = new StringBuffer();
    buff.append("delete from names where name =");
    buff.append(name);
    buff.append("");
    String sql = new String(buff);
    System.out.println(sql);

    StringBuffer buff1 = new StringBuffer();
    buff1.append("delete from datasources where name =");
    buff1.append(name);
    buff1.append("");
    String sql1 = new String(buff);
    System.out.println(sql1);

    //submit the statement
    try{
        Statement stmt = con.createStatement();
        stmt.executeUpdate(sql);
        stmt.executeUpdate(sql1);
        stmt.close();
        return true;
    }catch(SQLException e){
        System.out.println(e);
    }

    return false;
} //end function

//allow user access to a datasource
public boolean provideAccess(String name, String dsn) throws RemoteException
{
    //prep the SQL Statement
    StringBuffer buff = new StringBuffer();
    buff.append("insert into datasources values ('");
    buff.append(dsn);
    buff.append(",");
    buff.append(name);
    buff.append(")");
    String sql = new String(buff);

    System.out.println(sql);

    //submit the statement
    try{
        Statement stmt = con.createStatement();
        stmt.executeUpdate(sql);
        stmt.close();
        return true;
    }catch(SQLException e){
        System.out.println(e);
    }

    return false;
} //end function

```

```

//modify user access privileges
public void deleteAccess(String name, String dbase) throws RemoteException
{
    //prep the SQL Statement
    StringBuffer buff = new StringBuffer();
    buff.append("delete from datasources where name ='");
    buff.append(name);
    buff.append("' ");
    buff.append("and datasource = '");
    buff.append(dbase);
    buff.append("'");
    String sql = new String(buff);
    System.out.println(sql);

    //submit the statement
    try{
        Statement stmt = con.createStatement();
        stmt.executeUpdate(sql);
        stmt.close();
    }catch(SQLException e){
        System.out.println(e);
    }
}

//return names from name name table
public Vector getUsers() throws RemoteException
{
    Vector ansVect = new Vector();
    ResultSet rs = null;

    //submit the statement
    try{
        Statement stmt = con.createStatement();
        rs = stmt.executeQuery("select name from names" );
        while(rs.next()){
            String nameString = rs.getString("name");
            ansVect.addElement(nameString);
        }
        stmt.close();
    }catch(SQLException e){
        System.out.println(e);
        return null;
    }
    return ansVect;
}

//return datasource user has access to, client uses above function to get
// a list of names
public Vector getDBAccess(String name) throws RemoteException
{
    Vector ansVect = new Vector();
    ResultSet rs = null;

    //prep the SQL Statement
    StringBuffer buff = new StringBuffer();
    buff.append("select datasource from datasources where name ='");
    buff.append(name);
    buff.append("'");
    String sql = new String(buff);

    System.out.println(sql);

    //submit the statement

```

```

try{
    Statement stmt = con.createStatement();
    rs = stmt.executeQuery(sql);
    while(rs.next()){
        String dataSource = rs.getString("datasource");
        System.out.println(dataSource);
        ansVect.addElement(dataSource);
    }
    stmt.close();
}catch(SQLException e){
    System.out.println(e);
    return null;
}
return ansVect;
}

```

```

//*****
// Function: Vector getTableName()
//
// Purpose : uses the database meta data to
//           return a vector of table names
//*****
public Vector getDSNNames()throws RemoteException
{
    Vector ansVect = new Vector();
    ResultSet rs = null;

    //submit the statement
    try{
        Statement stmt = con.createStatement();
        rs = stmt.executeQuery("select name from ds");
        while(rs.next()){
            String dataSource = rs.getString("name");
            ansVect.addElement(dataSource);
        }
        stmt.close();
    }catch(SQLException e){
        System.out.println(e);
        return null;
    }
    return ansVect;
}
} //end getDSNNames()

```

```

public boolean addDSN(String name) throws RemoteException
{
    //prep the SQL Statement
    StringBuffer buff = new StringBuffer();
    buff.append("insert into ds values ('");
    buff.append(name);
    buff.append("')");
    String sql = new String(buff);

    System.out.println(sql);

    //submit the statement
    try{
        Statement stmt = con.createStatement();
        stmt.executeUpdate(sql);
        stmt.close();
        return true;
    }
}

```

```

        }catch(SQLException e){
            System.out.println(e);
            return false;
        }
    }//end function

} //end dbAdminBroker_Impl.java

//*****
// File:    dbAdminBroker_Impl.java
//*****

//*****
// File:    dbBroker_Int.java
// Purpose: Ineterface for generic database access
//*****

import java.rmi.*;
import java.sql.*; //for ResultSet
import java.util.*; //for vecor

public interface dbBroker_Int extends Remote
{
    //connection mgmnt
    public abstract boolean createConnection(String dataSource, String
userName, String pw) throws RemoteException;
    public abstract void closeConnection() throws RemoteException;

    //utilities
    public Vector getTableNames() throws RemoteException;
    public abstract String executeSQLGetString(String sql) throws
RemoteException;
    public abstract Vector getTableMetaData(String sql) throws RemoteException;
}

//*****
// END:    dbBroker_Int.java
//*****

//*****
// File:    dbBroker_Impl.java
// Purpose: Implementation for generic database broker access
//*****

import java.sql.*;
import java.util.*; //for Vector
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

public class dbBroker_Impl extends UnicastRemoteObject
    implements dbBroker_Int
{
    Statement stmt = null;
    Connection con=null;
    dbUtil dbUtilities = null;

```

```

//*****
// Function: dbBroker_Impl()
// Purpose : object default constructor
//           must be declared to throw RemoteException
//*****
public dbBroker_Impl()throws RemoteException, SQLException
{
    //create a dbUtil object
    dbUtilities = new dbUtil();
} //end constructor

//*****
// Function: Vector setConnection(String dataSource, String uid, String
password)
// Purpose : sets the connection to the datasource, returns a vector
//           containing tablenames
//*****
public boolean createConnection(String dataSource, String uid, String pw)
throws RemoteException
{
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    if(dataSource.equals("NSGDB"))
        uid = "sa";
    StringBuffer buff = new StringBuffer();
    buff.append("jdbc:odbc:");
    buff.append(dataSource);
    String url = new String(buff);

    con = dbUtilities.getConnection(driver,url,uid,pw);

    //get database table name
    return true;
}

public void closeConnection() throws RemoteException
{
    try{
        con.close();
    }catch(SQLException e){
        System.out.println(e);
    }
}

//*****
// Function: executeSQLGetString(String sql
//
// Purpose : user submits SQL statement gets back a string containing
//           resultSet
//*****
public String executeSQLGetString(String sql)throws RemoteException
{
    String resultString = dbUtilities.executeSQLGetString(sql);

    return resultString;
} //end executeSQL

//*****
// Function: Vector getTableName()
//
// Purpose : uses the database meta data to

```

```

//          return a vector of table names
//*****
public Vector getTableNames() throws RemoteException
{
    DatabaseMetaData dmd = null;
    ResultSet rs = null;
    Vector ansVect = new Vector();
    try{
        dmd = con.getMetaData();

        String[] types = {"TABLE"};
        rs = dmd.getTables(null, null, "%", types);

        while(rs.next()){
            String tableName = rs.getString("TABLE_NAME");
            ansVect.addElement(tableName);
        }
    }catch(SQLException e){
        System.out.println(e);
        return null;
    }

    return ansVect;
}

//*****
// Function :Vector getTableMetaData(String sql)
// Purpose  : user can submit a SELECT * FROM <table>
//           to get a list of the column names that exist
//           to display in a choice list
// Goal: return a vector of hash tables containing other information
//       such as name, size, type for displaying
//*****
public Vector getTableMetaData(String sql) throws RemoteException
{
    try{
        Statement stmt = con.createStatement();

        ResultSet results = stmt.executeQuery(sql);

        ResultSetMetaData rsmd = results.getMetaData();

        Vector colVect = new Vector();

        int cols = rsmd.getColumnCount();

        for(int ix = 1; ix < cols; ix++)
        {
            String colName = rsmd.getColumnName(ix);

            if (colName == null)
                colName = "was null";

            //store in vector
            colVect.addElement(colName);

            //other useful information for displaying results
            int colWidth = rsmd.getColumnDisplaySize(ix);

            //get the columns sql type
            int colType = rsmd.getColumnType(ix);

```



```

        } //end for

        stmt.close();

        return colVect;

    } catch (SQLException e) {
        System.out.println(e);
        return null;
    }
} //end funct

} //end databaseServer

//*****
// END:    dbBroker_Impl.java
//*****

//*****
// File:    navydb_Int.java
// Purpose: Hard coded connection to navy database, offers specific
//          navy database manipulation
//*****

import java.rmi.*;
import java.sql.*; //for ResultSet
import java.util.*; //for vecor

public interface navydbInt extends Remote
{
    /** business logic **
    //returns an object that implements the accessPolicy interface
    public abstract accessPolicy_Int getAccessPolicy() throws RemoteException;

    /** database functions **
    public abstract String executeSQLGetString(String sql) throws
    RemoteException;
    public abstract Vector getTableName() throws RemoteException;
    public abstract Vector getTableMetaData(String sql) throws RemoteException;
}

//*****
// END:    navydb_Int.java
//*****

//*****
// File:    navydb_Impl.java
// Purpose: Hard coded connection to navy database, offers specific
//          navy database manipulation
//*****

import java.sql.*;
import java.util.*; //for Vector
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

```

```

public class navydbImpl extends UnicastRemoteObject
    implements navydbInt
{
    Statement stmt = null;
    Connection con=null;
    dbUtil dbUtilities = null;

    /*******
    // Function: accessPolicy_Int getAccessPolicy()
    // Purpose : Returns a serialized object that contains the
    //             companies accessLogic
    /*******
    public accessPolicy_Int getAccessPolicy()
    {
        return new accessPolicyImp();
    }

    /*******
    // Function: acctbdbImpl()
    // Purpose : object default constructor
    //             must be declared to throw RemoteException
    /*******
    public navydbImpl()throws RemoteException, SQLException
    {
        //create a dbUtil object
        dbUtilities = new dbUtil();

        String driver    = "sun.jdbc.odbc.JdbcOdbcDriver";
        String url       = "jdbc:odbc:NSGDB";
        String uid       = "sa";
        String password  = "";

        con = dbUtilities.getConnection(driver,url,uid,password);
    }//end constructor

    /*******
    // Function: executeSQLGetString(String sql
    //
    // Purpose : user submits SQL statement gets back a string containing
    //             resultSet
    /*******
    public String executeSQLGetString(String sql)throws RemoteException
    {

        String resultString = dbUtilities.executeSQLGetString(sql);

        return resultString;
    }//end executeSQL

    /*******
    // Function: Vector getTableName()
    //
    // Purpose : uses the database meta data to
    //             return a vector of table names
    /*******
    public Vector getTableName()throws RemoteException
    {
        DatabaseMetaData dmd = null;
        ResultSet rs = null;
        Vector ansVect = new Vector();
        try{

```

```

        dmd = con.getMetaData();
        String[] types = {"TABLE"};
        rs = dmd.getTables(null, null, "%", types);

        while(rs.next()){
            String tableName = rs.getString("TABLE_NAME");
            ansVect.addElement(tableName);
        }
    }catch(SQLException e){
        System.out.println(e);
        return null;
    }

    return ansVect;

}

//*****
// Function :Vector getTableMetaData(String sql)
// Purpose  : user can submit a SELECT * FROM <table>
//           : to get a list of the column names that exist
//           : to display in a choice list
// Goal: return a vector of hash tables containing other information
//       such as name, size, type for displaying
//*****
public Vector getTableMetaData(String sql) throws RemoteException
{
    try{
        Statement stmt = con.createStatement();

        ResultSet results = stmt.executeQuery(sql);

        ResultSetMetaData rsmd = results.getMetaData();

        Vector colVect = new Vector();

        int cols = rsmd.getColumnCount();

        for(int ix = 1; ix < cols; ix++)
        {
            String colName = rsmd.getColumnName(ix);

            if (colName == null)
                colName = "was null";

            //store in vector
            colVect.addElement(colName);

            //other useful information for displaying results
            int colWidth = rsmd.getColumnDisplaySize(ix);

            //get the columns sql type
            int colType = rsmd.getColumnType(ix);

        }//end for

        stmt.close();

        return colVect;

    }catch(SQLException e){
        System.out.println(e);
        return null;
    }
}

```

```

    }
    }//end funct

} //end databaseServer

//*****
// END:    navydb_Impl.java
//*****

//*****
// File:    navydb_Int.java
// Purpose: Interface for accounts database, offers client various methods
//*****

import java.rmi.*;
import java.sql.*; //for ResultSet
import java.util.*; //for vecor

public interface acctsdbInt extends Remote
{
    //returns an object that implements the accessPolicy interface
    public abstract accessPolicy_Int getAccessPolicy() throws RemoteException;

    //database function
    //DBA Functions
    public abstract Vector executeSQL(String sql) throws RemoteException;
    public abstract String executeSQLGetString(String sql) throws
RemoteException;
    public abstract Vector getTableName() throws RemoteException;
    public abstract Vector getTableMetaData(String sql) throws RemoteException;

    //Basic User Functions
    public abstract void insertEmployee(String name, boolean faculty) throws
RemoteException;
    public abstract String viewEmployees() throws RemoteException;

    //administration functions
    public abstract void addUser(String uid, String pass) throws
RemoteException;
}

//*****
// END:    acctsdb_Impl.java
//*****

//*****
// File:    acctsdb_Impl.java
// Purpose: Hard coded connection to accounts database, offers limited
//          accounts database manipulation
//*****

import java.sql.*;
import java.util.*; //for Vector
import java.rmi.*;
import java.rmi.server.*;

public class acctsdbImpl extends UnicastRemoteObject

```

```

    implements acctsdbInt
{
    Statement stmt = null;
    Connection con=null;
    dbUtil dbUtilities = null;

    private boolean    debug = true;    //for debugging only

    //*****
    // Function: accessPolicy_Int getAccessPolicy()
    // Purpose : Returns a serialized object that contains the
    //           companies accessLogic
    //*****
    public accessPolicy_Int getAccessPolicy()
    {
        return new acctsPolicyImp();
    }

    //administration functions
    public void addUser(String uid, String pass) throws RemoteException
    {
        //write info to a file
    }

    //*****
    // Function: acctsdbImpl()
    // Purpose : object default constructor
    //           must be declared to throw RemoteException
    //*****
    public acctsdbImpl()throws RemoteException, SQLException
    {
        //create a dbUtil object
        dbUtilities = new dbUtil();

        String driver    = "sun.jdbc.odbc.JdbcOdbcDriver";
        String url        = "jdbc:odbc:acctsDataBase97";
        String uid        = "";
        String password   = "";

        con = dbUtilities.getConnection(driver,url,uid,password);
    }

    //end constructor

    //*****
    // Function: executeSQL(String sql)
    //           which processed and returns a vector with resultSet
    // Source:   java.sql.Connection
    //*****
    public Vector executeSQL(String sql)throws RemoteException
    {
        Vector resultVector = new Vector();

        resultVector = dbUtilities.executeSQL(sql);

        return resultVector;
    }

    //end executeSQL

```

```

//*****
// Function: executeSQLGetString(String sql
// Purpose : user submits SQL statement gets back a string containing
//           resultSet
//*****
public String executeSQLGetString(String sql) throws RemoteException
{
    String resultString = dbUtilities.executeSQLGetString(sql);

    return resultString;

} //end executeSQL

public void insertEmployee(String name, boolean faculty) throws
RemoteException
{
    StringBuffer buff = new StringBuffer();
    buff.append("insert into People values (' ");
    buff.append(name);
    buff.append(",");
    buff.append(faculty);
    buff.append(")");

    String sql = new String(buff);
    String junk = dbUtilities.executeSQLGetString(sql);
}

public String viewEmployees() throws RemoteException
{
    String result = dbUtilities.executeSQLGetString("select * from People");
    return result;
}
//*****
// Function: Vector getTableName()
//
// Purpose : uses the database meta data to
//           return a vector of table names
//*****
public Vector getTableName() throws RemoteException
{
    DatabaseMetaData dmd = null;
    ResultSet rs = null;
    Vector ansVect = new Vector();
    try{
        dmd = con.getMetaData();

        String[] types = {"TABLE"};
        rs = dmd.getTables(null, null, "%", types);

        while(rs.next()){
            String tableName = rs.getString("TABLE_NAME");
            ansVect.addElement(tableName);
        }
    } catch (SQLException e){
        System.out.println(e);
        return null;
    }

    return ansVect;
}

```

```

    }

    /*******
    // Function :Vector getTableMetaData(String sql)
    // Purpose  : user can submit a SELECT * FROM <table>
    //           to get a list of the column names that exist
    //           to display in a choice list
    // Goal: return a vector of hash tables containing other information
    //       such as name, size, type for displaying
    /*******
    public Vector getTableMetaData(String sql) throws RemoteException
    {
        try{
            Statement stmt = con.createStatement();

            ResultSet results = stmt.executeQuery(sql);

            ResultSetMetaData rsmd = results.getMetaData();

            Vector colVect = new Vector();

            int cols = rsmd.getColumnCount();

            for(int ix = 1; ix <= cols; ix++)
            {
                String colName = rsmd洗getColumnName(ix);

                if (colName == null)
                    colName = "was null";

                //store in vector
                colVect.addElement(colName);

                //other useful information for displaying results
                int colWidth = rsmd洗getColumnDisplaySize(ix);

                //get the columns sql type
                int colType = rsmd洗getColumnType(ix);

            }//end for

            stmt.close();

            return colVect;

        }catch(SQLException e){
            System.out.println(e);
            return null;
        }
    }//end funct

} //end databaseServer

/*******
// END:    acctbdb_Impl.java
/*******

/*******
// File:    accessPolicy_Int.java
// Purpose: Interface to funtions that are implemented by
//         accessPolicyImp.

```

```

//*****
public interface accessPolicy_Int
{
    public abstract int getAccessCode(String name);
}

//*****
// END:    accessPolicy_Int.java
//*****

//*****
// File:    accessPolicyImp
// Purpose: Contains the current accessPolicy, resides on the server,
//          The policy is downloaded to the client, ensures client
//          uses the current policy.
// Notes:   Object must implement Serializable since the object will be
//          downloaded to the client via RMI
//          Object runs on client virtual machine, not the servers.
//*****
import java.io.*;

public class accessPolicyImp implements accessPolicy_Int, Serializable
{
    //*****
    // Function: int getAccessCode(String name)
    // Purpose:  Based upon name, returns current access code
    //           which sets level of database manipulation
    //*****
    public int getAccessCode(String name)
    {
        int accessCode = -1;
        if (name.equals("dba"))
            accessCode = 1;
        else if(name.equals("fred") )
            accessCode = 1;
        else if(name.equals("ramis") )
            accessCode = 1;
        else
            accessCode = 2;
        return accessCode;
    }
}

//*****
// END:    accessPolicyImp
//*****

```


APPENDIX D. DEPLOYMENT

A. Deploying Jbuilder database aware Application

Before explaining the deployment of an application, some assumptions must be made concerning about both the development and client environment:

Classpaths in the JBuilder's IDE settings must be checked in order to point to the valid zip or jar files. See JBuilder help files for details.

If Borland DataGateway is being used, then the classpaths must include the location of the file datagateway.zip. To do that, open the JBuilder.ini file and make sure that classpath includes datagateway.zip. Also do the same thing for system classpath. This is required in order for the Deployment Wizard to gather the appropriate class files for setting the connection with the database.

Client computer has the capability to establish an Internet connection. Before running the application, this connection must be established in order to connect to the database server.

Client computer has a Java Runtime Environment (JRE) higher than or equal to version 1.1. JRE is the core Java Virtual Machine that allows the applications to run on different platforms. If the client environment does not have one, then the user must be told to download a JRE from Sun's JRE download page (<http://java.sun.com>) and to install it into his/her computer.

After satisfying the previous conditions, then follow through the steps to deploy the application:

Create a JBuilder application that uses the Borland DataGateway for connecting to a database.

Before compiling the project, change the OUTPATH in the project settings to point to the folder where the projName.jpr file resides. After the compilation, an extra directory structure inside the main project folder (the one that JBuilder creates automatically when a project is created). The very last folder of this new directory structure will contain the class files. If the developer does not change the OUTPATH, then JBuilder puts your class files under JBuilder/myClasses/*projectName*(folder) by default.

Save all of the files in the project and re-build and make sure that the application runs in the JBuilder environment by simply clicking the lightning button on the toolbar.

Bundle the class files into a compressed file by using the Deployment Wizard (from the menu bar *tools/deployment wizard...*). In the Deployment Wizard pop-up frame

select the check boxes beside JBCL, JGL and all others options and give a name to the file (zip/jar) that will be created by the wizard. But make sure that the path in the file window shows the folder that jpr file resides (that is, the created jar/zip file will be in the same directory with the jpr file). The reason of doing this is just to locate it easily.

Locate that jar or zip file and locate the *broker.zip* file (client side of datagateway) in the development environment. Broker.zip file contains the necessary class files for connecting to the Datagateway server.

Usually the clients expect to run the applications that they receive without typing or modifying anything. In general they just want to click an icon in order to run them. So, create a batch file which will contain the exact command line statements to run the JBuilder application and tell the client to create a shortcut to that batch file and put it on the desktop and run it.

Make the client create a directory structure necessary for the files that will be sent to him/her and let him/her know which file to put where. The batch file should change the directories down to where jar or zip file resides and should contain exact commands to run the application. The following is an example of a batch file, assuming the client has created the desired directory structure (in this case C:\testArea\test1\). And the files *fName.jar* and *broker.zip* are currently in the *test1* folder and the name of the class file that contains the main method is *myMainApplication*.

RunMe.bat may be like this:

```
cd testArea\test1
jre -cp broker.zip ; fName.jar test1.myMainApplication
```

Send the following files to the client:

- Jar or zip file that has been generated by the Deployment Wizard.
- Broker.zip file that will talk to DataGateway server
- The batch file
- A readme text file that explains what directories to create and where to put the received files.

If the user has JDK 1.1 or higher, then the line beginning with jre command inside the batch file might just be replaced with "java -classpath .;<exact path to the classes.zip of JDK>;broker.zip;fName.jar test1.myMainApplication". (Semicolon is used in Windows environment, so in unix or linux environment use colon to separate the classpath items). User can also modify the classpath of the system by adding the paths to broker.zip and fName.jar files into the classpath settings.

The user can run this batch file and it will automatically launch the application and bring up the GUI.

B. deploying jbuilder database aware applets

The followings are the assumptions for a successful applet deployment:

- Borland DataGateway has been setup correctly. And it runs without an error.
- System classpath points to datagateway.zip. If not, the user might get "No suitable Driver" error during the execution of his/her applet.

For Windows 95, autoexec.bat file should have the following settings (path has to be accurate):

```
set classpath = .;c:\Program Files\Borland\Classes\datagateway.zip
```

For Window NT machines, in the system properties window the classpath system variable must contain the exact path to datagateway.zip file.

- The classpath in the JBuilder's IDE options contains datagateway.zip file. If not, the user must go to Tools | IDE Options and edit the CLASSPATH, then find the zip file in his driver and add it.
- JBuilder.ini file must be edited for IDECLASSPATH and CLASSPATH to point to the file datagateway.zip. If not, the user should open the JBuilder.ini file and add the IDECLASSPATH the exact path of datagateway.zip. The same thing must be done for the line beginning with CLASSPATH also.
- Browsers must be Java enabled. For the time being, Netscape 4.3 and higher versions require a Java update (which they call as SmartUpdate). For that you should visit Netscape's home page and follow the instructions. Microsoft Internet Explorer 4.0 and Hot Java are built-in Java enabled.

After satisfying the previous conditions, then follow through the steps to deploy the applet:

Almost everyone who develops JBuilder applet has the same deployment frustration. The following errors are the most common ones: "No suitable Driver" or "security.checklink:BDEDriver" or "Applet could not be initialized". Before using the Deployment Wizard some code must be changed in order to let the wizard grab those class files that are not added into the jar file during the execution of usual Deployment Wizard.

Since we are talking about data-aware applets, there is at least one piece of code in your applet that looks like this:

```
database1.setConnection(new
    borland.jbcl.dataset.ConnectionDescriptor(
        "jdbc.BorlandBroker://127.0.0.1/DB","","",false,
        "borland.jdbc.Broker.RemoteDriver"));
```

In the above code the driver is not instantiated. So the user must instantiate the driver by simply writing the following code before the setConnection function:

```
Class.forName("borland.jdbc.Broker.RemoteDriver");
```

Then remove the jdbc driver name from the connectionDescriptor. After making these changes your code chunk will look like this:

```
Class.forName("borland.jdbc.Broker.RemoteDriver");
database1.setConnection( new
    borland.jbcl.dataset.ConnectionDescriptor(
        "jdbc.BorlandBroker://127.0.0.1/DB","","", false, ""));
```

Then build your applet and save everything frequently (JBuilder does not save your files if you only click the build button) while you are developing. Compile and make sure it runs in the JBuilder environment. Then deploy it by using Deployment Wizard. Once it is done, check the jar file (by extracting and re-jaring it) make sure that you have the directory structure \borland\jdbc\Broker in there. This indicates that the Deployment Wizard grabbed the necessary class files that your applet needed for Borland DataGateway connection.

Modify the html file (but do not change the code tag). Add the archive tag into the applet tag. (i.e., <APPLET ... ARCHIVE = "myApplet.jar" ...> </APPLET>). If the html file will stay in the same directory with the jar file and you will create a hyperlink to that html file in your main page, then you are done. If not or if you want to embed this applet directly into your main page without creating a link to it, then add the codebase tag into the applet tag which should point to the folder where the jar file resides. For example, if the path to myApplet.jar is C:\inetPub\wwwroot\databases\JBuilder\myApplet and wwwroot is the actual root of your web server. Then your html file might look like this:

```
...  
...  
<APPLET  
    CODEBASE = "http://131.120.1.91/database/JBuilder/myApplet"  
    CODE      = "lastApplet.myApplet.class"  
    ARCHIVE   = "myApplet.jar"  
    NAME      = "JDBC Database Connectivity Applet"  
    WIDTH     = "400"  
    HEIGHT    = "125"  
    HSPACE    = "0"  
    VSPACE    = "0"  
    ALIGN     = "middle">  
</APPLET>  
...  
...
```

Basically what an applet is just a panel that is being placed onto the html page. The dimensions of the panel are hard-coded inside the actual code. The height and the width parameters in the html file should match with those hard-coded values. If you specify smaller height and width values in the html file, your applet might get cut off (that is, you will not be able to see some parts of your applet).

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
8725 John J. Kingman Road, Ste 0944
Ft. Belvoir, Virginia 22060-6218
2. Deniz Kuvvetleri Komutanligi.....2
Personel Daire Baskanligi
Bakanliklar
Ankara, TURKEY
3. Deniz Harp Okulu Komutanligi.....1
Kutuphane
Tuzla, Istanbul, TURKEY 81704
4. Dudley Knox Library2
Naval Postgraduate School
411 Dyer Rd.
Monterey, California 93943-5101
5. Chairman, Code CS.....1
Naval Post Graduate School
411 Dyer Rd.
Monterey, CA 93943-5101
6. Dr. C. Thomas Wu, Code CS/KA1
Naval Postgraduate School
Monterey, California 93943-5100
7. LCDR Chris Eagle, Code CS/EA1
Naval Postgraduate School
Monterey, California 93943-5100
8. LTJG Ramis Akin..... 2
Ilk Adim Sanayi Sitesi
290/17 Kutlukent
Samsun, TURKEY 55267
9. CPT Fred O'Brien3
38 Beacon St.
Hyde Park, Ma 02136