Making Digital Libraries Flexible, Scalable and Reliable: Reengineering the MARIAN System in JAVA

Jianxin Zhao

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science

Edward A. Fox, Chair
Sallie M. Henry Dennis G. Kafura

June 16, 1999
Blacksburg, Virginia

Keywords: Online Public Access Catalog, Digital Library, User Information Layer, Personalization, Project Management, Reengineering

Making Digital Libraries Flexible, Scalable, and Reliable: Reengineering the MARIAN System in JAVA

Jianxin Zhao

(ABSTRACT)

There is a great need for digital libraries that are flexible, scalable, and reliable. Few such systems exist. Little is known about how to build them. This thesis addresses these problems by enhancing a prototype digital library system with the aim of making it more flexible, scalable, and reliable.

We hypothesize that: 1) adding a new (user information) layer and maintaining weak coupling in the design of a digital library system can help achieve system flexibility; 2) optimizing network connection usage and facilitating distribution of computation and disk operations in system design can help achieve system scalability; and 3) applying good software processes can help university students produce a very reliable system.

Approaches based on the above hypothesis were used in the project of Reengineering the MARIAN System in Java. The results of the project and experiments verified the correctness of the hypothesis.

The results of this thesis may help inform future digital library design and implementation projects to produce flexible, scalable, and reliable systems.

Acknowledgments

I thank Professor Edward A. Fox for his advice, help and support during my thesis and throughout my graduate program in the Department of Computer Science at Virginia Polytechnic Institute and State University. I also thank the other members of my advisory committee, Professor Sallie M. Henry and Professor Dennis G. Kafura, for their comments and suggestions. Help and suggestions from other members of the Network Research Group and Digital Library Research Laboratory at Virginia Polytechnic Institute and State University also are appreciated.

Special thanks go to Robert K. France, who directed me through the reengineering project, gave me many comments, and provided much information about the MARIAN system.

A grant from the National Library of Medicine supported the reengineering project and my graduate study.

Contents

1 Introduction 1

1.1 Problem Statement 1

1.2 Hypothesis 2

1.3 Organization 3

2 Related Works 4

2.1 Digital Libraries and Online Public Access Catalog Systems 4

2.2 Hyper Text Transfer Protocol (HTTP) 5

2.3 Common Gateway Interface (CGI) 6

2.4 Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) 6

2.5 Java 7

2.6 Capability Maturity Model (CMM) 8

3 The C/C++ MARIAN System 9

3.1 MARIAN History 9

3.2 Major Feature Descriptions 10

3.3 Top Level Architecture 13

3.4 Operations 14

4 Reengineering the "Formit" 17

4.1 The C/C++ "Formit" *17*

4.2 The Java "Formit" *19*

4.3 Flexibility Considerations *21*

5 Reengineering the "Webgate" **23**

5.1 The C/C++ "Webgate" *23*

5.1.1 Architecture 23

5.1.2 Operations 26

5.2 The Java "Webgate" *27*

5.2.1 Architecture 27

5.2.2 Operations 32

5.2.2.1 System Startup 32

5.2.2.2 Handle "Formit" Requests 33

5.2.2.3 System Shutdown 35

5.3 Flexibility Considerations *36*

5.3.1 Adding a New Layer in System Design 36

5.3.2 Maintaining Weak Coupling 37

5.3.2.1 Top Level Weak Coupling 37

5.3.2.2 Weak Coupling Inside "User_manager" 38

5.3.2.3 Weak Coupling Inside "Uip_manager" 39

5.3.2.4 Weak Coupling Inside "Request_response" Thread 40

5.3.2.5 Weak Coupling Among "Results", "Request_response", and "Call_back_processor" 41

5.4 Scalability Considerations *42*

5.5 Java "Webgate" Major Feature Descriptions *43*

6 User Information Layer **53**

6.1 Architecture Description *53*

6.2 Benefits of the User Information Layer *55*
6.2.1 Reducing Workload of Search Engine 56
6.2.2 Personalization 56
6.2.3 Active System 56
6.2.4 Billing Capability 57
6.2.5 Distance Learning 57
6.2.6 User Characteristics Analysis 57
6.2.7 Integrated Service 58
6.3 Summary *59*
7 Reengineering the MARIAN Server 60
7.1 The C/C++ MARIAN Server *60*
7.1.1 Architecture 60
7.1.2 Operations 63
7.2 The Java MARIAN server *64*
7.2.1 Architecture 65
7.2.1.1 "Client_uip" and "Server_uip" 66
7.2.1.2 "Session_manager" 68
7.2.2 Passing Functions Through "Uip" 70
7.2.3 Operations 71
7.3 Flexibility Considerations *74*
7.3.1 Top Level Weak Coupling 75
7.3.2 Weak Coupling Inside "Server_uip" 76
7.3.3 Weak Coupling Inside "Session_manager" 77
7.3.4 Weak Coupling With "Webgate" 79
7.4 Scalability Considerations *80*
7.4.1 Concurrency Control 80
7.4.2 Optimize the Usage of Network Connection 81
7.4.3 Facilitate Computation Distribution 82

8 Applying Good Software Processes 85

8.1 Project Development Lifecycle *85*

8.1.1 Requirements Analysis 87

8.1.2 Original Project Analysis 87

8.1.3 High Level Design 88

8.1.4 Detailed Design 88

8.1.5 Coding 89

8.1.6 Unit Testing 89

8.1.7 Integration Testing 90

8.2 Project Management *90*

8.2.1 Group Assignment 90

8.2.2 Training 91

8.2.3 Measurement & Estimation 92

8.2.4 Process Improvement and Defect Prevention 96

8.2.5 Software Reuse 98

8.3 Results *99*

9 System Performance Experiments 101

9.1 Experimental Design *101*

9.1.1 Measurements 104

9.1.2 Cases 105

9.2 Getting the Correct Data *106*

9.3 Experiment Results *107*

9.3.1 Removal of System Bottlenecks 107

9.3.2. Measuring the Cost of Measurement 112

9.3.3 Final Experiment Results 114

9.4 Summary *120*

9.4.1 Scalability 120

9.4.2 Reliability 121
9.4.3 Flexibility 122

10 Conclusions and Future Directions 123

10.1 Conclusions *123*

10.2 Limitations and Future Directions *123*

Bibliography 126

List of Figures

Figure 3.1: Old system search page *10*
Figure 3.2: Old system results page *11*
Figure 3.3: Old system records detailed description *12*
Figure 3.4: Old system top-level architecture *13*
Figure 4.1: C/C++ "formit" workflow *18*
Figure 4.2: Java "formit" architecture *19*
Figure 4.3: Java "formit" workflow *20*
Figure 5.1: C/C++ "webgate" top-level architecture *24*
Figure 5.2: Java "webgate" top-level architecture *28*
Figure 5.3: "Uip_manager" object architecture *29*
Figure 5.4: "User_manager" object architecture *30*
Figure 5.5: "Request_response" thread architecture and workflow *32*
Figure 5.6: Relation among "results", "request_response", and "call_back_processor" *41*
Figure 5.7: New system beginning page *44*
Figure 5.8: New system main menu page *45*
Figure 5.9: New system query page *46*
Figure 5.10: New system query history page *48*
Figure 5.11: New system super user main menu page *49*
Figure 5.12: New system log query page *50*
Figure 5.13: New system log contents page *51*
Figure 5.14: New system user management page *52*
Figure 6.1: Digital library architecture *54*
Figure 7.1: C/C++ MARIAN server top-level architecture *61*

Figure 7.2: Simplified C/C++ MARIAN top-level architecture *62*
Figure 7.3: Java MARIAN server top-level architecture *65*
Figure 7.4: "Client_uip" architecture *66*
Figure 7.5: "Server_uip" architecture *67*
Figure 7.6: "Session_manager" architecture *69*
Figure 7.7: Passing functions through "uip" *70*
Figure 7.8: Java MARIAN server operations *72*
Figure 7.9: Future "session_manager" architecture *79*
Figure 7.10: Distributed search engines architecture *83*
Figure 8.1: Basic development phases *86*
Figure 9.1: Experiment model *102*
Figure 9.2: Load generator architecture *104*
Figure 9.3: Java server cost of measurement graphs *113*
Figure 9.4: "Webgate" cost of measurement graphs *114*
Figure 9.5: All modules in one machine, performance graphs *115*
Figure 9.6: One "webgate", performance graphs *116*
Figure 9.7: Two "webgates", performance graphs *117*
Figure 9.8 Four "webgates", performance graphs *118*
Figure 9.9: Performance comparison graphs *119*

List of Tables

Table 3.1: C/C++ MARIAN system module sizes *14*
Table 5.1: Java "webgate" request types and descriptions *34*
Table 6.1: Database examples *55*
Table 8.1: Measurement and estimation table *93*
Table 8.2: Bug history table *95*
Table 9.1: Modification list *111*
Table 9.2: System configuration recommendations *120*

Chapter 1

Introduction

1.1 Problem Statement

A good digital library system can serve users with different backgrounds and information needs. Since in many cases users' needs keep changing, a good digital library system should allow their designers and developers to modify the system quickly to add new features and/or change existing features. Thus flexibility is very important for a good digital library system.

A good digital library system may become quite popular, so should be able to provide rapid response to large numbers of users accessing large numbers of digital objects. It should make use of system resources efficiently based on its workload. Thus scalability is also very important for such a system.

Reliability is very important too, since a problem in such a system can harm multiple users. System reliability is especially an issue for projects developed in universities by students. One reason is that students are doing the projects to learn, so they are much less experienced than professional engineers. Another reason is that since most of the projects developed in class are not used in real life, quality usually is not emphasized.

This thesis investigates the feasibility of making digital libraries flexible, scalable, and reliable through the project of Reengineering the MARIAN System in Java. Starting in the summer of

1998, the National Library of Medicine (NLM) provided funding for this project to reengineer the MARIAN (Multiple Access Retrieval of Library Information with Annotations) system from C/C++ to Java to improve the new system and make it platform independent.

MARIAN is a digital library system built around a library catalog search system. It allows users to search library records for a large collection. Currently the system contains over 1 million records from the Virginia Tech Library. The system has evolved since 1990 and is being gradually enhanced. It contains over 100,000 lines of C/C++ code and partly runs under the MACH UNIX operating system. The system provides a Web interface in addition to other interfaces for users to access it.

During one year's reengineering activity, part of the system has been converted to Java. Now the new system is running partially in Java and partially in C/C++.

1.2 Hypothesis

We believe that adding a user information layer and maintaining weak coupling in the design of a digital library system can make it flexible: on the one hand to fulfill user's individual needs, and on the other hand to make the addition and change of features easy for designers and developers.

We also believe a system's scalability can be improved by optimizing the usage of network connections and facilitating the distribution of computation and disk operations to multiple machines.

We believe that tailoring and applying good software processes can help increase the reliability of a digital library system even if it is developed by students in a university.

Approaches based on the above three-part hypothesis were used in the reengineering project. A number of experiments were designed and performed that validate each part of the hypothesis regarding flexibility, scalability, and reliability.

1.3 Organization

The remainder of this thesis is divided into 9 chapters. Chapter 2 reviews related work. Chapter 3 gives a detailed description of the C/C++ MARIAN system in terms of its features and architectural design, prior to the reengineering activity. Chapter 4 discusses how the “formit” module (the closest module to end-users) was reengineered from C/C++ to Java. Chapter 5 describes the reengineering activity of the “webgate” module. Chapter 6 discusses the user information layer of digital library systems. Chapter 7 describes the reengineering activity of the MARIAN server. Chapter 8 talks about applying good software processes in the reengineering activity. Chapter 9 describes the system performance experiments. The last portion, chapter 10, gives conclusions and future directions.

Chapter 2

Related Works

2.1 Digital Libraries and Online Public Access Catalog Systems

Digital libraries basically store materials in electronic format and manipulate large collections of those materials effectively[1].

There are many digital library and Online Public Access Catalog (OPAC) systems. Also there are many digital library and/or information retrieval related papers. But few of them discuss in detail the architectural design of a flexible, scalable, and reliable digital library system. Some common technologies used in digital library systems include indexing, federated searching, and digitization. For more information, see the online courseware at http://ei.cs.vt.edu/~dlib/.

It is generally acknowledged that the first large-scale implementations of online catalogs were at Ohio State University in 1975 and the Dallas Public Library in 1978[2]. Early online catalog systems (before 1986) include the Scorpio system of the Library of Congress, SULIRS system at Syracuse University, NLM Catline, NOTIS system at Northwestern University, Okapi system at Polytechnic of Central London, and others. There are many more OPACs now. A search from the homepage of the library of Virginia Polytechnic Institute and State University leads to scores of them.

Starting from 1994, a four-year, multi-agency Digital Library Initiative funded six projects at: University of Illinois, University of California at Berkeley, Carnegie Mellon University, University of California at Santa Barbara, Stanford University, and University of Michigan[1].

Stanford University proposed the concept of "infobus" which they believe can connect multiple domains, multiple document representations, and multiple collections together[3]. They also considered payment systems. University of Michigan proposed agent-based design since they believe "distributing tasks to numerous specialized fine-grained modules promotes modularity, flexibility and incrementality"[4]. University of California at Berkeley and Carnegie Mellon University worked on image or video searching but didn't focus very much on system architectural design. University of Illinois worked on SGML, vocabulary switching, and interfaces. They identified the need for a "stateful gateway" and "search history kept for each user"[5]. University of California at Santa Barbara focused on geographical information systems and maps[6].

The Networked Computer Science Technical Reference Library (NCSTRL) is also worth mentioning. The underlying system was created largely by Cornell University. NCSTRL involves over 150 universities or research laboratories. It is a distributed system running the Dienst protocol. For a detailed description of the Dienst protocol please see[7].

2.2 Hyper Text Transfer Protocol (HTTP)

HTTP is the protocol used between Web browsers and servers. It is an application layer protocol where the underlying layers provide reliable pipe connections (like TCP). It is also a client server based protocol. The client sends an HTTP request to the server, specifying which file to get. After receiving the request, the server finds the file and sends it to the client in an HTTP response. Version 1.1 of the protocol provides features like persistent connections, cache/proxy support, and so forth. Readers can find a detailed description of HTTP version 1.1 from[8].

Due to the popularity of the World Wide Web, almost all online information systems (including digital library systems) are using HTTP to provide their user interfaces, applying browsers to interact with their users. A big disadvantage of HTTP in those systems is that HTTP is a "stateless" protocol. No relationship can be assumed between different requests even if they are sent from the same client to the same server. Since most online searching systems are context sensitive and allow users to complete a search function over the course of multiple interactions with the system, they have to use some mechanism to maintain state information on top of HTTP between different client requests.

2.3 Common Gateway Interface (CGI)

CGI is widely used in many online searching systems. The MARIAN system also uses CGI to support interaction with users.

When a user clicks on a typical link on a Web page, the browser sends a request to the server specifying that the user wants the corresponding file. The server finds the file and sends it back to the browser. But if the file refers to a CGI program, instead of sending that program to the browser, the server runs the program and sends whatever output the program produces to the browser. Also in the request sent by the browser, there can be information other than the name of the CGI program (for example, information the user entered in a form). This information will be passed to the CGI program as input. Thus the browser, and the user, communicate with the CGI program through the Web server.

A very important feature of CGI is that after the output is sent to the browser the CGI program will "die". It will be run again by the Web server the next time a user clicks its name from the browser. This is one of the reasons that users observe slower speed when communicating with CGI programs than when directly accessing HTML files.

2.4 Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)

TCP is a transport layer protocol built on top of the IP layer. It is widely used in many applications. For example, HTTP is built on top of TCP. Also many digital library systems are using the services of TCP. TCP employs a complicated mechanism to guarantee that it provides a reliable pipe connection even on a network where there may be packet loss and/or disorder. TCP is a connection-oriented protocol: three packets are transmitted between two sides to set up a connection and four packets are transmitted to close a connection. TCP also uses a slow start mechanism[9] to avoid network congestion. For a detailed description of TCP please see[10].

UDP is another transport layer protocol built on top of the IP layer. Unlike TCP, UDP is a connectionless protocol. There is no connection setup and tear down. Also UDP does not handle packet loss or disorder over the network. But due to its simplicity, the speed of UDP is faster than that of TCP. A detailed specification of UDP can be found at[11].

2.5 Java

The Java language was created and made popular by Sun Microsystems. Compared with C/C++, Java is a pure object oriented language. Its biggest advantage is its platform independence. Written once, a program can run on almost any operating system. Also its automatic garbage collection feature makes programming much easier since programmers do not need to free memory or delete objects they created.

Its disadvantages include speed. Since it is an interpreted language, the speed is much slower than C/C++ (20 times according to[12]). Also because of its platform independence feature, it can not dive into very system specific features. This means that in terms of a particular operating system, it is less powerful than C/C++.

We decided to choose Java since we believe its advantages outweigh its disadvantages. We used JDK 1.1[13] where the functions provided are already more than enough for our system. As for the speed, we can depend on faster computers. Also, Sun is trying to improve the speed of the Java language.

2.6 Capability Maturity Model (CMM)

CMM was proposed by the Software Engineering Institute associated with Carnegie Mellon University. The purpose of it is to help software companies produce software products with high quality by tailoring and applying good software processes into project developments.

CMM has five maturity levels. Except for level one, which is the “initial” level, each level has several “key process areas” which they believe can increase the control of software product quality. Software companies proceed from level one to five, step by step, by improving their software processes. Detailed information about CMM can be found in[14].

Chapter 3

The C/C++ MARIAN System

This chapter first describes the history of the C/C++ MARIAN system through the most current version – 1.5. Then it uses screen shots to explain the major features of the system. After that it presents the top-level architecture of the system and gives a description about how the modules communicate with each other to form the whole system.

3.1 MARIAN History

The MARIAN system has been running as an alternate catalog for the main VT campus library catalog for the past six years. In 1992, it had minimal functionality on both back end and client for NeXTSTEP machines and had a small but realistic experimental collection. It was first released to the VT campus on 1993. Since then ongoing usability testing has resulted in several changes in the interface of the system[15]. Incremental improvements in both speed and functionality also have been made to the search system.

The aim of building the MARIAN system was in part to rectify mechanical and conceptual problems of OPACs with techniques like: morphology-based matching, query expansion, authority files, linking and terminological aids, and direct interfaces[16] [17] [18]. Advanced information retrieval technologies like an opportunistic matching algorithm and ad hoc weighting scheme were used in the system[19]. Many people have participated in the development of this system. Those include: Sangita Betrabet, Ben E. Cline, Robert K. France, Scott Guyer, Liya Huang, Steve H. Teske, Chris Kirmse, James Powell, Tim Rhodes, and Eskinder Sahle.

3.2 Major Feature Descriptions

In addition to other methods, the MARIAN system allows end users to search library records through the Web. Figure 3.1 shows its major search interface screen:

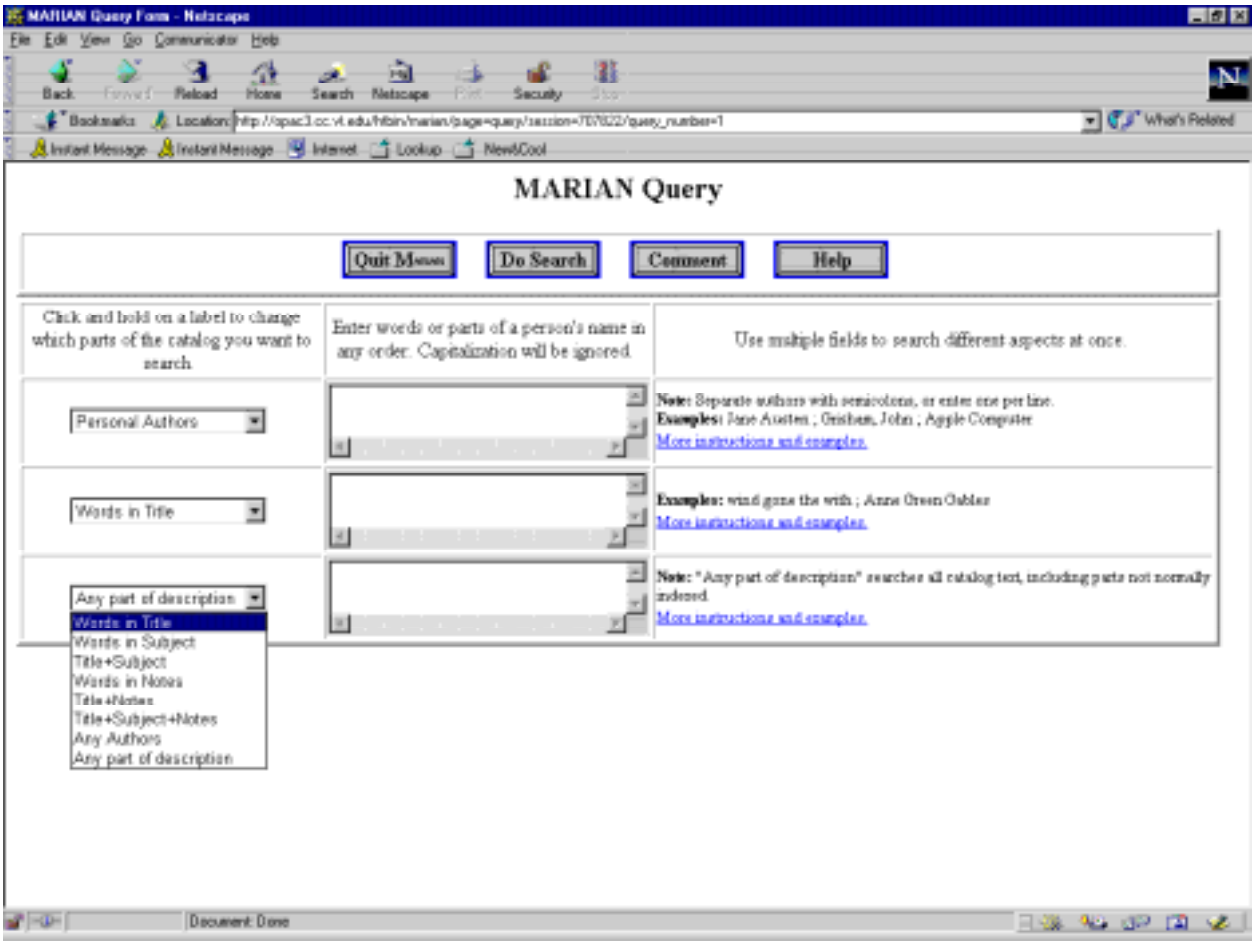

Figure 3.1: Old system search page

The form provides three text fields for the user to enter search terms. For each text field the user is allowed to select a part or combination of parts of the underlying library record (for example Words in Title or Words in Subject). This determines where the system will search for the data. After filling in the form the user can click the button “Do Search”. The system will search its current collection and present the results to the user. Figure 3.2 shows the results page for the

query “cat” entered in the second field, for performing a search on “Words in Title”.

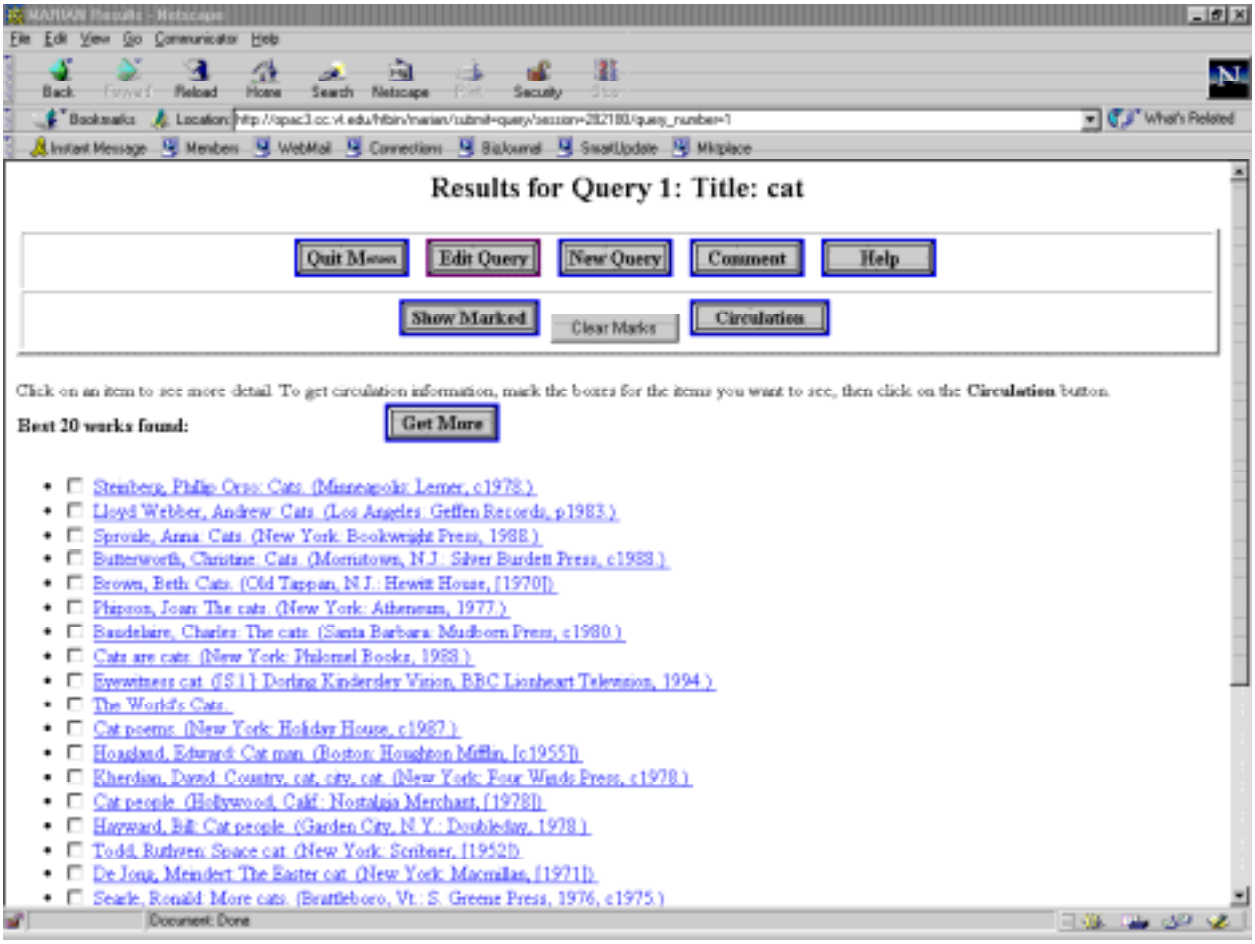

Figure 3.2: Old system results page

The system returned the top 20 records matched with the query. If the user wants to see more records regarding this query, he/she can click the button “Get More” and the system will show another 20 records after the first 20.

Each record is shown on one line. In the database all records are stored in US MARC format[20]. The system extracts a sentence summary for each record from its USMARC format and shows the summary information.

All the records are hyper-linked: if the user clicks on a link, the system will present a detailed

description of the record (also extracted from the US MARC format). The user can see the detailed description for multiple records at the same time by marking their checkboxes and pressing the “Show Marked” button. Figure 3.3 shows the detailed description of records 1, 5, and 6 that were listed in Figure 3.2.

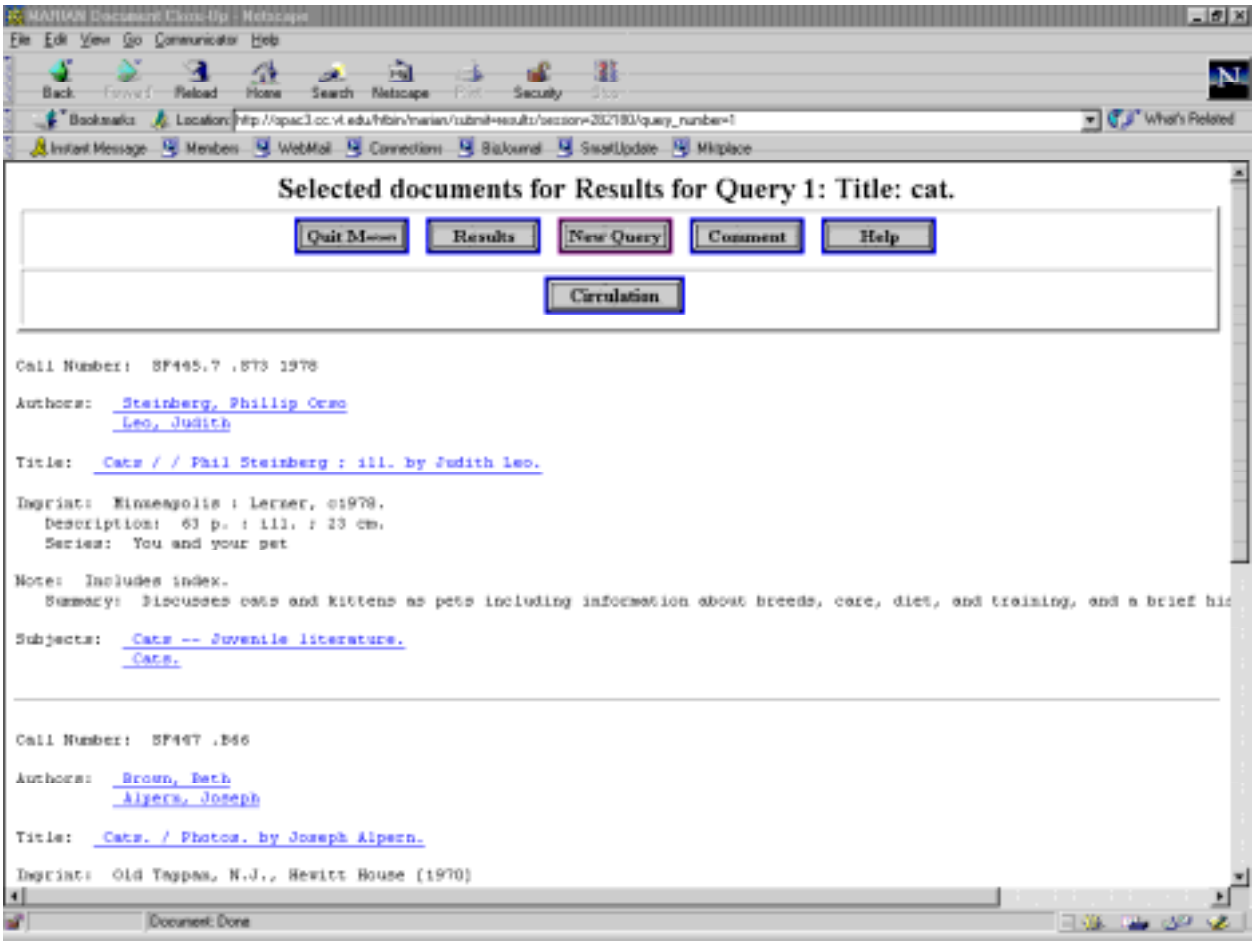

Figure 3.3: Old system records detailed description

The authors, title and subject here are hyper linked. If the user clicks a link, the system will perform another query and provide requested information. For example, if the user clicks on the link “Leo, Judith” under the Author section, the system will perform a search and bring back documents that have “Leo, Judith” as an author.

3.3 Top Level Architecture

Figure 3.4 illustrates the top-level architecture of the system.

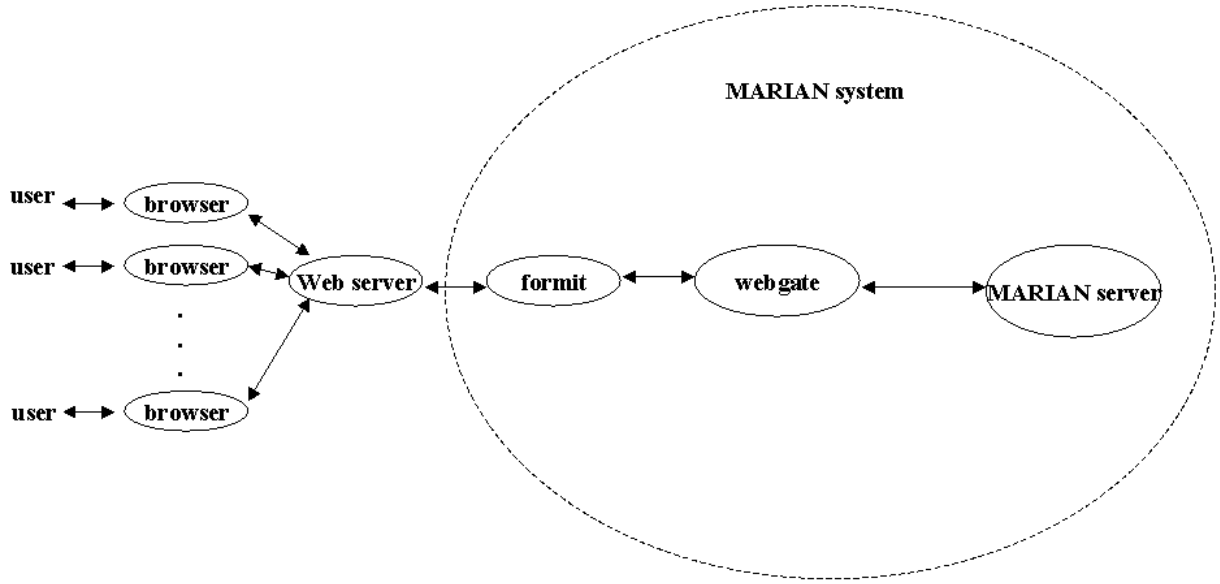

Figure 3.4: Old system top-level architecture

From the top-level view, the MARIAN system is composed of three modules – "formit", "webgate", and the MARIAN server.

"Formit" is a CGI program. Each time a user clicks a link or button from the browser (assuming he/she is accessing the system), "formit" will be executed. It acts like a bi-directional pipe: receives a user's request from the Web server, sends the request to "webgate", receives a response from "webgate", and sends the response back to the Web server.

“Webgate” sits between “formit” and MARIAN server. Unlike “formit”, it keeps on running whether or not there are users accessing the system. It manages a number of sessions based on the current users’ activities. A session can be defined as a user’s activity over a certain period of time. For example, when a user first accesses the system, a session will be created in “webgate” corresponding to this user. After some activities (like searching), the user will stop accessing the system. If “webgate” detects that the user is inactive for a certain period of time, it will release the session corresponding to that user and free all the resources occupied by it. “Webgate” also is responsible for creating a “query” object from the user’s input (provided by the “formit”), sending the “query” to the MARIAN server, receiving search results from the MARIAN server, creating an HTML page based on the results, and sending the HTML page to the end user through “formit”.

The main functions of the system are implemented in the MARIAN server. It contains all the databases of the system and implements the search functions. Like “webgate”, the MARIAN server keeps on running. It also maintains a number of sessions similar to those in “webgate”.

Table 3.1 gives an overview of the sizes of the modules in the C/C++ MARIAN system.

Table 3.1: C/C++ MARIAN system module sizes

Module name	Size (lines of code)
“formit”	Approximately 500
“webgate”	Approximately 9,000
MARIAN server	Approximately 100,000
Other interfaces	Approximately 47,000

3.4 Operations

We describe system operations using scenarios. Let’s start from the search page. Suppose the

user enters something in the text fields and then selects the button “Do Search”.

First, the browser sends an HTTP request to the Web server. This request contains the user input. The request also asks the server to launch the CGI program (module “formit”).

“Formit” is run by the Web server, with all the information contained in the request as input. “Formit” reads in the information, creates a MARIAN request from it, opens a socket connection to “webgate”, writes the information to “webgate”, and waits for the response.

“Webgate” receives the request sent by “formit”. It creates a session corresponding to this user (assuming this is the first time the user accessed the system, otherwise “webgate” will find the session corresponding to this user), forms a query from the information in the request, and sends the query to the MARIAN server along with the session ID.

The MARIAN server receives the query, performs a search on its collection(s), and returns the search results to “webgate”. Since many queries retrieve large numbers of record results, the server only sends back a small number of best matches. It sets a flag to tell whether more results are available.

“Webgate” receives the results from the MARIAN server, creates an HTML response page, inserts the results as well as the session ID and query ID into the page by replacing predefined tags in that page, and then sends the page back to “formit” which has been waiting for its response.

Since “formit” receives the response as an HTML page, it does not need to understand the content of the page. “Formit” just closes the socket connection and writes the response back to the Web server without any change.

The Web server writes the data it receives from “formit” to the browser as if it were a static HTML page.

The browser shows the HTML page to the user. This is the result page we have shown above. Since the session ID has been embedded inside the response HTML page, when the user clicks another button (for example “Get More”), the session ID as well as the information input by the user will be sent back to the system. The system maintains state between different HTTP requests in this way.

In summary, between the time the user clicked the button “Do Search” and he/she saw the results page, the following things happened in order:

Data (request): browser → Web server → “formit” → “webgate” → MARIAN server
Data (response): MARIAN server → “webgate” → “formit” → Web server → browser

If the user presses the button “Get More”, similar transfers will take place.

Chapter 4

Reengineering the “Formit”

“Formit” is much simpler than other modules of the system. The C/C++ “formit” contains only several hundred lines of code. Yet we were careful when reengineering it. The result is a Java “formit” which is more flexible.

4.1 The C/C++ “Formit”

Figure 4.1 illustrates the workflow of the C/C++ “formit”.

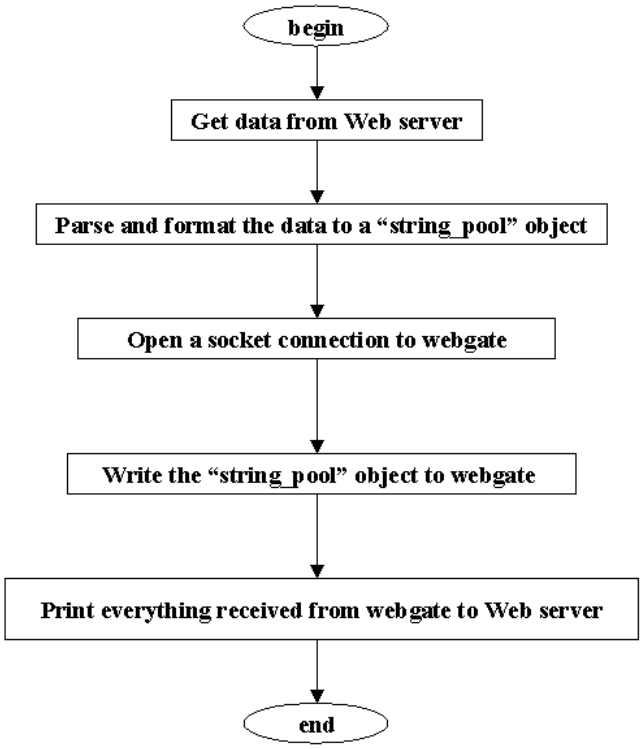

Figure 4.1: C/C++ "formit" workflow

As the name suggests, a "string_pool" object contains a number of strings. Each string may contain multiple lines. When a "string_pool" object is written to a socket, first the number of strings is written as an integer, then the strings are written one by one. For each of the strings the number of lines contained in this string is written first followed by the complete string.

When executed, "formit" first reads data from the Web server, decodes the data (the Web server uses some encoding for certain special characters), and then puts it into a "string_pool" object. Then it opens a socket connection to "webgate" and writes the "string_pool" object to "webgate". After writing the "string_pool" object, "formit" will print everything it received from "webgate" to the Web server since it assumes this is the response sent back from "webgate".

4.2 The Java "Formit"

Figure 4.2 illustrates the design architecture of the Java "formit".

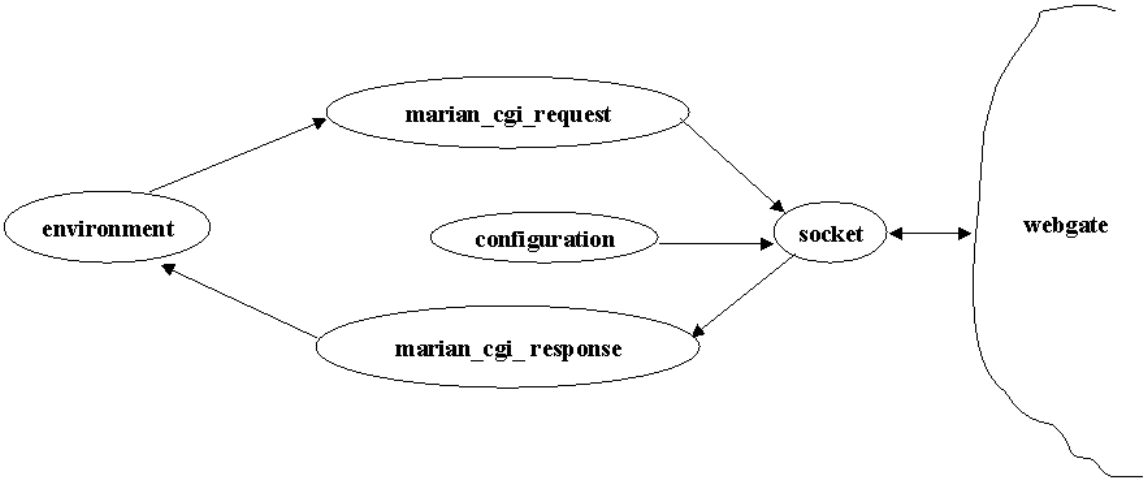

Figure 4.2: Java "formit" architecture

There are four major classes/objects in the Java "formit": "configuration", "marian_cgi_request", "marian_cgi_response", and "environment". (Socket is a Java system class.)

The "configuration" object contains configuration information about the "formit". This currently includes the host name and port number of "webgate".

The "environment" object encapsulates the communication mechanism with the Web server. It provides input to the remainder of the system and accepts output from it.

“Marian_cgi_request” and “marian_cgi_response” are objects that “formit” creates and sends to “webgate” or reads out from “webgate”, respectively. The “marian_cgi_request” object currently contains a “string_pool” object.

Figure 4.3 illustrates the operation of the Java “formit”.

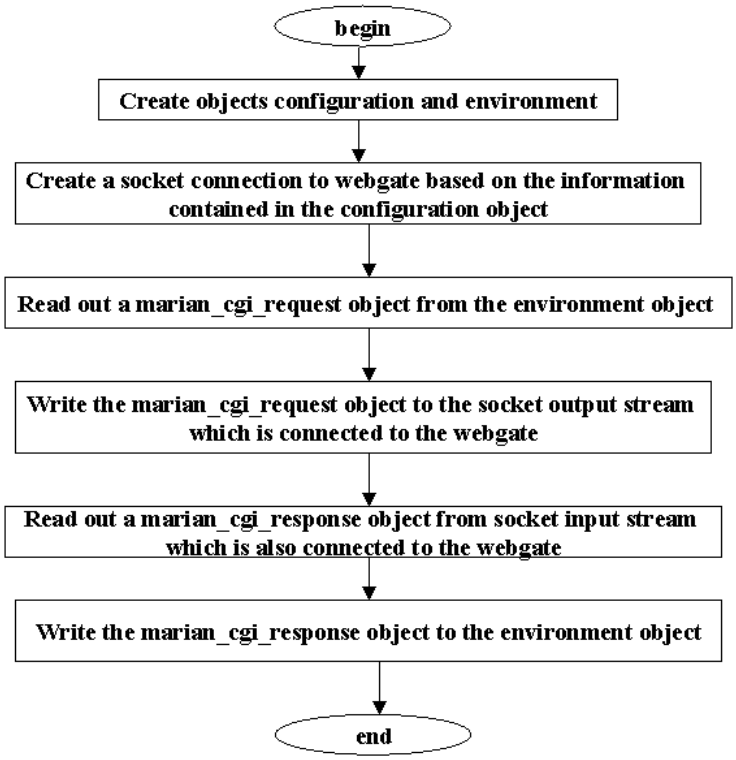

Figure 4.3: Java “formit” workflow

Upon execution, Java “formit” creates the “configuration” object and the “environment” object, then uses the information contained in the “configuration” object to open a socket connection to “webgate”. After that it creates a “marian_cgi_request” object from the “environment” object and writes the request to the output stream of the socket. Then it reads out a “marian_cgi_response”

object from the input stream of the socket and writes the response object to the “environment” object.

4.3 Flexibility Considerations

Currently, there is a “string_pool” object inside the “marian_cgi_request” object. Almost all the functions provided by the “marian_cgi_request” object are implemented through some methods of the “string_pool” object, but the outside world knows nothing about this (as a result of information hiding). This design decision was made to make the coupling between the “marian_cgi_request” class and the rest of the system weak. As a result, if we find a better data structure in the future, we can just implement it inside the “marian_cgi_request” class and no other classes need to be changed as long as we maintain the same service provided by the “marian_cgi_request” class.

The “environment” object encapsulates the communication mechanism with the Web server. Data read from it is in fact coming from the Web server. Data output to it is passed to the Web server. These are hidden from other classes of the system. Again this design decision was made to make the coupling between the “environment” class and rest of the system weak so that if in the future the “environment” is something else (for example a browser on a client’s machine), minimum changes are needed to other classes.

The “marian_cgi_request” object knows how to write itself to a stream and the “marian_cgi_response” object knows how to create itself from a stream. There are no specific requirements for the streams. They can be socket streams or file streams. The flexibility of stream implementation was achieved by maintaining weak coupling between the classes “marian_cgi_request”, “marian_cgi_response”, and the streams. If in the future there exists another protocol which can provide reliable pipe connections like TCP, “formit” can be easily ported to that protocol. For example, in our testing, the request object wrote itself to a file stream in addition to the socket input stream. Thus we obtained all request information easily, helpful for either debugging or logging.

"Webgate" needs to know the file name of the "formit" since it will put this name inside the HTML response page it sends back. In this way, when the end user clicks a button or link on that page the corresponding "formit" will be invoked by the Web server based on the file name). In the C/C++ system, the file name of "formit" is passed to "webgate" as a command line parameter when "webgate" is started. Thus if we change the name of "formit", we have to restart "webgate" with the new "formit" name. This is not very convenient. In the Java system, when creating itself, the "marian_cgi_request" object will ask the "environment" object for the name of this "formit". It then includes the name as part of the request sent to "webgate". "Webgate" gets the name of the "formit" from the request instead of the command line parameters. So there is no need to rerun it when we change the file name of the "formit".

Chapter 5

Reengineering the “Webgate”

“Webgate” is much more complex than “formit”. The C/C++ “webgate” contains about 9,000 lines of code while the Java one contains about 20,000 lines. During the reengineering, we greatly enhanced the flexibility and scalability of the system by making the Java “webgate” evolve to a new layer (user information layer) and maintaining weak coupling in the design. By allowing the Java “webgate” to communicate with multiple MARIAN servers in parallel and thus distribute searching to multiple machines, the scalability of the system is also increased.

5.1 The C/C++ “Webgate”

The C/C++ “webgate” creates and manages a number of sessions based on the user input it gets from “formit”. It is the “webgate” which maintains the state information on top of HTTP (since HTTP is stateless). The state information is embedded inside the HTML responses that “webgate” sends back to “formit”.

5.1.1 Architecture

Figure 5.1 describes the top-level architecture of the C/C++ “webgate”.

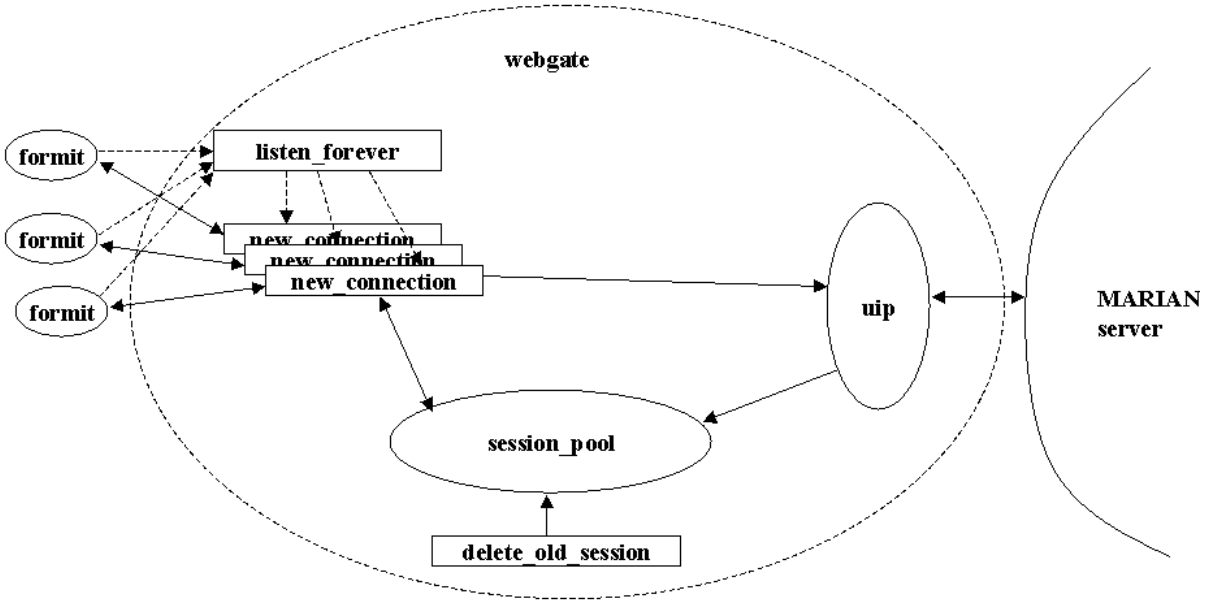

Figure 5.1: C/C++ "webgate" top-level architecture
(We use ellipses to represent objects and rectangles to represent threads in all the design diagrams in this thesis.)

It is composed of five major parts – the "listen_forever" thread, "new_connection" thread(s), a "uip" object, a "session_pool" object, and a "delete_old_session" thread.

The "listen_forever" thread is responsible for accepting "formit" connections. It keeps listening on the well-known port of the "webgate". Whenever a "formit" connection is detected, this thread will generate a "new_connection" thread to handle this connection, and then will listen on the well-known port again.

Each "new_connection" thread is responsible for getting user input from "formit", sending requests to the MARIAN server (through "uip"), getting result documents from "session_pool",

and returning an HTML response to “formit”. A “new_connection” thread is created for each “formit” connection detected. The thread will die after it sends back the HTML response.

The “uip” (user interaction protocol) object is responsible for communication between “webgate” and the MARIAN server. It can pass complicated data structures from one part to the other and vice versa. The complicated data structure here is represented as a *function* in the MARIAN system. Each function has a name and a number of parameters. There are several types of parameters. Some types, like integer and string, are very simple. Others are complicated, like array of array of integers. Readers can just regard functions passed through “uip” as complicated data structures. Functions passed by “uip” may contain query data, documents, and document circulation information.

The “session_pool” object manages a number of “session” objects. Each “session” object represents a user’s activity over a certain period of time. There is a timer inside each “session” object. Each time the corresponding user performs an action involving the system this timer will be updated. So the “session” can tell how long the corresponding user has been inactive based on the current time and the last time the timer was updated.

The “delete_old_session” thread is used to delete old “sessions” when they are inactive for a long period of time. This thread will check the “session_pool” from time to time. If it finds an “old” “session” it will remove it from the “session_pool” and release resources occupied by it.

Each “session” object contains a number of queries performed by the corresponding user within this session. Each “query” object contains query data and query results. Query data is the information submitted by the end user, and query results are the documents returned by the MARIAN server for this query. Each session object also has a state variable. This variable indicates the status of the last query performed in this session. The session behaves as a state machine. Its operation will be explained next.

5.1.2 Operations

Upon starting, “webgate” creates an empty “session_pool” object. Then the thread “delete_old_session” is created for house cleaning. After that a “uip” object is started for communication with the MARIAN server. At last the thread “listen_forever” waits for connection(s) from “formit” by listening on a well-known TCP port (using the “webgate” port number).

When an end user of the system performs an action in the browser, “formit” will be run by the Web server. “Formit” gets the user input from the Web server and opens a socket connection to the well-known port of the “webgate”. The “listen_forever” thread accepts the connection and creates a “new_connection” thread to handle this “formit”. The “new_connection” thread reads the user input from “formit”. If the user wants to submit a query, the thread tries to find a “session” from the “session_pool” corresponding to this user. If there is no such “session” (either this is the first time the user has accessed the system or the “session” has been removed by the thread “delete_old_session”), it will inform the “session_pool” to create one. Then it changes the state of the “session” to “query_sent_out”, indicating this “session” just submitted a query and is waiting for result documents. After that, it sends the query (user input) to the “uip” object. The query is passed to the MARIAN server by the “uip” object in two functions, “search_collection” and “biblio_query_text”. The parameters of the first function indicate on which collection the query should be performed (since one MARIAN server may support multiple collections). The parameters of the second function contain the query data, like the user’s input in the three text fields and the options for the three fields.

After all these things are done, the thread keeps on checking the state of the “session” to see if documents have come back for the query. Some time after the query is sent to the MARIAN server, the server may return a number of documents regarding this query to the “webgate”. The documents are passed to the “uip” object in the function “show_retrieval_coll”. The “uip” object stores the documents in the corresponding “session” object and sets the state of the “session” to “query_done”, indicating that the query results are ready. When the thread “new_connection”

finds that the state of the session has changed, it takes out the documents, creates an HTML page with them, and passes the HTML page to the “formit” it is connecting with.

As explained in chapter 3, the “formit” then passes the HTML page to the Web server which passes it to the browser, and the browser shows the HTML page to the end user.

The “new_connection” thread will die after it serves the “formit” request. Another “new_connection” thread will be created if a new “formit” connection is detected by the “listen_forever” thread. There may be multiple “new_connection” threads running at the same time since there may be multiple “formit” invocations active at the same time.

5.2 The Java “Webgate”

The Java “webgate” creates and manages a number of “user” objects. A “user” object contains more information than a session object in the C/C++ “webgate” and is persistent. The Java “webgate” also can communicate with multiple MARIAN servers at the same time and merge documents coming back from them.

5.2.1 Architecture

Figure 5.2 illustrates the top-level architecture of the Java “webgate”.

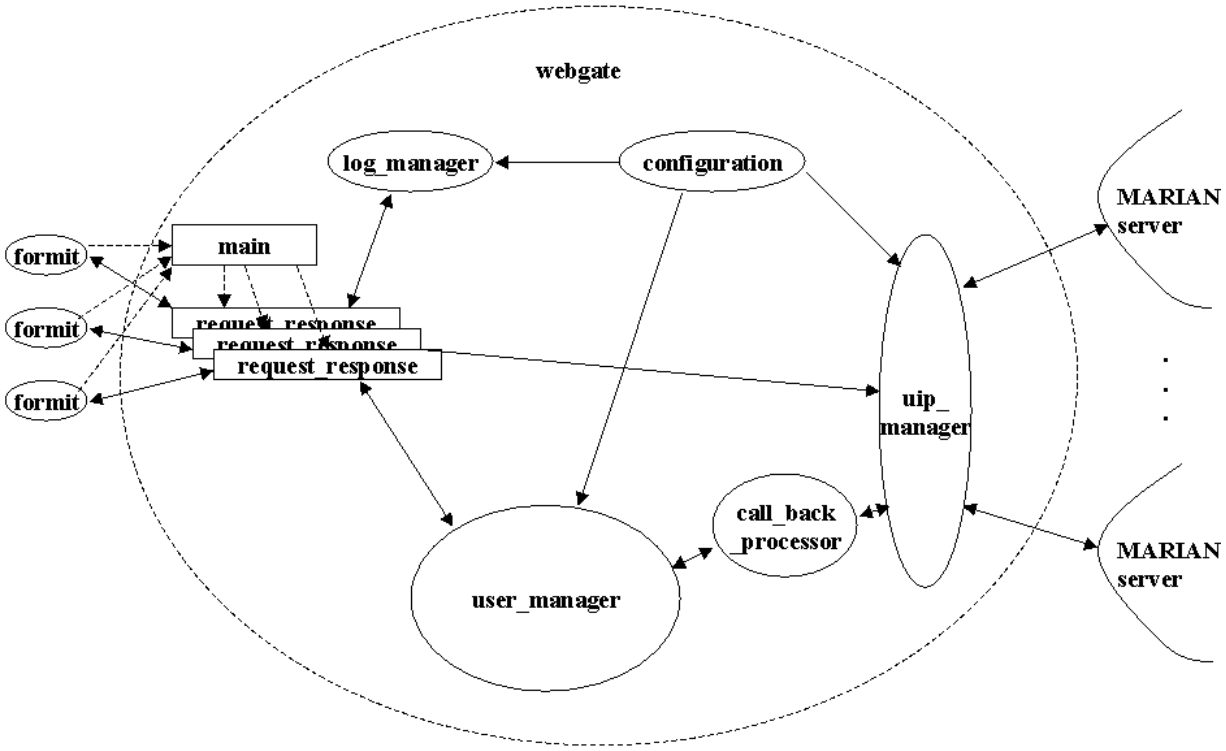

Figure 5.2: Java “webgate” top-level architecture

Viewed from the top-level, it has seven major parts: “main” thread, “request_response” thread(s), “log_manager” object, “configuration” object, “user_manager” object, “call_back_processor” object, and “uip_manager” object.

The “configuration” object contains configuration information for “webgate”. This currently includes the “webgate” well-known port number and the directories from which to create the “user_manager” object, “uip_manager” object, and “log_manager” object.

As its name suggests, the “log_manager” object manages the logs of the system. It is responsible not only for writing system logs to a file but also for retrieving logs from system log files.

The “uip_manager” object manages the communication between “webgate” and multiple MARIAN servers. Figure 5.3 illustrates its architecture.

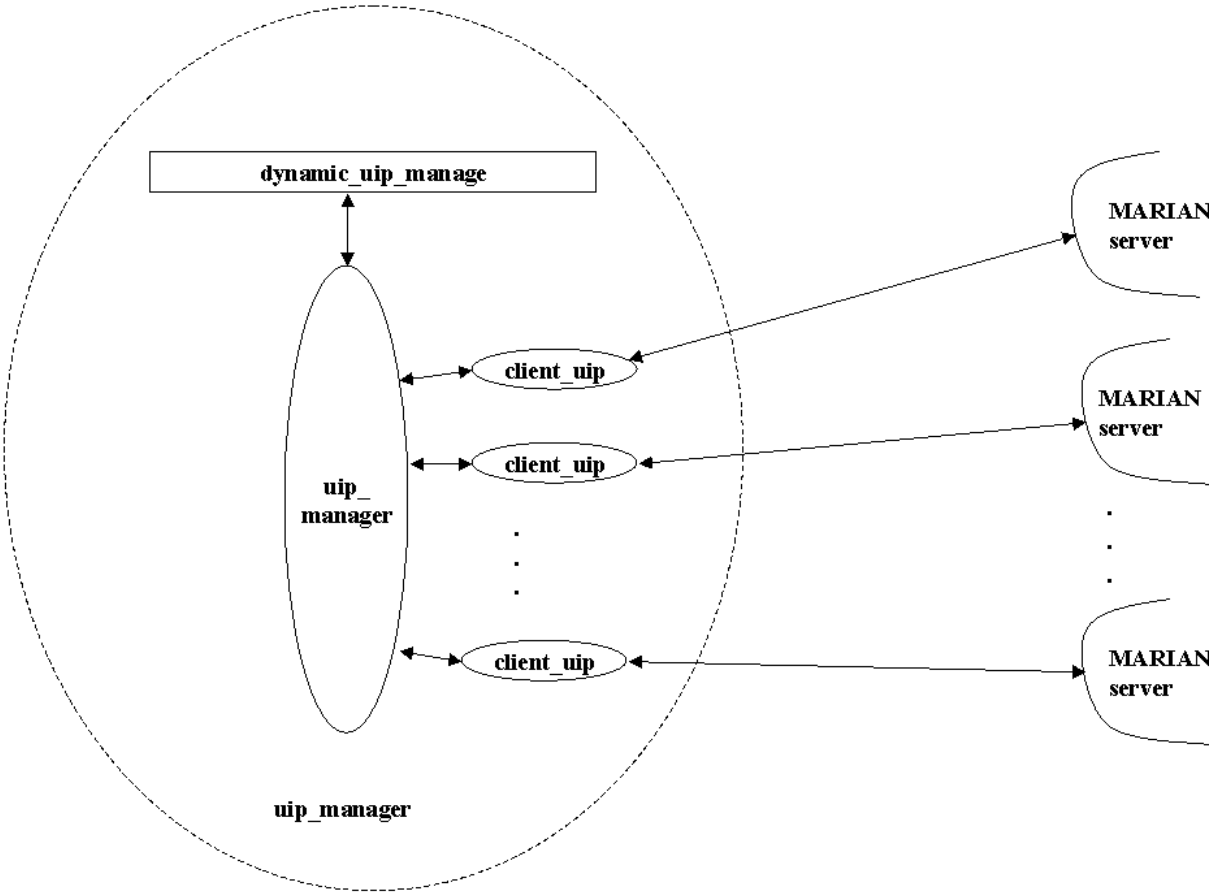

Figure 5.3: “Uip_manager” object architecture

This object creates and manages a number of “client_uip” objects, each of which is responsible for communication with one MARIAN server. All the data (queries or result documents) transmitted between the “webgate” and a MARIAN server go through the “client_uip” object connected to that MARIAN server. The “uip_manager” also creates a thread “dynamic_uip_manage”. From time to time, this thread searches the available MARIAN servers and then informs the “uip_manager” to create new “client_uip” objects to connect to new servers or delete some “client_uip” objects connected to old servers.

The “user_manager” object manages a number of “user” objects. Each “user” object corresponds to an end user of the system. Figure 5.4 illustrates the architecture of the “user_manager” object.

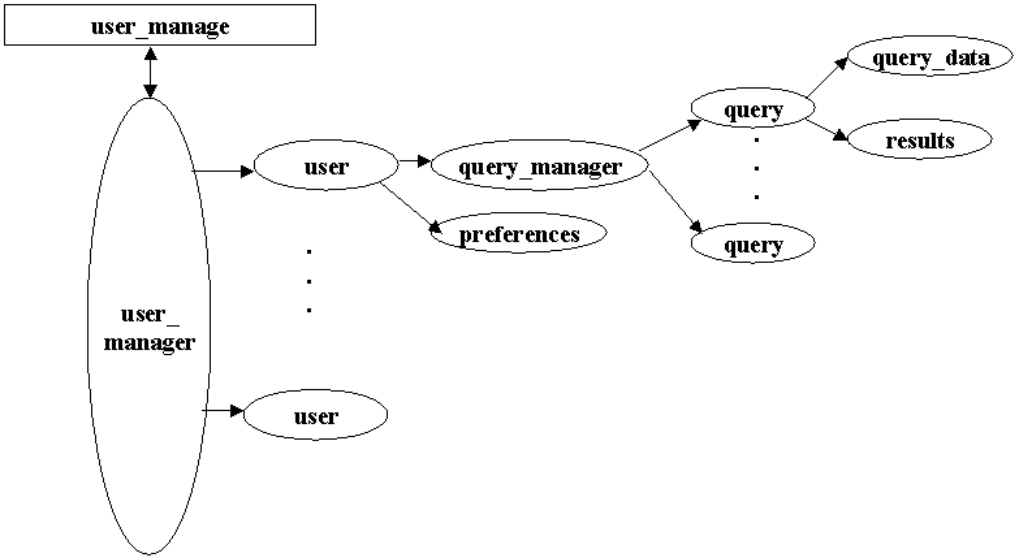

Figure 5.4: “User_manager” object architecture

Upon creation, the “user_manager” object creates a thread “user_manage”. From time to time, this thread asks the “user_manager” object to unload user information from memory to disk for those users who have been inactive for a long period of time. Each “user” object contains a “preferences” object and a “query_manager” object. The “preferences” object contains the preferences of this user such as the default time-out value for performing a query, the batch size, and so on. The time-out value specifies how long the system will block the user when waiting for the results of a query to come back. Batch size specifies how many documents the system should return each time the user submits a query or clicks the “Get More” button. The “query_manager” object contains all the queries performed on the system by this user to date. Each “query” object

contains a “query_data” object which is filled by the user when submitting the query and a “results” object which contains all the documents returned by the MARIAN server regarding this query.

The “call_back_processor” object in Figure 5.2 is used to process functions received from MARIAN servers. For example, if a function comes from a MARIAN server which contains documents corresponding to a certain query performed by a certain user, the “call_back_processor” will take the documents out of the function, find the corresponding “user” object, find the “query” object from the “user” object, and store the documents into the “results” object of the “query”.

The “main” (see Figure 5.2) thread’s responsibility is similar to the “listen_forever” thread in the C/C++ “webgate”. It listens on the well-known port of the “webgate” and generates a “request_response” thread when it detects a “formit” connection.

The “request_response” (see Figure 5.2) thread is similar to the “new_connection” thread in the C/C++ “webgate” except for being more object-oriented. It is created when a “formit” connection is detected. Figure 5.5 illustrates its architecture and workflow.

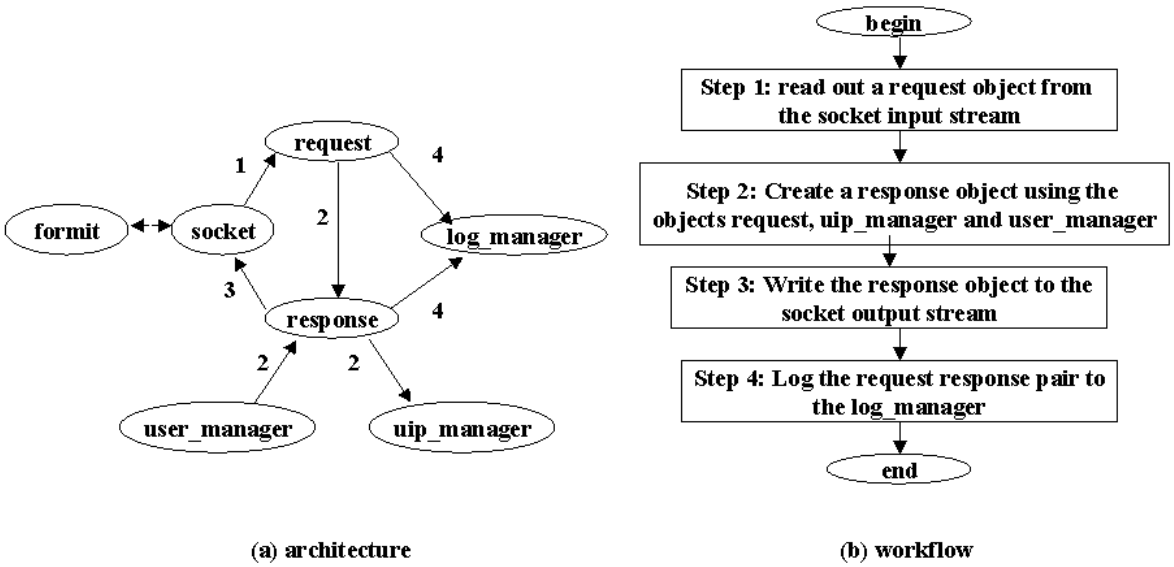

Figure 5.5: “Request_response” thread architecture and workflow

When the “main” thread creates a “request_response” thread, it passes to the “request_response” thread a socket which connects to a “formit”. The “request_response” thread first reads out a “request” object from the socket, then creates a “response” object using the “request” object and possibly the “uip_manager” and “user_manager” objects. After that it writes the “response” object to the socket. Finally it logs the “request”-“response” pair to the “log_manager” object and then dies.

5.2.2 Operations

5.2.2.1 System Startup

At system startup, the “configuration” object is created. This object reads from a configuration file all the configuration information of the “webgate”. Then the object “user_manager” is created from the directory specified by the “configuration” object. The “user_manager” will be empty if this is the first time the “webgate” is run. If the directory already contains user information the “user_manager” object will read it out. Then the “call_back_processor” object and “uip_manager” objects are created for communication with MARIAN servers. A “log_manager” object also is created to manage the logs of the system. At last the “main” thread begins to listen on the well-known port for “formit” connections.

5.2.2.2 Handle “Formit” Requests

When it detects a “formit” connection, the “main” thread will create and start a “request_response” thread. The thread first reads out a “request” from the socket input stream, then creates a “response” for this “request” possibly using “uip_manager” and “user_manager” objects.

Each request has a type. When the end user clicks different buttons on different pages of the system, requests of different types will be created. The C/C++ “webgate” recognizes about 10 request types while the Java “webgate” extended the number to more than 100. All types of requests are handled by the constructor of the “response” object separately. Table 5.1 illustrates some of the request types along with their descriptions in the Java “webgate”.

Table 5.1: Java “webgate” request types and descriptions

Request description	Request type
User clicks button “CANCEL” on change password page.	change_password_cancel
User clicks button “NEW” on main menu page	main_menu_new_query
Super user clicks on button “OK” on log file contents page	log_file_contents_ok
Super user clicks on button “ONLINE HELP” on log file contents page	log_file_contents_help
…	…

Suppose the “request” specifies that the user wants to submit a query. The constructor of the “response” object will first use the user name in the “request” (almost all the requests contain user name since this is a user oriented system) to find the corresponding “user” object from the “user_manager” object. Then it creates a query from the information contained in the “request” and adds the query into the “query_manager” object of the “user” object. After that it sends a “search_collection” function to “uip_manager” to pass to MARIAN server(s). The function will be passed to one or multiple MARIAN servers based on how many servers have been selected by the user in the “request”. After that the constructor of the “response” object keeps on checking the “results” object of the “query” object.

When the MARIAN server(s) receive the “search_collection” function(s), they will respond by asking for queries appropriate to their particular collection(s). A library catalog server, for instance, will respond with a “solicit_biblio_query” function asking the user to fill the form to do the search. The function(s) are received by “uip_manager” and passed to “call_back_processor”. All the functions passing through “uip_manager” contain user name and query number specifying which “query” object of which “user” object the function corresponds to. The “call_back_processor” uses the user name and query number to find the corresponding “query”

object, gets the user input from its “query_data” object, and sends back a “biblio_query_text” function with the user input in it to MARIAN server(s).

Sometime later, the MARIAN server(s) return(s) documents in the form of the function “show_retrieval_coll”. The function gets passed to “call_back_processor” by the “uip_manager” as before. This time the “call_back_processor” takes out the documents from the function, finds the corresponding “query” object from the user name and query number contained in the function, and stores the documents to the “results” object in the “query” object.

Later on, the constructor of the “response” object detects that there are new documents in the “results” object it is checking, extracts the documents, and forms an HTML page from them. Here the constructor of the “response” finishes its task and a “response” object corresponding to this “request” object is created.

The “request_response” thread writes the “response” object to “formit” through the socket output stream and then logs the “request”-“response” pair to the “log_manager” and dies.

Not all the “requests” require the constructor of the “response” object to communicate with “uip_manager”. Some of them (like change user password and preference) only require the communication with the “user_manager” to get and/or change the corresponding user information, and thus are much faster to process.

5.2.2.3 System Shutdown

A super user can shut down the system from any browser. The system first informs the “log_manager” to exit. The “log_manager” will close any log file(s) it opened. Then the system asks the “uip_manager” to exit. The “uip_manager” will disconnect all the connections to MARIAN server(s) and close all the sockets it created. It also will kill the “dynamic_uip_manage” thread it created. Finally the system asks the object “user_manager” to

exit. The “user_manager” object first stops the thread “user_manage” it created and then writes all user information to disk.

Since the system writes back all the user information to disk before its shutdown, the user information is kept between different runs of the system. This means that even if the system is restarted, all the queries performed by a user along with his/her preferences are not lost. Even if the system crashes during execution or the shutdown does not occur properly, since the “user_manage” thread informs the “user_manager” object to unload user information to disk from time to time, the majority of user information is not lost.

5.3 Flexibility Considerations

5.3.1 Adding a New Layer in System Design

By maintaining user information like queries performed and preferences, the Java “webgate” has evolved to a new (user information) layer in our system design. As a result, the new system can provide more flexible services to its users.

Due to this design, the system allows its users to view queries performed by them earlier. A user also is allowed to submit a query to the system based on one of his previous queries. Queries performed by different users may have different values in terms of the time-out value and batch size since each person may have different preferences.

Also due to this design, the super user of the Java “webgate” is allowed to manage individual users of the system. For example, a super user can check when a certain user first logged into the system. A super user also can check how many queries have been performed by a certain user to date. Of course, a super user can delete users along with their queries and preferences.

This design also makes the logs recorded by the "log_manager" object more valuable to system designers. With such a design, the "log_manager" is able to split logs by users, thus enabling user characteristics analysis not possible otherwise.

Since the system maintains persistent user information, it can provide some valuable functions not possible in traditional session based systems. For example, in the Java "webgate", after a user submits a query to the system and before documents come back, he/she can close the browser or even the computer. In this case, the "webgate" will take care of the result documents when they come back. So when the user comes back to the system later (for example, the next day), the system can still present the query along with the result documents to him/her.

Many other flexible services can be added easily to the system due to the user information layer. The next chapter gives a detailed description of the benefits associated with this layer.

5.3.2 Maintaining Weak Coupling

Weak coupling was maintained throughout the design of the Java "webgate". Viewed from the top level, weak coupling was maintained between major objects of the Java "webgate". Weak coupling also was maintained inside major classes as much as possible. All this makes the "webgate" very flexible.

5.3.2.1 Top Level Weak Coupling

Weak coupling was maintained between the object "user_manager" and the rest of the system (see Figure 5.2). Other objects of the system only know that the "user_manager" object creates and manages a number of "user" objects. They can inform the "user_manager" to create a new "user" from a user name and password or they can get a "user" object from the "user_manager" by passing it a user name or ID. They don't know that inside the "user_manager" object, there is the thread "user_manage" which takes care of the cache management. Thus if the mechanism

inside the “user_manager” were changed (for example, databases were used), there would be no need to change other classes.

Similar weak coupling was maintained between the “uip_manager” object and other classes of the system. Other classes don’t know that there is a thread “dynamic_uip_manage” which is responsible for searching current MARIAN servers from time to time. They even don’t know that the “uip_manager” is using multiple “client_uip” objects to communicate with multiple MARIAN servers. They only know that the “uip_manager” can communicate with multiple MARIAN servers, each server is identified by a hostname and port number, the server list may change from time to time, and the “uip_manager” can tell the current server list. These design decisions were made to achieve weak coupling between “uip_manager” and other classes. It allows the flexibility to change the mechanism of the “uip_manager” (for example, not use the service of “client_uip”) without affecting other classes in the system.

The “log_manager” object also has very weak coupling with other classes in the system. The “request_response” thread only knows to inform the “log_manager” to log a “request”-“response” pair. The “response” object only knows that it can get log content from the “log_manager” by passing it a query. How a “request” – “response” pair is written to logs and how the “log_manager” processes a log query are hidden from other classes. Those design decisions make the coupling between “log_manager” and other classes weak. Currently, the “log_manager” only creates one log file in its directory. If we change this mechanism to let the “log_manager” create multiple log files, or use databases to store logs, or even write the logs to another computer through a socket, no changes will be needed for other classes in the system.

5.3.2.2 Weak Coupling Inside “User_manager”

Weak coupling was maintained between the “user_manager” object and the “user_manage” thread (see Figure 5.4). From time to time, the “user_manage” thread asks the “user_manager” object to unload user information for those users who are inactive longer than a certain period of time. The thread does so by calling the method “unload(long time)” of the “user_manager”

object. Thus the thread is responsible for applying a caching algorithm to calculate values like the interval used to inform the “user_manager” object to unload users and how long a user must be inactive before considered inactive. It does not know how the user information is written back to disk. On the other hand, the “user_manager” object knows how to unload user information, but does not know what caching algorithm is being used. These design decisions make the coupling between the two weak. As a result, if we want to change the caching algorithm (for example, from fixed values to dynamically adjustable values), we only need to change the “user_manage” thread. Also if we want to change the way to unload user information to disk (for example, change number of directories and files written), we only need to change the “user_manager” class or classes in it.

Weak coupling was also maintained among objects “user_manager”, “user”, “query_manager”, and “query”. The “user_manager” object knows that a “user” object has a user name, password, and user ID. It knows to unload the information inside a “user” object to disk by calling the latter’s method “unload(…)”. But it doesn’t know what information there is inside a “user” object and how the information is written back to disk. The same thing applies to the “user” and “query_manager” objects. A “user” object calls the method “unload(..)” of its “query_manager” object to unload query information to disk. It doesn’t know how the queries are stored in memory or how they are written back to disk. Again, the “query_manager” object only knows to call the method “unload(..)” of a query object to do so. As a result of this weak coupling, if we want to change the way query information is stored in memory and/or it is written back to disk, we only need to change the class “query” and classes inside it. No other classes need to be changed.

5.3.2.3 Weak Coupling Inside “Uip_manager”

Weak coupling was maintained between the thread “dynamic_uip_manage” and the object “uip_manager” (see Figure 5.3). The thread “dynamic_uip_manage” is responsible for searching available MARIAN servers and for asking the “uip_manager” object to add new server(s) or delete old server(s). The thread doesn’t know how new server(s) are added or old server(s) are deleted in the “uip_manager” object. On the other hand, the object “uip_manager” doesn’t know

what protocol is used to search available MARIAN servers. Thus if we want to change the protocol used to search MARIAN servers, we only need to change the thread "dynamic_uip_manage" (in fact, only the method "search_server(String protocol)" of it). If we want to change the way to add or delete a server, we only need to change the object "uip_manager".

The coupling between the objects "uip_manager" and "client_uip" is weak too. "Uip_manager" knows that each "client_uip" object connects to a MARIAN server identified by hostname and port number. It doesn't know that the communication between a "client_uip" and a MARIAN server is currently using two sockets. It also does not know that the "client_uip" object is using the XDR protocol[21] to pass data through sockets to maintain platform independence. As a result of this weak coupling, if we want to change the mechanism to pass functions (for example, using one socket or not using XDR), we only need to change the "client_uip" class.

5.3.2.4 Weak Coupling Inside "Request_response" Thread

Please refer to Figure 5.5. The thread only knows a "request" object can create itself from a stream; a "response" object can be created by passing it a "request" object and objects "uip_manager" and "user_manager"; and a "response" object can write itself to a stream. The thread doesn't know what is really read out when creating a "request" object from a stream. It also doesn't know what is written to a stream when a "response" object writes itself to the stream. Those design decisions were made to keep the coupling among them weak. As a result, if we want to change the data read out from a stream when a "request" object is created from it, or the data written to a stream when a "response" object writes itself to the stream, the "request_response" thread need not to be changed.

When a "request" object is created from a stream, it only cares about the content read out from the stream. It doesn't care what is used to implement the stream. The same thing happens to the "response" object when it is written to a stream. Thus if we change the implementation of the streams, we don't need to change the two classes.

5.3.2.5 Weak Coupling Among "Results", "Request_response", and "Call_back_processor"

Figure 5.6 illustrates the relationship among a "results" object of a "query" object, a "request_response" thread, and the "call_back_processor" object.

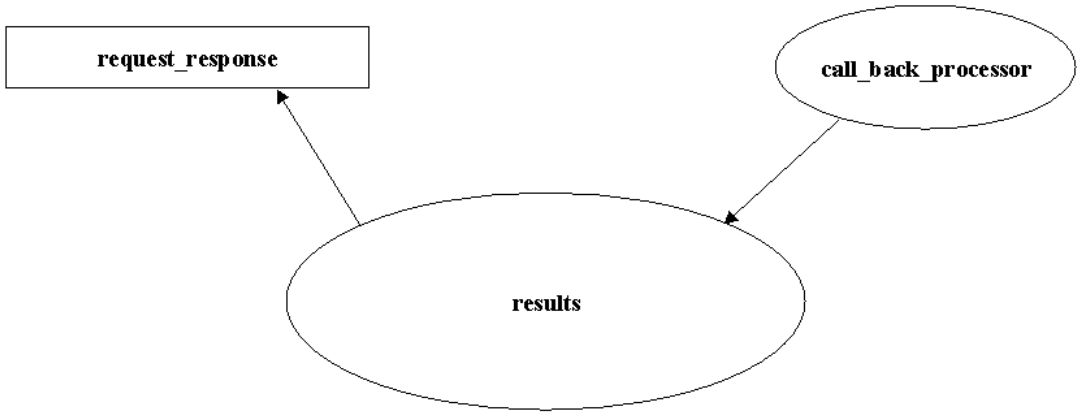

Figure 5.6: Relation among "results", "request_response", and "call_back_processor"

The "call_back_processor" object only knows to store documents returned by the MARIAN server(s) to a "results" object. The "request_response" thread knows to take out documents from a "results" object of a certain "query" object of a certain "user" object. How the "results" object stores documents possibly returned by multiple MARIAN server(s) and how it orders documents are the "results" object's own responsibility. They are hidden to other classes. These design decisions were chosen to maintain weak coupling between them. As a result of this weak

coupling, we can apply various sorting and merging rules inside the “results” object without affecting any other classes in the future. This brings flexibility to the system.

5.4 Scalability Considerations

The Java “webgate” evolved to a new (user information layer) layer in the system. It contains much more information than the C/C++ “webgate”. This includes all the “user” objects and all the queries performed by each user. Thus how to store and manage so much information becomes a problem in terms of system scalability. For example, suppose there are 1,000 users, each user performs 50 queries on the average, and each query results in 20KB documents. Then the information for all the queries will take 1GB. Obviously, storing all this in memory is not a good solution. Thus, how to store and manage so much information becomes a problem in terms of system scalability. The cache management of the “user_manager” was designed to solve this problem. From time to time, the thread “user_manage” informs the “user_manager” to unload user information. As a result, only those users who are currently active have their information in the memory of the system. Even for those users, only the queries they performed recently are in memory; queries they haven’t accessed for a long time are written back to disk. To increase the speed of the system, a change flag is maintained for each user. When the information of a user needs to be unloaded to disk, the system first checks the flag to see whether the information of this user has been changed since it was last read out from the disk. No writing will happen unless there are changes. This mechanism greatly reduces the memory occupied by the Java “webgate” and thus increases its scalability.

The design of the “uip_manager” improves the scalability of the system when the MARIAN server is the bottleneck. In this case, we can partition our document collection across several MARIAN servers and let the Java “webgate” send queries to them at the same time through the “uip_manager”. Though the query rate to each of them is not changed, they will respond faster because each contains databases of much smaller size.

The design of the “uip_manager” also facilitates the improvement of system scalability in terms of the number of MARIAN servers supported. If the MARIAN system becomes very popular in the future, there may be hundreds or even thousands of MARIAN servers all over the world. There may be even more “webgates”. Obviously, letting each “webgate” maintain persistent TCP socket connections to each MARIAN server will waste too many system resources. Ideally, each “webgate” should maintain semi-persistent socket connections to each MARIAN server: when there are data sent from one to the other, the connections exist. The connections will be removed when there are no data transmissions between the two over long periods of time. The design of the “uip_manager” makes this implementation easy. Only the “client_uip” class needs to be changed due to the weak coupling between it and the rest of the system (see section 5.3.2.3). A timer will be needed in the class; the timer will be updated each time a function is passed or received. If there is no transmission over a certain period of time (for example, 5 minutes), the “client_uip” will remove the socket connection with the MARIAN server. When the next function comes, it will reconnect to the server and repeat the above operation. All these changes are inside this class and transparent to the rest of the system. But these changes will greatly increase the scalability of the system by using the network connections more efficiently.

5.5 Java “Webgate” Major Feature Descriptions

Figure 5.7 illustrates the beginning page of the new system.

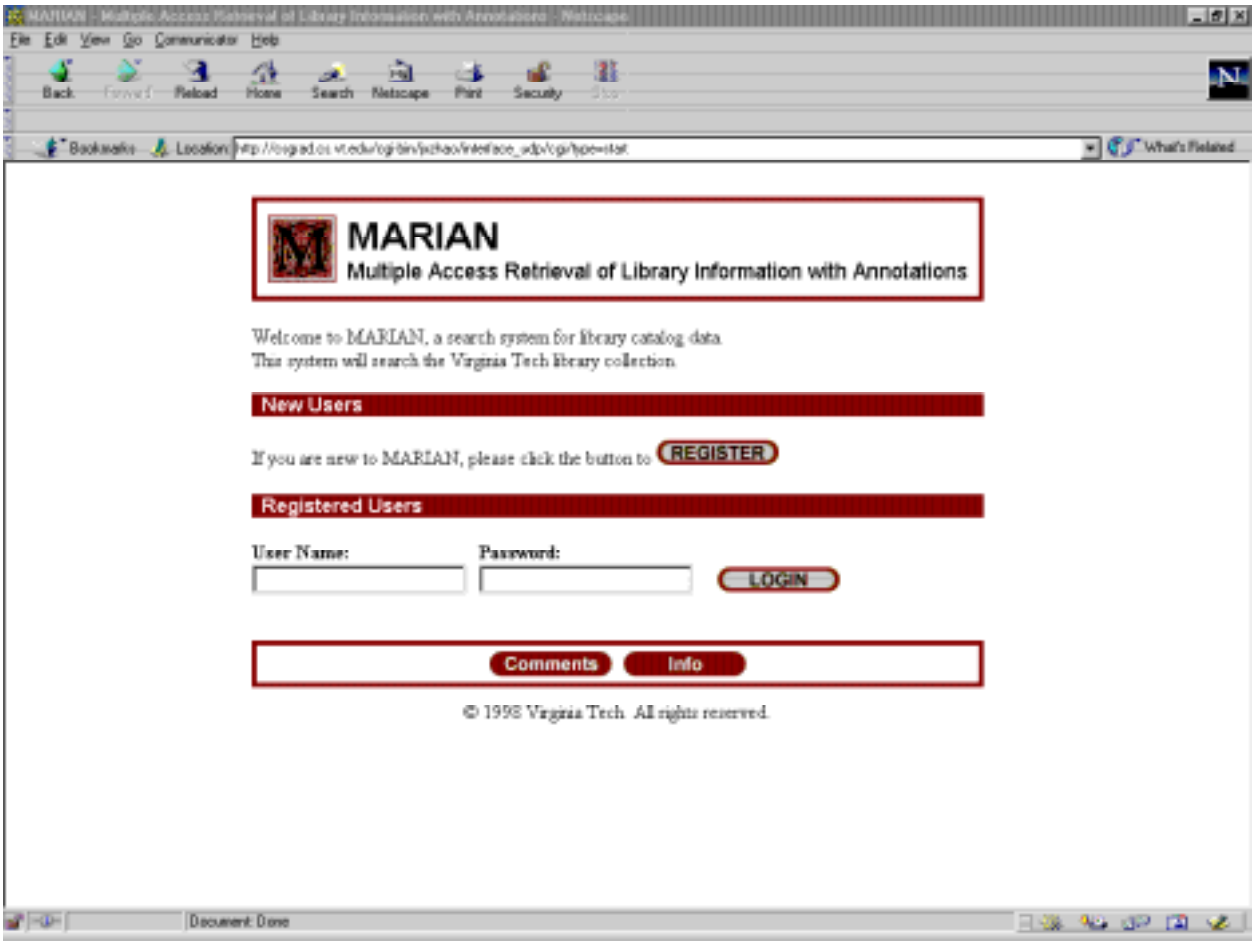

Figure 5.7: New system beginning page

From this page, a new user (a user who first uses this system) can register by clicking the button "REGISTER" and then entering his/her user name and password. A registered user (a user who has already opened an account in the system) can enter the system by typing in his/her user name and password directly. In either case the system will present the following page to the user, as in Figure 5.8.

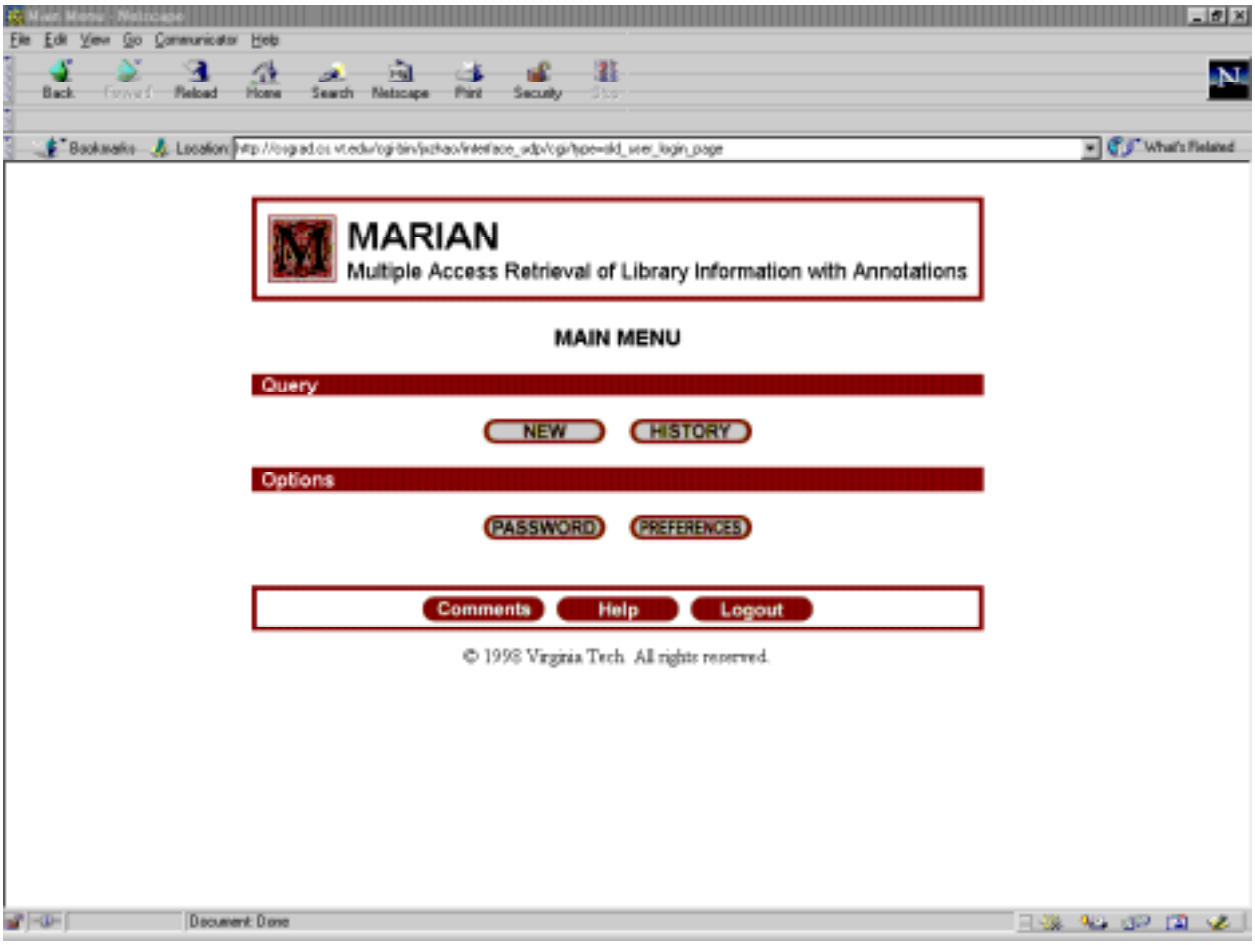

Figure 5.8: New system main menu page

Four major functions are provided from this page. The “PASSWORD” button allows the user to change his/her password. The “PREFERENCES” button allows the user to change his/her preferences, like query time-out value or batch size. If the user clicks on the button “NEW”, the system will bring the user to a page where he/she can enter a new query, as shown in Figure 5.9.

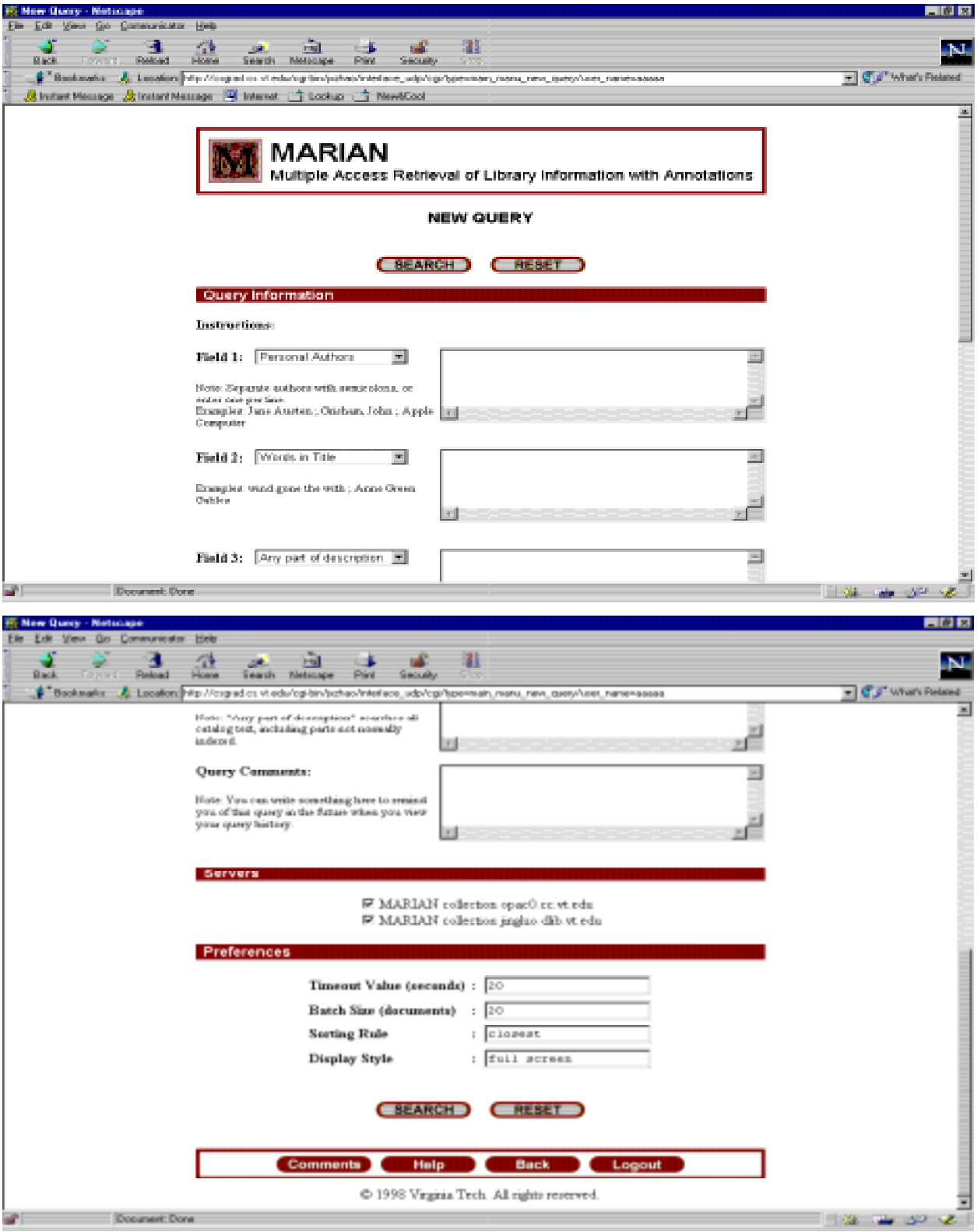

Figure 5.9: New system query page

Since this page is large, we used two screen drawings to show it. The top part of the page is similar to the query page in the old system, providing three fields to enter the query data. But in the new interface another field -- Query Comments -- is added. This field allows entering of a reminder comment about this query, for the user to view it from the query history page. Also there is a section “Servers” which lists all the MARIAN servers currently available. The user can select any or all of the servers to perform the query by marking the checkbox(es) corresponding to the server(s). Finally there is a section “Preferences” where the user can customize values for this query. The default values are this user’s most recent selections of preferences.

If the user clicks the “HISTORY” button from the main menu page, the system will bring the user to the query history page illustrated in Figure 5.10.

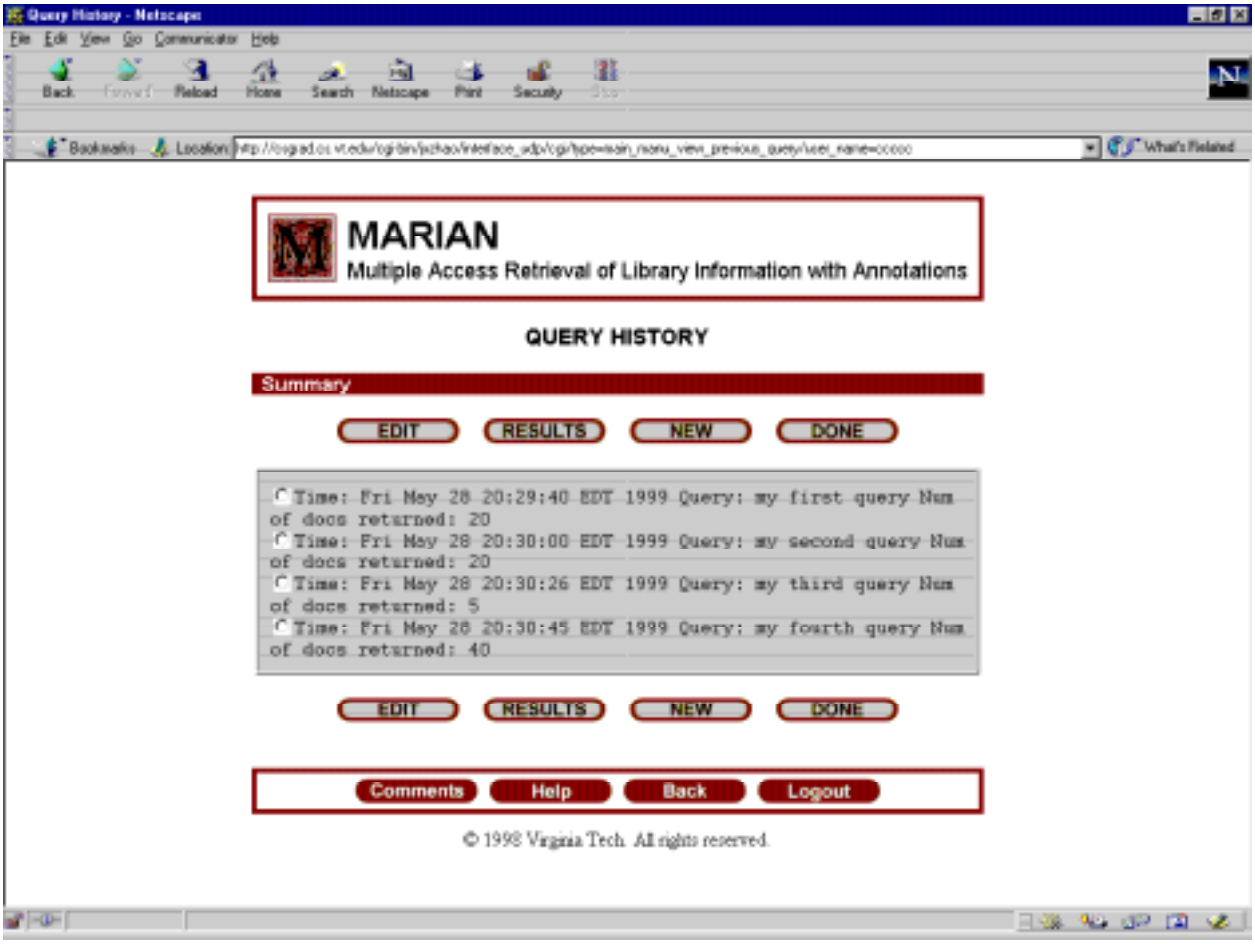

Figure 5.10: New system query history page

On this page, the system lists all the queries performed by this user to date. For each of them the system lists the time the query was performed, the comments the user entered when performing the query and the number of documents returned for the query. The user can mark a radio box near a query and then click the button “EDIT” to see and perhaps modify the query content he/she entered at that time, or he/she can click the button “RESULTS” to see the documents returned for this query previously. When viewing the query content, the user may choose to make some modifications and then invoke the query against the current collection. In this case, the modified query becomes a new query item in the query history. The user also is allowed to

perform a query from his/her query history without any modifications; the same query may result in different result documents since the collection may have changed.

Figure 5.11 illustrates the page the system shows to the super user when he/she logs into the system.

Figure 5.11: New system super user main menu page

From this page, the super user is allowed to change his/her password, shut down the “webgate”, do some system log analysis, as well as manage the users in the system. The following page, shown in Figure 5.12, will appear if the super user clicks on the button “LOGS” in the above page.

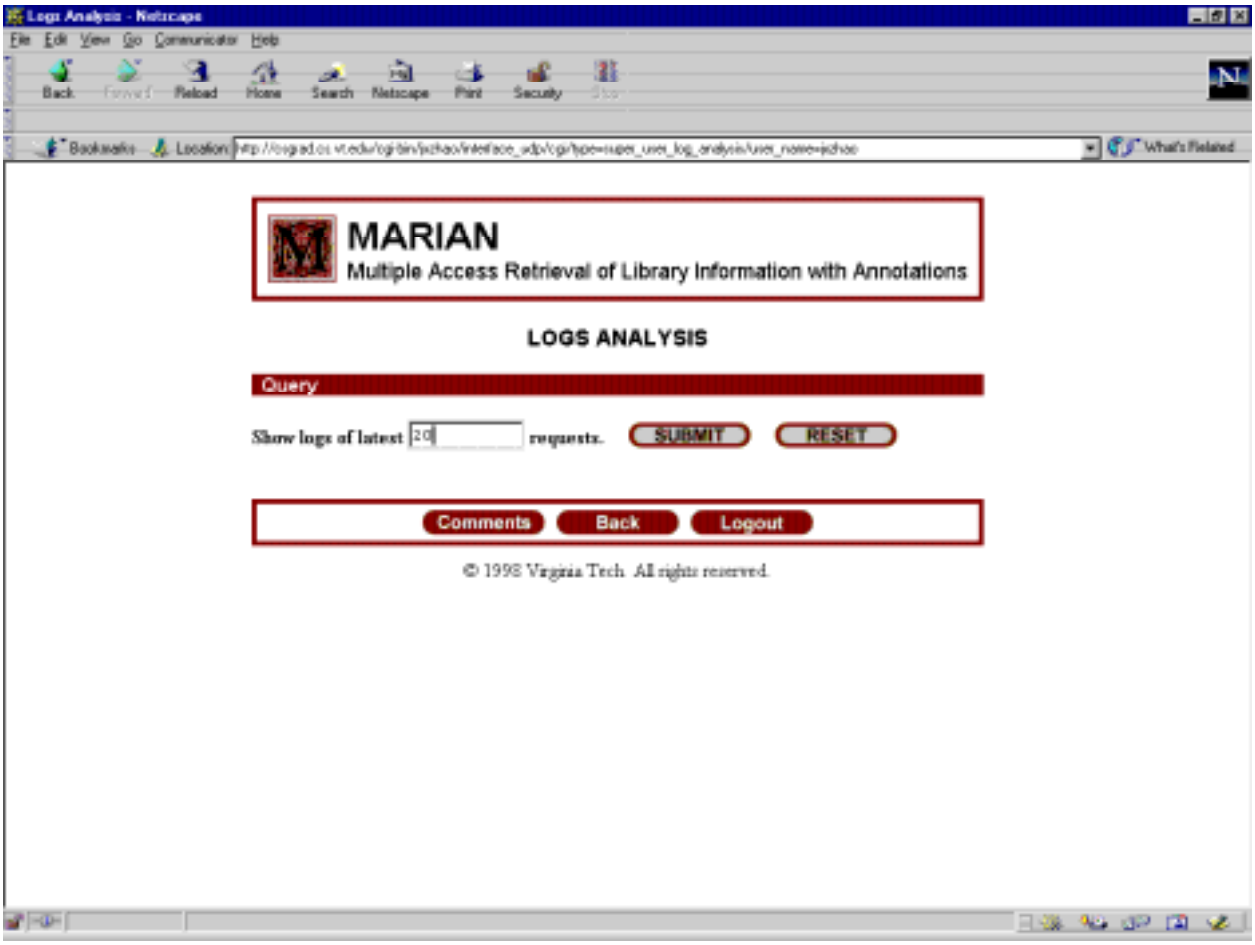

Figure 5.12: New system log query page

This page allows the super user to specify how many log elements he/she wants the system to show. The following page, illustrated in Figure 5.13, will be shown after the number of log elements is entered and the button “SUBMIT” is clicked by the super user.

Figure 5.13: New system log contents page

This page shows the latest 20 log elements of the system. For each log element, the “client” field specifies the file name of the “formit” from which this request came. The “user” field specifies the login name of the user who sent this request. The “request type” field specifies which button from which page was clicked which resulted in this request being sent out. The “response type” field specifies whether or not this request has been served correctly. The time the request is received and the time the response is created also are listed.

If the super user clicks the button “USERS” on the super user main menu page, the following page, as illustrated in Figure 5.14, will appear.

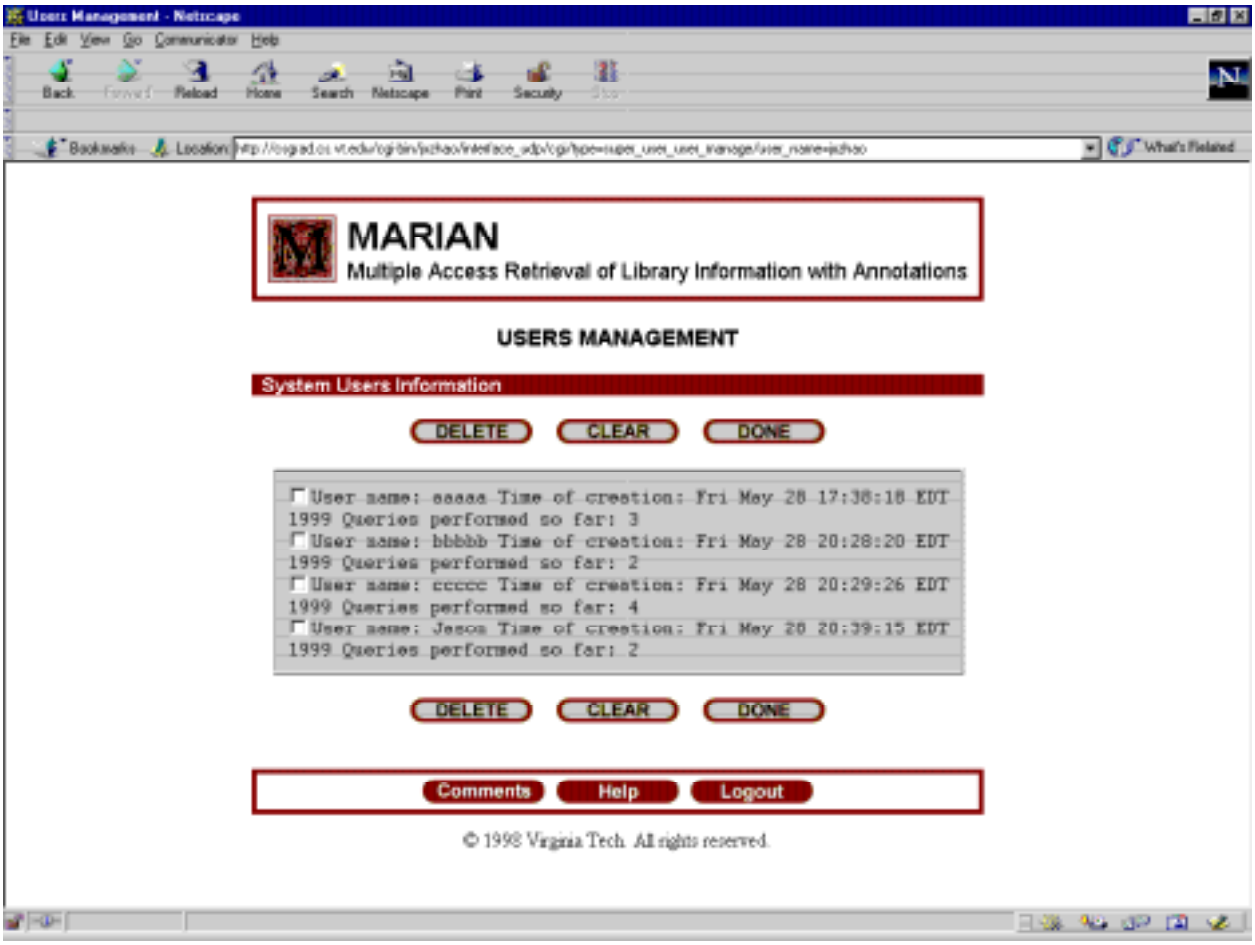

Figure 5.14: New system user management page

This page lists all the users currently registered in the system. Information about the creation time and number of queries performed so far is included for each user. The super user can delete user(s) by marking the corresponding checkboxes and then clicking the “DELETE” button.

Chapter 6

User Information Layer

The Java “webgate” has evolved to a new layer (user information layer) in our approach to increase the flexibility of a digital library system. This chapter discusses the benefits associated with this layer.

6.1 Architecture Description

We propose that viewed from the top-level, a good digital library system should be composed of at least the following layers, illustrated in Figure 6.1.

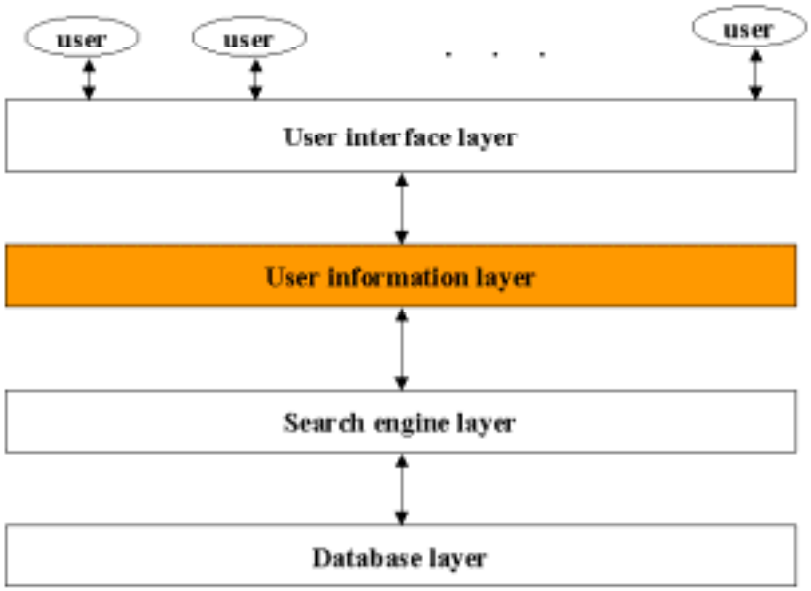

Figure 6.1: Digital library architecture

There are four layers in the model (from user to back-end): user interface layer, user information layer, search engine layer, and database layer.

The user interface layer is responsible for communicating with multiple end users. In terms of the new MARIAN system, the “formit”, Web server, and Web browser belong to this layer. If in the future we use Java applets in the user interface, they would belong to this layer too.

The user information layer sits between the user interface layer and the search engine layer. It is responsible for managing all the user information related to the system. For a large system, this layer may be distributed among multiple machines based on scattered user groups to optimize the performance of system.

The search engine layer sits on top of the database layer. This layer is responsible for converting the queries submitted by the user to database understandable commands (for example, Z39.50[22] or SQL commands), using databases in the database layer to find proper result documents. Most information retrieval happens here. This layer also can be distributed to multiple machines for large systems.

The database layer contains databases like document databases, string databases, link databases, and index databases. This layer is used by the search engine layer to fulfill users' queries. Table 6.1 gives examples for those kinds of databases.

Table 6.1: Database examples

Type	Type of objects stored	Example(s)
document	Full, raw document text or bibliographic records	SGML document, MARC record
string	Unique strings	Author name, English root
link	2-way links between objects	hasAuthor, hasSubject
weighted link	Weighted sets of objects linked to a term	occursInTitle, occursInPersonName

Compared with most digital libraries (for example, NCSTRL at http://www.ncstrl.org/, University of Berkeley digital library at http://elib.cs.berkeley.edu/, and CyberStacks at http://www.public.iastate.edu/~CYBERSTACKS/), this model has a new layer – the user information layer. The next section explains the benefits associated with this layer.

6.2 Benefits of the User Information Layer

Adding this layer into the design of a digital library system can greatly increase the flexibility of the system by allowing it easily to provide many useful services which are impossible (or very difficult) for traditional digital library systems to provide.

6.2.1 Reducing Workload of Search Engine

Many times, a user just wants to find a document he/she found before by using the system. With the addition of the user information layer, such requests can be handled solely through this layer. The search engine layer is not involved. This reduces the burden of the search engine so that it can support more users. Furthermore, the end user will observe better performance since little searching happens (the system only needs to find the corresponding user object and corresponding query).

6.2.2 Personalization

Briefly speaking, personalized systems will customize the interface and features of the system to different users based on their personal preferences. For example in the Java "webgate", after logging into the system, each user can see his/her query history, or perform queries based on his/her own preferences like time-out value and batch size. Many other personalized features like background color or screen arrangement can be added into the system easily within the user information layer.

Since different users have different backgrounds and preferences, making the system customizable lets the system meet the needs of users who would not be served by a fixed one-size-fits-all system. Even users who are comfortable with a non-personalized system are likely to welcome customization.

6.2.3 Active System

Most traditional digital library systems are "passive": they just sit there waiting for their users to do a search or some other operations. With the user information layer, we can make our system "active": it can inform its users when things of interests to them are entered into the system. For example, in our "webgate", we can let the "user" object periodically query the system based on

the user's personal interest, and then report the new documents found to the user through the user's email address. Such "filtering" systems bring more convenience to their users[23].

6.2.4 Billing Capability

Whether or not a digital library system should charge fees to its users is still arguable. But being able to add this ability easily is definitely desirable. With the introducing of the user information layer, each user has his/her own account, and adding billing features is not difficult. In the design of the "user" object in the Java "webgate", there is a object called "money_manager". Though this object does not provide many features at this time, it can be extended and used to implement a billing feature in the future.

Along with the billing feature, electric commerce is possible. With its popularity in recent days, adding this feature into the system may help attract additional types of databases and users.

6.2.5 Distance Learning

One purpose of a library system is to help users learn things. With the user information layer, where each user has his/her own account, it is possible for us to save information like a user's course record in the user object. Thus the user can take classes online. We could even support the offering of online degrees in the future.

6.2.6 User Characteristics Analysis

With the introducing of the user information layer, we are able to split system logs based on users. This makes log analysis much more efficient than in session-based systems. For example, the log analysis of a session-based system cannot answer questions like "what is the ratio of active users vs. total users of the system" since one user may create multiple sessions at the same time or different times and there is no way for the system to know which sessions are created by the same user or different users. Questions like "what percent of the students in the Computer

Science Department are interested in information retrieval related topics" are even more difficult to answer. But we can answer those questions through the user information layer since we can extract each user's background information.

Also with the addition of this layer, we can analyze the history of a user. Then we should be able to set his/her preference based on his/her history and background information. For example, imagine that there are two users. One is an animal expert, and most of the queries he submits are related to analyzing animal behavior. The other is a building design architect, where the majority of the queries he performs are related to how to build a good house. If we get a query which contains the word "home", for the animal expert, we'll emphasize documents related to the home of animals, while for the architect, probably documents related to how to build a comfortable house are more germane.

The user information layer also can facilitate remote user evaluation[24]. The reason is that in such a system, when the user sends some comments, we know not only the comments, but also both the activity of the user immediately before he/she sent the comments and the background information of this user. There is less need for us to send someone to the remote site to observe the user's behavior when using the system.

All this makes analyzing the user characteristics more efficient, and in return the results can be used to improve system performance: making it faster and more efficient.

6.2.7 Integrated Service

If we modify our user information layer a little bit so that it not only talks to our search engine layer but is also able to talk to other systems (through some common protocols/standards), we'll be able to provide more and more services in our system.

For example, we can add online shopping, newspaper reading, ticket booking, renting of automobiles, and many other services into the system in this way. In the end the system could evolve into an online home for our users.

6.3 Summary

Based on all the above reasons, we strongly recommend that digital library designers consider our model, especially the addition of a user information layer, when designing their systems. Many commercial systems are providing personalized features to attract their customers. These include companies like Microsoft, Yahoo, Excite, Netscape, and America Online[25].

Chapter 7

Reengineering the MARIAN Server

The C/C++ MARIAN server contains about a dozen independent modules (each module has its own main() method and can be run without other modules). In the last year, part of it (several modules) has been converted into Java. The current working MARIAN server is implemented partially in Java and partially in C/C++. During the reengineering, the flexibility and scalability of the system have been improved by maintaining weak coupling throughout the design (section 7.3 gives detailed description), optimizing network connection usage, and facilitating computation distribution in design.

7.1 The C/C++ MARIAN Server

In the original C/C++ MARIAN server, communication between different modules is implemented using operating system specific remote procedure calls. The function of the server is to accept queries from its client(s) ("webgate"(s)), process each query, and send result documents back to clients.

7.1.1 Architecture

Figure 7.1 illustrates the top-level architecture of the original C/C++ MARIAN server.

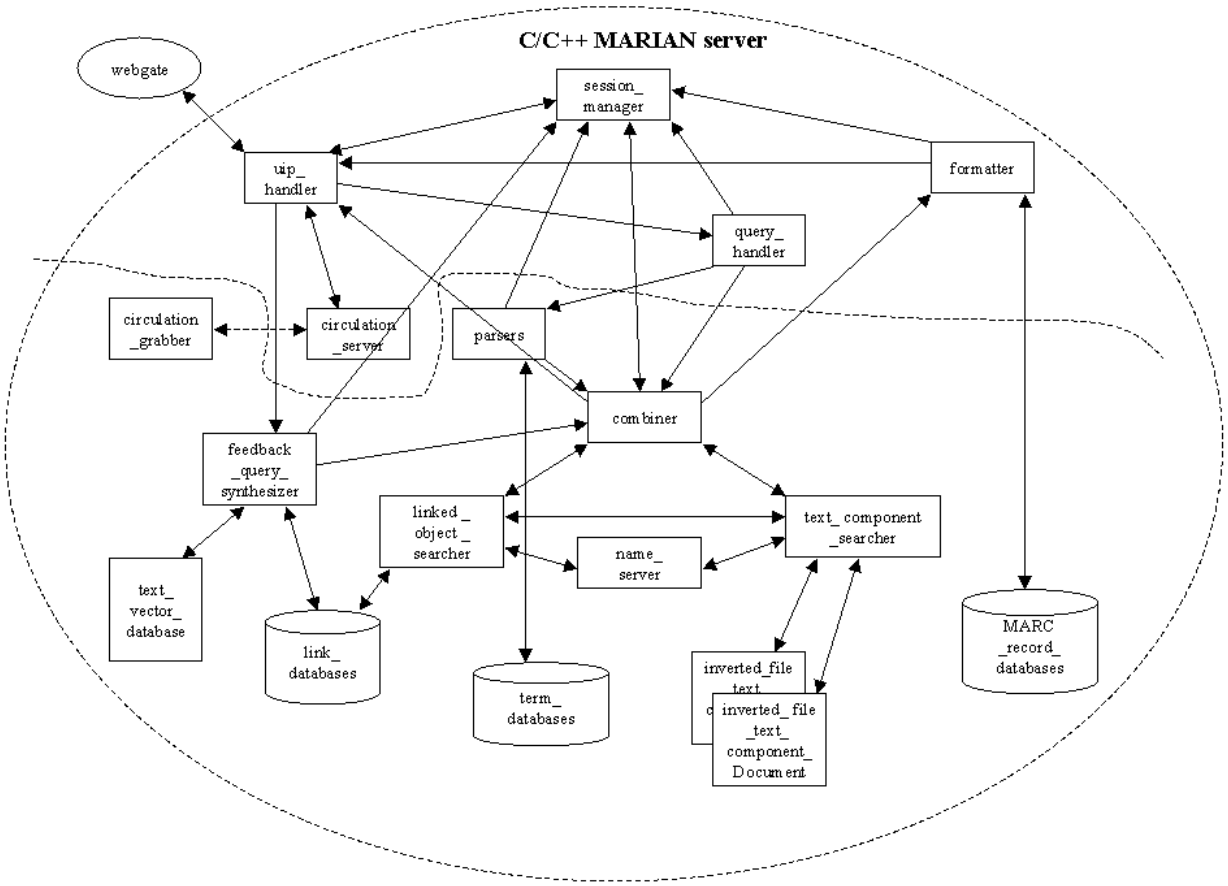

Figure 7.1: C/C++ MARIAN server top-level architecture

This design diagram looks very complicated. Due to time constraints, only modules above the dotted line have been converted to Java. These include modules “session_manager”, “uip_handler”, “circulation_server”, “query_handler”, and “formatter”. To make it easy to understand, we simplify the design diagram to focus on these modules. Figure 7.2 illustrates the simplified design diagram.

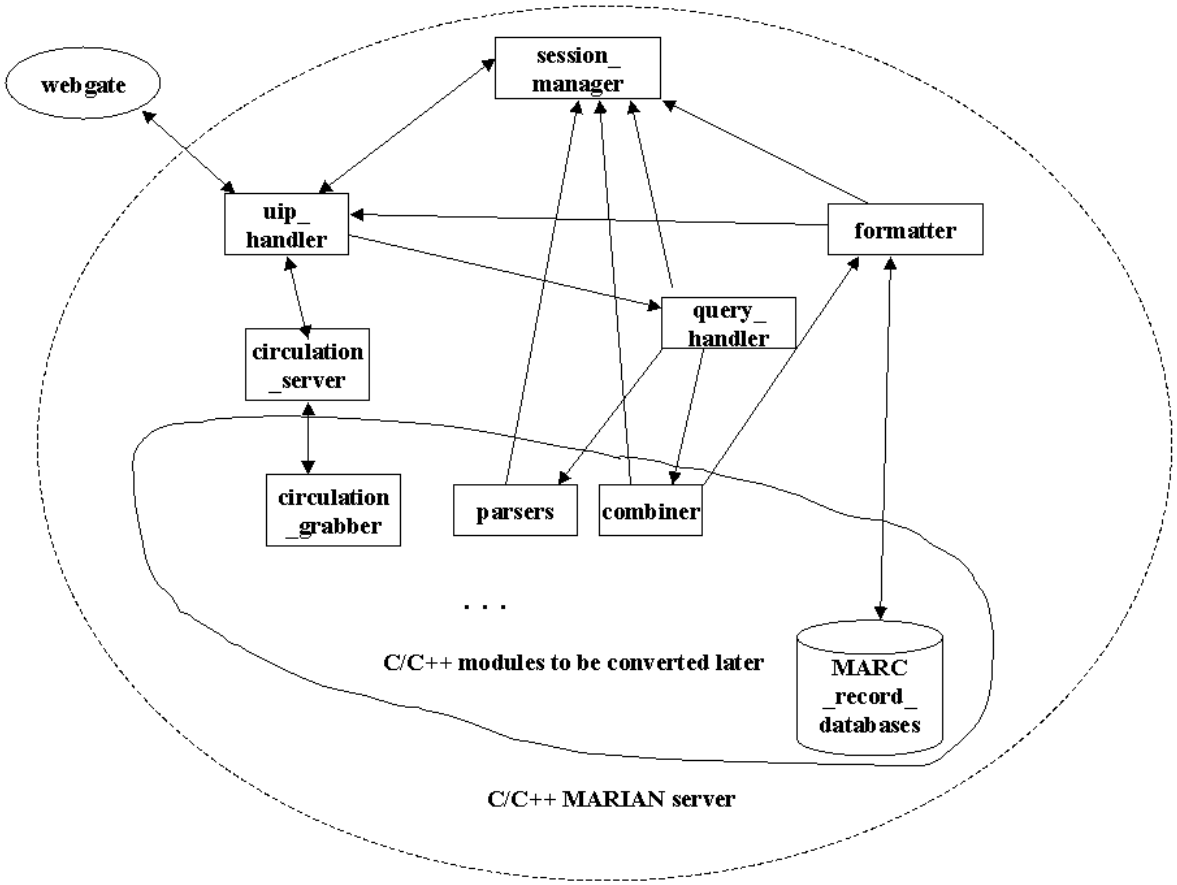

Figure 7.2: Simplified C/C++ MARIAN top-level architecture

The “uip_handler” module is responsible for communicating with “webgate”(s), in particular, passing functions to and receiving functions from “webgate”(s). Functions may contain user queries or result documents. This module is also responsible for informing the “session manager” when a new MARIAN session is created. Detailed design of this module can be found in[26].

The “session_manager” module manages all the sessions of the system. Almost all the processing modules (like “uip_handler”, “query_handler”, “circulation_server”, “formatter”, and others) will report to the “session_manager” during their processing of a query. The “session_manager” also communicates with the “uip_handler” to oversee the user logon process.

The “circulation_server” is responsible for getting circulation information for library documents selected by users.

The “query_handler” module handles all the queries of the system. It uses a tree structure to manage all the queries for each session. It also is responsible for some preprocessing of the queries it receives – parsing them into individual text strings and counting the total number of strings in a query.

The module “formatter” is responsible for getting documents from the MARC record databases using the IDs of the documents. Since the documents are stored in the database in US MARC format, this module is also responsible for formatting the US MARC record into human readable strings. Each formatted document contains two strings. The short description is usually only one line and contains the name, author, and creation time of the document. The long description has several lines and contains detailed information about the document like all the authors, title, imprint, notes, and subject. Documents in other formats may be supported by the system by adding the corresponding formatting function to the “formatter” module.

7.1.2 Operations

When a user from a “webgate” submits a query to a MARIAN server, the query is first accepted by the “uip_handler” as a function (see section 5.1.1 page 25 for explanations of functions). The “uip_handler” then informs the “session_manager” that a new query is received. It also will inform the “session_manager” that a new session is created if it has not seen the session before. Then the “uip_handler” passes the query to the “query_handler” module. The “query_handler” maintains a tree structure for all the queries within a session and adds this query into the proper position in the query tree structure. Then it does some processing of the query, parses the contents of it to a number of searchable strings, and sends the strings to the “parsers” module. It also counts the total number of strings and sends this number to the “combiner” module.

Things that happen in the “parsers” and “combiner” modules are complex, but in general they do some searching resulting in a set of document IDs each of which represent a document identified by this query.

The IDs are passed to the module “formatter” which uses them to retrieve the actual documents from the databases. Since the documents are stored in the database in US MARC format, the “formatter” formats them to short and long descriptions and sends two such strings for each document to the “uip_handler”.

The “uip_handler” passes the documents it receives from the “formatter” to the “webgate” in the function “show_retrieval_coll”.

When the “combiner” module fetches the first batch of document IDs for a query it partially constructs the rest of the set. As a result, when “uip_handler” receives a function which asks for the next batch of documents for the query, much of the work has been done and the system responds much faster even though the information goes through the same path.

Sometimes, a “webgate” may send the “uip_handler” a number of document IDs and specify that it wants the circulation information for those documents. In this case, the “uip_handler” passes the document IDs to the “circulation_server” module. The “circulation_server” here opens a socket connection to the “circulation_grabber”, writes the document IDs to the grabber in a special format (the format is provided by Virginia Tech Library System Inc. since we are getting circulation information from their VTLS system), then reads the response from the grabber. The response is also in a special format, so the “circulation_server” formats it to human readable format and then sends the formatted string to the “uip_handler” inside the function “append_to_text”. The “uip_handler” passes the function through “webgate” to the user who asked for the circulation information.

7.2 The Java MARIAN server

The Java MARIAN server has over 20,000 lines of code and performs the functions of modules "session_manager", "circulation_server", "query_handler", "uip_handler", and "formatter".

7.2.1 Architecture

Figure 7.3 illustrates the top-level architecture of the Java MARIAN server.

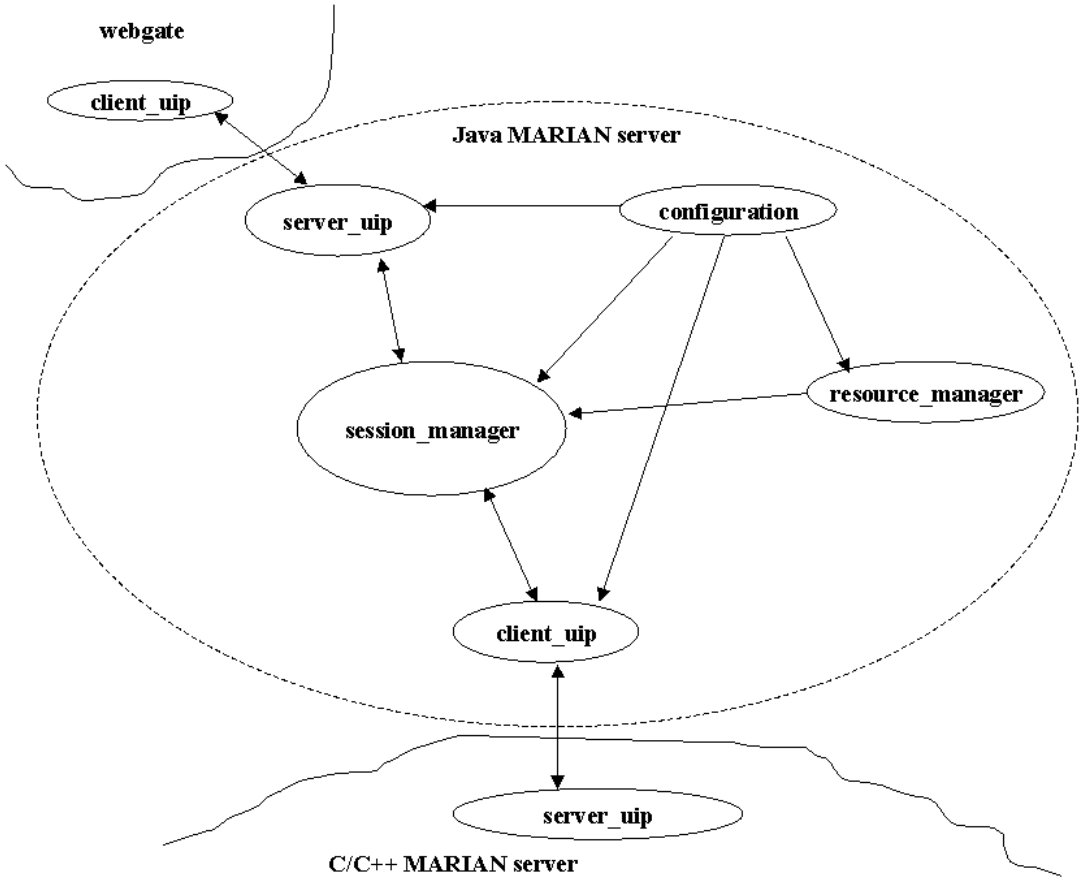

Figure 7.3: Java MARIAN server top-level architecture

There are five major modules – "client_uip", "server_uip", "configuration", "resource_manager", and "session_manager".

The “configuration” object contains the names of the directories used to create the “session_manager”, “resource_manager”, “server_uip”, and “client_uip” objects.

The “resource_manager” object currently contains the “log_manager” of the system.

7.2.1.1 “Client_uip” and “Server_uip”

Both “client_uip” and “server_uip” are used for the communication between the Java MARIAN server and other subsystems (in this case the C/C++ MARIAN server and “webgate”). They are responsible for sending functions to and receiving functions from the other side. Figure 7.4 illustrates the architecture of the “client_uip”.

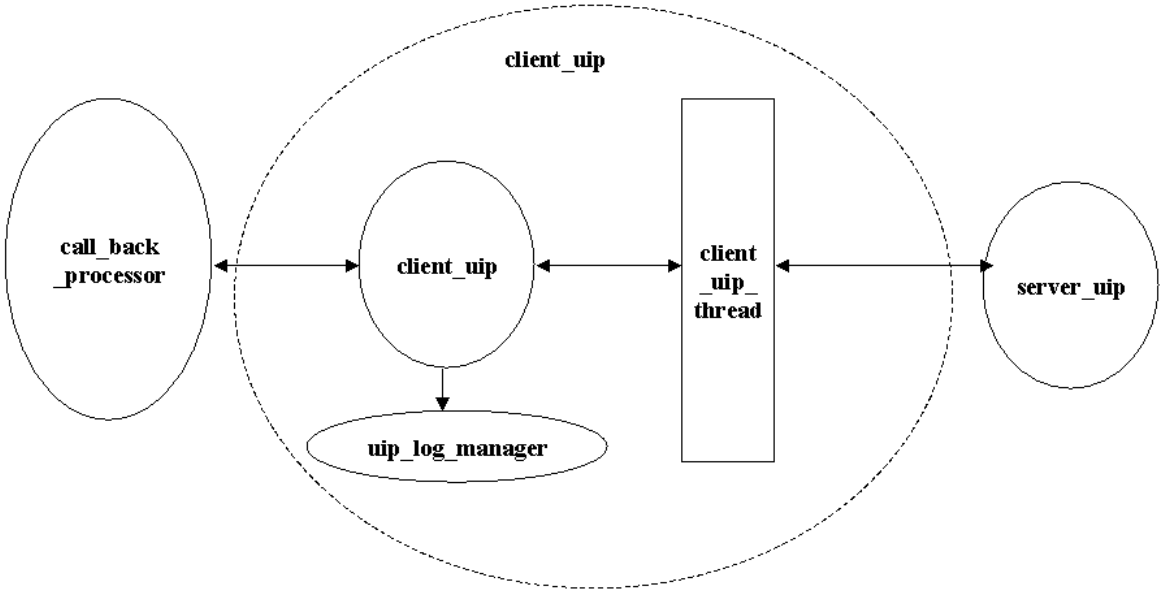

Figure 7.4: “Client_uip” architecture

"Call_back_processor" can be any system using the service of "client_uip". In our case, it is the "webgate" or the Java MARIAN server. The "client_uip" object is responsible for passing functions between the "call_back_processor" and the "client_uip_thread". It is also responsible for informing the "uip_log_manager", to log functions passed. "Client_uip_thread" manages the socket connections to the "server_uip" and is responsible for sending functions to and receiving functions from the "server_uip" through sockets.

The "server_uip" is more complicated than the "client_uip". It keeps listening for new "client_uip" connections on a well-known port and can connect and communicate with multiple "client_uips" at the same time. Figure 7.5 illustrates its architecture.

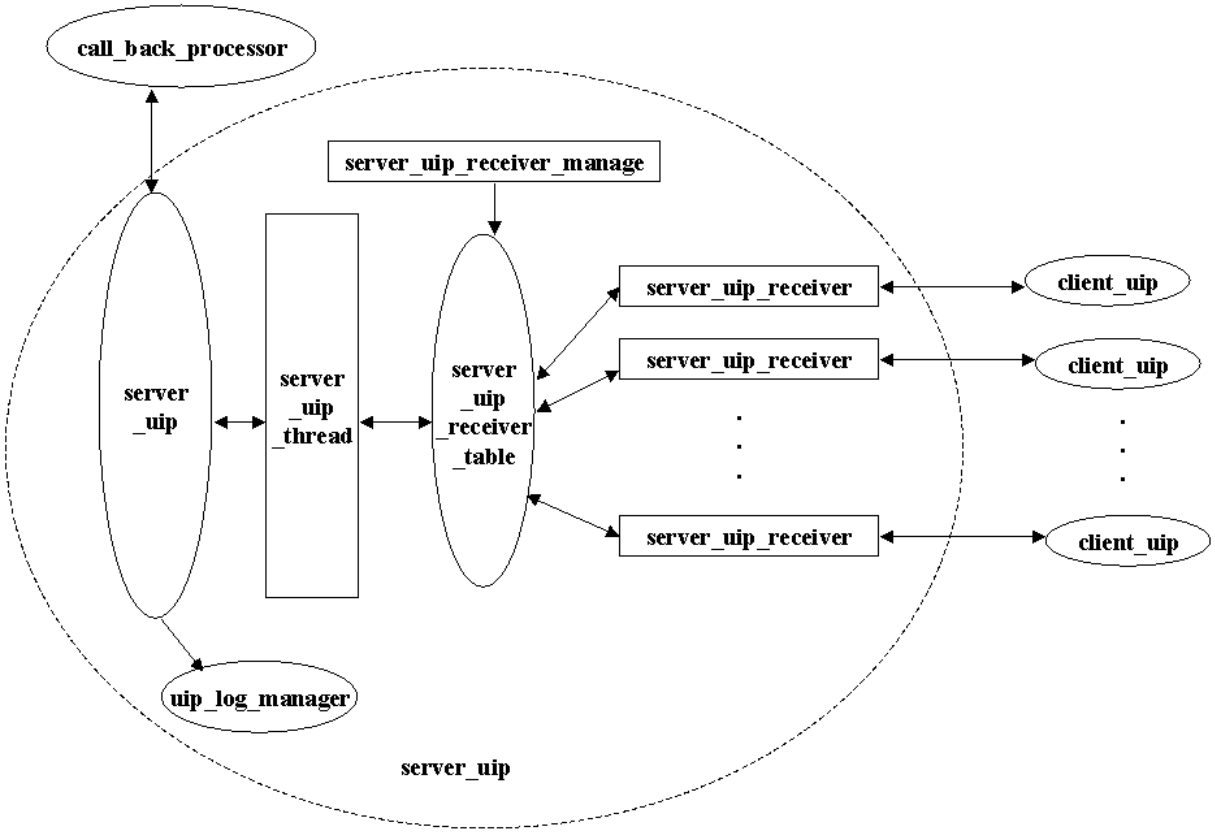

Figure 7.5: "Server_uip" architecture

The functionality of “call_back_processor” and “uip_log_manager” are the same as those in the “client_uip”. The “server_uip” object is responsible for passing functions and client IDs between “call_back_processor” and “server_uip_thread”. It is also responsible for requesting the “uip_log_manager” object to log functions passed. “Server_uip_thread” is responsible for passing functions and client IDs between “server_uip” and “server_uip_receiver_table”. It is also responsible for accepting new “client_uip” connections and informing “server_uip_receiver_table” to add a new “server_uip_receiver” to communicate with that “client_uip”. “Server_uip_receiver_table” creates and manages a number of “server_uip_receivers”. It is also responsible for deleting them. “Server_uip_receiver_manage” thread’s responsibility is to inform the “server_uip_receiver_table” to delete bad “server_uip_receievers” from time to time – bad “server_uip_receiver” means that the connection between it and its “client_uip” has some problems. Each “server_uip_receiver” thread manages the socket connections to a “client_uip” and is responsible for sending functions to and receiving functions from the “client_uip”.

7.2.1.2 “Session_manager”

The “session_manager” module manages a number of sessions in the system. It is probably the most complicated object in the system. Figure 7.6 shows its architecture.

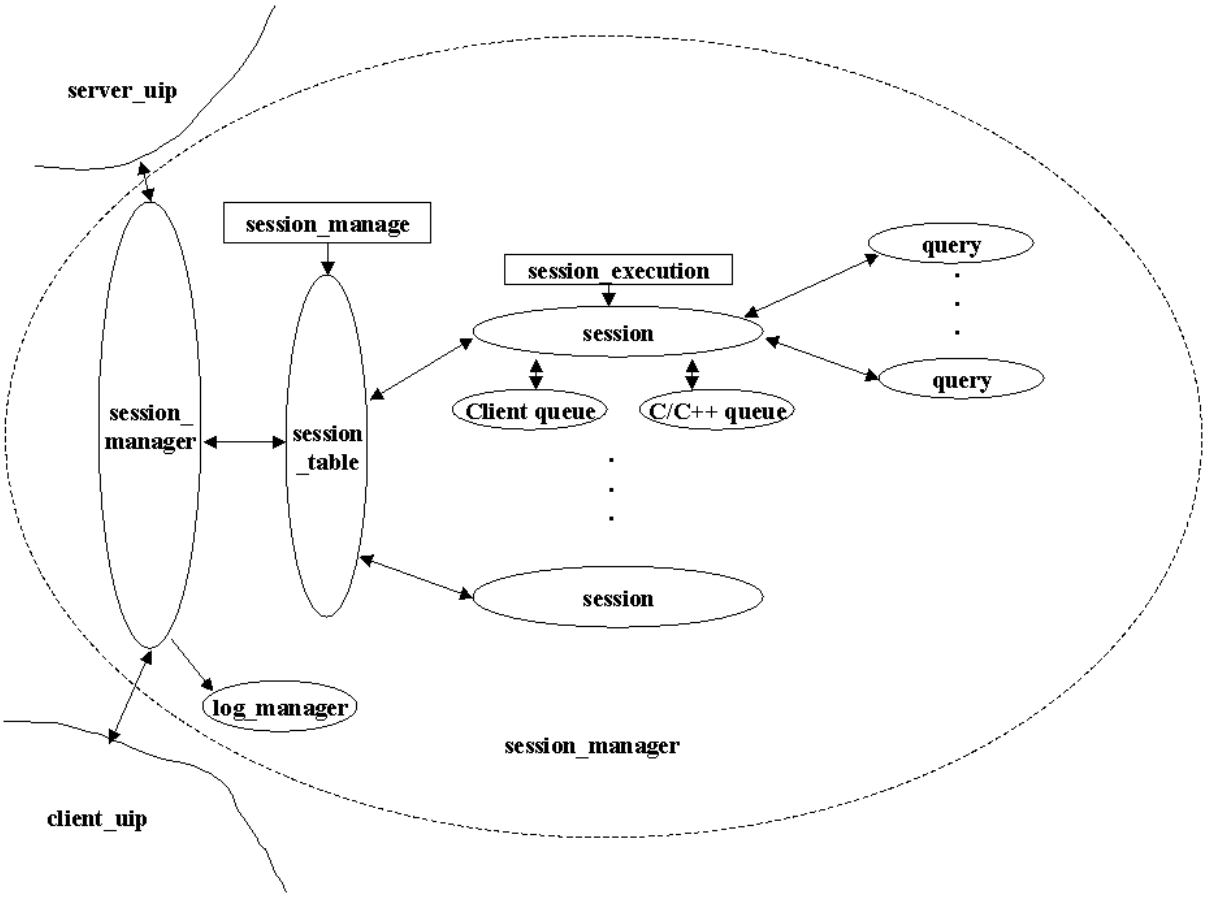

Figure 7.6: “Session_manager” architecture

The “session_manager” object communicates with objects “client_uip” and “server_uip”. In the Java MARIAN server, it is the “call_back_processor” for both of them. “Session_manager” uses “session_table” to manage all the sessions and queries of the system. Since the C/C++ MARIAN server only recognizes session ID and query ID, “session_table” maintains the mapping between client ID (“webgate” ID) and user ID on the one hand, and session ID on the other. It also creates and manages all the sessions of the system. The thread “session_manage” is responsible for garbage collection – it informs the “session_table” from time to time to delete inactive sessions.

Each “session” object maintains all the queries performed within this session so far. It has two “function_queues”, one (client queue) used to store functions passed to it from “webgate”, the other to store functions passed to it from the C/C++ MARIAN server. There is a thread

"session_execution" associated with each session. This thread informs the corresponding "session" object to take out functions from the two function queues and execute them in turn.

7.2.2 Passing Functions Through "Uip"

Figure 7.7 illustrates the mechanism of passing functions between "client_uip" and "server_uip".

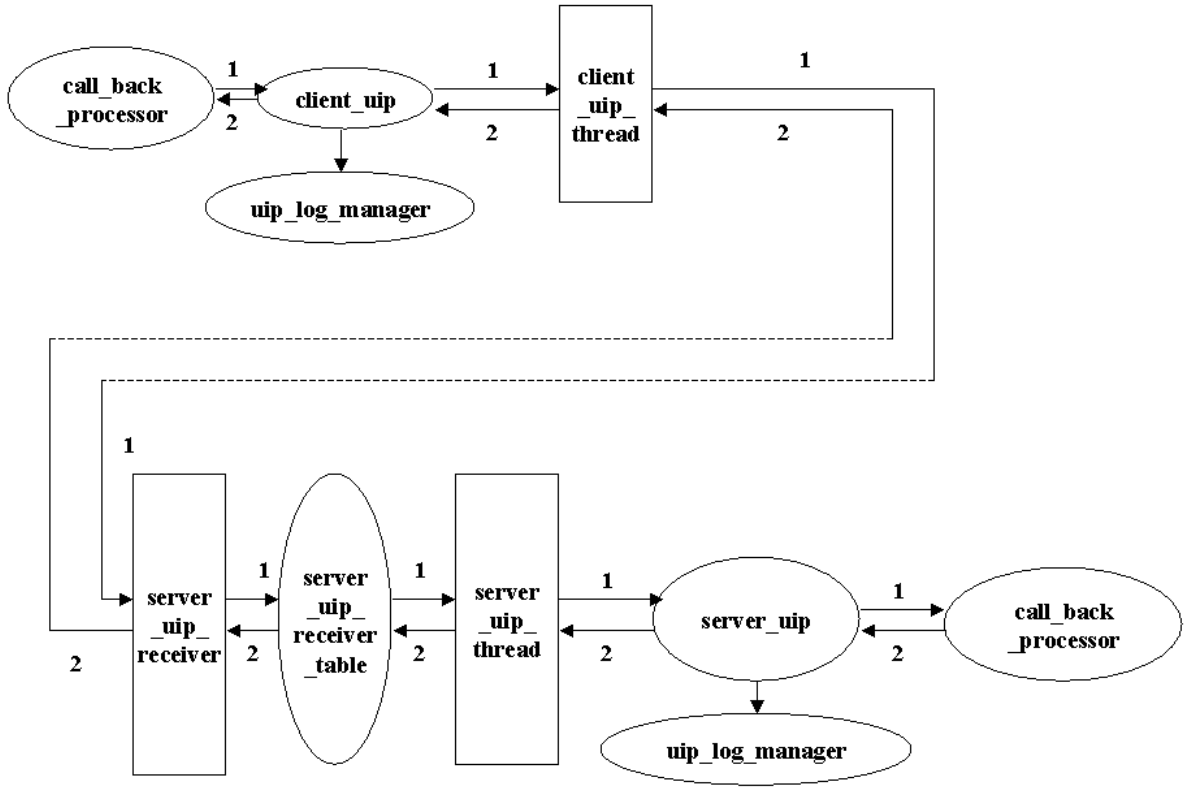

Figure 7.7: Passing functions through "uip"

(Note: 1 labels the path of functions from the client, while 2 labels the path of functions back from the server.)

When the "call_back_processor" in the "client_uip" side wants to pass something to the other side, it first creates a function, then passes the function to the "client_uip" object. The

“client_uip” first asks the “uip_log_manager” to log this function, then passes the function to “client_uip_thread”. “Client_uip_thread” writes the function to the “server_uip” it is connecting with through sockets.

The function is received by the “server_uip_receiver” which is connecting with this “client_uip”. Each “server_uip_receiver” has a unique ID assigned to it by the “server_uip_receiver_table”. The receiver passes the function along with its ID to the “server_uip_receiver_table”. “Server_uip_receiver_table” passes the function and ID to “server_uip_thread” which passes them to “server_uip”. “Server_uip” first informs its “uip_log_manager” to log this function, then passes the function and ID to its “call_back_processor” to process.

From the “call_back_processor” of the “client_uip” part to the “call_back_processor” of the “server_uip” part, a function goes through the path labeled 1 in Figure 7.7.

When the “call_back_proccessor” in the “server_uip” side wants to pass something to the other side, it creates a function, and then passes the function along with an ID (identifies a “client_uip”) to “server_uip”. “Server_uip” first asks the “uip_log_manager” to log the function, and then passes the function and ID to “server_uip_thread” which passes them to “server_uip_receiver_table”. “Server_uip_receiver_table” uses the ID to find a “server_uip_receiver” and then passes the function to that receiver. “Server_uip_receiver” passes the function to the “client_uip” it is connecting with through sockets.

The function is received by the “client_uip_thread”. It passes the function to “client_uip”. “Client_uip” first asks the “uip_log_manager” to log the function and then passes the function to its “call_back_processor” to process. In this case, the function goes through the path labeled 2 in Figure 7.7.

7.2.3 Operations

Figure 7.8 illustrates the operations of the Java MARIAN server.

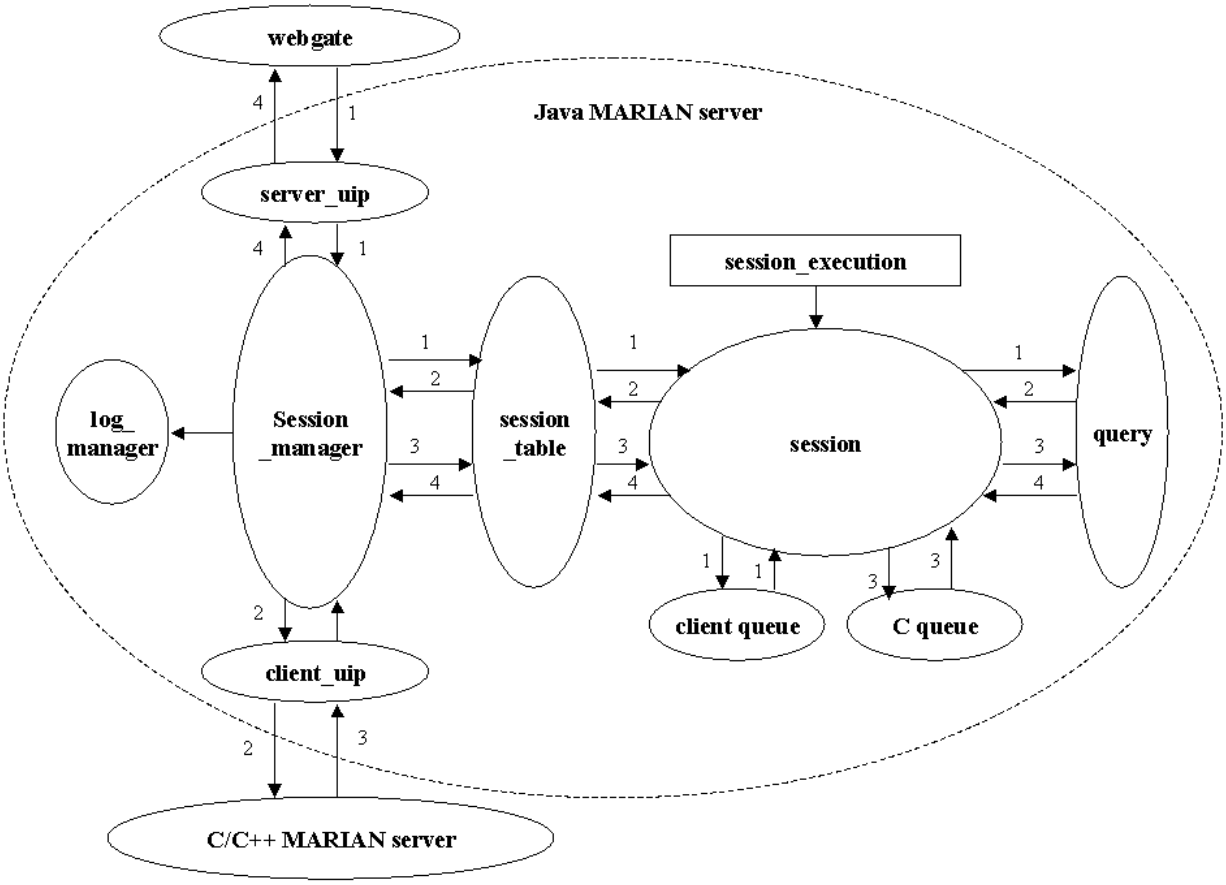

Figure 7.8: Java MARIAN server operations

(Note: numbers label steps in processing for a given user.)

When a “webgate” sends a request to the Java MARIAN server, the request reaches the “server_uip” object as a function. This function may contain a query a user wants to submit, a request to get more results from an old query, or several other possibilities. The “server_uip” passes the function, along with an ID identifying which “webgate” sent this function, to its “call_back_processor”: the “session_manager” object. The “session_manager” object just passes the function along with the ID to the “session_table” object. The “session_table” object uses the ID and the first parameter of the function (constrained to always be the user ID) to search the mapping it maintains between client ID, user ID, and session ID. If it finds a “session”, it passes the function to that “session” object with the first parameter (the user ID) removed. If the

“session_table” cannot find a corresponding “session”, it will create one. When a “session” object receives the function it simply puts the function into its client function queue.

Sometime later the associated “session_execution” thread asks this “session” to execute a function in the client queue. The “session” takes this function out of the queue, uses its first parameter (constrained to always be the query ID) to search the queries in this session. If it finds the “query”, it passes the function to the “query” object to process. If it cannot find a “query” with the same ID, it will create a “query” object with the ID.

Sometime later, after processing the function passed to it, the “query” object passes a function to the “session” object specifying that the “session” object send this function to the C/C++ MARIAN server. If the original request was a new query, this function will contain preprocessed query information. The “session” object inserts the query ID into the function and passes the function to the “session_table”, specifying that this function needs to be passed to the C/C++ MARIAN server. The “session_table” object inserts the session ID into the function and then passes the function to the “session_manager” object with the same specification. The “session_manager” knows that the “client_uip” is connected to the C/C++ MARIAN server, so it passes the function to the “client_uip” which writes the function to the C/C++ MARIAN server through its socket.

Sometime later -- after it finishes its own processing -- the C/C++ MARIAN server sends a function to the “client_uip”. For the query search task, the function will contain documents identified by the query. When it receives the function, the “client_uip” passes it to its “call_back_processor” which again is the “session_manager” object. The “session_manager” passes the function to the “session_table”. The “session_table” uses the first parameter of the function (which is session ID) to find a “session”, removes the first parameter, and passes the function to the “session”. The “session” object puts the function into the function queue used for the C/C++ MARIAN server.

Some time later, the associated “session_execution” thread informs the “session” to execute a function in the C/C++ function queue. The “session” takes this function out of the queue, uses the first parameter of the function (which is query ID) to find the corresponding “query” object inside this “session” object, removes the first parameter, and passes the function to the “query” to process.

Sometime later, after processing the function, the “query” object sends a function to the “session”, asking the “session” object to pass this function to “webgate”. The function may contain documents identified by this query. The “session” object inserts the query ID into the function and then passes it to the “session_table”. “Session_table” uses the session ID to search the mapping it maintains and find the corresponding client ID (the “webgate” this function corresponds to) and user ID. It inserts the user ID into the function and passes the function along with the client ID to the “session_manager”. The “session_manager” simply passes the function along with the client ID to the “server_uip” which uses the client ID to pass the function to the correct “webgate”.

In summary, between the time the “webgate” sends a query and it receives result documents, data (functions) go through the path 1→ 2 → 3 → 4 in Figure 7.8. Since multiple users may access the system at the same time, the above operations are done independently and in parallel for each user.

“Session_manager”, “session_table”, “session” and “query” objects may all want to log information at any time (such as when a session or query is created or deleted). All the log messages first are passed to the “session_manager” object. Then the “session_manager” object asks the “log_manager” object to log them. Whether or not anything will really be written into a log file depends on the logging level of the “log_manager”.

7.3 Flexibility Considerations

Weak coupling was maintained throughout the design of the Java MARIAN server. This resulted in a very flexible system.

7.3.1 Top Level Weak Coupling

In the C/C++ MARIAN server, the “uip_handler” module is strongly coupled with the rest of the system in two ways. First, the number of sockets maintained by the “uip_handler” to communicate with the “webgate” is directly related to the number of sessions in the system. Each session will result in two sockets created inside the “uip_handler” module. Second, the “uip_handler” module has to know the names of all the functions which may be passed through it in advance. For each function, it has to know the number of parameters, the type of all the parameters, as well as the order of them, in advance. Thus to change the function set passed through “uip”, we have to change the code of the “uip_handler”, recompile it, and then run it – very inflexible.

The “client_uip” and “server_uip” is weakly coupled with their “call_back_processor” (the “session_manager”) in the Java MARIAN server (see Figure 7.3). The “client_uip” and “server_uip” are responsible for passing functions between the Java MARIAN server, Java “webgate”, and the C/C++ MARIAN server. They do not know the meaning of the functions passed. They also don’t know how many sessions there are inside the “session_manager”. In contrast, the “session_manager” knows the meaning of the functions passed but doesn’t know how the functions are passed to the other system by “uip”. It also doesn’t know how many sockets there are inside “uip”. These design decisions resulted in very weak coupling between “session_manager” and “uip”. If we want to change the mechanism of passing functions, we only need to change the “client_uip” and “server_uip”. If we want to change the explanations of functions passed, we only need to change the “session_manager”.

Also, there is no need for the Java “uip” to know the name, parameter number, type and order of all the functions before another class uses the service of “uip” to pass functions. Thus the Java “uip” is more flexible. Other classes can create whatever functions they want and inform the

"uip" to pass on the condition that the type of the parameters in those functions are currently supported by "uip".

Due to all this weak coupling, the Java "uip" becomes a convenient tool for passing complicated data structures between systems on different machines. It is possible to build many information protocols (like Z39.50 and Dienst) on top of it, and thus let the system support federated searching.

7.3.2 Weak Coupling Inside "Server_uip"

Please refer to Figure 7.5. The coupling between the "server_uip_receiver_table" object and the "server_uip_receiver_manage" thread is weak. Their relation is similar to that between the thread "user_manage" and object "user_manager" in the design of the Java "webgate". While the "server_uip_receiver_manage" thread takes care of the protocol used to delete bad receivers (i.e., receivers whose connections with corresponding "client_uips" have problems), the object "server_uip_receiver_table" takes care of how to delete a receiver.

Weak coupling was also maintained between thread "server_uip_receiver" and object "server_uip_receiver_table". "Server_uip_receiver" takes care of managing the socket connections to "client_uip" and passing functions to and receiving functions from it through sockets. The receiver also uses XDR to pass data through sockets. "Server_uip_receiver_table" doesn't know and doesn't care about these. It maintains the mapping between client IDs and receivers. Thus if we want to change the mechanism to pass functions (for example, change the number of sockets or not use XDR), we only need to change the class "server_uip_receiver". If we want to change the mapping between client ID and receiver (for example, allowing a special ID to be mapped to all the current receivers), we only need to change the class "server_uip_receiver_table". In addition to that, we can add a queue inside a "server_uip_receiver" object to implement flow control with the corresponding "client_uip" without affecting any other classes. We also can add a queue inside the

“server_uip_receiver_table” object to do the flow control among different receivers without affecting any other classes.

Finally, the coupling between “server_uip” object and other classes is weak too. The “server_uip” object takes care of passing functions to the “call_back_processor” to process and receive functions from it. It also takes care of informing the “uip_log_manager” object to write to the log. All the other classes do not know about these activities. As a result of this weak coupling, if the function passing mechanism between the “server_uip” and the “call_back_processor” changed, we only need to change the “server_uip” class. We also don’t need to change other classes when the service provided by the class “uip_log_manager” changed.

7.3.3 Weak Coupling Inside “Session_manager”

Please see Figure 7.6. There is very weak coupling among classes inside the “session_manager”.

The “session_manager” object takes care of the communication with “client_uip”, “server_uip”, and “log_manager”. Other classes (like “session_table”, “session”, and “query”) don’t know this. So if the communication between “session_manager” and “uip” or the service provided by “log_manager” changes, we only need to change the “session_manager” class.

The relationship between the “session_manage” thread and “session_table” object is similar to that between “server_uip_manage” and “server_uip_receiver_table”. While the “session_manage” thread takes care of the protocol used to delete old sessions, the real deleting work is done by the “session_table” object.

The coupling between the object “session_table” and the object “session” is weak. “Session_table” maintains a mapping between client ID (that identifies the appropriate “webgate”), user ID, and session ID. “Session” knows nothing about this. On the other hand, “session” uses the thread “session_execution” and two “function_queues” to do the flow control between “webgate” and the C/C++ MARIAN server. “Session_table” knows nothing about this.

As a result of this weak coupling, only “session_table” needs to be changed if we want to change the mapping between the three IDs (for example, we may want multiple user IDs mapped to one session ID or one user ID mapped to multiple session IDs). Also we only need to change class “session” or classes inside it if we want to change the flow control mechanism.

Even the thread “session_execution” and object “session” are weakly coupled. The thread only takes care of the flow control protocol and informs the “session” object to execute a function in the client “function_queue” or C/C++ “function_queue”. The “session” object doesn’t care about the protocol but it knows how to execute functions inside the two “function_queues”.

When executing a function, the “session” object takes out the first parameter (query ID) and uses it to find the corresponding “query” object. After that it passes the function to the “query”. Thus all the classes except the “query” know little about how to execute a function. This design decision was made to maintain weak coupling between “query” and other classes. The flexibility of the system has been greatly improved as a result. We illustrate this by presenting the proposed future design of the “session_manager” in Figure 7.9.

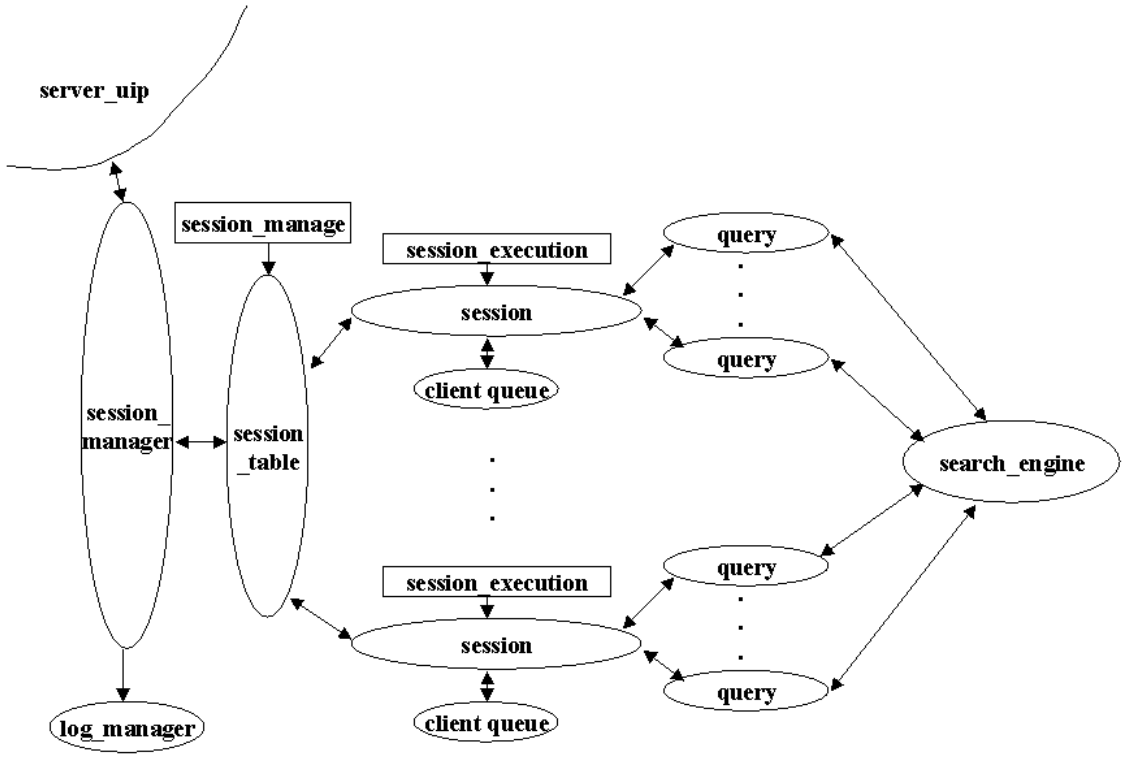

Figure 7.9: Future "session_manager" architecture

Due to the weak coupling, the changes in the design will be very small and easy. Only one class – "query" in the above figure, needs to be changed. When a "query" object wants to send preprocessed query data to the search engine, it just passes them directly to the "search_engine" object (it gets this object from the "resource_manager" object passed to it during construction) instead of sending them to the "session" object who created it.

7.3.4 Weak Coupling With "Webgate"

The coupling between the Java MARIAN server and the Java "webgate" is weak compared with that between the C/C++ ones. This also makes the Java system more flexible in terms of providing support for new document formats.

In the C/C++ system, after the documents of a query are found, the MARIAN server formats the documents from an internal format (for example, US MARC) to human readable strings and sends the strings to “webgate” which shows them to end-users. The format in which a document is shown to the end-user belongs to the user’s personal preferences. It is proper that “webgate” (user information layer) knows this since “webgate” contains the preferences of all the users. But letting the MARIAN server know this without knowing the user preferences unnecessarily increases the coupling of the system.

In the Java system, we broke this coupling by letting the MARIAN server only transmit the raw document strings to the “webgate”. It is inside the “webgate” that raw strings are formatted to human readable strings and presented to end-users. Different users may want different formats of the documents, then the “webgate” formats the document raw strings it gets from the MARIAN server in accord with the format of each user’s preferences. If we want to add some new document formats to show to our end-users, we only need to change the “webgate”.

7.4 Scalability Considerations

The design of the Java MARIAN server makes it very scalable by increasing concurrency control, optimizing usage of network connections, and facilitating the distribution of computation.

7.4.1 Concurrency Control

Based on the design diagram of the “session_manager” (see Figure 7.6), when there are multiple users accessing the system at the same time, there may be multiple threads executing independently in the “session_manager” at the same time. There is a possibility that some threads are searching the table while others may want to delete some entries from the table. They need to be synchronized to maintain the consistency of the table (critical data). The synchronization support of the Java language is not powerful enough to deal with the reader-writer problem: if

several methods of an object are specified to be synchronized, at any time only one thread is allowed to execute any one of the methods even if the method only reads some critical data.

To increase the concurrency and thus the scalability of our system, we designed, implemented, and tested a "reader_writer_mutex" class. The "reader_writer_class" successfully solved the reader-writer problem by allowing multiple reader threads (threads that only read critical data) to execute at the same time. The class also guarantees that neither reader thread(s) nor writer thread(s) will be starved. In the testing, a maximum reader parallelism of 92 was reached in the case of 100 readers and 5 writers. This shows that the "reader_writer_mutex" can significantly increase the parallelism and thus the scalability in the case of heavy load (many readers).

This class was used for the concurrency control of the "session_table" and "server_uip_receiver_table" in the Java MARIAN server. No bugs were found.

7.4.2 Optimize the Usage of Network Connection

In the C/C++ system, due to the strong coupling between the "uip" and the rest of the system, two sockets will be created between the "webgate" and the MARIAN server whenever a session is created. Since a session corresponds to a user's activity in a certain period of time, there will be many socket creations and shutdowns when there are many users who enter and exit the system. These many connection establishments and tear-downs will consume significant system resources due to the complicated mechanism used by TCP (please see section 2.4). It has been shown in[27] that at least a quarter of the total elapsed time is spent in establishing a network connection in HTTP/1.0. In addition to that, the slow-start mechanism used by TCP to avoid network congestion will make the network usage very inefficient especially in the case when many users access the system only to perform one or two queries each time. Both these result in poor scalability of the C/C++ system.

In the Java system, the "uip" has the flexibility to choose any number of sockets used for the communication due to the weak coupling between it and its "call_back_processor". Currently,

there are only two sockets between a “client_uip” and a “server_uip”. All the requests share these two sockets. Since there are only two connection setups and tear-downs, also because the system only suffers slow-start once at the very beginning of the connection, the system is making good use of the network connections. The Java system achieved better scalability in this regard.

7.4.3 Facilitate Computation Distribution

As mentioned before, the document formatting function is implemented by the MARIAN server in the C/C++ system. The formatting involves many string operations and thus puts a heavy burden on the MARIAN server, especially when there are many users using the system through multiple “webgates”. It is even possible that the formatting may become the bottleneck of the system thus affecting the scalability of it. In the Java system, the formatting function has been moved to “webgate” which is probably running on another machine. Thus we reduced the burden on the MARIAN server. When there are multiple “webgates” in the system, the formatting is essentially done in parallel and this increases the performance and thus the scalability of the system.

The design of the Java MARIAN server also facilitates the distribution of the search engine in the future. Please refer to Figure 7.9. We mentioned that due to the weak coupling, the search engine can be converted into Java in the future without affecting any classes except “query” and “resource_manager”. When we distribute search engines on different computers, even the classes “query” and “resource_manager” may find nothing changed. Figure 7.10 illustrates the future design of distributing search engines to multiple machines.

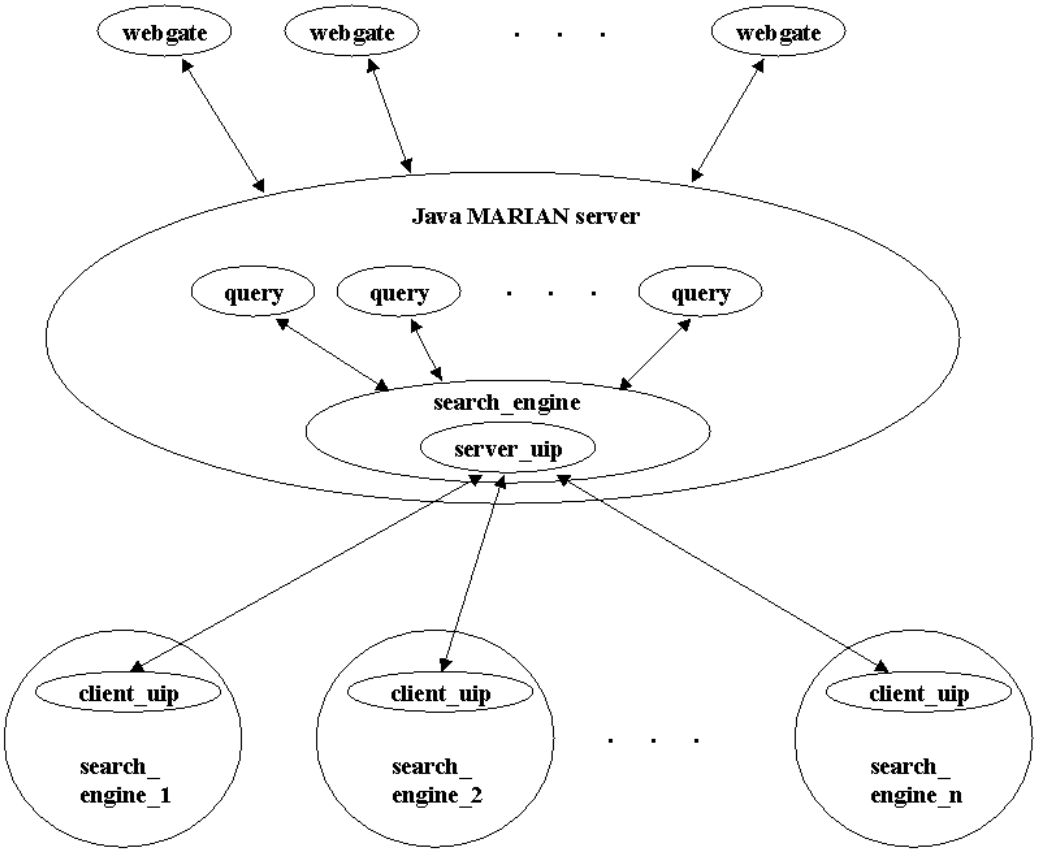

Figure 7.10: Distributed search engines architecture

In Figure 7.10, the “search_engine” object in the Java MARIAN server distributes the requests it gets from “query” objects to multiple “search_engines” possibly running on multiple machines. The “query” objects may still think that the searching is done locally and thus don’t need to be changed at all.

The scalability improvement is obvious: each “search_engine” gets a smaller workload in terms of query rate, and thus will respond faster.

Our “server_uip” can automatically detect new “client_uip” connections and delete bad “client_uip” connections. Thus this design makes the system very robust: when one “search_engine” crashes, the “search_engine” object in the Java MARIAN server will

automatically direct requests to other working ones. This design also allows the scalability of the system to increase smoothly: when the system is heavily loaded, the super user can just add one or several “search_engine(s)”. In this case, the “search_engine” object inside the Java MARIAN server will automatically connect to the new “search_engine(s)” and distribute requests among all available “search_engines”. There is no need to shut down the system.

Chapter 8

Applying Good Software Processes

The Java MARIAN server was mainly developed by 15 graduate students involved in a CS5604 class project in the fall semester of 1998. The students involved selected this for their term project. As mentioned in Chapter 1, system reliability is especially an issue in class projects. To verify that our approach to system reliability is effective even in such an environment, software processes based on the Software Engineering Institute's Capability Maturity Model were tailored and applied throughout this project. These include software product engineering, training, measurement and estimation, process improvement, and defects prevention. This chapter describes how we applied and tailored these software processes in the development of the Java MARIAN server.

8.1 Project Development Lifecycle

Basically speaking, the development of the Java MARIAN server followed the waterfall lifecycle. Figure 8.1 illustrates the basic phases we followed.

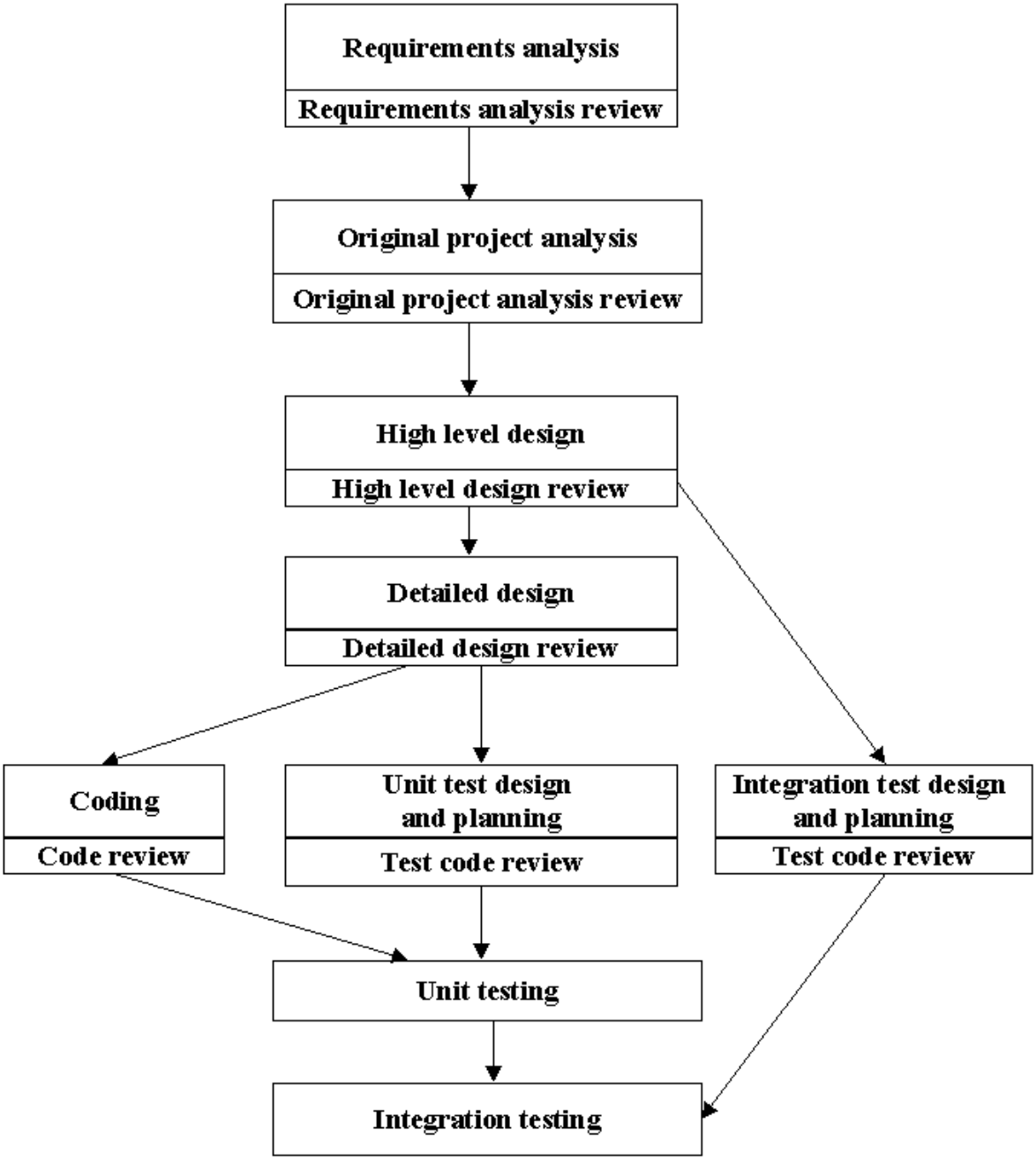

Figure 8.1: Basic development phases

We first did the requirements analysis, then the analyzing of the original project (the C/C++ MARIAN server), then the system high-level design, and then the detailed design. After that the source code of the system and the test code were written in parallel. Finally, we performed unit testing and integration testing (the design of the integration testing began after we finished the high level design). At the end of each phase, we performed one or more reviews to make sure all the problems found in the phase had been solved.

8.1.1 Requirements Analysis

In the requirements analysis phase, we considered our resources in terms of the number of students available, the number of hours each student would spend on this project, and the complexity of converting each module of the C/C++ MARIAN server to Java. We changed our requirement from reengineering 6 modules from C/C++ to Java to 5 after we found that we had 15 students enrolled in this project versus the 19 to 26 we expected originally. The group assignments were done in this phrase. Six groups were identified. The *original project understanding* group was responsible for analyzing the C/C++ modules which needed to be converted to Java. The *high-level design* group was responsible for the high level design of the new system based on the work generated by the *original project understanding* group. The *uip* group was responsible for designing, implementing and testing the Java "uip". There were two *detailed design, implementation, and testing* groups which were responsible for the work generated by the high level design group. The last group, i.e., the *webgate interface* group, was responsible for producing an attractive new system interface and optimizing the code in the Java "webgate". At the end of this phrase, a review meeting was held to make sure the things we proposed to do were reasonable, given the resources available. The size of our product was also estimated in the meeting.

8.1.2 Original Project Analysis

Although this is not a typical phrase in the waterfall lifecycle model, we found it was necessary since this project was a reengineering project. In this phrase, the members in the *original project understanding* group read the code of the parts of the C/C++ MARIAN server which were to be converted to Java. They wrote documentation describing the detailed operation of the parts they were analyzing. Figures and graphs were widely used in the documentation to help following group(s) understand these operations quickly, without diving into the sea of C/C++ source code. Several review meetings were held between the members in the *original project understanding group* and *the high level design* group to make sure that the documents generated in this phrase were easily understood and detailed enough to carry on the following work – the high level design of the system.

8.1.3 High Level Design

In the high-level design phrase, the designers read the work generated by the understanding group and then produced the design of the new system. Major classes of the system were identified. The high-level design document contains the class relationship diagram of the top-level classes in the system. Each relation between them (message passing or function call) was explained in detail. The signatures of all the public methods of those classes were specified. Furthermore, the high level designers wrote scenarios for the high level design to help following group(s) understand the operation of the system better. Several review meetings were held and a significant amount of time was spent in design discussions. We compared several design choices in terms of their flexibility, scalability, complexity, testability, reusability, and efficiency. In the end we believed we found the best choice for the system in the long run.

8.1.4 Detailed Design

In the detailed design phase, all the methods of all the classes were identified, and the flowchart of each method was generated. The flowcharts were detailed to the extent that on the average each page of flowchart corresponded to 60 ~ 70 lines of source code. Detailed designs were reviewed at the flowchart stage, before coding began. In the review meetings, we removed tricky

algorithm errors and optimized parts of the system. Testers participated in those meetings so that they could start designing test cases for branch coverage without seeing the source code of the system.

8.1.5 Coding

In the coding phase, developers were required to write code based on the detailed design document (flowchart). Code reviews were performed after they produced the code but before they compiled it, since according to[28], reviewing code before compiling it can remove syntax defects not detected by a compiler and at the same time generate satisfaction that comes from doing a quality job. We checked boundary conditions and typos as well as misuse of the Java language, during the code reviews. We verified that the code written really followed the flowchart produced in the detailed design. We also emphasized the places where inline comments should be added to make the code easily readable. Some good coding styles were found from the meetings and applied in the development of the project. Based on the number of problems found per thousand lines of code, we performed additional code reviews to guarantee that as many bugs as possible were removed from the code before we tested it. On average, two to three reviews were needed.

8.1.6 Unit Testing

The unit test design and implementation were done in parallel with the development of the source code. Some test code was written even before the source code to be tested was ready. Code reviews were performed on the test code too. On the one hand this removed bugs that existed in the test code. On the other hand some bugs in the source code also were found and removed just by reviewing the test code since developers participated in the test code review. The review meetings also generated some good coding styles especially suitable for test code: they make the test code easy to follow and test results easy to check by human beings.

Unit testing was done following a bottom-up strategy. We first tested and debugged classes that are not using services provided by other classes. Then we tested and debugged classes that use the services of the classes we had already tested and debugged. Sometimes, testers coded dummy classes. Due to the weak coupling of the system, coding those dummy classes was relatively easy.

8.1.7 Integration Testing

There is no clear line between unit testing and integration testing. Since we were using a bottom-up strategy, the later phase of unit testing was really testing the cooperation of many classes. The integration testing here refers to the testing of the communication between "session_manager" and "uip", and also the testing of connecting the Java "webgate", Java MARIAN server, and the C/C++ MARIAN server together.

8.2 Project Management

A significant amount of time and effort has been put into the management of this project since we believe this is crucial to the success of a multi-person project.

8.2.1 Group Assignment

Group assignment was done at the very beginning of the semester. Six groups along with their responsibilities were identified for this project. For each group, workload at different times during the semester had been specified so that members could arrange their work accordingly. For example, the *original project understanding* group would be very busy in the first third of the semester while the *detailed design, implementation and testing* groups had almost nothing to do until the middle of the semester.

To choose the most suitable persons for all the positions, also to respect each member's interest as much as possible, a group preference questionnaire was sent to each member. Each member was required to list four positions he/she preferred, in order. Also each was required to explain

why he/she liked that position in terms of his/her background and interest. Analyzing the questionnaire we found 8 students were interested in the *webgate interface* group position (first preference) while we only needed three. On the other hand only one student expressed tentative interest in testing while we needed at least 3 testers. Thus there was strong competition for the *webgate interface* group position and in the end we chose those who had usability engineering experience and had done some interface design before.

8.2.2 Training

Training was used to make sure each member had the required skill to perform his/her task effectively. The Java programming training performed during the development of this project proved to be very effective.

Many members said that one of the reasons they chose this project was because they didn't know or were not familiar with the Java language. They hoped they could learn some Java programming skills by working on this project. Therefore we decided they needed to be trained in Java language programming. Otherwise they would end up spending a lot of time studying the language by themselves. Further, many things they would learn in this way might not be used in the project. So training was aimed to help them focus more time on the development of the project.

A training questionnaire was sent out to 9 members based on their positions and roles in the project. We only trained those who needed to know Java for their task. There were 12 questions in the questionnaire. Some questions were very simple (like "Do you know how to compile a Java program?") while others were complicated (like "Do you know how to use the synchronization control in Java?"). To save time and meet the deadline of the project, only questions related to the development of the project were put in the training questionnaire. The purpose of the questionnaire was to identify for each member how much needed to be taught.

Based on the answers to the survey, members were assigned to three groups. The members of the basic level group knew nothing about Java including how to compile and run a Java program. The middle level group members knew some Java but were not familiar with complicated operations. The advanced level group members knew and had used Java before but needed some reminding since they had not used it for some time. There were two members who knew the answers to all the questions in the questionnaire and thus were excused from the training.

Training for the three groups was performed in parallel by three trainers (Java experts) at the same time. Different training strategies were used for different groups since members had different backgrounds. A training page was posted online after the training. The training page explains how to download and install JDK, the usage of some system classes as well as some features of the Java language. We built the page in such a way that everything on it was useful for the development of this project, and almost everything used in this project (in terms of Java language) was on the page. This page helped greatly in our software development.

The result of the training was amazing. Even those members who knew nothing about Java at the beginning could write code with high speed in the coding phrase by consulting the training page. They said the training saved them a tremendous amount of time in learning Java programming.

8.2.3 Measurement & Estimation

Measurement and estimation also were very important in the development of this project. On the one hand they helped us make reasonable plans based on previous performance. On the other hand by analyzing the data collected we figured out ways to improve the processes we followed in developing the system, thus increasing our efficiency.

The project manager held weekly meetings with each group and individual members. In the meeting they planned the next week's task for the group or member. A public Web page was set up with all the deadlines and task assignments for group and members. When the deadline came, another meeting was held to determine whether or not the tasks had been accomplished. After

that a colored percentage number was posted on the page near the deadline. 100% success was shown as green, 50% as yellow, 0% as red. If much less than 100% was achieved, a reason was posted too. Since all the members want to see deadlines under his/her name posted green (which was considered in grading), this page acted as a tool for motivation. The result was that a majority of students achieved all green, some even achieving some deep green (indicating more than 100% success – i.e., ahead of schedule).

That page alone was not enough since we still had the problem of making reasonable estimates in the meetings. We used a table with the following format (see Table 8.1) to measure the performance of all the members.

Table 8.1: Measurement and estimation table

date	name	position	task description	estimated time	actual time	comments
12/01/98	XXX	Developer	XXX	20 hours	18 hours	
12/03/98	XXX	Tester	XXX	10hours	20 hours	Understand the design document took too much time
…	…	…	…	…	…	…

The column “date” records when the corresponding task has been finished. The “name” column records who performed the task. The “position” column records the position of the person when he/she performed the task. The “task description” column describes the task performed. This description should be understandable in terms of workload – for instance, “code classes AAA and BBB based on detailed design”. Where possible, numbers were used to help measure it quantitatively. The “estimated time” and “actual time” columns show the estimated and the real time taken to perform the task, respectively. The “comments” column is used if there is a big difference between the estimated and the actual time used.

This table and the public deadline page were introduced into the project late in the semester. We benefited from them very much in terms of making accurate estimations based on a member's previous performance. Many estimates made at the beginning were far from accurate since we had nothing to base them on. By the end of the semester, our estimates were almost 90% accurate. We found that design time is more difficult to estimate than coding since it involves much more creativity. Also we made "poor" estimates at the beginning of the testing because we didn't consider the significant amount of time taken for the tester to understand the operation of the system.

During the unit testing phrase, we built a bug history table (see Table 8.2) to record all the bugs we encountered.

Table 8.2: Bug history table

time	bug description	severity	bug reason	time taken to fix	comments
12/10/98	Class “coverage_string_pair”, couldn’t set coverage using the constructor	Severe	Typo in the parameter of the constructor of the class “coverage_string_pair”, “converage” should be “coverage”	15 minutes	In the constructor when assigning variables try to use different names, like “my_coverage = coverage”
02/10/99	Class “uip_log_manager” couldn’t log data correctly when the logging level is 2	Not very severe	Class “rpc_function” method “to_stream()” forgets to flush the stream when the logging level is not high enough	20 minutes	
…	…	…	…	…	…

There is one row for each bug found during the testing of the system. The “time” column specifies the date the bug was fixed. The column “bug description” specifies the behavior that created awareness of the bug. The “severity” column specifies how severe the bug would be to the whole system if not fixed. The column “bug reason” specifies what caused the bug, and the

column “time taken to fix” specifies how long it took the corresponding developer to find the reason and fix the bug. The “comments” column is used to write something which the developer or tester believes can prevent similar bugs from occurring again. Analyzing this table helped us improve the efficiency of our programming.

We kept records for all the review meetings we held during the development of the project. For each meeting we recorded persons attending, time taken, things covered as well as conclusions reached. The time taken and things covered were used to estimate future review meetings.

8.2.4 Process Improvement and Defect Prevention

Though we believed we had followed very good processes, we knew they could never be perfect. So we kept an eye on possible improvements of our process throughout the development of the project. We observed the result of our review meetings, and analyzed the data we collected using the estimation and measurement table. We also asked group members for their thoughts on the bottlenecks in the development during our weekly meetings.

For example, during several high level design and detailed design review meetings developers and testers complained that they had difficulty understanding the design and operation of the system. They said they only saw “a batch of functions there, but don’t know how they communicated with each other”. We analyzed the reason for this and reached the conclusion that our high-level design document (which was only a class relationship diagram with explanations at that time) only gives a static view of the system. To perform their task efficiently, developers and testers also need a dynamic view of the system in terms of how different methods/objects communicate with each other. Then the high level designers were required to write another document describing how the system operates in terms of scenarios. Since the high level designers were very familiar with the design (they made it), it took them little time to write the operation scenarios (only 2 ~ 3 hours for 4~5 thousand lines of code). This saved developers and testers much time in understanding the operation of the system. They reported that it only took

them 20 to 30 minutes to understand the operation of a system of several thousand lines of code using the scenarios.

Another process improvement was the adding of synchronization control checking in detailed design review meetings. This system provides service to multiple users at the same time. No one can predict what each user will do at a certain time. There may be multiple threads running independently inside the system. To make sure our system will not crash or behave undesirably under all possible situations, we decided that a synchronization-control checking process needed to be added to our detailed design review. In this process, we first examined each method of an object, to see whether or not it would be possible that more than one thread would execute it at the same time. If the answer is yes, we check whether or not such execution will produce undesirable results to our system. If that is possible, we mark this method as a synchronized method. Second, for each method of an object we check whether or not it will be executed at the same time when other methods of this object are executed by other threads. If there is such a possibility we check whether or not this will produce undesirable results. If so, we mark both methods as synchronized methods. Many methods were marked as synchronized as a result of this process. It was also in this process that we identified the need for the "reader_writer_mutex" class. Mainly due to this process, no synchronization bugs were found in the Java MARIAN server during testing while such bugs were found both in the Java "webgate" and the C/C++ MARIAN server which were developed without such a process.

A significant amount of effort also was put into avoiding duplicated mistakes or preventing mistakes from happening at all. From the beginning of the semester we maintained a common error page (http://video.dlib.vt.edu:90/marian/cs5604/management/err.html) for our programmers. The content of this page was collected from several Java experts based on the mistakes they made when they did Java programming before. All our programmers were required to read the common error page (the time it took them to read the page was also estimated and counted into their workload) to avoid making similar mistakes. In addition to that, during our code reviews, if we found a mistake and believed it was not a special case, we posted it on our common error page and informed other programmers about this. Sometimes we even made use

of mistakes made by students in other projects (like those who were doing projects in the software engineering class), posting them on our common error page. We maintained a page (http://video.dlib.vt.edu:90/marian/cs5604/management/code_style.html) about the coding style in this project. Again programmers were required to read this page before they started coding. We updated this page dynamically when we found some good coding styles which we believe could help other people understand the program better or reduce the bug rate. All these were very effective: our programmers became more and more efficient and wrote more and more professional code with higher speed.

8.2.5 Software Reuse

We believe software reuse can reduce time and increase system quality. So during the development of this project we kept an eye on software reuse and also developed some reusable components when our time constraints permitted.

When designing the services provided by a class, we not only considered those which were needed in this project at that time, but also those which should be provided to make the services provided by this class complete. This increased the possibility that this class could be used in other projects or in the future development of this project.

For example, when we designed our “uip”, we allowed two ways of passing and receiving functions through it. Running in “DIRECT_CALL” mode, when it receives a function from its “call_back_processor” to pass to the other subsystem, it will block the “call_back_processor” until the function is written to the other subsystem through a socket. Running in “THREAD_CALL” mode, it will generate a thread to write the function while at the same time returning the control to the call back processor immediately. The same thing happens in the other direction – when the “uip” receives a function from the other subsystem it either generates a thread to pass the function to its “call_back_processor” to process or passes the function to the “call_back_processor” directly, depending on the mode it is running in. The advantage of “THREAD_CALL” is that it will not block the caller when the function takes the callee a long

time to process. The disadvantage is that more threads are created in the system and they consume system resources. The user is allowed to choose either mode for the “uip” or even to choose different modes for different directions (sending or receiving functions) by changing a configuration file. Being able to provide this feature makes our “uip” suitable for different environments, thus increasing its reusability. The design of our “reader_writer_mutex” class also illustrates this by allowing the user to specify a maximum number of concurrent readers.

We reused code throughout the development of this project. The class “client_uip” was used both in “webgate” and the Java MARIAN server. It also will be used in our future design to distribute search engines, as illustrated in Figure 7.10. The “uip_log_manager” class was used both in “client_uip” and “server_uip”. The “reader_writer_mutex” class was used in “session_table” and “server_uip_receiver_table” for synchronization control. Since the constructors of many classes need to read configuration information from a file, we developed and tested some code to do this and let all our developers use this piece of code in writing the constructors of their classes.

We also reused design patterns we created earlier. The design of “server_uip” and that of the “session_manager” follow similar patterns. Also similar design patterns can be found in the relationship between “server_uip_receiever_manage” thread and “server_uip_receiver_table”, the relationship between “session_manage” thread and “session_table”, the relationship between “user_manage” thread and “user_manager”, and the relationship between “dynamic_uip_manage” thread and “uip_manager”.

We also reused some design patterns from other projects – the design of “request_response” thread in the Java “webgate” made use of the design of one of the projects of the Networking class (CS5516) to develop an HTTP server. In addition to that, after we found possible improvements in one design, we reviewed all the similar designs to see if it were possible for them to benefit from those improvements.

8.3 Results

The Java MARIAN server contains about 40 classes and 20,000 lines of code[*]. About 1,500 man-hours were spent in developing this subsystem. These include the time spent in requirements analysis, original project analysis, high level design, detailed design, coding, unit testing, and integration testing. These also include the time spent in all the review meetings and training.

During the unit testing phase, eleven bugs (including non-severe ones) were found and fixed. The total time spent in correcting those bugs was less than 3 hours (not including the time to document the fixes). After that, no bugs were found to date. This shows that the system we developed following such processes is very reliable.

If we assume a full-time employee works 40 hours per day, and 4 weeks per month, then the time we spent in developing the 20,000 lines Java server is roughly 9 man-months. Considering the quality of the system, and considering the background of our members at the beginning of the semester (they had little experience in developing digital library systems, and quite a few developers knew almost nothing about Java), the results we achieved are very encouraging.

The results of the experiments described in the next chapter further verified that the system developed following those processes seems to be reliable.

[*] The large average class size (about 500 lines) is due to the focus of functionality instead of a pure object oriented design to meet project deadlines.

Chapter 9

System Performance Experiments

To verify that the reengineering activity did produce a scalable system, also to further verify that the system developed following good software processes is reliable, several series of experiments were designed and performed. This chapter describes the design of our experiments and their results.

9.1 Experimental Design

Figure 9.1 shows the model adopted in our experiments.

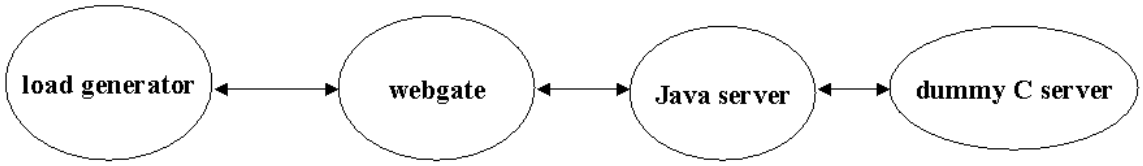

Figure 9.1: Experiment model

The load generator acts as the “formit” module, sending requests to “webgate” and receiving responses from “webgate”. Since we are only interested in the performance of the Java part developed in the past year, a dummy C server was coded for our experiments. After receiving a request, the dummy server will delay 0.5 to 4.5 seconds (average 2 seconds) on purpose to simulate the processing time of a query before it sends back a response. The real query log of the MARIAN system has been used by the load generator to generate requests in all the experiments. Also to simulate the real workload to the system, each query will result in 20KB of result documents being sent back to the client (since the 20KB size is the average observed in logs).

Many values are measured in our experiments. These include the response time of requests observed by the load generator, the response time observed by the “webgate” and the response time observed by the Java server, as well as the CPU and memory usage of the load generator,

the “webgate”, and the Java server. For the response time, we measured the average value as well as standard deviation. We divided requests into two types. Type 1 requests are those which reach both the “webgate” and the servers (for example, a query submitted by a user) while type 2 requests are those which only reach the “webgate” (for example, user login to the system). We measured the response time and standard deviation for the two types of requests separately.

Figure 9.2 illustrates the design of the load generator. It is multi-threaded. Upon execution, the “user_creator” thread will create a number of “user” threads. Each “user” thread represents a user. Upon starting, the thread first logs in to the system, then performs queries one after another. It acts exactly as “formit” – opens a socket to the “webgate”, writes the request, and then waits for the response. It closes the socket after it reads out a response, and sends out another request immediately. Note this is a more demanding situation than that of a typical user in the real world, who will not send requests one after another without any delay. Each “user” thread in the load generator records the time it observes the “webgate” taking in response to its requests and reports the values to a shared object – “test_statistics”. To avoid synchronization problems (many “user” threads send requests to the system at the same time, which causes congestion in the system), the “user_creator” thread will not create a large number of users all of a sudden at the beginning of an experiment – it delays 500 ms on purpose before creating a new “user” thread.

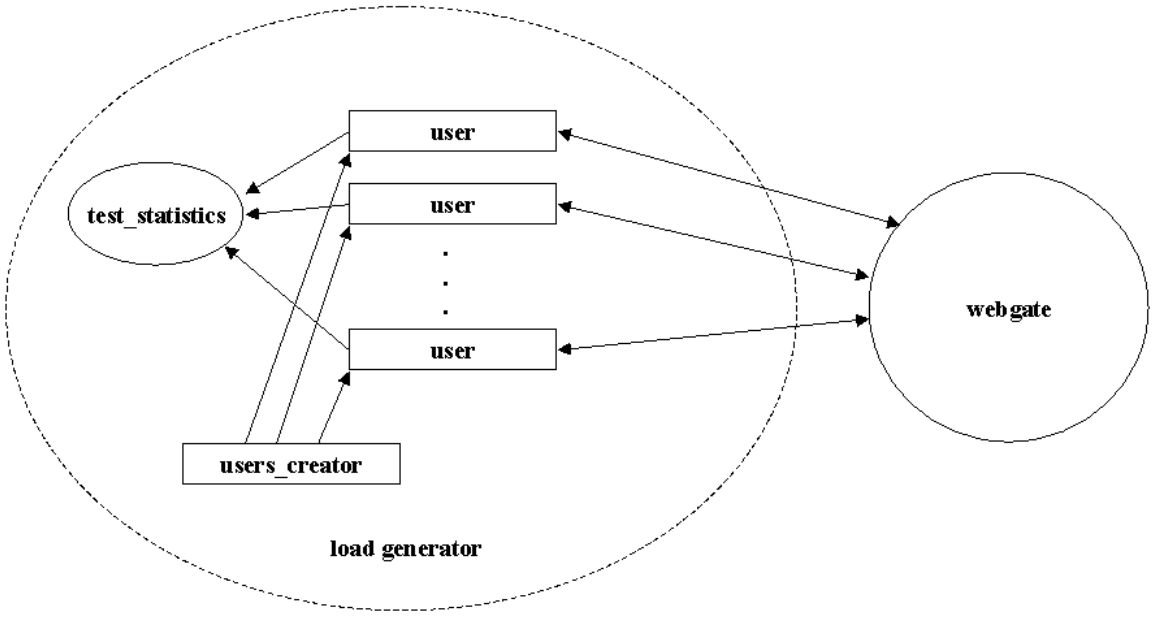

Figure 9.2: Load generator architecture

9.1.1 Measurements

To measure the response time observed by the "webgate" and the Java server, we inserted some code inside them to record time values at different stages. When the "webgate" sends a "search_collection" function to the Java server, it uses the first two parameters of the function (user ID and query ID) to form a unique request ID and then informs the shared object "test_statistics" to record the start time for this request. When the "webgate" receives a function "solicit_biblio_query" from the Java server, it uses the first two parameters to form the request ID again and then informs the "test_statistics" object to record the end time for the request. The same thing happens to the functions "biblio_query_text" and "show_retrieval_coll". The Java server uses similar mechanisms for functions it sends to and receives from the dummy C server but uses session ID and query ID to form the request ID. Thus the object "test_statistics" records

the start time and end time of many requests during the execution of the system and it is possible for it to calculate the average value and standard deviation of them at the end.

The class “test_statistics” is designed to minimize the CPU and memory usage of the system yet collect as much statistical information as possible. During the running of the system, it only records the start and end time of each request (identified by request ID) and stores them. The real calculation of average value and standard deviation is done immediately before the system exit. Thus it takes trivial CPU power until the system exit, after all the testing has been finished.

9.1.2 Cases

There are four experimental cases. In the first case, all the Java modules (load generator, “webgate”, and Java server) run on the same machine; thus all the computation and disk operation happens on one machine and this can be considered as a centralized system. In the second case, we have the computation and disk operation distributed to two machines by separating the load generator and “webgate” from the Java server. In the third case, we have two load generator “webgate” pairs running on two machines communicating with the same Java server. Computation and disk operation are distributed to three machines in this case. In the last experiment case, we distributed the computation and disk operation to five machines by having four load generator “webgate” pairs communicating with the Java server from four different machines. Workload is distributed equally in the experimental cases where multiple “webgates” are involved. In all the experiments, we increased the workload (query rate) of the system until quit a few queries took the system more than 60 seconds to process. Since it is unlikely that a user in the real world will wait more than 60 seconds for the results of a query, we consider the system reaches its upper-limit with such a workload.

All experiments were run in the Virginia Polytechnic Institute and State University Department of Computer Science’s graduate lab where all the machines are GATEWAY 2000 Pentium II 233 MHz with 64M memory running the Windows NT 4.0 operating system. All the machines are

connected to the same switched 10Mbps ethernet. We ran all experiments when there were few students in the lab, so the background traffic can be ignored.

The software system was set up as follows: In “webgate”, the thread “user_manage” will inform the “user_manager” object to unload inactive users/queries every 10 seconds (we set this to such a small value to minimize the memory usage of the webgate). The “uip_log_manager” object in the Java server records the name of all the functions sent and received. The “log_manager” object in the Java server records the creation of a query. This constitutes a medium level of detail logging, which we believe will likely happen in real world systems.

9.2 Getting the Correct Data

To get the correct measurement data from such a complicated system is not very easy. For example, the time a query spends in the Java server is measured by the “webgate”. If the computer on which the “webgate” is running is heavily loaded, the “webgate” will report the wrong data – it may report that a query takes a long time in the Java server while in fact delay is caused by the “webgate” itself. The same thing will happen if the Java server is heavily loaded – it will report the wrong data about the time a query stayed in the dummy C server.

To get the correct measurement data in all the above cases we coded a dummy “webgate” and a dummy Java server. The reason is that when the measurement is taken from a computer that is not heavily loaded, the result is more believable. The dummy “webgate” will send functions to the Java server as the real “webgate”. That way it does not process complicated user information, and thus consumes only a trivial amount of CPU power. Thus it can measure the time a query stays in the Java server more reliably than the real “webgate”. Similarly, the dummy Java server will send functions to the dummy C server as the real Java server would, but consumes much less CPU power. It can get more reliable data about the time a query stays in the dummy C server. We used the data we obtained in this way to calculate the time a request spent in the whole Java system as well as in the “webgate” and the load generator.

We also did some processing of the data concerning the memory usage of modules. Since the Java language is an interpreted language, the operating system will load the Java interpreter into memory first before running any Java program. Thus the memory usage reported is in fact the total memory consumed by both the Java interpreter and our program. To find out how much memory is consumed by our program we have to figure out how much is consumed by the Java interpreter. We obtained that by running a dummy program which consumes almost zero memory. We found that the Java interpreter will consume roughly 2.83MB. Consequently, we adjusted memory usage figures down by 2.83 MB to compensate for the memory taken by the Java interpreter.

9.3 Experiment Results

9.3.1 Removal of System Bottlenecks

At the beginning of the experiments, we found that the performance of the system was far worse than expected. Especially in “webgate”, the average response time delay was very long (about 16~17 seconds) when the query rate reached 30/minute. We decided that there must be some bottlenecks which prevented the system from achieving high performance.

After a careful analysis of the system design and behavior, several modifications were made to the “webgate” and Java server. As a result we achieved significant performance improvement – less than 4 seconds average response time delay when the query rate reaches 200/min in the case all modules are on the same machine.

One modification was made to the Java server. We changed the mechanism the thread “session_execution” uses to execute the functions in the queues of the corresponding “session” object. Instead of sleeping a constant amount of time and then waking up to inform the associated “session” object to check functions in the queues, the thread will suspend itself when there are no functions in the queues to execute. When a function is entered in either queue, the “session” object will resume the execution of the “session_execution” thread and the thread will inform the

"session" to execute the functions in the queues until both queues are empty. This significantly improved the performance of the Java server since when there are no functions in the queues the corresponding "session_execution" thread will be suspended and not consume any CPU resource at all. Also, when a function is entered in a queue, the thread is woken up immediately to inform the "session" to execute that function. Thus we achieved minimum delay yet used the CPU resource efficiently.

Several modifications have been made to "webgate", each of which speeds up this part several times. At the beginning of our experiments, we found that it took a very long time for the system to unload a query from memory to disk (700~800 ms on the average). Since a user will be blocked when the system is unloading his/her queries, some users experience long delays during queries while a significant amount of time is actually being spent waiting for the system to finish unloading earlier queries. Another bad effect of this slow unloading is that it prevents the system from reaching a high query rate. The reason is that if the speed of unloading queries is slower than the speed of new queries entered into memory, the system will run out of memory eventually. For example, if the system spends 750 ms unloading a query, the query rate can not go over 80/min for long periods of time. Something must be done to break this limit. After analyzing the design of the query object we found that the slow speed is mainly caused by two factors. First, the "query" object used complicated directory structures for storing on the disk. When a "query" object is unloaded to disk, about 7 directories and 10 files were created to store the content of it. Creating so many directories and files definitely slows down the system. Another factor is that when writing the content of a "query" object to file(s), the buffer was flushed many times before all the data has been written to a file. This results in inefficient usage of the hard disk and thus slows down the unloading. Based on those two observations, we optimized the "query" object so that when it is unloaded to disk only one directory and two files are created. Also the system only flushes the buffer after it has written all the data about a file into it. We observed the performance of the system after the optimization. This time it took the system less than 200 ms to unload a query. Thus the upper limit of the system performance has been raised to 300 queries/min.

Another bottleneck of the “webgate” is the amount of data transmitted between it and the module “formit”. Each time the “formit” sends a request to the “webgate”, the “webgate” sends back an HTML page that will be 10K to 20K bytes, through the socket. This amount of data caused a bottleneck in the system even though the two modules are on the same machine and socket connections are much faster than those connecting modules on different machines. We found this because when we reduced the amount of data sent through sockets for each request on purpose, we observed significant performance improvement. After analyzing the design of the system we found that it is possible for us to reduce the amount of data sent through sockets, by moving the HTML template page reading and replacement from “webgate” to “formit”. In this case the “webgate” only needs to send two things to the module “formit” – the HTML template file name and the value of tags to replace in the template page. After reading out these two values, the “formit” opens the template file, replaces tags in it using the tag values it gets from “webgate”, and then prints the result to the “environment” object which is a wrapper of the communication mechanism with the Web server. Since we moved the function of reading the template HTML file and replacing tags from “webgate” to “formit”, also since the two modules are running on the same machine, no CPU resource consumption has been changed for the system, but much less data was passed per request. To make sure that our experiments reflect the real time the system takes to process a query, we added reading the HTML template file and tag replacement into our load generator and let it count the time for doing this into the response time of a query. Significant performance improvement has been observed after this change.

After the above modification, we observed that in most of the cases, the data sent between “formit” and “webgate” is less than 9KB. Thus it can be put inside one IP packet. However, due to the mechanism of TCP, the system will establish a connection first and this requires three packets being sent between the two sides. Also at the end of the data transmission TCP will tear down the connection by sending four packets between the two sides. Thus in most of the cases, only two packets are used to transmit system data (one containing the request from “formit” to “webgate” and the other containing the response from “webgate” to “formit”) while 7 packets are transmitted between the two sides for TCP management (3 for connection setup and 4 for connection tear down). Since in our case the module “formit” and “webgate” are on the same

machine, it is unlikely that transmission between them will have errors (unless the computer memory has problems). Furthermore, passing data on the same machine will not result in any scrambling of the order of the packets. All this suggests that using TCP unnecessarily wasted system resources. So, we decided to change the connection between these two modules to UDP to reduce the unnecessary overhead. Since the system assumes the connection between the two modules are streams, while UDP is a connectionless packet transmission based protocol, also since in some cases the response sent from “webgate” to “formit” will occupy multiple packets, three classes were added into the system to simulate stream connections between the two modules on top of UDP. These classes guarantee that the system can still get the correct data even in case one response is made up of multiple packets, by maintaining a packet counter in the first packet. After this change, we observed significant system performance improvement.

Table 9.1 gives a summary of the modifications we made and the performance improvements brought by them to the system.

Table 9.1: Modification list

Modification Description	Performance Improvement
In the Java server, changed the mechanism of the thread "session_execution" used to inform associated "session" object to execute functions in the function queues, using suspending and resuming instead of sleeping a constant amount of time.	Significantly improved the response time of the Java server. Reduced the CPU usage by avoiding executing the thread when the corresponding queues are empty.
In "webgate", reduced the number of directories and files needing to be created, also reduced the number of times the flushing statements are called when unloading queries from memory to disk.	Improved the query unloading speed of the system from 700~800ms/query to less than 200ms/query. This brings the upper limit of the query rate from less than 80/min to more than 300/min.
Moved the HTML template file reading and tag replacing from "webgate" to "formit", and thus reduced the amount of data transmitted between the two modules for each request.	Improved the performance of the system 6~7 times, from 30 queries/min and 16~17 seconds average response time delay to 70 queries/min and 5~6 seconds average response time delay.
Changed the connection between the module "formit" and "webgate" from TCP to UDP.	Improved the performance of the system by another 3~4 times, from 70 queries/min and 5~6 seconds average response time delay to 200 queries/min and 4 seconds average response time delay.

All these modifications were easy to make due to the weak coupling maintained in the system design, a result of our focus on flexibility.

For the first modification listed in Table 9.1, only classes "session" and "session_execution" were changed due to the weak coupling between the "session" classes and other classes in the

system (mentioned in section 7.3.3). As a result, making this modification and testing it took less than 2 man-hours effort.

The second modification benefited from the weak coupling among classes “user_manager”, “user”, “query_manager”, and “query” (mentioned in section 5.3.2.2). To make this modification, we only changed class “query” and classes in it like “query_data” and “results”. As a result, this modification only took 2~3 man-hours to make and test.

The third modification benefited from the weak coupling between the “response” class and other classes in “webgate” (mentioned in section 5.3.2.4). It also benefited from the weak coupling between the “marian_cgi_response” class and other classes in “formit” (mentioned in section 4.3). As a result, only the class “response” in “webgate” and the class “marian_cgi_response” in “formit” were changed to enable this modification, and this took less than 1 man-hour of work.

Though the last modification was a little bit difficult, we still benefited greatly from the weak coupling in the design of “webgate” and “formit” (mentioned in section 4.3 and 5.3.2.4). After we added three classes to build a stream on top of UDP, the two subsystems communicated correctly without knowing that the underlying protocol has been changed. As a result, this modification took less than 8 man-hours of work.

9.3.2. Measuring the Cost of Measurement

Though we designed our load generator and the time measurement code carefully so that they consume as little system resources as possible, we still want to know the overhead added by them to the system. Two series of experiments have been designed and performed to get these data.

The first experiments were designed to gauge the measurement cost for the Java server.

The dummy "webgate" was used to establish this time value for the Java server. We ran the experiments using the Java server with and without the time measurement, under different workloads. Figure 9.3 shows the results in this experiment series.

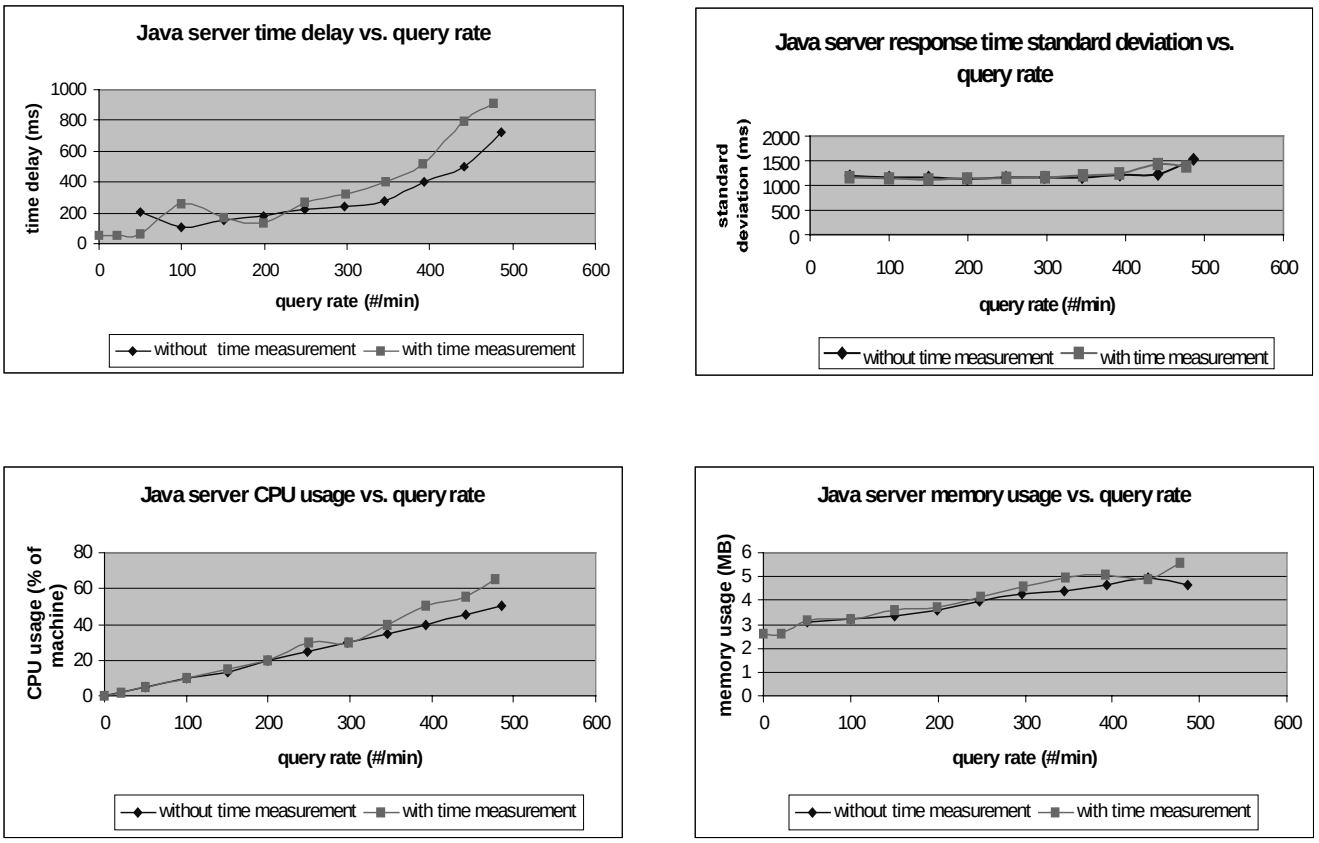

Figure 9.3: Java server cost of measurement graphs

The above curves show that time measurement added some cost in terms of time delay (in the worst case the cost is about 25% when the query rate is 480 queries/min). The overhead of CPU usage is about 30% in the worst case while that of memory usage is pretty small.

The second series of experiments were used to measure the overhead of the time measurement to the "webgate". We used the real Java server in our experiments. We ran the experiments on one "webgate" with and without time measurement under different workloads. Figure 9.4 gives the results in this series of experiment cases.

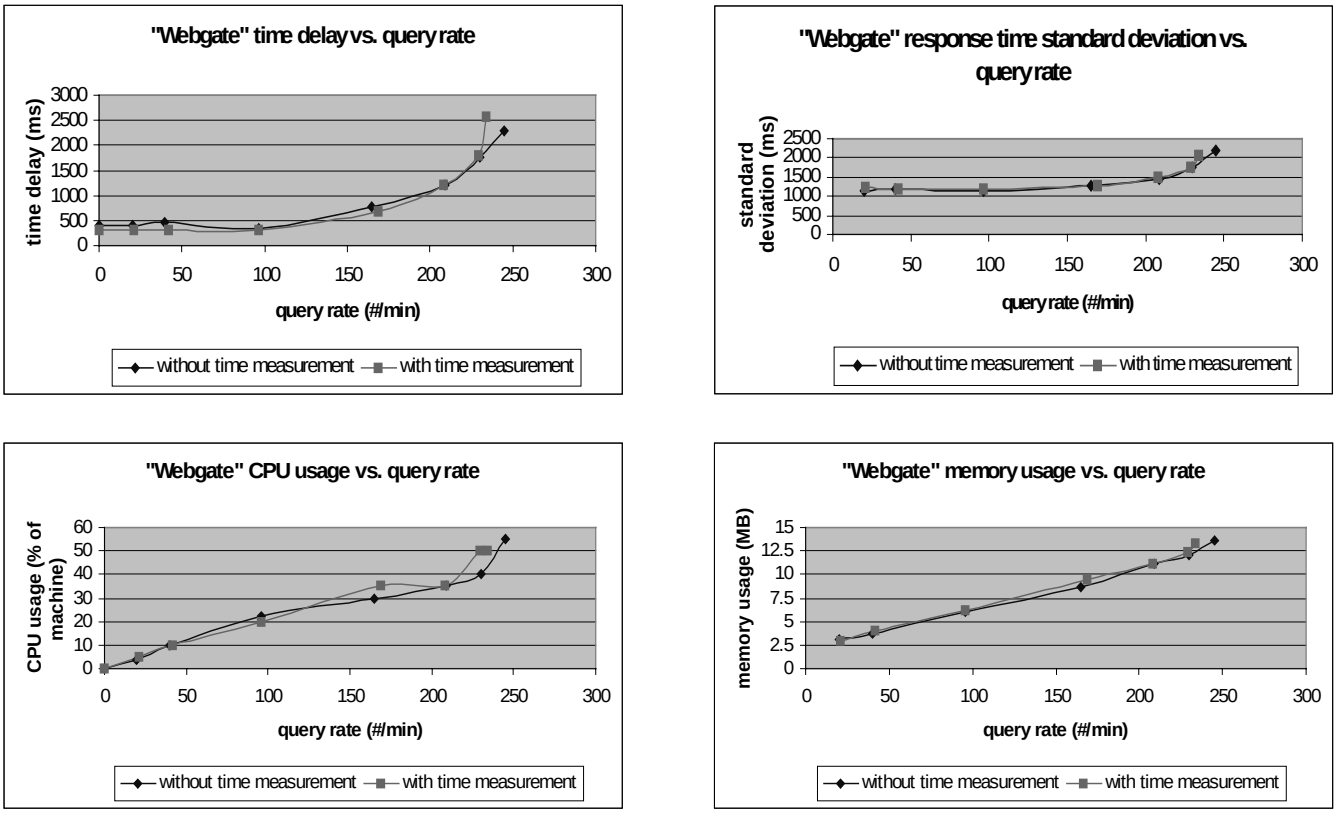

Figure 9.4: "Webgate" cost of measurement graphs

These curves suggest that the time measurement didn't add much overhead to the system in terms of time delay, time delay standard deviation, or CPU or memory usage.

The results of these two series of experiments suggest that the performance of the system will be improved a little bit, but not significantly when running without the time measurement code.

9.3.3 Final Experiment Results

Figure 9.5 gives the results in the first series of experiments where all the modules (load generator, "webgate" and Java server) are on the same machine – the centralized system case.

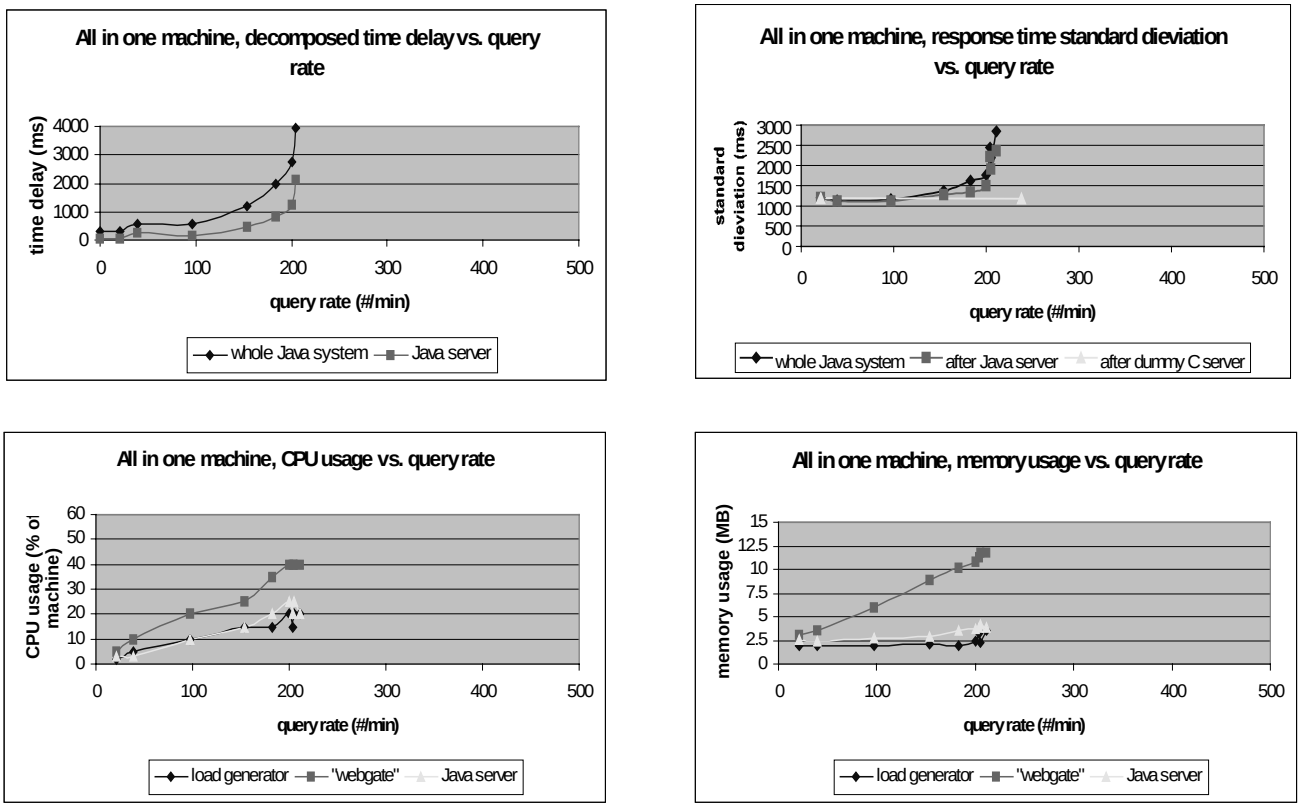

Figure 9.5: All modules in one machine, performance graphs

These graphs suggest that the system workload has an upper limit of 200 queries/min. In this case, most of the response time delay and standard deviation are caused by “webgate” and “formit”. The “webgate” also consumes larger amounts of memory and CPU power than other modules.

Note that since all the modules are running on the same machine, they are competing with each other in terms of CPU and memory usage. Since “webgate” uses the majority of the CPU power when the system reaches its upper limit of performance, the time it reports about the time delay caused by the Java server may not be correct.

Figure 9.6 shows the result when the load generator and “webgate” are running on one machine while the Java server is running on another.

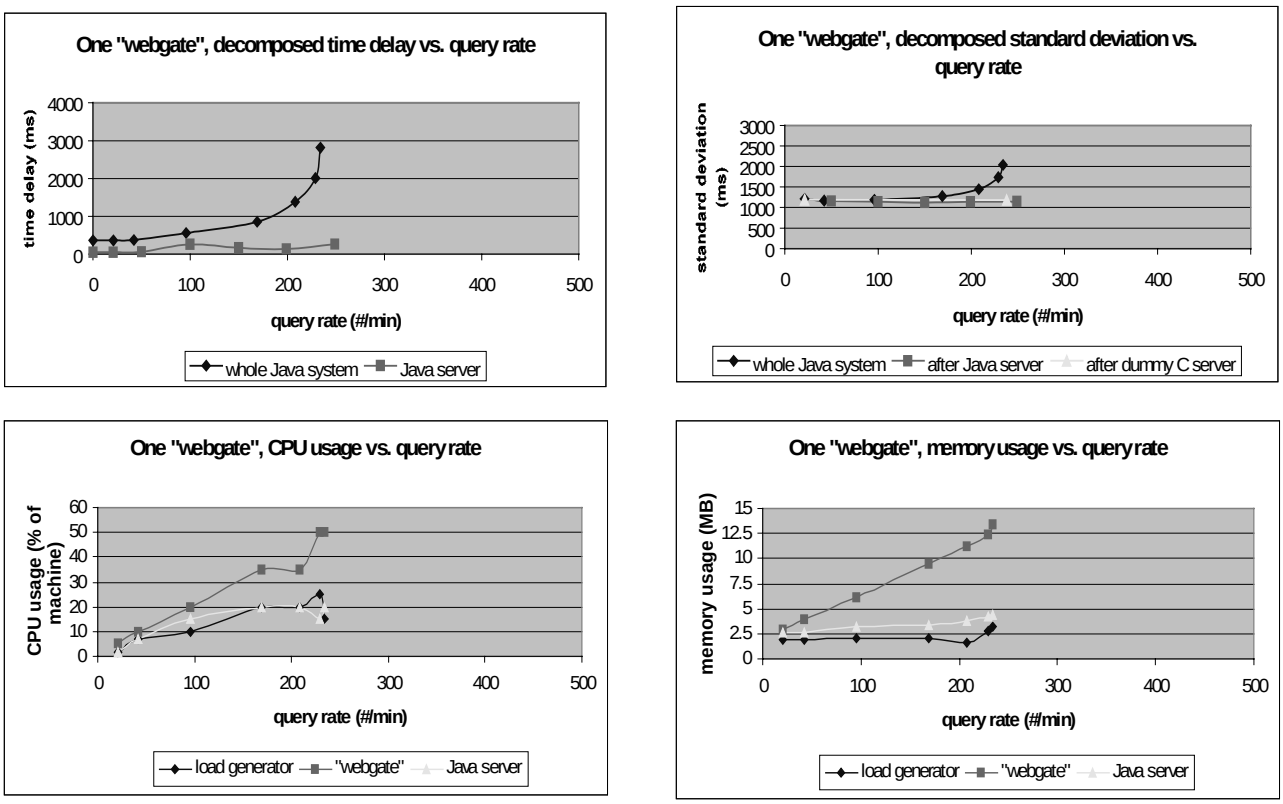

Figure 9.6: One “webgate”, performance graphs

We can see from the above graphs that the upper limit of the system has been raised to about 240 queries/min and the response time has been reduced from 4000ms to less than 3000ms. The majority of the delay and deviation are again caused by “webgate”. “Webgate” also consumes much more memory and CPU power than other modules.

Figure 9.7 shows the experiment results for the case where computation and disk operation are distributed to three machines by having two load generator “webgate” pairs running on two machines communicate with the Java server, which is running on a third one.

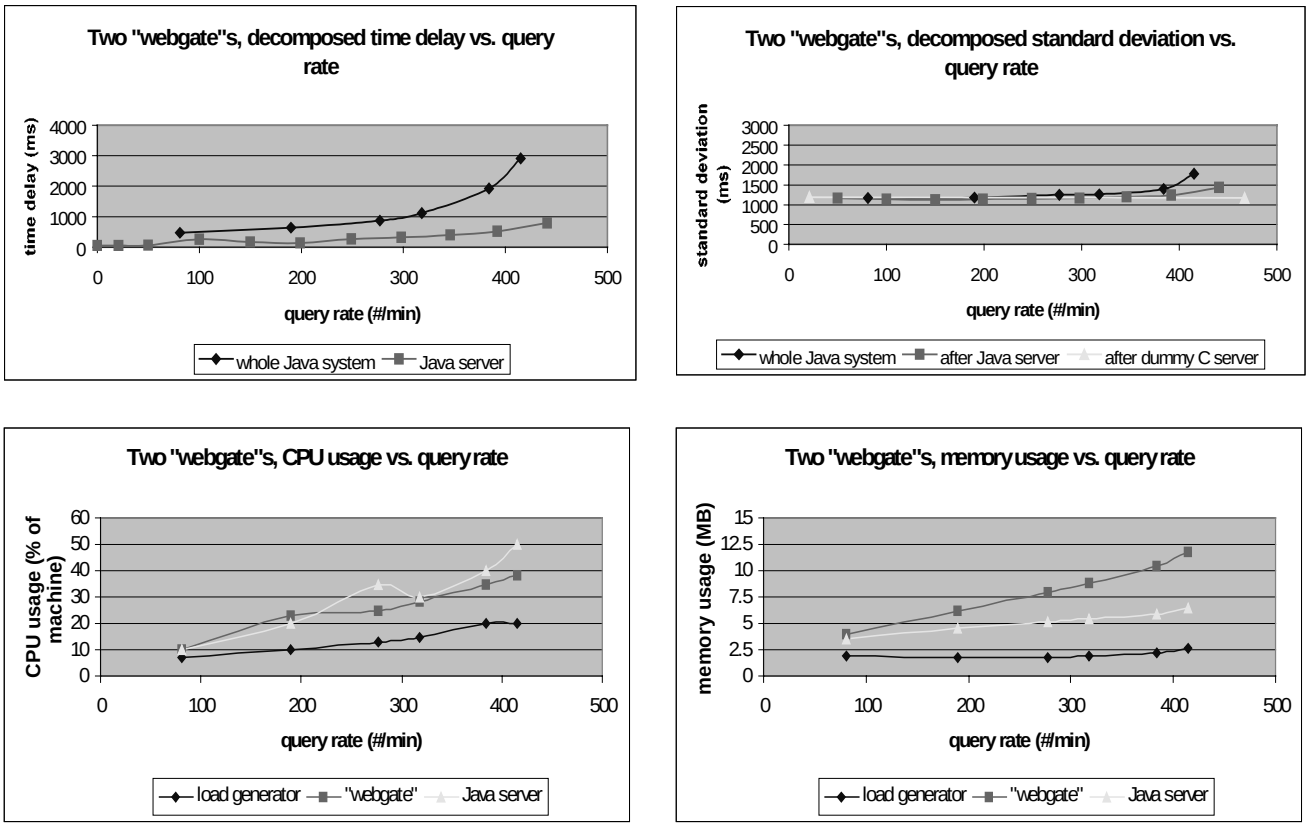

Figure 9.7: Two “webgates”, performance graphs

Much better performance has been achieved in this case. The response time is less than 3000 ms when the workload reaches over 400 queries/min. Though the time delay caused by the “webgate” is still much larger than that caused by the Java server, a higher percentage of time is spent in the Java server. The standard deviation caused by the “webgate” is comparable to that caused by the Java server. The CPU usage of the “webgate” and the Java server are similar. “Webgate” still consumes much more memory than the Java server does.

Figure 9.8 gives the results of the case where four “webgates” are involved.

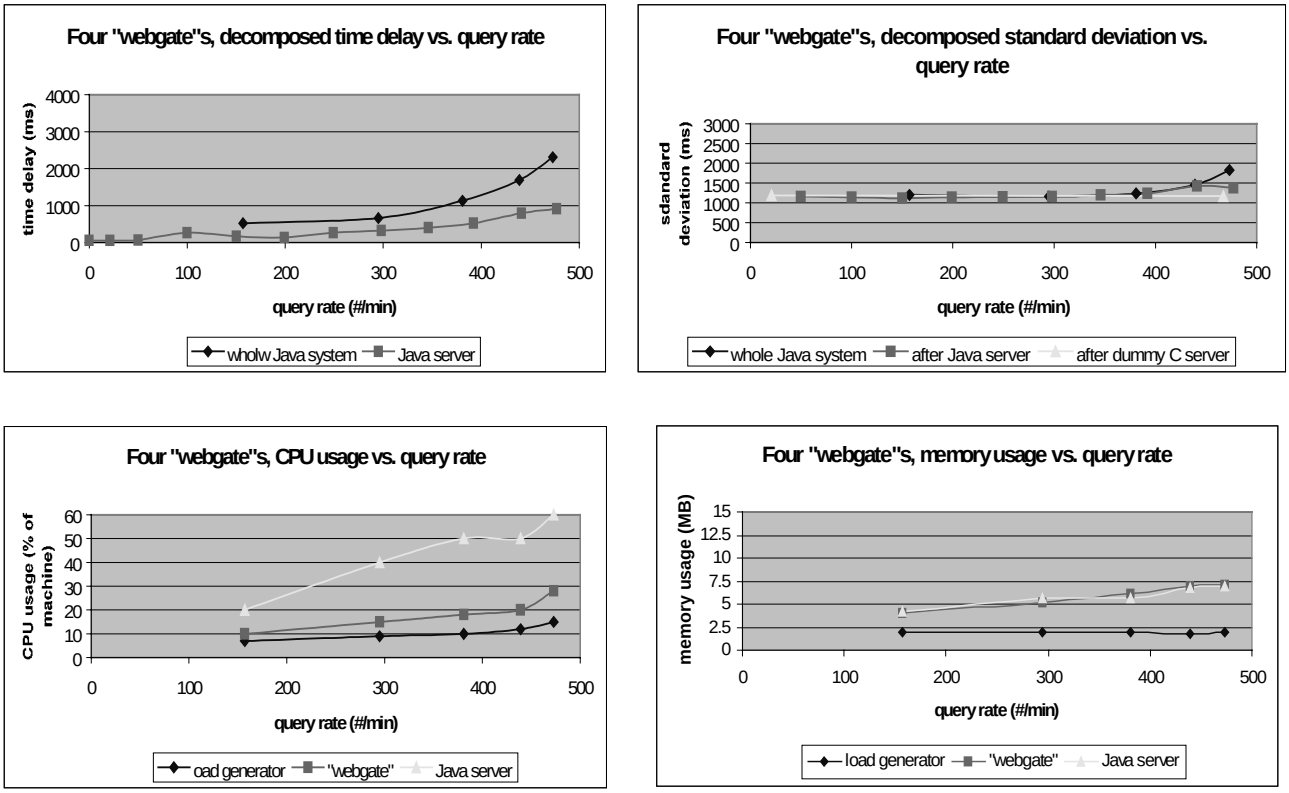

Figure 9.8 Four “webgates”, performance graphs

From the above graphs we can see the performance of the system is improved further. The average response time is less than 2500ms when the workload reached 473 queries/min. The delay caused by the “webgate” is comparable to that caused by the Java server. The “webgate” values exhibit small standard deviation until the query rate reaches 450/min, after which the standard deviation caused by the “webgate” is comparable to that caused by the Java server. In this case the CPU usage of the Java server is much higher than that of the “webgate” while the memory usage of the two are similar. Based on these results, we guessed that the CPU usage of the Java server becomes the bottleneck of the system in this case. To verify this, we used the dummy Java server to replace the real Java server. Since the dummy Java server consumes trivial CPU power, this simulated a very fast CPU. Then we achieved performance of nearly 800 queries/min with average delay less than 2500ms. This verified that our guess is correct – the CPU power of the Java server is the bottleneck of the system in the case of four “webgates”.

Figure 9.9 summarizes the system performance results from all the experiment cases.

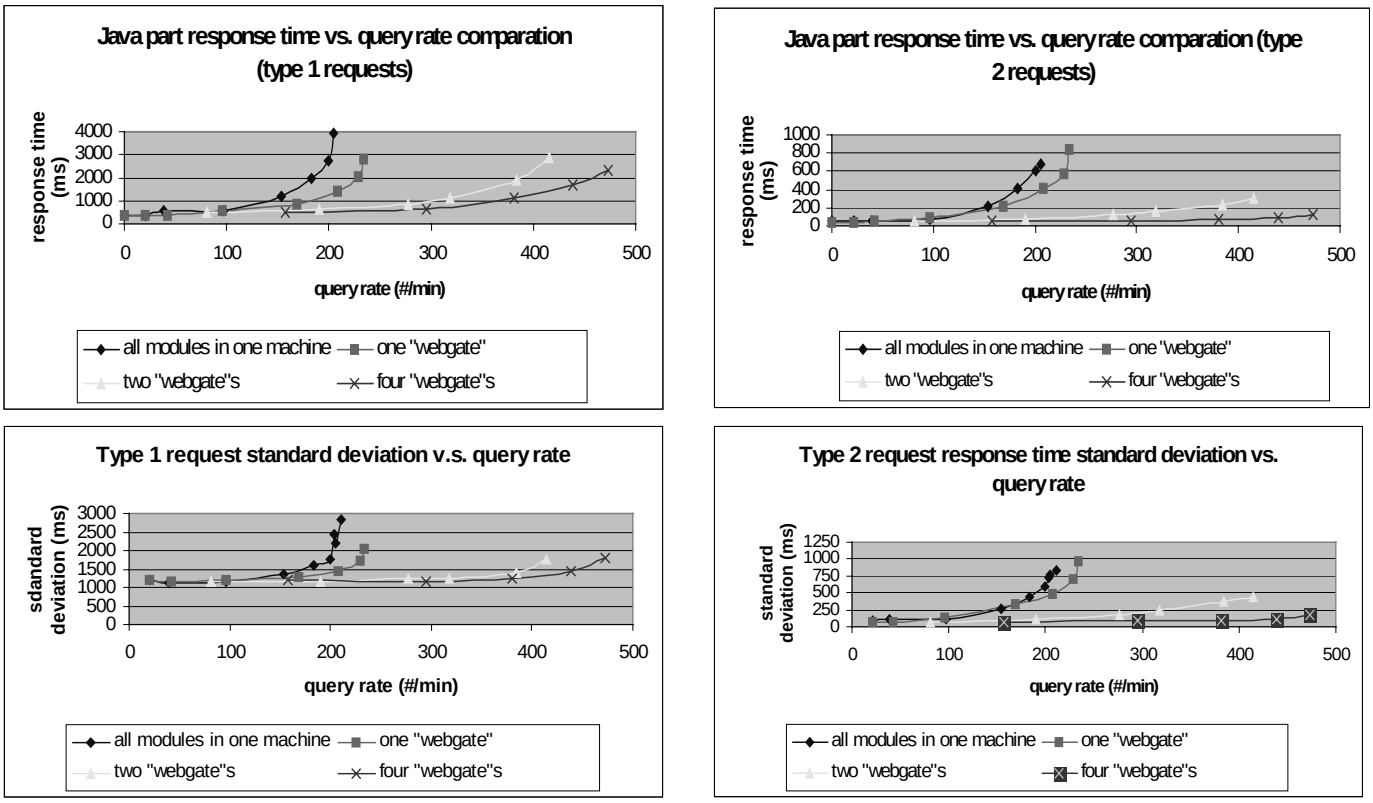

Figure 9.9: Performance comparison graphs

Based on the results in the four experiment cases, when the system's workload is less than 200 queries/min, using one "webgate" the delay caused by the whole Java system will be less than 1200ms. When the workload reaches 400 queries/min, using two "webgates" will result in 2300ms delay in the Java part while four "webgates" can reduce the delay to 1300ms. Type 2 requests always take the system much shorter time to process but exhibit a similar trend. Table 9.2 shows the system configuration we recommended for various workloads along with the corresponding performance.

Table 9.2: System configuration recommendations

workload (queries/min)	recommended configuration	type 1 request Java system average time delay (ms)	type 2 request Java system average time delay (ms)
Less than 140	All modules on one machine	Less than 1000	Less than 160
140 - 185	"Formit" and "webgate" on one machine, Java server on another	Less than 1000	Less than 260
185 - 310	Two "formit" "webgate" pairs	Less than 1000	Less than 160
310 - 425	Four "formit" "webgate" pairs	Less than 1500	Less than 90

Note: Above table assumes machines used are Pentium II 233 MHz with 64 MB memory, and multiple machines are connected to the same 10 Mbps switched ethernet. When query rate reaches over 425/min, four or more "formit"--"webgate" pairs should be used, the Java server should either be duplicated or run on a machine faster than Pentium II 233 MHz.

9.4 Summary

9.4.1 Scalability

The results of all the experiments show the system is scalable. It achieves higher performance when the computation and disk operation are distributed to multiple machines and the network connection was used efficiently to reduce communication cost between modules. By reducing the data transmitted between "webgate" and the load generator, also by changing the communication between the two modules from TCP to UDP, over 20 times performance improvement has been achieved. When we changed from all the modules in one machine to having four "webgates"

running on four machines, the performance was improved from 4000 ms response time for a workload of 210 queries/min to less than 2500 ms for a workload of 473 queries/min. The reason is that in the later case the system is consuming the computing power of five CPUs (four "webgates" and one Java server) and writing query data to four hard disks in parallel.

9.4.2 Reliability

No bugs were found in the Java MARIAN server in all the experiments. This demonstrates the high reliability of this subsystem. We conclude that the reliability was the result of the processes we followed when developing it. The Java "webgate" verified this from another point of view.

Both the Java "webgate" and the Java MARIAN server are of the same size – containing about 20,000 lines of code. The Java "webgate" was developed not following the processes used in developing the Java MARIAN server. There was no detailed design and detailed design review processes. There also was no synchronization checking process. Code reviews were only performed by the person who wrote the code. As a result, more than 100 bugs were found in unit testing (the comparable number was 11 in the Java MARIAN server). Because of this, the unit testing took several weeks' time. Even after that, bugs were found from time to time and usually took hours to locate and fix.

Before the experiments, we did a code review and applied the synchronization checking process on the code of the Java "webgate". More than 10 bugs were found and fixed as a result. After that, only one bug was found in the "webgate" during the experiments and this bug cost us about 10 hours to locate and fix.

All these show that different processes result in systems of different reliability. The high reliability of the Java MARIAN server was mainly due to the good processes followed when developing it.

9.4.3 Flexibility

Though not one of the original areas of investigation considered in the experiments, the results nevertheless show that the Java system is very flexible. Due to the weak coupling maintained throughout the system design, the four modifications made to remove system bottlenecks took less than 14 man-hours to make and test.

Chapter 10

Conclusions and Future Directions

10.1 Conclusions

This thesis investigates the feasibility of making digital libraries flexible, scalable, and reliable through the project of Reengineering the MARIAN System in Java.

Results of experiments and the project show that our approaches to achieve flexibility, scalability, and reliability are beneficial. Better flexibility can be achieved by adding the user information layer and maintaining weak coupling in system design. Good scalability can be achieved by optimizing the usage of network connections and facilitating distribution of computation and disk operations in system design. Reliability can be achieved by applying good software processes in the development of the project.

These approaches may be used to guide the design and implementation of future digital library systems.

10.2 Limitations and Future Directions

With the addition of the user information layer, more personalized services can be provided by a digital library system. This means a lot more user oriented log data can be obtained. Future work

needs to be done on effectively analyzing those data, and using the conclusions reached to direct future design of digital library systems to make them more flexible to their users.

Additional scalability improvements might be possible by further separating the “formit” from the “webgate” in our system. We achieved 2 - 3 times performance improvements when we did that in our experiments. However, since UDP was used for the communication between modules on different machines, result documents for some queries didn’t come back due to the packet loss over the network. Further study needs to be done to make use of TCP’s error handling as well as UDP’s fast speed. In our case, it might help to have a “connector” module running on the same machine with the “formit”. It communicates with the “formit” through UDP and at the same time maintains persistent TCP connections with the “webgate”. Future study needs to be done to verify this approach.

In our experiments, the load generators generated equal workload for different “webgates”. Achieving load balancing may not be as easy in the real world. Further study needs to be done in designing smart load balancing algorithms/protocols so that the system can automatically balance the load on multiple modules and thus achieve the best scalability possible.

The results of experiments verified that system performance was improved significantly when having “webgates” run on different machines with the Java server in the same ethernet. In theory, the performance of the system also can be improved by distributing “webgates” to remote user groups since persistent TCP connections are maintained between “webgates” and the Java server, and all the requests share these connections. Further study needs to be done to verify this.

Finally, good software processes and document formats need to be generalized and made easily accessible to students in the Computer Science Department so that they can benefit from them in future project development. We hope that our Web site (http://video.dlib.vt.edu:90/marian/marian.html) will be extended to support these goals.

In conclusion, we hope that our efforts may directly guide subsequent work on the MARIAN system, and also may be of interest regarding other digital library development projects.

Bibliography

[1] Bruce Schatz and Hsinchun Chen, "Building Large-Scale Digital Libraries", *Computer* theme issue on the US Digital Library Initiative, Vol 29, No 5, May 1996
http://computer.org/computer/dli/

[2] Borgman, C. L., "Why are online catalogs still hard to use? Lessons learned from information retrieval studies", *JASIS* 47, 6 (July 1996), 493-503

[3] Andreas Paepcke, Steve B. Cousins, Hector Garcia-Molina, Scott W. Hassan, Steven P. Ketchpel, Martin Roscheisen, and Terry Winograd, "Using Distributed Objects for Digital Library Interoperability", *Computer* theme issue on the US Digital Library Initiative, Vol 29, No 5, May 1996
http://computer.org/computer/dli/r50061/r50061.htm

[4] Daniel E. Atkins, William P. Birmingham, Edmund H. Durfee, Eric J. Glover, Tracy Mullen, Elke A. Rundensteiner, Elliot Soloway, Jose M. Vidal, Raven Wallace, and Michael P. Wellman, "Toward Inquiry-Based Education Through Interacting Software Agents", *Computer* theme issue on the US Digital Library Initiative, Vol 29, No 5, May 1996
http://computer.org/computer/dli/r50069/r50069.htm

[5] Bruce Schatz, William H. Mischo, Timothy W. Cole, Joseph B. Hardin, Ann P. Bishop and Hsinchun Chen, "Federating Diverse Collections of Scientific Literature *Computer* theme issue on the US Digital Library Initiative, Vol 29, No 5, May 1996
http://computer.org/computer/dli/r50028/r50028.htm

[6] Terence R. Smith, "A Digital Library for Geographically Referenced Materials", *Computer* theme issue on the US Digital Library Initiative, Vol 29, No 5, May 1996
http://computer.org/computer/dli/r50054/r50054.htm

[7] "Dienst protocol version 4.1 Draft", *NCSTRL Documentation,* February 1998
http://www.cs.cornell.edu/NCSTRL/protocol.html

[8] R. Fielding , J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", *Network Working Group, Request for Comments: 2068,* January 1997
http://www.ietf.org/rfc/rfc2068.txt

[9] R. Braden, RFC: 1122, "Requirements for Internet Hosts -- Communication Layers", Internet Engineering Task Force, October 1989
http://www.ietf.org/rfc/rfc1122.txt

[10] Information Sciences Institute, University of Southern California, RFC: 793, "Transmission Control Protocol", *Darpa Internet Program, Protocol Specification,* September 1981
http://www.ietf.org/rfc/rfc0793.txt

[11] J. Postel, RFC: 768, "User Datagram Protocol", ISI, August 1980
http://www.ietf.org/rfc/rfc0768.txt

[12] Arthur Van Haff, Sami Shaio, and Orca Starbuck, "Hooked on Java™", Sun Microsystems, Inc., Addison-Wesley Publishing Company, February 1996

[13] Sun Microsystems, Inc, "Java™ Platform 1.1 Core API Specification",
http://java.sun.com/products/jdk/1.1/docs/api/packages.html

[14] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis, "The Capability Maturity Model: Guidelines for Improving the Software Process" Carnegie Mellon University, Software Engineering Institute, Addison-Wesley Publishing Company, 1995

[15] Robert K. France, Lucy Terry Nowell, Edward A. Fox, Roni A. Saad, and Jianxin Zhao, "Use and Usability in a Digital Library Search System", *Unpublished Report*, Jan. 1999, Virginia Polytechnic Institute and State University, Blacksburg, VA

[16] Fox E. A., France, R. K., Sable, E. Daoud, A. and Cline, B. E., "Development of a Modern OPAC: From REVTOLC to MARIAN", *Proc. ACM SIGIR-93 16th Int'l Conference on R&D in Inforrmation Retrieval*, Pittsburgh, PA, 248-259

[17] Borgman, C. L., "Why are online catalogs hard to use? Lessons learned from information retrieval studies", *JASIS* 37, 6 (Nov. 1986), 387-400

[18] N. N. Mitev, "Human computer interaction and online catalogues". In *OPACs and Beyond, Proceedings of a Joint Meeting of the British Library, DBMIST, and OCLC*, 1988

[19] Robert K. France, Edward A. Fox, "Indexing large collections of small text records for ranked retrieval", *Unpublished Report*, 1993, Virginia Polytechnic Institute and State University, Blacksburg, VA
http://www.dlib.vt.edu/reports/LargeCollsSmTexts.pdf

[20] "MARC Documentation", *Princeton University Library's Cataloging Documentation*
http://infoshare1.princeton.edu/katmandu/marc/marctoc.html

[21] R. Srinivasan, RFC: 1832, "XDR: External Data Representation Standard", Sun Microsystems, August 1995
http://www.ietf.org/rfc/rfc1832.txt

[22] Z39.50-1995 Maintenance Agency, Z39.50 – 1995 Protocol Specification, Library of Congress, March 1996
http://lcweb.loc.gov/z3950/agency/19995doce.html

[23] Nicholas J.Belkin, and W. BruceCroft; "Information filtering and information retrieval: two sides of the same coin?" *Commun.* ACM 35, 12 (Dec. 1992), pages 29 - 38
http://www.acm.org/pubs/articles/journals/cacm/1992-35-12/p29-belkin/p29-belkin.pdf

[24] Hartson, H. R. & Castillo, J. C. (1998). "Remote evaluation for post-deployment usability improvement". Invited paper in Proceedings of AVI '98 (Advanced Visual Interfaces) International Working Conference, May 25-27, 1998, L'Aquila, Italy, in cooperation with: ACM-SIGCHI, ACM-SIGMM, and SIGCHI Italy, 22-29.

[25] "Home on the Web", *PC Magazine* Vol. 17, No. 15, (Sep. 1, 1998)
http://www.zdnet.com/pcmag/features/webportals/index.html

[26] Ben Cline, "User Interaction Protocol – Implementation Guide, Version 1.0", Aug. 1993, Unpublished Internal Communique

[27] Binzhang Liu, "Characterizing Web response Time", MS Thesis, Virginia Polytechnic Institute and State University Department of Computer Science, April 1998
http://www.cs.vt.edu/~chitra/docs/nrgpub/thesis.pdf

[28] Watts S. Humphrey, "Using a Defined and Measured Personal Software Process", *IEEE Software, Vol. 13, No. 3, May 1996,* 77-88

Vita

Jianxin Zhao was born on June 28, 1972 in Liaoning Province, People's Republic of China. He earned a B.E. degree in the Computer Engineering Department at Beijing University of Posts and Telecommunications in 1994. His defense topic for the degree is "Voice Mail System on Microcomputer".

Mr. Jianxin Zhao worked as a software engineer in the China Academy of Telecommunication Technology from 1994 to 1997.

Jianxin Zhao started his graduate study in the Computer Science Department of Virginia Polytechnic Institute and State University in August 1997. He has completed his M.S. degree requirements in July 1999.