XVIEW: An XML Visualization Interface for Biological Information Systems

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of
Master of Engineering
in
Computer Science and Engineering

by

Amitabha Ghosh

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

July 2000

Dedicated to my loving parents and

to Pinky

Contents

Acknowledgements **iv**

Abstract **v**

1 Introduction **1**
1.1 Motivation 1
1.2 Contribution 2
1.3 Paper Organization 2

2 Oshadhi Architecture **4**
2.1 Overall System Architecture 4
2.1.1 Query Processor 4
2.1.2 Object Services 5
2.1.3 Spatial Services 5
2.1.4 Genome Services 5
2.2 Visualization Architecture 6

3 Biodiversity Data and System Requirements **8**
3.1 Biodiversity Information 8
3.1.1 Taxonomic Data 8
3.1.2 Geospatial Data 9
3.1.3 Genome Data 9
3.1.4 Bibliographic Data 9
3.2 System Requirements 10
3.2.1 Modeling requirements 10
3.2.2 Typical queries 10

4 System Design and Modeling **12**
4.1 Library Package 12
4.1.1 Swing Library package 13
4.2 Network Interconnection 14
4.2.1 Java sockets 14
4.2.2 Remote Procedure Call 15
4.2.3 Remote Method Invocation 15
4.3 The Native Interface 15

4.4 Displaying Query Results . . . 16
4.4.1 XML . . . 16

5 Implementation Details 18
5.1 Data Transaction between the GUI and the Database server . . . 18
5.2 OQL Query Generation . . . 20
5.2.1 Example queries in OQL form . . . 20
5.3 Query Results Exchange . . . 21

6 Related Work 24

7 Current Status and Future Work 26

8 Conclusion 28

Bibliography 30

A Remote Method Invocation 32
A.1 Architectural Overview . . . 33
A.2 How RMI works . . . 35
A.2.1 Develop Remote Object Code . . . 36
A.2.2 Develop the Server Code . . . 36
A.2.3 Setting up a security manager . . . 39
A.2.4 Develop the Client Code . . . 40
A.2.5 Compile the code . . . 41
A.2.6 Start the Registry . . . 41
A.2.7 Start the Server . . . 41

B Java Native Interface 42
B.1 Where JNI might be used . . . 42
B.2 Steps to write a JNI application . . . 44

List of Figures

2.1 Oshadhi Architecture 6
2.2 Oshadhi Visualization Architecture 7

4.1 Swing's Component Hierarchy 14

5.1 Meta-Data Export Format in BNF 19

6.1 GUI screen shot for Taxonomic Database Operations 25

7.1 GUI screen shot for Spatial Database Queries: Spatial characteristics are specified using this form 27

8.1 GUI screen shot for Taxonomic Database Queries: Taxon characteristics are entered using this form 29

A.1 RMI system architecture 34
A.2 RMI system uses a Web server to load class byte codes, from server to client and vice versa. 35

B.1 How legacy programs use the JNI 43
B.2 How Java applications use the JNI 44
B.3 JNI with Java and native applications 45
B.4 Constructing the HelloWorld native method name 46
B.5 Steps to write a Java program with native methods 47

Acknowledgements

When I think of people I should thank, an endless list of names runs across my mind and here I'm restrained to mention only a very few of them, who really influenced me.

I express my deep sense of gratitude to my project guide Dr. Jayant Haritsa for his encouraging advice, which was always forthcoming, whether it be in technical or personal matters. I consider myself very fortunate to have him as my guide, and can only hope I proved myself worthy of his guidance.

I owe a lot to each and every one of my friends, especially my fellow labmates and the coffee gang. Special thanks to Srikanta for bearing with me and also fighting and arguing with me to accept or to refute my opinion during the course of this project. Although, most of the times it turned out to be fruitful. Also thanks to Raja, Satheesh and Madhav who made my stay in DSL a memorable one.

I would also like to thank to all the faculties of the CSA department, for being great teachers and to all the staff in the department who helped me a lot during the presentation.

Special thanks to Anup, Amit and Mota who have always been there with me since the good old days of IIT Kgp, and who have made my life a cherishable one. Also thanks to all my other friends in this institute for the lesson teaching experiences and for all the spice they added to my life in the past four years.

Finally, thanks to "basant" (my favourite computer in the lab) who has never given up on my efforts to install many new things and trying out even crazy things on it.

Abstract

Oshadhi is an object oriented Database Management System tuned for bio-diversity applications. In such applications, visualization plays a very important role because of the complex nature and enormous size of this ecological data, with arbitrary nested inherent aggregation hierarchies. In this report, we have talked about the visualization interface, XVIEW [1]*, that we have built to suit the needs of bio-diversity applications. In our system, the database server can be accessed through an Internet browser, because the visualization interface is designed to be a web-enabled front end. We have also designed a DTD (Document Type Definition) which validates the XML formatted meta-data and query results, that are exchanged over the network. The architecture of the interface has been made generic, to some extent, so that it can handle issues related to database schema change.*

[1] XML Visualization Interface for Ecological World

Chapter 1

Introduction

1.1 Motivation

In recent times there has been a growing need for biological and ecological information to be conserved and documented. The rich pharmaceutical value of medicinal plants and discovery of new plant species in every year, make it essential, that the bio-diversity of the plant kingdom be conserved. There is also a need to visualize this information in sophisticated ways, because query results in biological and ecological domains, tend to be either enormous in size (geospatial queries, sequence homology searches etc) or complex with deep aggregation hierarchies (phenotype classification results). It case of spatial queries, which concern about the geographical distribution of plants and different climatic conditions, the results can vary from simple text to complicated overlapping images and multidimensional spatial objects. In order to make sense of these results, a proper visualization interface is needed to enable the end user query the database and visualize the results of such queries. Thus, visualization techniques play a very important role in Biological and Geographical Information systems.

1.2 Contribution

In *Oshadhi*, the Biological Information System being developed in the Database Systems Lab, we have built a visualization interface which can be used to query the database server and view the results of such queries in sophisticated ways. With the rapid growth of Internet throughout the world, there cannot be a better medium for information exchange than the World Wide Web. Keeping this view in mind, the visualization interface has been designed to be accessible through a web-enabled browser (Netscape, Internet Explorer 5 or above etc) over the Internet.

The visualization module consists of a GUI-client and a GUI-server that interacts with the query processor. The GUI-server is built as a layer of the query processor and the GUI-client is built as a Java applet, using the Swing library package of JDK1.2. The whole application is deployed on the web. During execution, server-GUI classes are downloaded to the client machine, where queries are formed and passed over the network, using the framework of Remote Method Invocation, to the query processing layer. The query processor then executes the query and sends the object results back to the GUI-client. These query results are then displayed in the GUI-client using XML, which conforms to a DTD. The interface provides facilities to browse species characteristics for taxonomic database and geographical characteristics for spatial database. Generic database queries can be formed using the GUI. It also provides facilities to insert and delete tuples from the database.

1.3 Paper Organization

The rest of the report is organized in the following fashion. Section 2 describes the overall Oshadhi architecture and specifically the visualization architecture. Section 3 provides details on bio-diversity information and about the system requirements for the visualization module. Several design issues and guidelines are presented in Section 4. In Section 5, we talk about the implementation details, the query generation and about the query results exchange between the GUI and the database server. Section 6 contains related work. Current status of the work and future research avenues have been described in Section 7. Finally, Section 8 summarizes our work.

Chapter 2

Oshadhi Architecture

2.1 Overall System Architecture

The goal of the Oshadhi project is to design and implement a database management system which enables efficient processing of bio-diversity data. It is designed as an object oriented system [D$^+$94b] tuned for ecological applications. The system is built on top of an object-oriented storage manager, SHORE [C$^+$94] (Scalable Heterogeneous Object Repository), from the University of Wisconsin, Madison, which provides the basic database features like transaction processing, persistence and recovery. The high level architecture of the system is depicted in Fig 1. As the figure shows, the Oshadhi system comprises mainly five components, viz., Query Processor, Spatial Services, Object Services, Sequence Services and Visualization Client. In the rest of this subsection we describe briefly about the first four modules. The Visualization Module is discussed in more detail in the next subsection.

2.1.1 Query Processor

The Query Processor is built on top of a freely available rule based query processor optimizer, Lamdba-DB [Kon00], which works with ODL and OQL as its data definition language and query language respectively. The query processor is also responsible for interacting with the client. A separate layer is built on top of Lambda-DB which converts the results obtained from the database query into a tag based data exchange format and pass it to the visualization client. The query processor module is discussed in more details in [Kon00].

2.1.2 Object Services

In an object-oriented system, schema basically consists of inheritance hierarchies, which are built by deriving attributes and behaviors from existing data-types and aggregation hierarchies, which are built by combining the existing data-types. In such systems, queries can be based on the position of the object in inheritance hierarchy and/or in aggregation path. In order to facilitate processing of such queries, Object Services are designed to provide index structures to the system. More information on Object Services can be found in [Tad00]

2.1.3 Spatial Services

The spatial part of the bio-diversity data is atleast a two dimensional map with longitude and latitude forming the two independent dimensions. The Spatial services module provides multidimensional index structures on geospatial data. In addition, certain primitive data types such as: point, polyline, polygon, layer, network are needed for modeling of spatial data. The spatial objects which are built on these data types possess certain inter relationships, among which only six such as: disjoint, in, touch, equal, cover and overlap are important in geo-spatial applications. The Geometric Algorithms part of the Spatial Services currently implements these six relationships. The Spatial services are discussed in [Che00].

2.1.4 Genome Services

Biological sequences are DNA strings and genes are subsequences of such strings which are responsible for determining certain species characteristics (such as color of hair, sex, etc). The similarity of such subsequences is an indicator of similarity in species characteristics.

Thus, in genome databases, a common query is to locate similar sequences to a given DNA or Protein sequence. Sequence Services provide similarity search algorithms over biological sequences. In addition, the Genome Services also provide datamodel consisting of DNA, RNA and Protein sequence classes.

Figure 2.1: Oshadhi Architecture

2.2 Visualization Architecture

The visualization interface is a web enabled front-end consisting of a GUI-client and a GUI-server. There is a fixed set of protocols to communicate between the GUI-client, the GUI-server and the Query Processor. The architecture is depicted in Fig 2. During execution server-GUI classes are downloaded to the client machine using the security mechanism provided by the Java RMI framework. Once the client GUI is generated, queries are formed through user interactions by filing appropriate, customized forms and menus. These queries are then sent to the GUI-server over the network through the same RMI framework. GUI-server application program then interacts with the Query Processor through the Java Native Interface and passes the query into it. The query processor processes the queries using the services of Taxonomy, Spatial and Genome modules and passes the results back to the GUI-server which in turn passes the results to the GUI-client where they are displayed appropriately using XML.

Figure 2.2: Oshadhi Visualization Architecture

Chapter 3

Biodiversity Data and System Requirements

3.1 Biodiversity Information

Biodiversity data can be broadly classified into four major groups: taxonomic data, geographic data, genome data and bibliographic data.

3.1.1 Taxonomic Data

Classification of plant species based on their similarities and differences are made in taxonomy and each of these classifications is called a taxon. The four major taxons studied in taxonomy are Order, Family, Genera and Species. These taxons form a four level aggregation hierarchy and they are interrelated. An order consists of several families which exhibit certain similar characteristics. Likewise, a family consists of a collection of genera and a genera comprises a set of related species [Vas74]. These taxons are typically identified based on certain unique characteristics. The taxonomic data also exhibits several inter-relationships: for example, two species might have a parasitic, symbiotic or predator relationship between them.

3.1.2 Geospatial Data

Spatial or geographical data consists of bio-geographic provinces, forest reserves, biomes, botanical gardens and herbariums. Every geographical region has a map and a set of environmental conditions such as soil characteristics, temperature, humidity and vegetation type, associated with it. There is a rich inter-relationship between the taxonomic data and the geographic data. The distribution of species in a geographic region is an example of this kind of relationship.

3.1.3 Genome Data

Genome data mainly consists of biological sequences. These are the DNA sequences that form the blueprint of a cell, which differ from species to species. Protein sequences derived from DNA are important in analysis of species characteristics. Similarity between two such gene sequences is a measure of similarity between species. Analysis of DNA and protein sequences from different sources gives us important clues about the structure and function aspects of proteins, evolutionary relationships between organisms and perhaps helps us in discovering drug targets for various related pathogenic organisms.

3.1.4 Bibliographic Data

Bibliographic data includes abstracts and references of bio-diversity related journals and textbooks. A journal has a collection of articles which refer to species or geographical regions.

Although the data model developed for the Oshadhi system supports all the above four kinds of data, in this paper we primarily focus on the visualization interface for taxonomic data and geographic data because these are the main components of the bio-diversity domain.

3.2 System Requirements

3.2.1 Modeling requirements

Modeling the graphical user interface to support the above kind of ecological data needs several issues to be looked at:

1. The system should support the inherent aggregation hierarchy of the taxonomic data, the maps and the environmental conditions associated with the geographic data.

2. Typical queries on taxonomic data require specifying the identifying properties of a taxon and getting information about that taxon matched on the specified properties. The GUI should provide provisions for entering the characteristics of the taxons and form the query.

3. Spatial queries usually require displaying of maps and extracting information about geographic regions which have certain kinds of environmental characteristics or in which certain kinds of species are distributed.

4. Sometimes there are needs to modify the schema in the bio-diversity application. For instance, new types of characters may be added to aid the process of identification or new geographical organizations may evolve with time. The model for the GUI should be flexible enough to support these features.

5. In today's world, the world wide web is the best medium for information exchange. Keeping this view in mind, it would be most suitable for the visualization interface to be a web-enabled front-end, where the end user fills up the customized forms to form their specific queries and submit it to the GUI server over the network in an efficient and secure way and get back the results.

3.2.2 Typical queries

Here we list some of the typical queries in the bio-diversity application:

- View the information about an order, a family, a genera or a species having a certain set of characteristics.
- Display the map of a geographic region which grows certain vegetation types and possesses certain soil properties.
- Display the distribution of a species in a particular geographic location.
- Identify a species based on certain flower / leaf / seed / fruit characteristics.

Chapter 4

System Design and Modeling

In this section, we discuss the issues involved in the system design for visualization architecture, the other choices we had for implementing the system, the ones which we chose and reasons for our choices.

4.1 Library Package

Over the years various interface libraries have come up that provide configurable interface components, which allow the developers to implement sophisticated graphical user interfaces.

The cross-platform GUI library, wxWindows, allows C++ applications to compile and run on several different types of computers. This toolkit consists of one library per supported GUI, such as Motif [Bra92] or Windows. Besides providing a common API (Application Programming Interface) for GUI functionality, it provides functionality for accessing some commonly-used operating system facilities, such as copying or deleting files. wxWindows is a 'framework' in the sense that it provides a lot of built-in functionalities which the application can use or replace as required, thus saving a great deal of coding effort.

4.1.1 Swing Library package

Inspite of the advantages provided by the above mentioned software packages, we have chosen the Swing package [E+98] from the Java Foundation Classes (JFC) [E+98] for the following reasons:

- Swing development has its roots in the Model-View-Controller (MVC) architecture. The MVC-based architecture allows Swing components to be replaced with different kinds of data models and views. The Swing library supports a cross-platform look-and-feel, also called the Java look-and-feel, that remains the same across all platforms where ever the program runs. It also allows the developer to create his/her own look and feel.

- Swing supports lightweight component development. For a component to qualify as lightweight, it cannot depend on any native system classes, also called "peer" classes. In Swing, most of the components have their own view supported by Java look-and-feel classes instead of depending on any peer classes. Swing supports a good number of sophisticated user interface lightweight components such as buttons, check boxes, combo boxes, tables, text fields and so on as depicted in Fig 3.

- Swing components function based on the delegation event model (introduced in JDK1.1). The delegation event model supports a clean separation between the core program and its user interface. The delegation pattern allows robust event handling that is less error prone due to strong compile checking. In addition, the delegation event model improves performance because the toolkit can filter out undesirable events (like high frequency mouse events) that are not targeted to execute any function.

- As the underlying platform for the Swing package is Java it allows an application to be platform independent and to be easily deployed on the web browser.

Figure 4.1: Swing's Component Hierarchy

4.2 Network Interconnection

Having decided which software library to use for the GUI development we had a couple of choices for deploying the network interconnection between the user interface and the query processor.

4.2.1 Java sockets

The first choice for implementing the basic network communication was Java sockets [N+99], which are flexible and sufficient for general purpose communication. Here, the server address needs to be hard coded in the source code and whenever the client (GUI) sends a request to the server it sends to the fixed server IP and port number. The server always listens to that port number for incoming requests and when it gets one, it forks one process to which it gives the request to handle and it continues to listen to the same port for more incoming calls. However, sockets require the client and the server to engage in application level protocols to encode and decode messages for exchange. Designing such protocols is cumbersome and can be error-prone.

4.2.2 Remote Procedure Call

An alternative to sockets is Remote Procedure Call (RPC) [Sta97], which abstracts the communication interface to the level of a function call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR.

4.2.3 Remote Method Invocation

However, RPC doesn't translate well into distributed object systems, where communication between program-level objects residing in different address spaces is needed. Remote Method Invocation (RMI) [Dow98], provided by Java platform matches the semantics of object invocation, where a local surrogate (stub) object manages the invocation of a procedure on a remote object. It can dynamically resolve method invocations across Virtual Machine (VM) [L+99] boundaries, thus providing a fully object-oriented (OO) distributed environment. Besides, RMI supports various reference semantics for remote objects, such as live references, persistent references and lazy activation. It also provides a robust security mechanism for downloading classes and a distributed garbage collection algorithm for active objects. Thus, the network interconnection between the GUI client and the Database server has been deployed using Remote Method Invocation.

4.3 The Native Interface

The GUI server application program in our visualization architecture needs to call C++ methods from the query processor. For this, we have chosen the Java Native Interface (JNI) [Lia99]. It allows Java code that runs inside a Java Virtual Machine (JVM) to interoperate with applications and libraries written in other programming languages, such as C, C++ and Assembly. JNI also lets the native method to create, update and access Java objects and share them. The native methods, using the JNI can call the existing Java method, pass it the required parameters and get the results back when the method completes.

4.4 Displaying Query Results

4.4.1 XML

Once the query has been executed in the database server, the results have to be transported back to the visualization interface for rendering on the screen. There are several data exchange formats, such as: DELTA (Description Language for Taxonomy), MEDATLAS (for exchanging oceanographic data), CSDGM (Content Standards for Digital Geospatial Metadata), DIGEST (Digital Geographic Information Exchange Standards) etc, which are used by different commercial applications for exchanging taxonomic and geospatial information. In Oshadhi, we have used *XML* (eXtensible Markup Language) for meta-data and query-results exchange between the GUI and the database server. Reasons for choosing XML as the data exchange format are as following:

XML is structured : Using XML, structural relationships that exist inside the data can be embedded into it. A document type definition (DTD) is used to create these relationships and the XML document conforms to this DTD. The validity and integrity of the document is checked by an XML parser.

XML is platform independent, textual information : Information in an XML document is stored in plain text. There are several advantages of keeping things in plain text. First, it is easy to write parsers and all other XMl enabled technology on different platforms. Second, it makes everything very interoperable by staying with the lowest common denominator approach.

XML is language independent : By being language independent, XML bypasses the requirement to have a standard binary encoding or storage format. Language independence also fosters immense interoperability amongst heterogeneous systems and it is also good for future compatibility.

XML is web enabled and totally extensible : XML is derived from SGML (Standard Generalized Markup Language). So in essence, the current infrastructure available today to deal with HTML content can be re-used to work with XML. Even if clients donot support XML natively, there are Java Servlets which can convert XML with style-sheets, to generate plain HTML that can be displayed in all web browsers. Using XML to pass parameters and return values on servers, makes it very easy to allow these servers to be web-enabled. Finally, by not predefining any tags in the XML Recommendations, the W3C (World Wide Web Consortium) allowed developers full control over customizing the data. This makes XML very attractive to encode the data that already exists in legacy databases

Chapter 5

Implementation Details

5.1 Data Transaction between the GUI and the Database server

Database Schema and Meta-Data Exchange : There are several occasions when data transaction between the client-GUI and the query processor is needed.

The first transaction takes place when initial connection is made to the database server and the GUI classes are loaded to the client machine requesting for database schema and any other meta-data that might be useful in processing the queries at a later stage. The query processor takes up this request and searches the database for such information and ships it over to the client GUI. The format in which this meta-data is exported to the visualization interface follows the BNF (Backus Naur Form) grammar which is shown in Fig 4. This information transaction is very much essential, because the database schema might be updated at any point of time and the GUI must take care of the changes. The schema might change for various reasons such as:

- new character types may be added to the taxonomic database to aid the process of species identification and their distribution in different geographical locations,
- or new geographical organizations may evolve with time

Figure 5.1: Meta-Data Export Format in BNF

Once this meta-data are exported from the query processor, the user interface is generated according to that schema. For example,

- the data types that are present in the database such as: taxonomy, spatial, genome, bibliographic;
- the kinds of operations that are supported by the query processor; such as: adding a taxon to the taxonomy database, viewing special characteristics of a species or searching the database that meets certain specified characteristics on a taxon and a geographical location;
- the characteristics of each individual species;
- the properties of the geographical locations, etc

are to be taken into consideration when generating the user interface.

5.2 OQL Query Generation

Once the user interface is generated from the database schema the end user can navigate through the GUI and form the query by filling appropriate customized forms and menus. This query is then translated into an Object Query Language called OQL [Kon00].

OQL is a declarative query language standard proposed by ODMG (Object Data Management Group) [K^+93] and has been widely accepted as a query language for OODBMS. OQL, by itself is not computationally complete; however queries can invoke methods written in the host language and any host language method can include OQL queries. OQL supports only B+ tree [Che00] index structures which is not enough for Oshadhi. So it has been extended to support several index structures.

5.2.1 Example queries in OQL form

Here, we give a flavor of the query language designed for Oshadhi by showing some example queries.

1. **Object Creation:**

   To create a persistent object, say of type City the OQL query takes the following form:

   po := City(location:p, belongsToStates:s);

   where s and p are state and point objects created similarly. po is the newly created City object.

2. **Query on a Class hierarchy:**

   To search for a species which meets certain characteristics:

   s := Species(name : A, Properties(

   [Stalk] : SUB_SESSILE,

   [Arrangement] : DISTICHOUS,

   [OvaryPosition] : SUPIRIOR));

FOR EACH G in Genera

WHERE G.name = "B"

G.Search(S);

This will first create a Species object s, which has name A and properties as specified, and will search in Genera having name B for information about that Species.

3. **Query with a Join:**

   To find all lakes which flow through the city Bangalore the query will be:

   FOR EACH i IN (

   *SELECT * FROM c IN cities, w IN waterbodies*

   WHERE c.name = "Bangalore"

   AND c.location.overlap(w.location)

   AND w.type = LAKE)

   DO cout << i →w→name << endl;

Once these OQL queries are generated by the client-GUI, they are passed over the network through Remote Method Invocation to the server GUI and then, to the query processor through Java Native Interface. The query processor first parses the query for its correctness and then optimizes it. Then the physical plans are generated and the C++ code is produced for the query in terms of these physical plans. Finally, the query is executed and the results are shipped back to the visualization interface.

5.3 Query Results Exchange

There are several issues which come up when the query results are shipped back to the visualization interface. First of all, the mechanism by which the results are going to be passed back, and then the way in which the visualization module is going to interpret these results.

The first problem is tackled by using Remote Method Invocation, where the results are packaged into an object and then transmitted through the network using the concept of Java Object Serialization.

The way in which the visualization interface interprets the query results is very important, because there can be various types of results and each of them might need special treatment for displaying them. So the GUI server needs the type of these results. For example, output of a query can be a set of numbers which might represent a set of points describing a polygon or it might be the salary of an employee. In the former case, the visualization interface has to render the points and display the polygon, whereas in the latter case it has to just print them. So the results must also have associated meta-data describing the structure and context of the output. But the issue is who is responsible for generating this meta-data. There are two solutions to this:

1. The user writes the code for displaying the object's attribute values and the meta-data in the action part of the query in a format acceptable by the GUI-server.

2. Oshadhi will provide a function which does all this chores of displaying the data and meta-data and the user can just call this object that is to be displayed as parameter.

We have followed the second approach by providing a function *OshadhiDisplay(object)*, which traverses through the object recursively and outputs all the attributes of the object including the meta-data. A DTD (Document Type Definition) has been designed to structure the query results. These results are then packaged into XML format which conforms to the DTD and validated by an XML parser and then shipped back to the GUI-client. In the following, we provide some part of the DTD that we have written based on the different data models implemented in Oshadhi.

Having decided how to generate the meta-data and how to pass it over to the visualization module the next issue is of displaying the query results according to its meta-data. This meta-data which specifies the structure and context type of the object is nothing but XML tags [B$^+$98], which have meanings as described in the DTD. So an XML supported browser (such as, IE5) can be used to view the query results. A Formatter class in the

```
<!-- Top level declarations -->
<!ELEMENT OSHADHIDISPLAY    (TAXONOMY | SPATIAL | (TAXONOMY, SPATIAL))>
<!ELEMENT TAXONOMY          (ORDER*, FAMILY*, GENERA*, SPECIES*)>
<!ELEMENT ORDER             (NAME, FAMILY*, GENERA*, SPECIES*)>
<!ELEMENT FAMILY            (NAME, GENERA*, SPECIES*)>
<!ELEMENT GENERA            (NAME, SPECIES*)>
<!ELEMENT SPECIES           (NAME, PROPERTIES*)>
<!ELEMENT NAME              (#PCDATA)>
<!ELEMENT PROPERTIES        (FlowerChar*, LeafChar*, BranchChar*, FoliageChar*,
                             CanopyChar*, StemChar*, HabitChar*)>

<!-- Toplevel Flower Properties -->
<!ELEMENT FlowerChar        (Arrangement?, Stalk?, Placentation?, InfloChar?,
                             SepalChar?, PetalChar?, Claw?, Aestivation?,
                             AntherChar?, OvaryChar?, StyleChar?, StigmaChar?)>

<!-- Enum values -->
<!ELEMENT Arrangement       (keyword)+>
<!ELEMENT Stalk             (keyword)+>
<!ELEMENT Placentation      (keyword)+>

<!-- InfloChar -->
<!ELEMENT InfloChar         (InfLocation?, FertilePart?, AxisNature?,
                             AxisThickness?, AxisOutline?)>
<!ELEMENT InfLocation       (keyword)+>
<!ELEMENT FertilePart       (keyword)+>
<!ELEMENT AxisNature        (keyword)+>
<!ELEMENT AxisThickness     (keyword)+>
<!ELEMENT AxisOutline       (keyword)+>

<!-- SepalChar -->
<!ELEMENT SepalChar         (NerveChar?, SymmertyChar?, Form?, FusionChar?, Color)>
<!ELEMENT NerveChar         (keyword)+>
<!ELEMENT SymmertyChar      (keyword)+>
<!ELEMENT Form              (keyword)+>
<!ELEMENT FusionChar        (keyword)+>
<!ELEMENT Color             (keyword)+>
...
```

Table 5.1: Part of the Document Type Definition (DTD) for the Taxonomic Object Model

query processor does the job of converting the query results into an XML format. Some part of the DTD, which is based on the Oshadhi Object Model is depicted in Table 1.

At the end of this thesis we have shown some of the screen capture imagaes of the GUI.

Chapter 6

Related Work

There have been several efforts to build visualization interfaces for biological and GIS applications. The educational software, *Paradise* [D$^+$94a], from the University of Wisconsin, has a GUI mainly for GIS applications. The Paradise front-end has been implemented using Tcl/Tk and it includes key features such as: display of objects with spatial attributes on a 2-D map, layered display of overlapping objects, updating objects from the GUI etc. But the GUI cannot be deployed on the web because of the underlying platform. The *Idrisi* software package for GIS and image processing applications provides raster geographic analysis and a set of extensive tools for image processing. There are other commercial GIS applications such as *ArcView, MapInfo, ERDAS, ERMAPPER* which are mainly used for map display but they are not capable on being deployed on the web through Internet supported browsers.

Figure 6.1: GUI screen shot for Taxonomic Database Operations

Chapter 7

Current Status and Future Work

Currently the Visualization interface supports display of taxonomic and spatial database query-results. The format of information exchange between the query processor and the GUI is XML. The system has been designed to be web-enabled so that database queries can be executed over the network through the world wide web. The architecture of the GUI has been implemented to be generic and flexible so that it can take care of issues like update of database schema.

The code for the visualization interface is around 7000 lines, which includes the code for the RMI and the JNI interface with the query processor and the code for the XML DTD. A total number of 57 classes have been implemented for the GUI code.

Visualization of genome sequences has to be designed and implemented and much more sophisticated display facilities of Spatial database queries have to be done. The GUI can be made much more generic by exchanging more amount of meta-data thus more effectively taking care of schema change. A multi-lingual support can also be provided for end users to access the database, because the biological community consists of researchers and experts from various parts of the world coming from different linguistic background. So a multi-lingual support will make their work much simpler. A better way of customizing the visualization interface can also be provided, for example, the end user can view the GUI with his own specified look and feel.

Figure 7.1: GUI screen shot for Spatial Database Queries: Spatial characteristics are specified using this form

Chapter 8

Conclusion

Oshadhi is a general purpose OODBMS tuned for biological applications. It supports ODL and OQL for data definition and database querying respectively. In this paper we have talked about the visualization architecture and implementation details for a general purpose ecological application. The whole application can be accessed through a web enabled browser and database queries can be formed and executed over the network. The mechanism for meta-data and query results exchange between the database server and the end user is XML, which conforms to a document type definition (DTD). It also supports display of maps and species-specific characteristic for geospatial and taxonomic queries.

Figure 8.1: GUI screen shot for Taxonomic Database Queries: Taxon characteristics are entered using this form

Bibliography

[B$^+$98] Tim Bray et al. *Extensible Markup Language Specifications (XML) 1.0.* W3 Consortium, W3C Recommendation, 1998.

[Bra92] Marshall Brain. *Motif Programming : The Essentials...and More.* X and Motif Series. Digital Press, 1992.

[C$^+$94] Michael J. Carey et al. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD Conference*, 1994.

[Che00] Madhav Rao Cheethirala. The spatial perspective of oshadhi. Master's thesis, Dept. of Computer Science and Automation, Indian Institute of Science, Jan 2000.

[D$^+$94a] David J. DeWitt et al. Client-server paradise. *Proceedings of the 20th VLDB Conference, Santiago, Chile*, 1994.

[D$^+$94b] Asuman Dogac et al., editors. *Advances in Object-Oriented Database Systems*, volume 130 of *F:Computer and Systems Sciences.* NATO ASI Series, 1994.

[Dow98] Troy Bryan Downing. *Java RMI : Remote Method Invocation.* DG Books Worldwide, 1998.

[E$^+$98] Robert Eckstein et al. *Java Swing.* O 'Reilly & Associates, 1998.

[K$^+$93] Morgan Kaufmann et al. *The Object Database Standard: ODMG-93.* ODMG-93, 1993.

[Kon00] Satheesh Konidala. Query processing and optimization in oshadhi. Master's thesis, Dept. of COmputer Science and Automation, Indian Institute of Science, Jan 2000.

[L+99] Tim Lindholm et al. *The Java Virtual Machine Specification.* Java Series. Addison-Wesley Pub Co., 1999.

[Lia99] Sheng Liang. *The Java Native Interface : Programmer's Guide and Specification.* Java Series. Addison-Wesley Pub Co., 1999.

[N+99] Patrick Naughton et al. *Java2 : The Complete Reference.* Osborne McGraw-Hill, 1999.

[Sta97] Open Group Technical Standard, editor. *Remote Procedure Call.* The Open Group, 1997.

[Tad00] Rajarao Tadimeti. Database habitats for biological species: Handling taxonomy data in oshadhi. Master's thesis, Dept. of Computer Science and Automation, Indian Institute of Science, Jan 2000.

[Vas74] P.C. Vasistha. *Taxonomy of Angiosperms.* S. Chand and Co., Delhi, 1974.

Appendix A

Remote Method Invocation

Remote Method Invocation (RMI) provides a means of communication between Java applications using normal method calls, and offers the capability for the aplications to run on separate computers - located perhaps as far as on opposite sides of the globe.

One important feature of RMI is that it presents a programmatic interface for networking rather than relying on the sockets and streams approach. The method's major advantage is that it offers you a higher-level, method based interface in which a remote object is treated as though it were local. RMI is also convenient to use and more natural in many ways thatn using sockets. A requirement, however, is that you must run Java on both ends of the network connection-rather than simply both systems implement the same networking protocol.

Here are the steps for building a working RMI application:

1. Develop the remote object, server, and client code
2. Compile the code to generate the class files
3. Run the RMI compiler (rmic) to generate the stub and skeleton
4. Move the .class files to an appropriate location
5. Start the RMI registry
6. Start the server

7. Start the client

In the following paragraphs, the RMI framework, the architecture and the way in which RMI works is explained through an example code.

A.1 Architectural Overview

The RMI system consists of three layers:

- The stub/skeleton layer - client-side stubs (proxies) and server-side skeletons
- The remote reference layer - remote reference behaviour (such as invocation to a single object or to a replicated object)
- The transport layer - connection set up and management and remote object tracking

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.

A remote method invocaion from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote objest as a conduit to the remote object. A client held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote reference layer. Stubs are generated using the rmic compiler.

The remote reference 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 communication with multiple locations. Each remote object implementation chooses its own remote references semantics - whether the server is a single object or is a replicated object requiring communications with its replicas.

Also handled by the remote reference layer are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring objects

Figure A.1: RMI system architecture

that are implemented in (a) servers that are always running on some machine, and (b) servers that run only when some method invocation is made on them (activation). At the layers above the remote reference layer, these differences are not seen.

The transport layer si responsible for connection setup, connection management, and keeping track of and despatching to remote objects (the targets of remote calls) residing in the transport's address space.

In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behaviour that needs to occur before handling off the request to the server-side skeleton. The skeleton for a remote object makes an up call to the remote object implementation which carries out the actual method call.

The return value of the call is sent back through the skeleton, remote reference layer, and transport on the server side, and then up through the transport, remote reference layer, and stub on the client side.

A.2 How RMI works

With RMI, client and server programs are running, with both using Java. You specify a Java interface to describe each object to be shared remotely, and enumerate the public methods to be invoked on the object. A server will use the interface and create objects that implement it and that are also registered with a name registry service using a URL-based naming scheme.

A client will contact the registry, attempt to retrieve an object by name, and obtain a remote reference to it. You handle method invocation and data passing via skeleton and stub classes that are generated automatically through an RMI compiler from user code, and through communication using object serialization and TCP/IP.

A.2.1 Develop Remote Object Code

RMI supports Java objects communicating via their methods, regardless of where the objects are located. The first step in creating a class to be accessed remotely is to define an interface describing the methods that are available remotely. Following is a code fragment for the remote interface that we have used in our source code:

```
// Hello.java
...
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Hello extends Remote
{
    String sayHello(String queryString) thrown RemoteException;
    String[] ReadFile(String filename) throws RemoteException;
}
```

java.rmi.Remote is an empty marker interface that indicates that an object is remote, so class objects implementing Hello are marked as being remote references. All the methods in a remote interface must declare that they can throw an exception of type *java.rmi.RemoteException*, which gets thrown whenever a remote method invocation fails.

A.2.2 Develop the Server Code

Once you specify the remote object definition via an interface, the enxt step is to develop the server code. The server implements the objects interface and creates instances of the object to be shared remotely. Here we present a fragment of the source code:

```
// HelloImpl.java
...
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public String Format[] = new String[500];
    String jni_message;

    public HelloImpl() throws RemoteException {
        super();
    }
    public String sayHello(String queryString) {
        try {
            HelloImpl obj2 = new HelloImpl();
            ....
        }
        catch (Exception e) {
            System.out.println ("Exception..!!");
        }
```

```
        ...
    }
    public String[] ReadFile(String filename) {
        try {
            BufferedReader stdin = new BufferedReader(new
                 InputStreamReader (new FileInputStream(filename)));
            String in = "";
            int i = 0;
            try {
                while ((in = stdin.readLine()) != null) {
                    Format[i] = new String(in);
                    System.out.println (Format[i++]);
                }
            }
            catch (Exception e) {}
        }
        catch (FileNotFoundException fe) {
            System.out.println ("File not found!");
        }
        return Format;
    }
    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            HelloImpl obj = new HelloImpl();
            Naming.rebind("//144.16.79.148/HelloServer", obj);
```

```
                System.out.println("HelloServer bound in registry");
            }
            catch (Exception e) {
                System.out.println("HelloImpl err: " + e.getMessage());
                e.printStackTrace();
            }
        }
    }
```

HelloImpl extends *java.rmi.server.UnicastRemoteObject* and implements *Hello*. The first of these classes supplies soem of the basic properties needed for the remote objects, and the second defines the methods to be made available remotely.

A.2.3 Setting up a security manager

The tricky part of this code is what occurs in *main*. The first thing is to establish a security manager. RMI involves loading remote .class files, something like a web browser does with an applet, but this opeartion entails some security risk. If a security manager is not installed, teh default is to load only local files, and RMI by definition cannot work with this restriction. So a security manager has to be set to make explicitly clear that loading remote .class files is acceptable.

An instance of *Hello* then gets created and registered with the name registry service using *Naming.rebind*. This process associates an object reference with a specific name so that the client can look up the object by name through the registry. To actually invoke the remote method, the server and the client use skeleton and stub entities to communicate. These .class files are generated from the server .class files by the RMI compiler described below.

Conceptually, the stub class is:

```
public class HelloImpl_Stub extends java.rmi.server.RemoteStub
             implements Hello, java.rmi.Remote { ... }
```

and the skeleton class is:

```
public class HelloImpl_Skel extends java.lang.Object
                implements java.rmi.server.Skeleton { ... }
```

Usage like: *javap -c HelloImpl_Stub* displays the bytecodes and illustrates what goes on behind the scenes.

The stub is a surrogate for the remote object, and the skeleton is a server-side entity that dispatches calls to the remote object implementation. The stub takes care of collecting (or marshalling) the data and transmitting/receiving it on the client side. The skeleton performs similarly on the server side. Object serialization changes the data into a stream of bytes, which are transmitted via TCP/IP.

A.2.4 Develop the Client Code

```
// QueryView.java
...
public void actionPerformed (ActionEvent evt) {
    ...
    try {
        obj = (Hello)Naming.lookup("//" + M + "/HelloServer");
        message = new String(obj.sayHello(queryString));
        System.out.println ("Message from JNI & RMI: " + message);
        new DisplayResults(message, M,
               http://144.16.79.148/~amitabh/classes/cashew.htm");
        System.out.println("\nJNI & RMI over and Query end\n");
    }
    catch (Exception e) {
        System.out.println("HelloApplet Exception: " +
                                         e.getMessage());
    }
}
```

Once the server is implemented, implementing the client is straightforward. The security considerations for the client are the same as for the server. A URL is formed to contact the name registry, and *Naming.lookup* gets called with the server object name.

A.2.5 Compile the code

The three files Hello.java, HelloImpl.java, and QueryView.java are compiled using the javac compiler. Then the skeleton (HelloImpl_Skel.class) and the stub classes (HelloImpl_Stub.class) are generated using RMI compiler on the HelloImpl class: (rmic HelloImpl).

A.2.6 Start the Registry

An object being accessed remotely needs to be entered in a registry of objects. JDK1.2 provides a registry tool (*rmiregistry*) that can be used for this purpose. rmiregistry can run in a separate window or as a background process. It runs independently of the particular server and client but is required by both of them.

A.2.7 Start the Server

Next the server is started using the following command. Here URL is the location where the server class files are located and securityfile is a file which grants permission for a security manager to be installed on the client machine.

```
java -Djava.rmi.server.codebase=<URL>
     -DJava.security.policy=<securityfile> HelloImpl
```

In the final step the client is started.

Appendix B

Java Native Interface

The Java Native Interface (JNI) is the native programming interface for Java, using which, Java code that runs inside a Java Virtual Machine (JVM) can interoperate with applications and libraries written in other programming languages, such as C, C++, assembly etc. In addition, the *invocation API* allows to embed Java Virtual Machine into the native applications.

B.1 Where JNI might be used

Programmers use the JNI to write native methods when an application cannot be written entirely in Java. For example, native methods and JNI might be useful in the following situations:

- The standard Java class library may not support the platform dependent features that an application might have.
- A library or an application already written in another programming language might be needed to be accessed through Java applications.
- Sometimes a small portion of time critical code needs to be implemented in a lower level programming language, such as assembly, and then the Java application might want to call those functions.

Programming through the JNI framework lets you use native methods to do many operations. Native methods may represent legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

The JNI framework lets your native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

Figure B.1: How legacy programs use the JNI

Native methods can also easily call Java methods. Often, you will already have developed a library of Java methods. Your native method does not need to "re-invent the wheel" to perform functionality already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

The JNI enables you to use the advantages of the Java programming language from your native method. In particular, you can catch and throw exceptions from the native method and have these exceptions handled in the Java application. Native methods can also get information about Java classes. By calling special JNI functions, native methods can load Java classes and obtain class information. Finally, native methods can use the JNI to perform runtime type checking.

For example, the above figure shows how a legacy C program can use the JNI to link with Java libraries, call Java methods, use Java classes, and so on.

Figure B.2: How Java applications use the JNI

The above figure illustrates calling native language functions from a Java application. This diagram shows the many possibilities for utilizing the JNI from a Java program, including calling C routines, using C++ classes, calling assembler routines, and so on.

It is easy to see that the JNI serves as the glue between Java and native applications. The above diagram shows how the JNI ties the C side of an application to the Java side.

B.2 Steps to write a JNI application

In the following we will describe briefly the steps involved to invoke a native method from Java applications.

- The first step is to write the Java program. Create a Java class that declares the native method; this class contains the declaration for the native method and also

Figure B.3: JNI with Java and native applications

some function which calls the native method. The declaration of the native method looks as follows:

```
public native void displayHelloWorld();
```

- Compile the Java class, which declares the native method and also other functions.
- Generate a header file for the native method, using javah with the native interface flag -jni. Once the header file is generated, the formal signature of the native method is known. A sample signature of a native method is shown in the following:

```
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
                            (JNIEnv *, jobject);
```

The name of the native language function that implements the native method consists of the prefix Java_, the package name, the class name, and the name of the native method. Between each name component is an underscore ”_” separator. Graphically, this looks as depicted in the above diagram.The first parameter for every native method is a JNIEnv interface pointer. It is through this pointer that

Figure B.4: Constructing the HelloWorld native method name

your native code accesses parameters and objects passed to it from the Java application. The second parameter is jobject, which references the current object itself. In a sense, you can think of the jobject parameter as the "this" variable in Java.

- Write the implementation of the native method in the programming language of your choice, such as C, C++ etc.

- Compile the header file and the implementation files into a shared library. The runtime system later loads the shared library into the Java class that requires it. Loading the library into the Java class maps the implementation of the native method to its declaration.

- Finally, use the java interpreter to run the program.

Figure B.5: Steps to write a Java program with native methods