THREE-TIER ARCHITECTURE FOR SENTINEL APPLICATIONS AND TOOLS: SEPARATING PRESENTATION FROM FUNCTIONALITY

By

SANG-WOO HAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1997

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this interesting topic and for providing me great guidance and support through the course of this research work. I am also thankful to Dr. Eric Hanson and Dr. Herman Lam for serving on my committee.

I would like to express my special thanks to Sharon Grant for maintaining a well administered research environment and being so helpful in times of need. Sincere appreciation is due to Hyoungjin Kim for his invaluable help during the implementation of this work. I also would like to thank all my friends for their constant support and encouragement.

I would like to thank the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center, RDT&E Division, for supporting this work.

Last, but not the least, I thank my family for their endless love and support. My mother and father and my brother Dong-Ho give me unlimited amounts of patience, support, and extraordinary encouragement to get through difficult moments.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ii

LIST OF FIGURES v

ABSTRACT vi

CHAPTERS 1

1 INTRODUCTION 1

1.1 Current Architecture–Two-Tier Structure 2
1.2 Limitations of Two-Tier Structure 3
1.3 WWW Browser for an Alternative Interface 4
1.4 A New Approach 5

2 OVERVIEW OF CURRENT SENTINEL APPLICATION 7

2.1 MDP Application with Motif 7
2.2 MDP Application with CGI 9
2.2.1 Common Gateway Interface 9
2.2.2 CGI MDP–WWW Form-based Application with CGI 11
2.3 Dynamic Rule Editor with Tcl/Tk 12

3 THREE-TIER ARCHITECTURE FOR SENTINEL APPLICATIONS . . 14

3.1 Three-Tier Architecture 14
3.2 Sentinel Application with Three-Tier Architecture 17
3.2.1 Java Interface 17
3.2.2 Proxy Server 18
3.2.3 DMDP–WWW Browser-based Application with Applet 18

4 IMPLEMENTATION 20

4.1 The Use of Java 20
4.2 Java Interactive Visualization 21
4.2.1 Limitations and Related Work 25
4.2.2 Implementation Issues 25
4.2.3 Interface Details 30
4.3 Distributed MDP 32
4.3.1 Implementation Issues 35
4.3.2 Interface Details 37

5 CONCLUSION AND FUTURE WORK 41

5.1 Conclusion 41
5.2 Future Work 41
5.2.1 Multi-Tier Architecture 41
5.2.2 Distributed Object Architecture 42

APPENDIX : SAMPLE STATIC/DYNAMIC LOG FILES 43

REFERENCES 45

BIOGRAPHICAL SKETCH 46

LIST OF FIGURES

1.1 Monolithic Structure vs. Two-Tier Structure 2

2.1 MDP Layout 8

2.2 CGI MDP Layout 10

2.3 CGI Interface Model 12

3.1 Three-Tier Structure vs. Two-Tier Structure 15

3.2 Network connection of Three-Tier Structure 16

3.3 Network connection of Two-Tier Structure 16

3.4 Proxy Server in Three-Tier Structure 18

3.5 3-Tier Structure with Proxy in WWW environment 19

4.1 Functional Modules of Interactive Visualization Tool 24

4.2 Processing of Static Information in Java Interactive Visualization Tool 26

4.3 Layout of Java Interactive Visualization Tool 28

4.4 AWT Components of Interactive Visualization Tool Interface 31

4.5 AWT Components Hierarchy of Interactive Visualization Tool Interface 31

4.6 Functional Modules of DMDP. 33

4.7 Original Application vs. Web-based Application 36

4.8 Layout of Monitoring Desk 39

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

THREE-TIER ARCHITECTURE
FOR SENTINEL APPLICATIONS AND TOOLS :
SEPARATING PRESENTATION FROM FUNCTIONALITY

By

Sang-Woo Han

December 1997

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

During the last few decades, graphic user interfaces (GUIs) have evolved considerably to meet the demanding requirements of the application domains. One of the major requirements of the GUI was the use of graphics to communicate information to the users visually. Typically, a GUI presents a finite number of options to the users rather than requiring the users to memorize and manually enter commands. In this way, the GUI has been developed to put more emphasis on the needs of human beings than on computers.

The advent of these intuitive and easy-to-use design elements does not, however, ensure that applications incorporated with GUIs can be easily extended or redesigned. Historically, the implementation of GUIs has been coupled with applications and hence forms an indistinguishable entity. The complete integration of application logic and presentation makes it very difficult to extend.

With the phenomenal growth of networks, now developers also have to think about distributed applications. Applications need to be able to migrate easily to a wide variety of platforms and operating systems. In the conventional design, the implementation of the interface is coupled with architecture specific codes of an application. Developing for a specific target platform becomes unmanageable since it requires deep knowledge of the different platforms. The growth of the World Wide Web, although complex, distributed, and heterogeneous in nature, has made it an alternative popular user interface for interactive applications.

A new approach, a 3-tier architecture proposed in this thesis, supports separation of the three basic database application elements: presentation (user interface), functionality (the engine), and data. This clear separation gives flexibility and independence in application design which provide open choices about each tier and better CPU utilization and cost. The choice of an architecture-neutral and portable programming language for the user interface provides a simple solution to the problem of distributing applications across heterogeneous network-based computing platforms. The new design makes it simpler to manage applications and support modifications. Moreover, the new architecture makes applications suitable for use in the WWW environments with the use of Java.

This thesis concentrates on the design and implementation of the 3-tier architecture for application design in which we try to solve the problems of the conventional integrated architecture with an emphasis on the development of the Interactive Visualization Tool and the Plan Monitoring Application. We discuss various issues associated with the application of the 3-tier approach to the Sentinel application and tool.

CHAPTER 1
INTRODUCTION

The term user-interface refers to the method and devices that are used to accommodate interaction between machines and human beings who use them (users). User interface can take on many forms but it is always expected to accomplish the following fundamental tasks: communicating information from the machine to the user, and communicating information from the user to the machine. With efforts of following the principle, user interfaces entered the modern era when innovative designers at the Xerox Palo Alto Research Center broke away from the character-based interface paradigm and invented the Graphic User Interface [1] (GUI). This new paradigm significantly reduced the training that was necessary to use a computer, and for the first time uninitiated users were able to become productive almost immediately. In workstation computers, X windowing environment with the Motif standard began to demonstrate that easy to use, graphics interfaces could be wrapped around the majority of scientific and engineering software that already existed. These archaic FORTRAN, COBOL, ADA applications suddenly had a re-birth. They became easy to use. Suddenly major computer companies were seeking high dollar defense contracts to place shiny front ends around other companies executables. The X/Motif became the standard for workstation GUIs.

So where is all this historical information taking us? One gigantic collision course. The implementation of the Motif needs to be closely integrated with applications and hence functional core of the application is not easily recognizable or understandable. As the applications evolve over a period of time, this design makes it difficult to differentiate between the interface and the programming structure. Redesign or

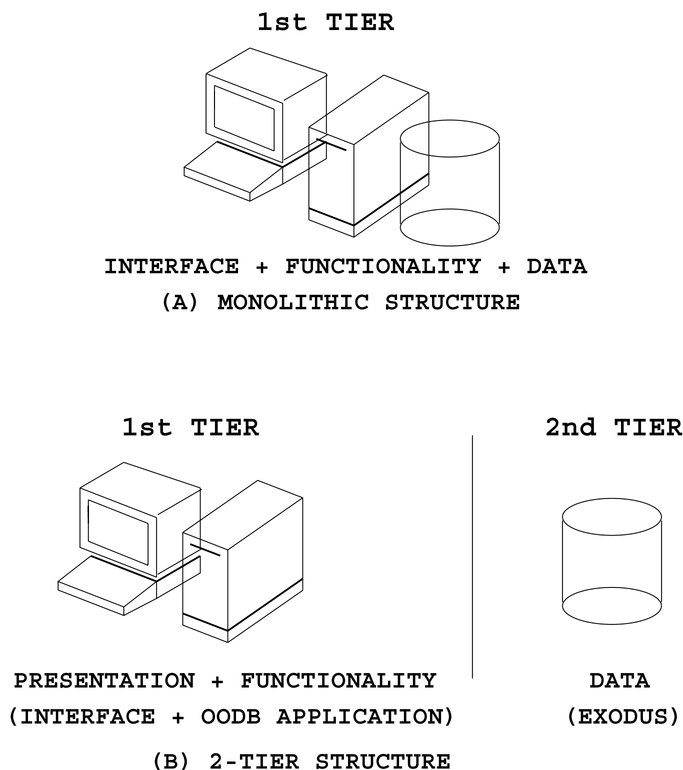


Figure 1.1. Monolithic Structure vs. Two-Tier Structure

modification process can be complex and challenging. Sometimes, it is impossible to break down the systems into manageable pieces.

1.1 Current Architecture–Two-Tier Structure

The traditional, or monolithic, application illustrated in Figure 1.1 A, does not separate user interface from its functionality or its data. All three individual strands are woven together like a bowl of spaghetti. Usually these applications are hand-tooled and work as one piece. Parts are not interchangeable and modifications are often difficult and time-consuming. An analogy would be to the time before interchangeable components, when most things were hand-crafted. if a part broke, one had to find a craftsman that could hand-craft a unique replacement. This caused a significant drag in productivity. The onset of interchangeable components caused an explosive rise in industrial productivity.

Two-tier client/server architecture, the current architecture of Sentinel application, begin to disassemble this “spaghetti” structure. In this category, data are decoupled from the presentation/functionality piece but functionality is still entwined with another element. This architecture is an improvement over the monolithic case described above, but it is not as open as it could be and is still likely to result in same problem.

1.2 Limitations of Two-Tier Structure

In order to understand the limitations of the current design, one must look inside the code and recognize that there are three inherently different components for any database application. These components are user interface, application logic, and data access. In current Sentinel applications, two-tier structure, all three or two components of the application are intertwined, built as a single entity, with no modularity to separate them. One tier contains all the application logic and the other tier contains the data server.

Applications built with a two-tier architecture do not tend to scale well. Large-scale applications, which usually require complex processing, high transaction volumes, and frequent maintenance typically “hit the wall” when a two-tier architecture is employed. First, there exists the complexity wall. It refers to the complexity of the application, measured by complex algorithms, large number of objects, or large number of functions. In most two-tier applications, the user interface logic, the application logic, and the database access logic are intertwined, not separated and modular. The potential for code reuse of the kind that a multi-tier, object-oriented system provides is largely eliminated, and as the complexity and size of the application grows, the difficulty of keeping it bug-free grows as well.

Next, supportability wall will hold back the application from flexible maintenance. Supportability is the problem common to the two-tier application architecture. As

stated above, each tier is intertwined, inexorably linked together. Suppose we had a customer service application that was built using a two-tier approach. If the management determines that customers should be allowed to service themselves by having Web-based interface or Voice response-based interface, the entire application may have to be re-written to support Voice response-based system or Web-based system. The development group faces the burden of supporting two code bases for one application. Worse, this two-tier approach which is a typically client/server structure, can cause supportability problems on the other side of the application, data access side, as well. What would happen if some new requirements were introduced that necessitated the move from the current relational DBMS to an object-oriented DBMS? If the data access logic is intimately intertwined with the application logic, major changes to the application are likely to be necessary to support a later migration to a relational database.

1.3 WWW Browser for an Alternative Interface

WWW (World Wide Web), various browsers and Java language were the major factors that transformed the Internet and made it possible for users to participate in this global network. Today programs like Netscape Navigator and Internet Explorer have further enhanced Web browsing. Numerous search engines have eased the task of finding information. It has been within the past three years that the Internet has seen its greatest expansion to date, and it is still expanding at a high rate today. This phenomenal growth of Internet persuaded many developers to use WWW browsers as their alternative user interface for various kinds of applications because WWW is accessible for many different platforms and even unexperienced user can use a WWW-based interface.

Common Gateway Interface [2] (CGI) and HTML Forms can be used to provide WWW as an alternative interface to users. Since WWW is another major

client/server system, with probably millions of users, interface between clients and server has to be developed. They talk to each other in a WWW protocol specification called HTTP [3] (HyperText Transfer Protocol). HTML (HyperText Markup Language) format is used for interfacing between the clients and the Web server and forms were introduced into HTML to allow interaction to occur from the browser to the server. It allows dynamic documents to be created in the server in response to browser information by CGI.

This WWW forms-based application using CGI runs on the server side in response to a request from the browser. This request specifies the script to run and makes available form information accessible through environment variables and the script's standard input. Application still has to re-engineered for use with CGI. CGI suffers from performance delay since the request has to go through server side HTTP protocol and CGI handler. CGI-MDP is an example of re-engineered application and is described in more detail in chapter 2. It is obvious that there exists another requirement for application developers that applications need to be used with Web across different platforms with high performance.

1.4 A New Approach

To solve the problems mentioned in previous sections, we need a better way to construct applications where the application logic should be expanded easily and long-term software maintenance has to be manageable with well-defined boundaries of application. Moreover, new applications need to run on the Web to provide cross-platform usability. These are the new requirements for the solution to the current design.

The remainder of this thesis is structured as follows. Chapter 2 overviews current Sentinel applications and describes their limitations and disadvantages. Chapter 3 proposes a solution to the current design approach and describes its advantages

and how it is applied in the design of two Sentinel applications. In chapter 4, we discuss design/implementation issues of the new design and how the new approach has been used to implement an Interactive Visualization Tool and a Distributed Plan Monitoring Application. Chapter 5 has conclusion along with future directions.

CHAPTER 2
OVERVIEW OF CURRENT SENTINEL APPLICATION

In this chapter, we overview the current Sentinel applications designed using the two-tier approach and their disadvantages and limitations.

2.1 MDP Application with Motif

First MDP, Plan Monitoring Application, is a single-user application, which means every user runs an instance of MDP locally. In this environment the Motif user interface is tightly coupled and integrated with the system, which is a typical two-tier architecture.

The Sentinel DBMS and the underlying Open OODB are accessed by MDP as libraries. The benefit of this design approach is that the application and the visualization program run in the same address space. Hence data can be readily shared by parameter passing and procedure invocation. The user interacts with the Sentinel system in a simple and asynchronous manner: he/she submits one or many reports at a time and wait for the system to process the messages, detect events and carry out actions according to rules that would fire. Results of actions are sent back to the Motif front end for the user to visualize. Since there are two types of critical situations to monitor (weather related and non-weather related) they must be treated differently to suit user's need.

In this design, non-weather related result messages are shown as plain text. For weather related messages in addition to text output we have arranged a 2-D raster image representing the geographic area we are currently monitoring, which is a partial world map containing most of the Pacific Ocean and its Asian and American coast lines. A region is defined as a rectangular area in the map. Currently there are 10

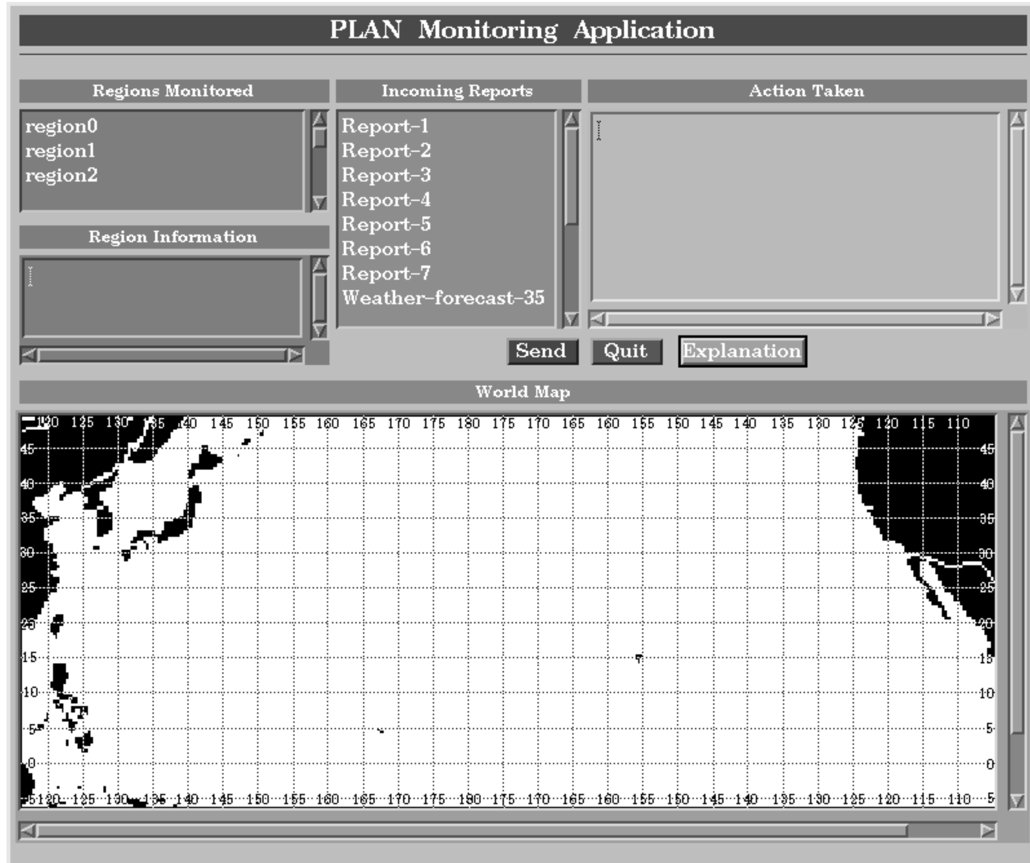


Figure 2.1. MDP Layout

regions of interest. When severe weather change occurs in a region, and an appropriate alert is issued, the region will be highlighted around its border in a eye-catching color. Newly created color rectangle will blink to enhance user attention until a newer update happens. Corresponding to severity of weather, a region can be highlighted in one of three colors : violet, orange, and red representing the weather condition in the ascending order of severity. Figure 2.1 shows the layout of the MDP graphical user interface. It contains a group of subwindows and a set of action buttons.

MDP is a military application utilizing the active capability of the Sentinel DBMS. The first version of MDP is implemented with X/Motif interface. The Motif graphical user interface provides application developers, and users with the industry's most widely accepted environment for standardizing application presentation on a wide

range of platforms. It is the industry standard graphical user interface (as defined by the IEEE 1295 specification), used on more than 200 hardware and software platforms. Motif has been the leading user interface for the UNIX operating system.

However, the extension to multi-user, and distributed application means building the application from the start all over again. The Motif interface code is closely coupled with application logic. Also this Motif version of MDP is architecture specific where running on a different platform means re-writing of a new application. WWW Form-based MDP is one of this example. The application has to be re-engineered for a use with CGI handler.

2.2 MDP Application with CGI

The growth of the Internet has made it is the newest wave of communication for electronic mail, file transfer, telnet access, transaction applications, and much more. Once the WWW concepts and the protocols were placed in the public domain, programmers and software developers around the world began introducing their own modifications and improvements.

2.2.1 Common Gateway Interface

CGI (Common Gateway Interface) is one of the improvements in this field. Standard HTML with CGI gives a reasonable amount of functionality on its own. It can build look and feel of the user interface by placing formatted text and graphics, and it gives a decent set of controls to pass on to the user through forms and their widgets, such as fields, check boxes, and list boxes. Hypertext links turn out to be a reasonable substitute for multiple window manipulation: clicking on a link is like creating a new window, and the “back” and “forward” arrow buttons on most HTML browsers are ways of changing your window focus. Figure 2.2 shows the layout of the CGI-MDP interface.

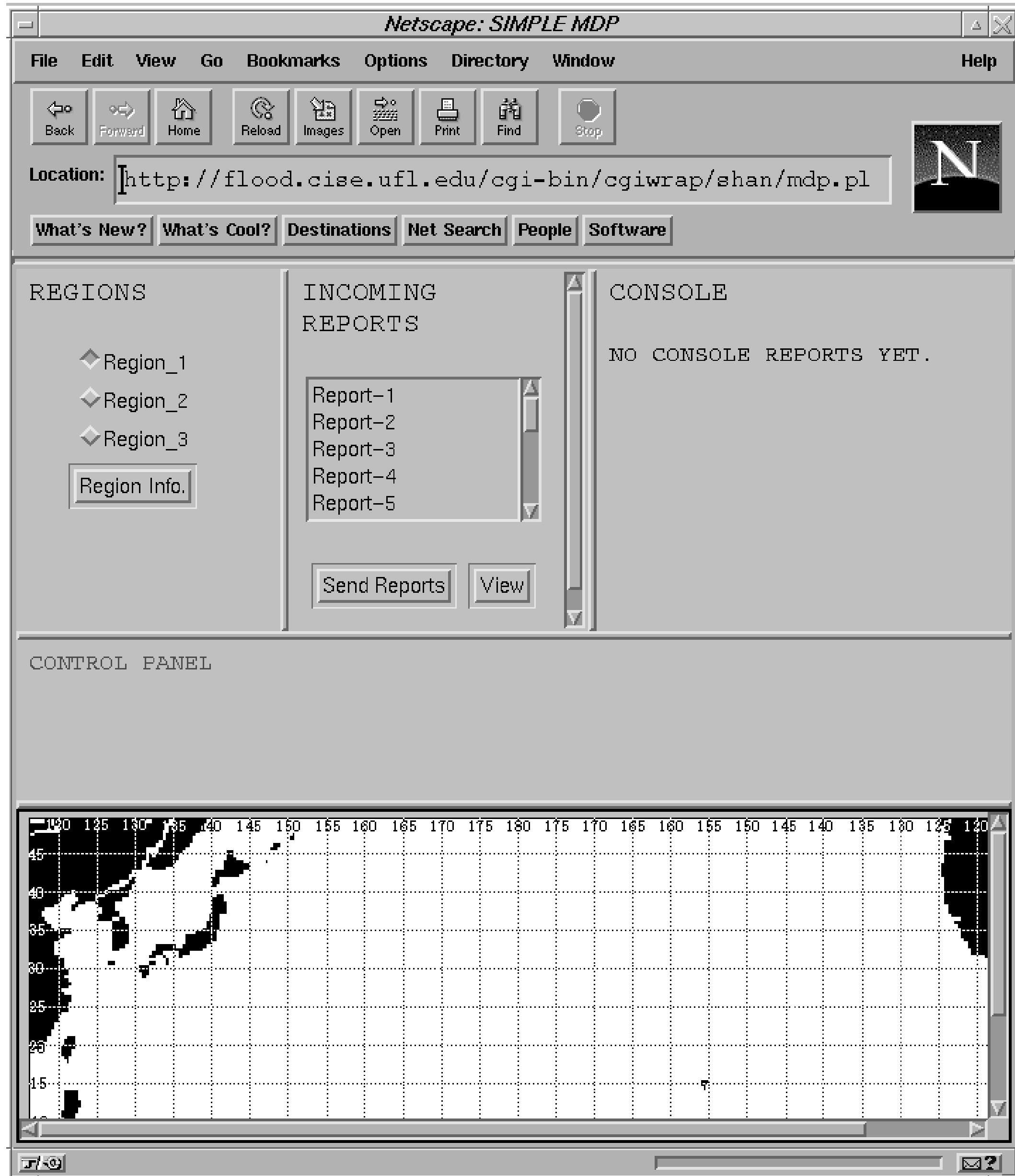


Figure 2.2. CGI MDP Layout

The web architecture gives other important features for client/server development: it includes protocols like HTTP that abstract away many of the grungy details of TCP/IP communication; HTML is a simple language to develop in and best of all, your resulting client code is guaranteed to execute on UNIX, Macintosh, and all flavors of Windows, without any portability issues.

In the server machine, the HTTP server forks a CGI handler which in turn forks a new session process at initialization time. For each transaction it makes a connection to the running session process based on the state key information contained in the HTTP request. With CGI the entire HTTP request can be recreated, so the handler recreates and sends it onto the session process through the named pipe or the open file descriptor. The handler waits until the session process has completed the transaction, reads the response, forwards it to the HTTP socket and exits.

2.2.2 CGI MDP–WWW Form-based Application with CGI

Using this scheme, MDP is ported to WWW environments. As we mentioned in Chapter 1, the redesign process–separating interface codes from the whole application–took substantial amount of time and effort. Also, the application suffers from the drawbacks of CGI.

The forms/CGI scheme is quite flexible but has two major drawbacks. First, the communication is only one-way. HTML pages cannot receive information back from the server and act on it. Because of this drawback, in the CGI-MDP, application behaves in discontinuous manner. For each request sent, the application waits for a response from the server and application terminates. Every request is one distinct "session" where data has to be loaded every time, and in the DB application this causes low performance.

Next, the approach is semantically weak and requires lots of ad-hoc CGI programming effort to set up correctly. This is because after the control of user's request is

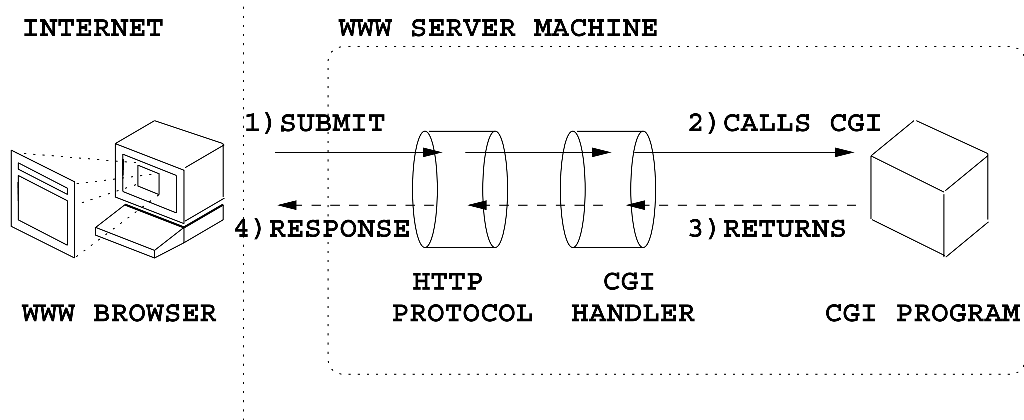


Figure 2.3. CGI Interface Model

sent to CGI, we have to do is to wait for the result. We have no idea about what's happening in the middle of CGI scripts' process. The error messages from the CGI program are somewhat "translated" to WWW browser terms which makes it hard to debug. The error message is distorted in the process between the steps 3 and 4 in the Figure 2.3. Since some of the data exchange is done through the UNIX environment variables in the CGI handler, data loss can happen. In order to prevent this problem, remote shell is used in the CGI-MDP and this caused severe performance delay. Simple diagram of CGI session is shown in Figure 2.3.

2.3 Dynamic Rule Editor with Tcl/Tk

Tcl/Tk [4] is another choice for developing GUI tools for Sentinel. Dynamic Rule Editor [5] has been implemented with Tcl/Tk interface.

Tcl stands for Tool Command language. Tcl is a library package that can be embedded in application programs. The Tcl library consists of a parser for the Tcl language, routines to implement the Tcl built-in commands, and procedures that allow each application to extend Tcl with additional commands specific to that application.

Tcl/Tk is used over Motif since Tcl commands provides a higher-level interface than most standard C library toolkits. Simple user interfaces require just a handful

of commands to define them. However, the X Toolkit Tk is itself tightly coupled with the Tcl language, which has disadvantages such as lack of modularity, and the current interpreter is slow. It is not suitable for migrating applications to WWW environments. Tcl/Tk only provides rapid prototyping of X-Window system applications and has to suffers from the same problems mentioned above for Motif-based applications.

CHAPTER 3
THREE-TIER ARCHITECTURE FOR SENTINEL APPLICATIONS

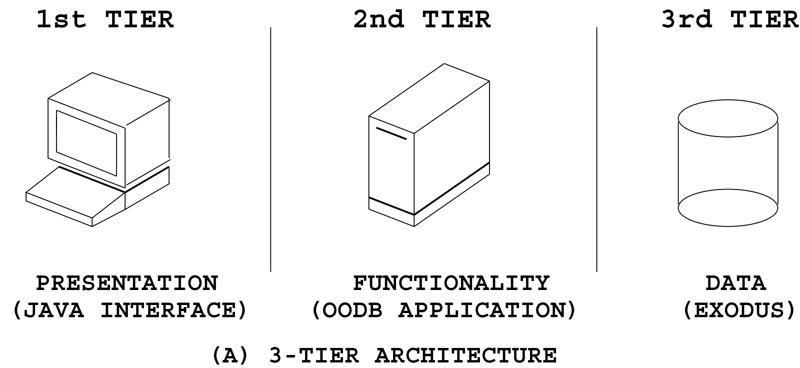
This chapter defines three-tier architecture which is widely accepted and provides a solution to the problems of the previous approach and describes how the new architecture can be applied in the Sentinel application design with Java language environment.

3.1 Three-Tier Architecture

The term client/server architecture is a general description of a networked system where a client program initiates contact with a separate server program (possibly on a different machine) for a specific function or purpose. The client exists in the position of the requester for the service provided by the server.

The new generation of the client/server system, termed three-tier architecture, is an architecture that has clear separation over the application elements. This structure supports separation of the three basic database application elements: presentation (user interface), functionality (the engine), and data. These well-defined boundaries give developers more flexibility and independence in application design which provides open choices about each tier and better CPU utilization and cost. As a result, long-term software maintenance improves resulting in rapid, low cost development environment. Software maintenance can be applied to each manageable entity which makes the process rapid and low cost development possible. Three-tier architecture proposed in this thesis is illustrated in Figure 3.1.

With Java, the new approach provides a way to eliminate lack of client side processing which was a significant weakness in HTML-based CGI application. It also provides many other features, such as security and the ability to run applications



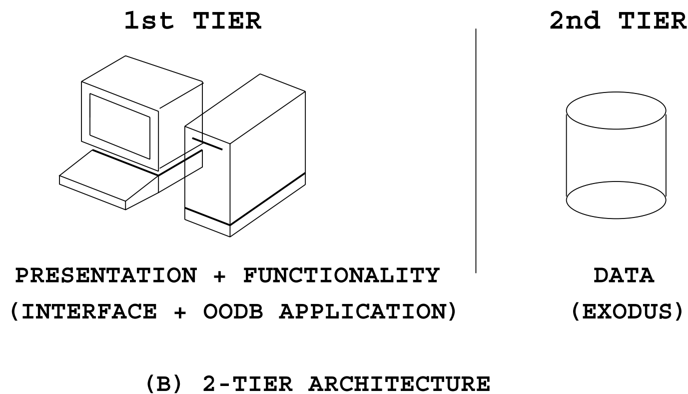


Figure 3.1. Three-Tier Structure vs. Two-Tier Structure

on different platforms without re-compilation, and makes application suitable for use on the Web. In addition, it returns to us the ability to build applications using any number of tiers we require for our application, providing all of the performance, flexibility, and supportability advantages. It allows us to combine all of the advantages of three-tier architecture with all of the great advantages of the WWW.

The network performance is better compared to the two-tier structure. A significant reduction in network footprint and enhancement of performance is possible in the alternative three-tier client/server architecture. For example, a middle tier may be inserted as server, thus achieving a three-tier structure. The client interacts with the middle tier via standard protocol such as API, or RPC. The middle tier interacts with the server via standard database protocols. The middle tier contains most of the application logic, translating client calls into database queries and other actions, and

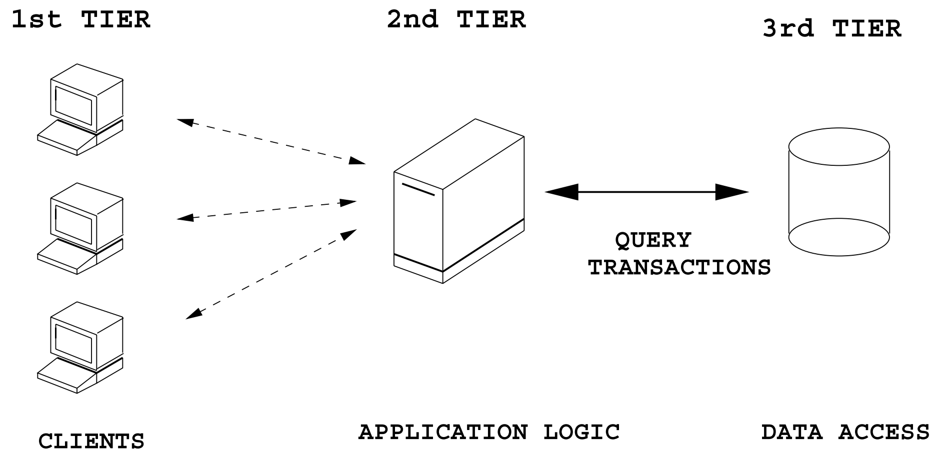


Figure 3.2. Network connection of Three-Tier Structure

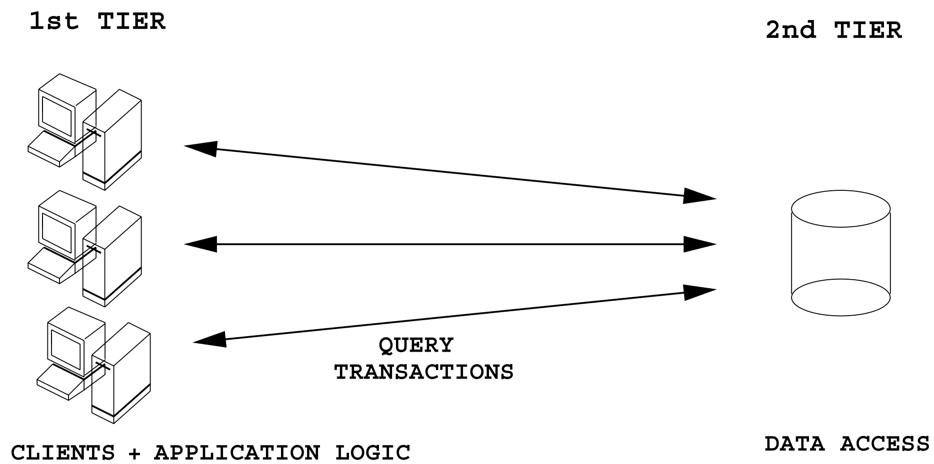


Figure 3.3. Network connection of Two-Tier Structure

translating data from the database into client data in return. The network connection of three-tier approach is illustrated in Figure 3.2.

Typically, the first generation systems use a two-tiered architecture where a client presents a GUI interface to the user, and uses the user's data entry and actions to perform requests of a database server running on a different machine. In the latter, application logic is typically tied to the client application and a heavy network process is required to mediate the client/server interaction. The network connection of two-tier approach is shown in Figure 3.3

The remote database transport protocols are used to carry out the transaction. The network "footprint" is quite large per query so that the effective bandwidth of the network, and thus the corresponding number of users who can effectively use the network, is reduced. Furthermore, not only the network transaction size, but query transaction speed, is slowed by this heavy interaction. This architecture is not intended for mission critical applications.

3.2 Sentinel Application with Three-Tier Architecture

As you see the figures above, two-tier structure overloads client side by having multiple copies of application logic running and server side by heavy network transactions. In order to apply this new design in Sentinel application, application elements are broken down into manageable pieces.

3.2.1 Java Interface

In the three-tier approach proposed in this thesis, the first tier (user interface), is implemented with Java language in the new three-tier design. The Java Virtual Machine is part of the Java specification. The code that is distributed in Java applets is in byte-code format which makes the interface protable. The interface can be run on different platforms without re-compilation and it also makes the implementation suitable for use on the Web.

Networking classes are included in the Java language, and applets downloaded into a browser are permitted to open socket connections. Java classes can connect indirectly to a database server by opening a socket connection to an intermediate server, which in turn opens a connection to the database. Although this method is less direct, it is portable, since the middleware, Proxy [6], can translate between platform-dependent database calls and a custom platform-independent socket-based protocol. This is also the way we wanted to organize our application elements in the three-tier structure. The Proxy Server is illustrated in Figure 3.4.

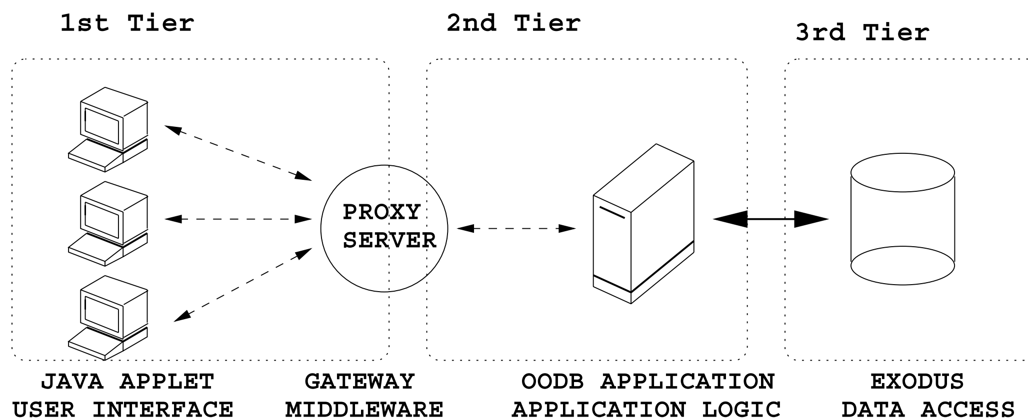


Figure 3.4. Proxy Server in Three-Tier Structure

3.2.2 Proxy Server

In Webster's dictionary, it states as following : prox.y (PROK-see) n., 1. An agent authorized to act for another. The Proxy is the protocol (socket-based) used in between first tier and second tier. In middleware terminology, a proxy which translates between a client and a server is called a gateway. One proxy can easily service multiple clients and multiple servers by using multi-threading. A proxy is a very general concept. It is anything which stands in-between a client and a server and is given the authority to accomplish a particular purpose.

3.2.3 DMDP–WWW Browser-based Application with Applet

Java applet is another way of writing java program where the java byte-code can be executed through WWW browsers. The applets can use networking features just as any Java program can, with the restriction that all communication must be with the host that delivered the applet to its current host. If the first tier is implemented as a Java applet and Proxy server is running in a Web server which is the same host as the applet is running, we can use WWW browsers as our user interface. This is how three-tier structure with Java can make applications suitable for use on the Web. This is also an example of how to use a server-side application to get around applet security restrictions and Figure 3.5 illustrates this example in detail. In the example,

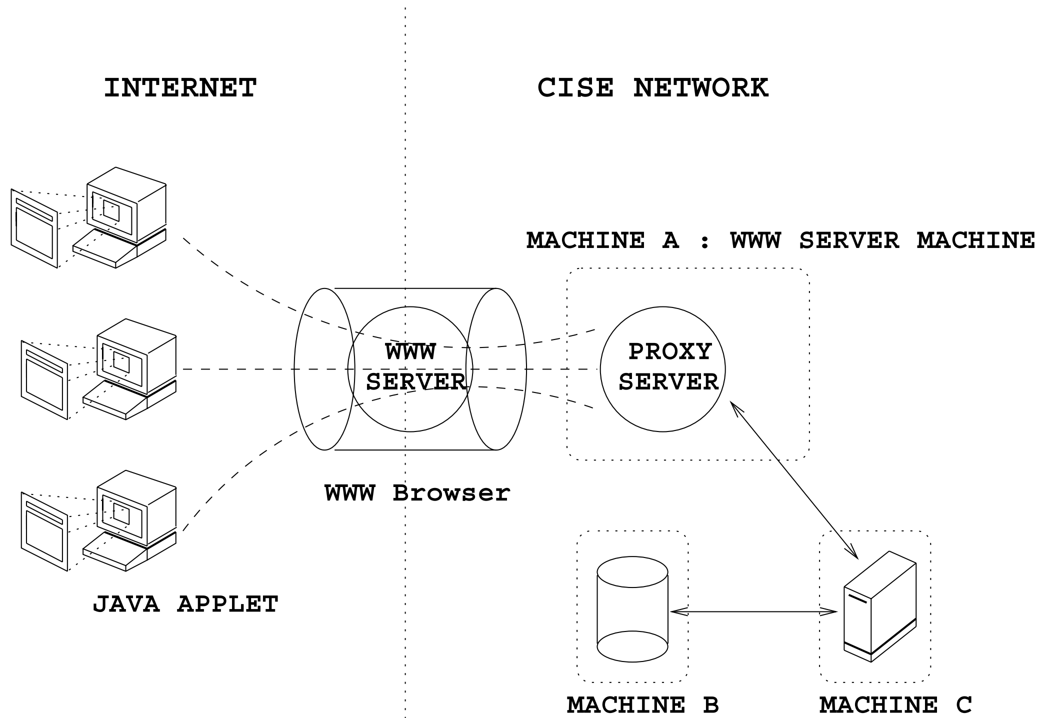


Figure 3.5. 3-Tier Structure with Proxy in WWW environment

applets originating from the same host but running on different machines talk to each other using a server-side application as an intermediary. Implementation issues regarding applying the three-tier approach to WWW environment are discussed in more detail in Chapter 4.

CHAPTER 4
IMPLEMENTATION

In this chapter, we discuss various issues related to the implementation and design of Java Interactive Visualization and Distributed Plan Monitoring Application. Both of them are new Sentinel applications developed with Java interface and the Distributed Plan Monitoring Application (DMDP) is the first example of Sentinel application with three-tier client/server architecture.

Sentinel's Interactive ECA Rule Visualization and Explanation Tool (SIEVE) has been implemented with Java language to explore the user interface aspects of Java language environment. DMDP has been implemented using three-tier architecture with Java interface. Both of the applications have been also converted for Web-based version with Proxy server to utilize the use of Java. We begin with a brief introduction of Java Language Environments [7]. Implementation of both systems are discussed in detail after that.

4.1 The Use of Java

Java is hailed as the latest silver bullet to cure software ills by making all of your software Internet aware, running across different vendors platforms.

The Web is now synonymous for many people with the Internet that underlies it, and usage figures support this. Distributed programming to many will be programs that affect their Web browser, and wander through Web documents on their behalf. Many mechanisms have been attempted to bring programming to the Web. These include Forms with CGI scripts, Plugins, VBScript, etc. Of all these, Java is the most flexible and the one with the most potential to be delivered across all platforms. Unless things go astray, Java is set to be the primary language for Web programmers.

Applets and applications commonly present information to the user and invite the user's interaction using a GUI. The part of the Java environment called the Abstract Window Toolkit [7] (AWT) contains a complete set of classes for writing GUI programs. The toolkit has menus, lists, text boxes, buttons and many other standard user-interface elements. GUI environments such as Windows, the X Window System and the Mac interface all use event-driven mechanisms to support user-driven . The AWT is built on top of each of these and also uses an event driven model. This means that applications can be built to conform to the standard user-interaction models.

The Java Virtual Machine [7] is part of the Java specification. The code that is distributed in Java applets is in byte-code format which is machine independent. The use of virtual machine implies that a local interpreter is used to run each Java program. Thus the same code runs everywhere.

4.2 Java Interactive Visualization

In the active database management systems (ADBMSs), the active feature is incorporated into DBMSs by the event-condition-action [8] (ECA) rule abstraction. Using ECA rules in active database systems for real-life applications involves implementing, debugging, and maintaining large number of rules. For this, a graphical debugging and explanation facility to assist understanding of the interactions - among rules, among events, between rules and events, and between rules and database objects would be useful.

SIEVE is a tool that enables the user to visualize the event detection and rule execution details at run-time in on-line mode as well as at post-run-time on a replay basis. Moreover user gains a new dimension of interactivity through the debugger's two-way communication channel. Rather than passively receiving information delivered out to the user interface, the user is able to input changes to the rule system,

modify and monitor the execution in an interactive manner. With Java's cross-platform architecture, this tool with a Java interface can be run on any machine and even on the Web browser.

The input to the tool consists of:

- *class and instance level rule/event definitions*: This information is supplied by the user in the application program using the event [9] and rule [10] definition language. The preprocessor gathers this static information in the form of a file.
- *event detection information*: This is the run-time information obtained on the occurrences of events and the creation of event objects. This information is provided by the local event detector and written to the run-time log file.
- *rule firing information*: This is furnished to the visualization tool in the same way as the event detection information. In addition, the transaction in which rules were fired and the information concerning locks acquired/released on database objects are also furnished.
- *transaction information* The TID of the transaction in which a rule is fired is provided to the tool by the transaction manager in Open OODB kernel. This association between the rule and the transaction helps the user visualize the nested transaction approach Sentinel adopted in rule execution.

The visualization tool's parser reads the static information, processes them to construct event tree graphs and stores the event and rule information in the in-memory event and rule repositories, which are linked list structures. The data structure which captures the nested execution of rules is an n-ary tree. The root node represents the top-level transaction of the application. When this transaction triggers a rule, and since rules are executed as sub-transactions, the child node of the top-level transaction represents the first rule fired. This node in turn could trigger another rule and

it is represented as the child node (subtransaction of subtransaction) and so on. The transaction tree grows in a top-down way: it starts from the top-level transaction and spans to the descendents.

There are three modes are available for users. User can visualize events and rules when they occur at run-time (Run-Time-Mode) or can do so later via a stored event/rule log (Post-Analysis-Mode). To support online mode, we need to have some mechanism by which there is instantaneous notification of event occurrences and firing of rules. This can be done by means of socket mechanism. Interactive mode allows the user to run the tool allowing the interactive activities. User can sets up a break points where application interrupts and wait for a user to go on. User can also input changes to the system in the course of execution and visualize the effect of these changes.

The Interactive Visualization Tool offers uniform supports for the following and Figure 4.1 illustrates the functional modules of the Interactive Visualization Tool.

- *step/batch mode tracing of rules:* The user can switch between step and batch mode during execution. When the step is chosen, the unit of consecutive execution is the interval between any event, rule or break point. When batch mode is chosen, the unit of trace is the entire application (if break points are ignored) or the interval between two break points (when break points are accepted). The user can enable/disable break points at run-time.

- *using information from preprocessing - post-analysis/run-time execution:* The user can specify if preprocessor should be used by the debugger. Also, he/she can choose from post-execution and run-time analysis before and a trace and change this preference afterwards. SIEVE utilizes either files or sockets accordingly, but the appearance of the front-end remains the same.

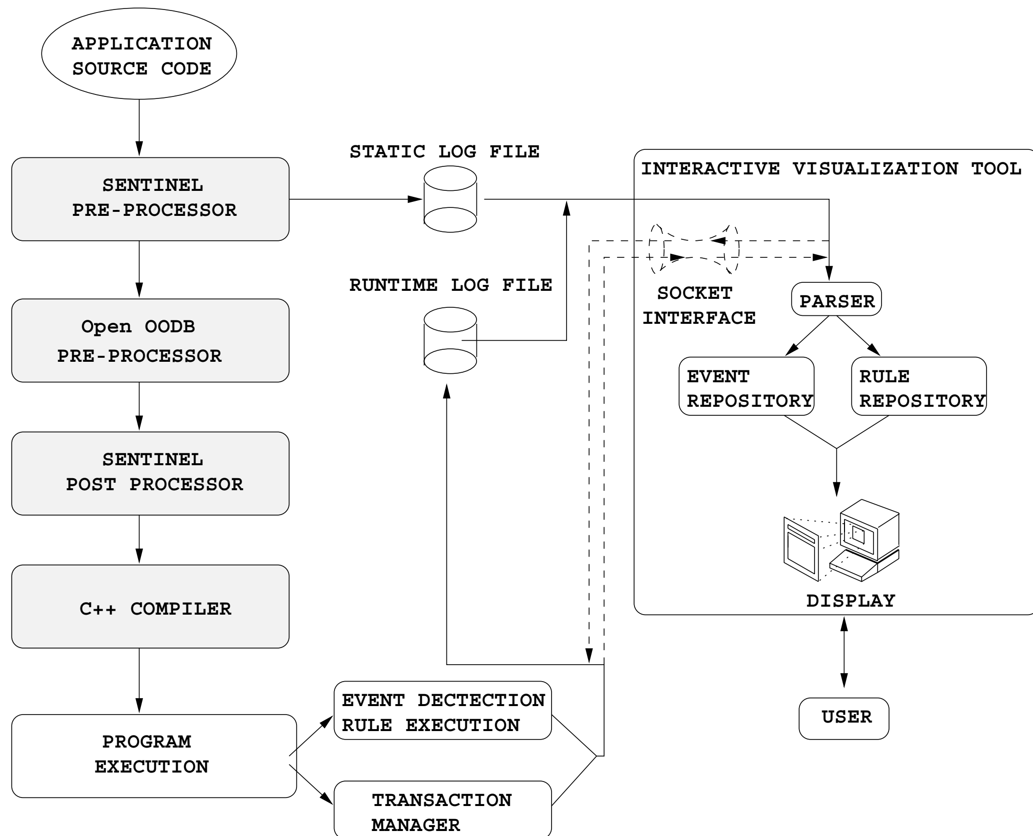


Figure 4.1. Functional Modules of Interactive Visualization Tool

- *using information from transaction manager and event detector - interactive mode on run-time:* In order to support predefined break points in an application, an additional feature is added to the preprocessor that it recognizes break points and maps them into a kind of explicit primitive event. This approach enables the event detector to notify the user interface at run-time when break points occur. Moreover, the preprocessor creates a data structure which contains all procedures (pointers) that are potentially conditions/actions with their names (strings). This one-to-one correspondence between the names and pointers makes it possible to search pointers by name at run-time. This feature is extremely useful for run-time rule creation, when the condition and action part need to be defined dynamically without recompiling.

4.2.1 Limitations and Related Work

Several systems have addressed the use of tools to support the analysis of rule behavior. The current Visualization Tool doesn't have cycle checking during execution to alert the user of potential non-terminating rules. It needs an indication of the existence of potential cycles in rule execution.

The DEAR project [11] which presents a debugging tool for an active, object-oriented database detect the existence of cycles among rules. The DEAR allows that a subset of rules can be viewed to provide for a more focused tracking.

The PEARD's [12] debugging features also include detecting potential cycles in rule execution and a utility to examine different rule execution paths from the same point in the rule triggering process.

These tools are similar to our Visualization tool especially with respect to rule browsing, the ability of set breakpoints, the ability to activate and deactivate rules.

4.2.2 Implementation Issues

The visualization involves drawing static graph from Sentinel pre-processor [10]. The static information of class level and instance level of rules and events is provided to the Tool, and parsed to generate a graph. Then the graph is displayed in the form of abstract syntax trees. This information is furnished by the pre-processor at compile time and at the run time, information about event detection and rule execution is provided. Figure 4.2 illustrates the process of the static information in Java Interactive Visualization Tool.

The static information, a log file generated by Sentinel pre-processor or dynamic information of event detection and rule execution are sent to Java Interactive Visualization Tool through C++ Proxy Server. The purpose of this middle gateway server is for support of a use of the tool on the Web and separation between presentation from another. The Proxy server is just a standard socket protocol between the 1st

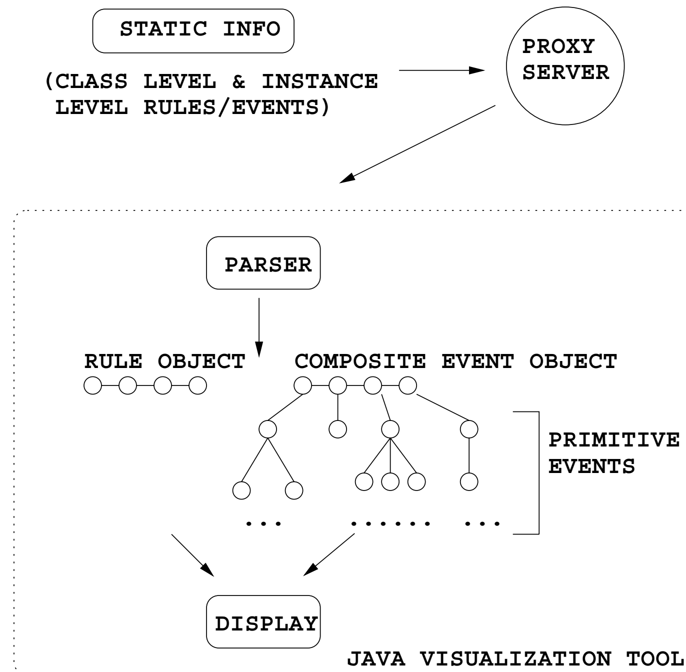


Figure 4.2. Processing of Static Information in Java Interactive Visualization Tool

tier, user interface and second tier, application. Each rule and event from the static information is constructed locally in the tool as a Java class.

The event class definition of the objects used in the tool is defined as below. Each object that needs to be drawn in the Canvas area of the tool has to have data about its location as well as its event or rule. When the static information from Proxy is processed in the tool, for each objects (rules and events), appropriate location data is computed and assigned to them. Those objects are in memory and used while the tool is up and running. This means that the tool can draw those objects anytime since all the data about objects are available in the tool. In Java Interactive Visualization Tool, user can rearrange the event graph by pointing any object and dragging to a desired place. This is advantageous since the event graph can be really tangled if the syntax of the composite events is complex.

```
public class VisNode {
```

```
  int id, x, y, jx, jy, rw, rh;
  String classname, name, pdata, etc;
  boolean primitive;

  static final int VISNODE_HEIGHT = 17;


  static final int XMARGIN = 3;
  static final int YMARGIN = 5;
  static final int char_width = 6;

  public VisNode () {classname = new String(); }

  ////////////////////////////////////////////////////////////
  //     Coordinates of the Event node
  //
  //     A---------------+    A = (x,y)
  //     |            |  C    B = XMARGIN (constants) 3
  //     |    XXXXXX  F  |    C = YMARGIN (constants) 5
  //     |            |  |    D = char_width * (# of chars)
  //     +---------------+    E = D + 2 * B
  //     |-B--|--D--|--B-|    F = VISNODE_HEIGHT
  //     |-------E-------|
  ////////////////////////////////////////////////////////////

  .....
  }
```

The run time information is provided to the tool in the form of a log file or socket stream. In either case, the tool has to draw the run time information in the Canvas along with the static event graph. For each run time information (event detection or rule execution), drawing in the Canvas is updated according to the the information. The updated information has to be drawn in the special manner. Figure 4.3 shows the layout of Java Interactive Visualization Tool in action.

```
.....
  while (Get_Runtime_Info()) {
    Update_Objects();
    Draw_Updated_Info();
  }
.....
```

This drawing code which seems to be reasonable will not work properly in the Java. The function `Draw_Updated_Info()` invokes AWT Component's function `paint()`

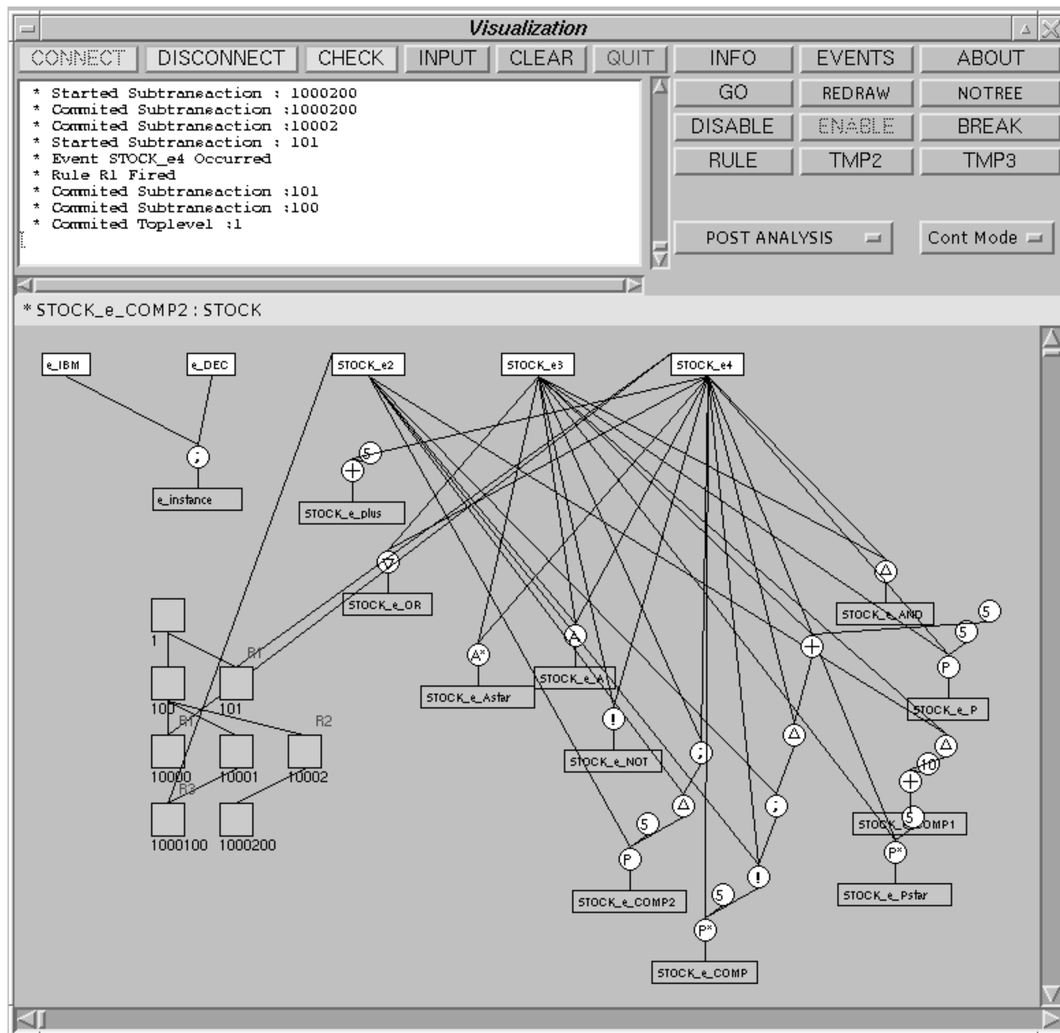


Figure 4.3. Layout of Java Interactive Visualization Tool

which in turn draws AWT Components from the top element to the bottom in the hierarchy. The `paint()` is called as a job of AWT Thread not as a separate thread for drawing. This results in weird behavior. Instead of executing `Update_Objects()` and `Draw_Updated_Info()` functions in turn as it is written in the code above, the `Update_Objects()` function only will be called as many as the loop executes, then the `Draw_Updated_Info()` is invoked repeatedly later. In order to fix this problem, one thread dedicated to painting objects has to be created so that AWT thread which is responsible for drawing AWT Components can do its work without any disturbance.

```
.....
  Thread animator = new Thread(this);

  public void stopPaint() { animator = null; }
  public void startpaint() {
    if(animator == null)
       animator = new Thread(this);
    animator.start();
  }

  public void run() {                 // public void run() is the
    while (Get_Runtime_Info()) {      // function invoked when
      Update_Objects();               // thread.start() function
      Draw_Updated_Info();            // is called
    }
  }
.....
```

In this case, another thread is created specially for drawing the updated objects. Now the AWT thread can do its work and animator thread can do its work respectively without causing any problems each other.

Java applet is another way of writing java program where the java byte-code can be executed through WWW browsers. The applets can use networking features just as any Java program can, with the restriction that all communication must be with the host that delivered the applet to its current host. With Proxy and Java, we were able to make the tool running on the Netscape. As long as the Proxy is running on the Web server machine, Java applets can work correctly. We have to specify the

codebase of the applet so that no matter where the HTML page is served from, the right machine handles the proxying and runs applets. The HTML keyword `CODEBASE` make sure that applets are download from that codebase machine .

```
<APPLET CODEBASE = ``http://www-pub.cise.ufl.edu/~shan/Java/''
CODE=''Visualization.class'' WIDTH=150 HEIGHT=30>
</APPLET>
```

4.2.3 Interface Details

When a Java program with a GUI needs to draw itself - whether for the first time, or in response to becoming unhidden or because its appearance needs to change to reflect something happening inside the program - it starts with the highest component (the top Component in the hierarchy) that needs to be redrawn and works its way down to the bottom-most Components. This is orchestrated by the AWT drawing system. In this way, each Component draws itself before any of the Components it contains. This ensures that a Panel's background, for example, is visible only where it isn't covered by one of the Components it contains. The AWT Components of Interactive Visualization Tool is shown in Figure 4.4.

The interface of Interactive Visualization Tool is composed of several levels in its Component hierarchy. The parent of each level is a Container (which inherits from Component). Many Components can inserted into a Container by appropriate Layout Manager. Different layout manager is available according to different needs. At the top of the hierarchy is the window (Frame instance) that displays the program. When the program runs as an application, the Frame is created in the program's main() method. Under the Frame is a Converter object, which inherits from Applet and thus is a Container (specifically, a Panel). Depending on what viewer the applet is displayed in, one or more Containers might be between the Converter object and the Frame at the top of the Component hierarchy. Figure 4.5 shows the hierarchy of interface elements in the Interactive Visualization Tool.

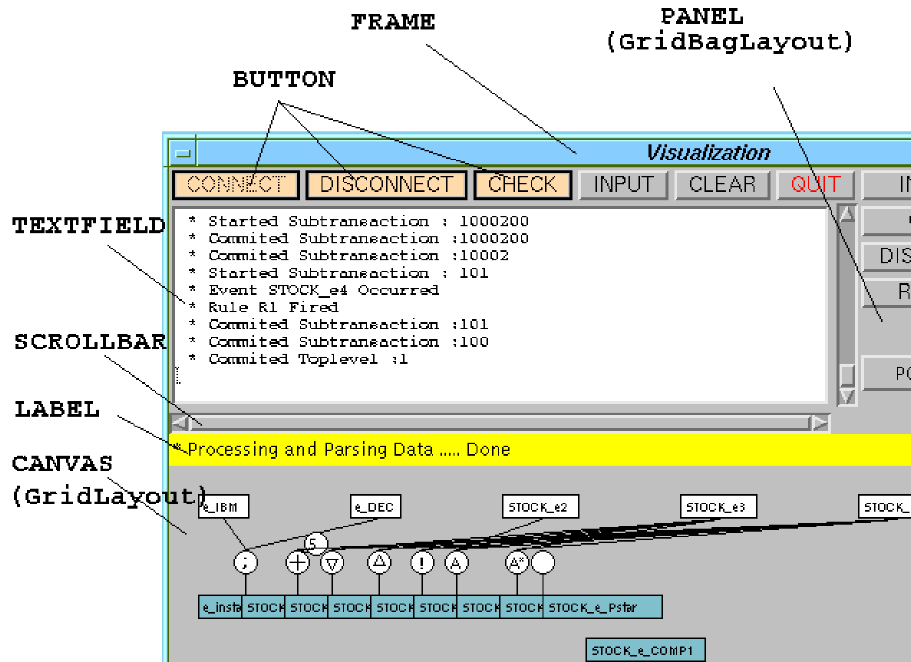


Figure 4.4. AWT Components of Interactive Visualization Tool Interface

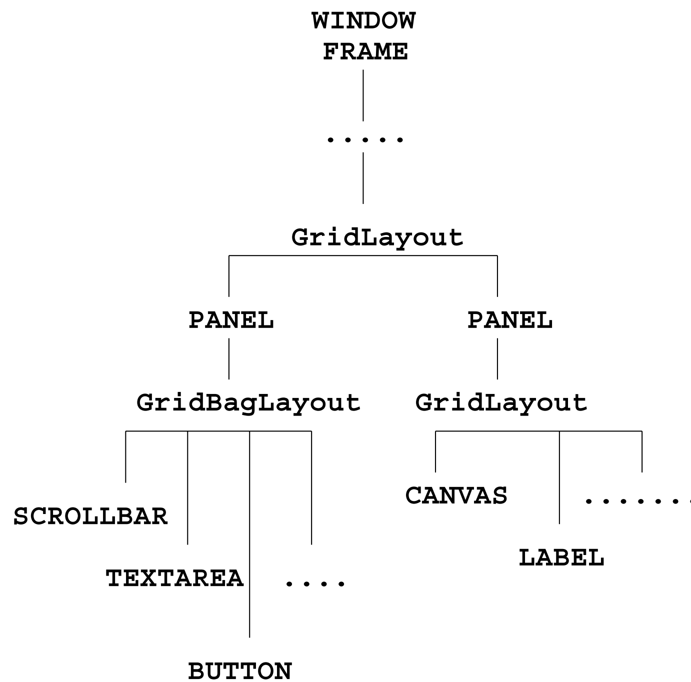


Figure 4.5. AWT Components Hierarchy of Interactive Visualization Tool Interface

4.3 Distributed MDP

The first version of MDP which is a single-user, and two-tier application is re-engineered to multi-user, distributed, and three-tier application. The DMDP (Distributed Plan Monitoring application) utilizes the ECA rules in a real-life situation (distributed environment) with a Global Event Detector [13] (GED) and OQL usage in conditions and actions. The GED handles event requirements of different component applications for supporting distributed monitoring. The GED which uses consumer/producer paradigm for receiving and sending events supports both global composite and locally composite global events. More issues on GED are discussed by H. Liao [13]. The DMDP uses several independent databases used by its component applications (Planbase, Unit/Force, and Weather/Intelligence application). In each component application, OQL is used for conditions and actions and the Java user interface is decoupled from those applications. The architecture of DMDP is the exact three-tier model where each level is separated clear. Figure 4.6 shows the functional modules of DMDP and the three-tier architecture constitutes separate three entities.

The DMDP is composed of the following modules.

- **Weather/Intelligence** : is dynamic and updates weather and intelligence data. This application is responsible for enemy and weather information. If either it receives a bad weather message or a enemy movement message, it will send this information to the other application which needs it.

- **Unit/force**: stores and tracks Unit and Force movements and their status. This application keeps the information on the current status of Units and Forces. If it receives the information of the significant degredation of a certain Unit and the unit belongs to a certain force, it sends this information to the Plan Application.

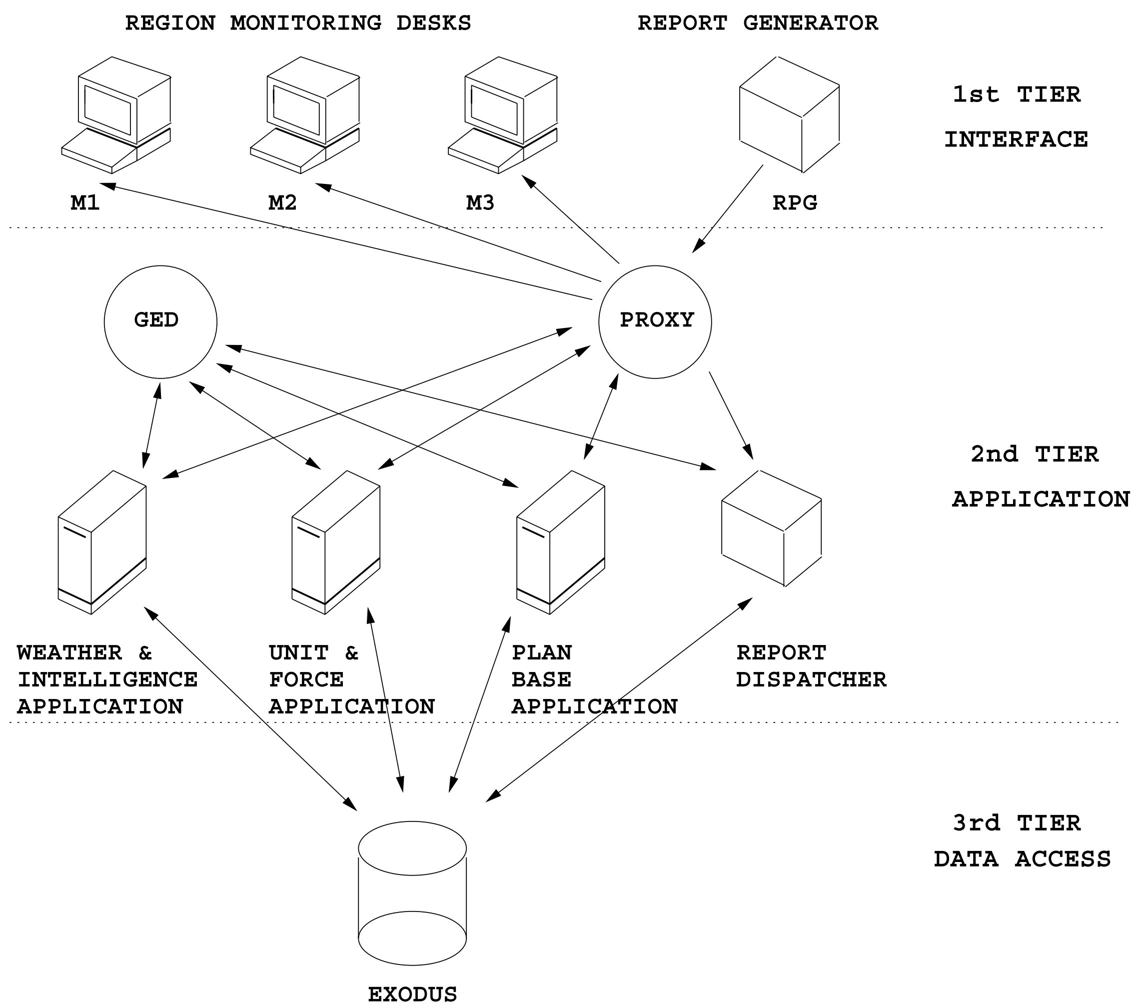


Figure 4.6. Functional Modules of DMDP.

- **Planbase** : holds plan segments along with regions and participating forces. This application maintains the whole plans. If any plan segment is affected due to either a bad weather message or the degredation of a certain unit, it will warn the commander by displaying the affected plan segment in the proper window.

- **Monitoring Desks** : allow desired portion of the world to be monitored.

- **RPG** : simulates a report generator that can send reports to update databases. This application is responsible for dispatching incoming reports to the proper application by raising its local events which are in turn subscribed by the other applications through Global Event Detector.

- **GED** : handles event requirements of different applications for supporting distributed monitoring. All of these applications communicates each other through this Global Event Dectector.

- **Visual Data Server (Proxy)** : mediates between Monitoring Desks and Applications.

In this application there are two kinds of critical situations that may create alerts when they occur:

- *non-weather related* The navy consists of various units (termed as *unit objects*), positioned at various locations for performing some task. Each unit has a *readiness* status indicating whether they are in a position to perform certain operations. Readiness can be defined based on personnel, training, supplies etc. and is maintained in terms of ratings (e.g., 1 signifying Combat Ready and 5 signifying Overhaul). A readiness rating of 2 or below is desired for any unit. As a crisis arises a *plan* has to be prepared to deal with it. A set of units have

to be assigned to carry out each plan. Once this is done any change to either the plan or a unit's readiness status has to be monitored continuously.

- *weather related* There are a fixed number of geographic regions in which the weather will be monitored. The weather is described by wind speed, wave height and the date on which this weather is valid. When severe weather condition is reported for a certain region, the following action will be taken: identify units in the affected region and warn commanders of the severe weather.

It is clear that the active capability of Sentinel can be used to achieve automatic monitoring of the above situations.

4.3.1 Implementation Issues

As for the Visualization, there was no problem in running the Visualization tool on the WWW as an applet. The architecture of the Visualization is a client structure. It makes a request first and gets appropriate data from the Sentinel applications. The current Java Development Kit 1.1.2 is for programming "client" side applets. Applets are subject to fairly strict security restrictions limiting, among other things, what they can communicate with. Since DMDP's Java interface is a "server" where it has to listen to incoming messages from the component applications, we had to combine the RPG (Report Generator) and MD (Monitoring Desk) to make it as client for a web-based application. In an applet, to be able to get data from the applications, it has to be the one that initiates the session (make a request to server) and in that request session applet interface can get the data. That is the main reason why RPG which initiates requests and MD which receives data has to be combined. Figure 4.7 shows architecture change after the migration to WWW. The diagram (A) is the original architecture and (B) is the architecture after the WWW migration.

In the diagram (B), user interface can freeze while the communication delay occurs. This is because only one thread which is the AWT thread is doing all the

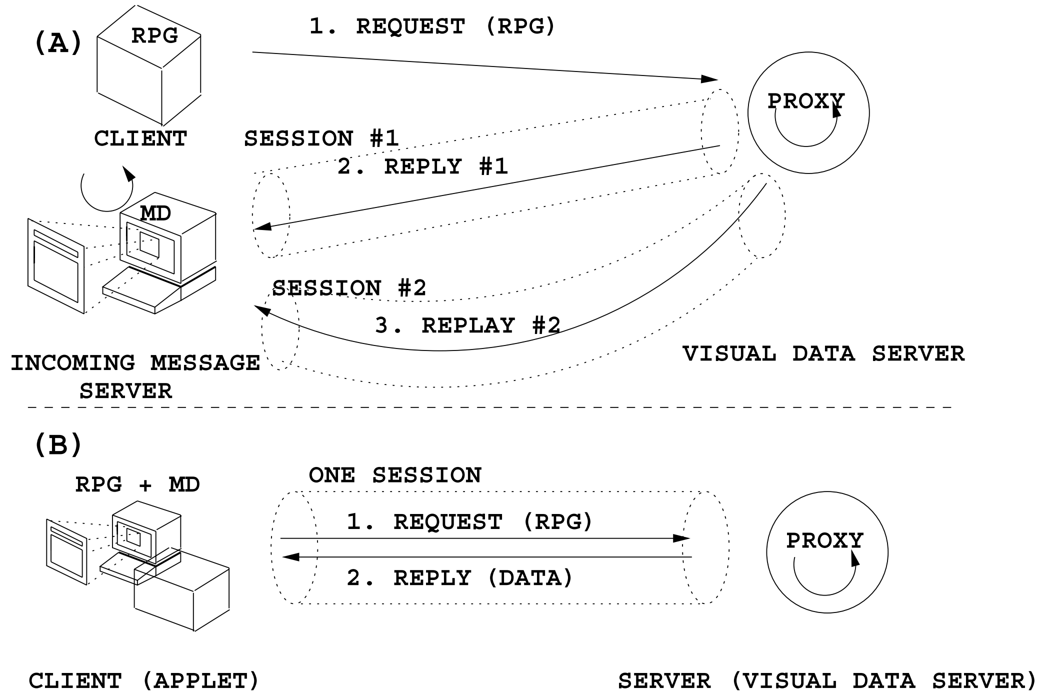


Figure 4.7. Original Application vs. Web-based Application

jobs. The AWT main thread is taking care of both interface events and also receiving/computing data from the Proxy. The freezing problem can occur while the AWT thread is waiting for the data. Whereas the original design (diagram A) doesn't have any problem where another thread is created and dedicated to receiving/computing data. Thus AWT can concentrate on its own job which is to take care of user events.

Getting the Image object corresponding to an image has to be taken care of different in the web-base architecture(diagram B). As long as the image data is in GIF or JPEG format and its filename, it's easy to get an Image object for it: just use one of the Applet or Toolkit `getImage()` methods. The `getImage()` methods return immediately, without checking whether the image data exists. The actual loading of image data normally doesn't start until the first time the program tries to draw the image. In a web-based architecture, we have to use URL instead of filename. The Toolkit class supplies two `getImage()` methods:

```
//
// In Diagram (A): filename is used for opening graphics
//
public abstract Image getImage(String filename)

//
// In Diagram (B): URL is used for opening graphics
//
public abstract Image getImage(URL url)
```

Only applets can use the Applet `getImage()` methods. Moreover, the Applet `getImage()` methods don't work until the applet has a full context (AppletContext). For this reason, these methods do not work if called in a constructor or in a statement that declares an instance variable. You should instead call `getImage()` from a method such as `init()`.

We can get a Toolkit object either by invoking Toolkit's `getDefaultToolkit()` class method or by invoking the Component `getToolkit()` instance method. The Component `getToolkit()` method returns the toolkit that was used (or will be used) to implement the Component.

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image1, image2;

// Opening local files
// (Used in the diagram (A))
image1 = toolkit.getImage(``image.gif'');

// Using URL for opening files in the WWW
// (Used in the diagram (B))
image2 = toolkit.getImage(new URL(``http://www...edu/~shan/image.gif''));
```

These are examples of using the Toolkit `getImage()` methods. Every Java application and applet can use these methods, with applets subject to the usual security restrictions.

4.3.2 Interface Details

The user interface of DMDP interacts with two-tier's applications asynchronously. The Monitoring Desks (MD) are implemented with Java to have a server capability

to support asynchronous distributed event monitoring. Figure 4.8 shows the layout of the Monitoring Desks. Each MD has information about its own responsible region to be monitored. While the Visual Data Server just mediates information between MDs and Applications, each MD can filter out those information. By creating a server thread in the interface, each MDs can listen to the Visual Data Server continuously for any incoming messages. Following is the JServer class which listen for new connections, once a connection is made, spawn off to another process (JServerConnection class) to handle requests.

```
//
// JServer Class
//
public class JServer extends Thread {
  int s_port;            // Server port number
  ServerSocket s_socket; // Server socket
  ....

  public JServer (int port) {
    s_port = port;
    try {
      s_socket = new ServerSocket(port);
    }
    catch (IOException err) {
      System.err.println("Error:Creating Server Socket:" + err);
      System.exit(1);
    }
  }



  public void run() {
    try {
      while(true)
        svr = new JServerConnection(s_socket.accept(), this);
    }
    catch (IOException err) {
      System.err.println("Error:Waiting for connection:" + err);
      System.exit(1);
    }
  }
}
```

JServerConnection class is a thread process spawned from JServer process (parent) and the actual thread process which handles the requests. This is the same example as

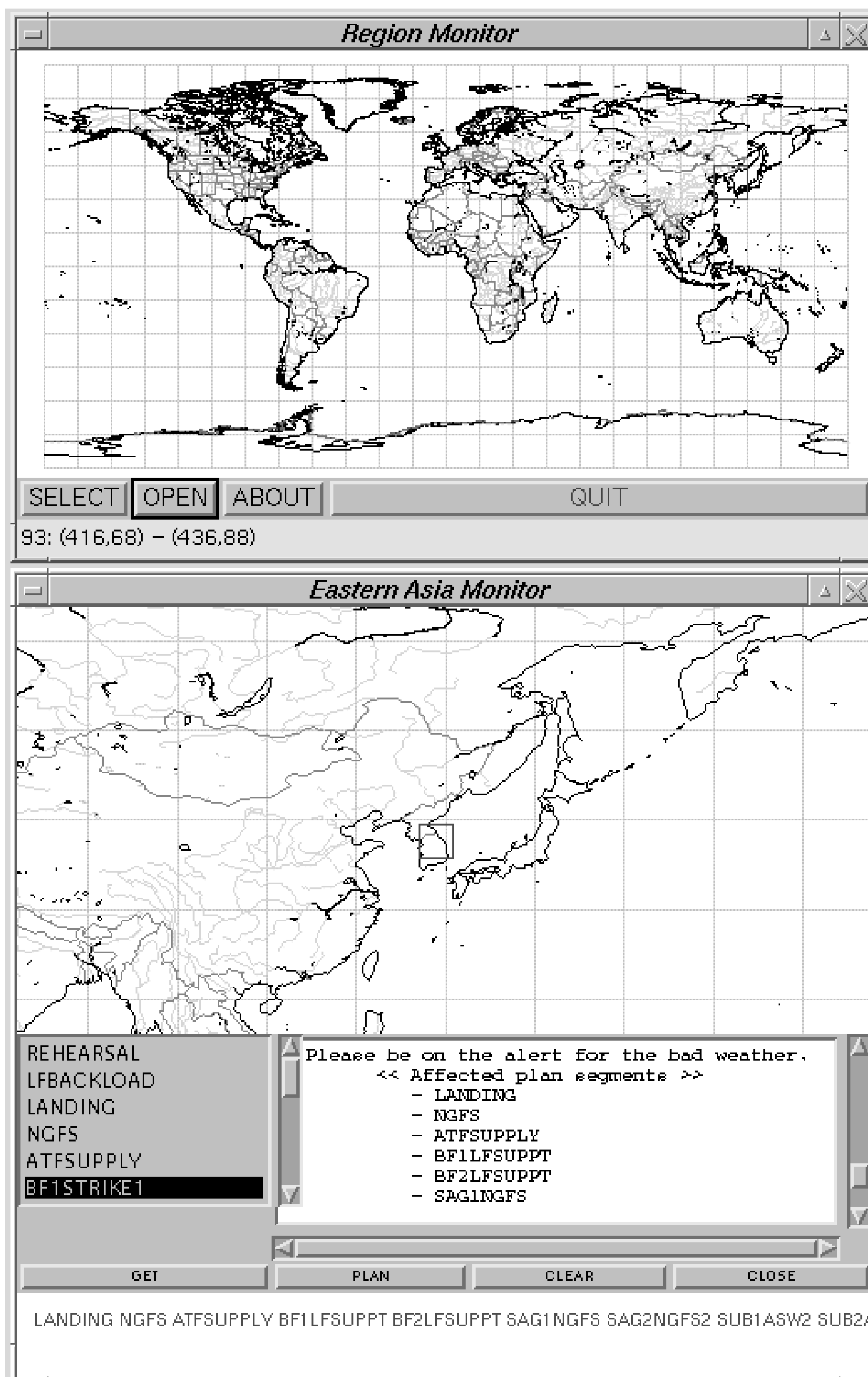


Figure 4.8. Layout of Monitoring Desk

the painting problem mentioned in the previous section. In order to prevent freezing the user interface, dedicated thread has to be created to service a special purpose process. While the DMDP interface is getting user's input continuously, independent thread can be working separately. Since JServer class is not affiliated with any other classes, it can be used in any other classes to accomplish its purpose. In the main module of the interface, JMdp class, it is created to process incoming messages from the applications as below.

```
....
static final int MONITOR_PORT_NUMBER = 7779;
JServer myserver;
MdpMaincanvas canvas;

public JMdp() {
myserver = new JServer(MONITOR_PORT_NUMBER)
....
}
```

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

In today's multi-platform world, traditional application architecture is giving way to new approaches. Generally, client/server can be implemented in a n-tier structure where application logic is partitioned. This leads to faster network communications, greater reliability, and greater overall performance. Increasingly, people need to be able to move traditional two-tier client/server applications to three-tier model that provides modular reusable services on a network.

The new architecture provides a solution to the problems of the current architecture. The three-tier client/server model makes it simpler to manage the application because it *separates the user interface*, the business logic and the data access. This allows us to gain high performance, maintaining data on a robust, very reliable system. Since *changes are limited to well-defined components*, it also gives us a way of managing the application codes and understand them easily so that it resides on a particular departmental or group tier, making code updates and redirection of data easier. User interface can be *run on cross-platform* with Java and standard network protocols and this general architecture allows the *applications running with and without Web.*

5.2 Future Work

5.2.1 Multi-Tier Architecture

The three-tier architecture separates the user interface code from the application logic, but fails to separate the application logic from the data access code. If we swap out our favorite relational database for a new object oriented database, what

happens? We probably face massive rewrites of the application, and a future filled with supporting two separate versions of our application. The three-tier architecture makes it difficult to change the backend data storage mechanism without seriously disrupting the application logic. The 4-tier approach is the solution and clearly separates the application logic from the data access logic.

5.2.2 Distributed Object Architecture

For maximum flexibility and scalability, an object-oriented approach, using a uniform communication mechanism between objects generally could work best. An application built using this approach could actually be fully implemented before a decision is made as to where the code should reside, and the code can be distributed as needed for each installation of the application.

APPENDIX : SAMPLE STATIC/DYNAMIC FILE

The sample static file (`Demo_stock.static`) would contain following information:

```
EVENT NULL L_PRIMITIVE e_IBM [int sell_stock(int qty)] end
EVENT NULL L_PRIMITIVE e_DEC [int buy_stock(int qty)] end
EVENT NULL L_COMPOSITE e_instance [e_IBM SEQ e_DEC]
RULE R_instance NULL cond1 test_action8 RECENT  e_instance
EVENT STOCK L_PRIMITIVE STOCK_e2 [int sell_stock(int qty)] end
EVENT STOCK L_PRIMITIVE STOCK_e3 [int buy_stock(int qty)] end
EVENT STOCK L_PRIMITIVE STOCK_e4 [int get_price()] end
EVENT STOCK L_COMPOSITE STOCK_e_plus [STOCK_e4 PLUS 5 sec]
EVENT STOCK L_COMPOSITE STOCK_e_OR [STOCK_e4 OR STOCK_e3]
EVENT STOCK L_COMPOSITE STOCK_e_AND [STOCK_e4 AND STOCK_e3]
EVENT STOCK L_COMPOSITE STOCK_e_NOT [ NOT (STOCK_e4,STOCK_e3,STOCK_e4)]
EVENT STOCK L_COMPOSITE STOCK_e_A [A(STOCK_e3,STOCK_e4,STOCK_e2)]
EVENT STOCK L_COMPOSITE STOCK_e_Astar [A*(STOCK_e3,STOCK_e4,STOCK_e3)]
EVENT STOCK L_COMPOSITE STOCK_e_P [P(STOCK_e3,5 sec,STOCK_e4)]
EVENT STOCK L_COMPOSITE STOCK_e_Pstar [P*(STOCK_e3,5 sec,STOCK_e4)]
EVENT STOCK L_COMPOSITE STOCK_e_COMP1 [(STOCK_e2 AND STOCK_e3) PLUS 10 sec]
EVENT STOCK L_COMPOSITE STOCK_e_COMP2 \
  [P((STOCK_e4 SEQ STOCK_e3) AND STOCK_e2,5 sec,STOCK_e2)]
EVENT STOCK L_COMPOSITE STOCK_e_COMP \
  [P*( NOT ((((STOCK_e3) PLUS 5 sec) AND STOCK_e4) SEQ \
  STOCK_e2,STOCK_e4,STOCK_e2),5 sec,STOCK_e4)]
RULE R1 STOCK cond1 test_action3 RECENT  STOCK_e_plus
RULE R2 STOCK cond1 test_action4 RECENT  STOCK_e_AND
RULE R3 STOCK cond1 test_action5 RECENT  STOCK_e_OR
RULE R4 STOCK cond1 test_action6 RECENT  STOCK_e_SEQ
RULE R5 STOCK cond1 test_action7 RECENT  STOCK_e_NOT
RULE R7 STOCK cond1 test_action9 RECENT  STOCK_e_A
RULE R8 STOCK cond1 test_action10 RECENT  STOCK_e_Astar
RULE R9 STOCK cond1 test_action11 RECENT  STOCK_e_P
RULE R10 STOCK cond1 test_action12 RECENT  STOCK_e_Pstar
RULE R11 STOCK cond1 test_action13 RECENT  STOCK_e_TEMPORAL
RULE R12 STOCK cond1 test_action16 RECENT  STOCK_e_COMP
```

The sample dynamic file (`Demo_stock.dynamic`) would contain following information:

```
Toplevel 1
SubTransaction 100
SubTransaction 10000
Event STOCK_e4 234
Rule R1 567
SubCommit 10000
SubTransaction 10001
SubTransaction 1000100
Event STOCK_e2 235
Rule R3 987
SubCommit 1000100
SubCommit 10001
SubTransaction 10002
Rule R2 364
SubTransaction 1000200
SubCommit 1000200
SubCommit 10002
SubTransaction 101
Event STOCK_e4 234
Rule R1 567
SubCommit 101
SubCommit 100
Commit 1
```

REFERENCES

[1] Sun Microsystems Inc. Open Look, Graphical User Interface, Application Style Guidelines. Mountain View, 1989.

[2] R. McCool, L. Masinter. Introduction to Common Gateway. Interface `http://hoohoo.ncsa.uiuc.edu/cgi/overview.html`, 1994.

[3] T. Berners-Lee, H. F. Nielson, R. T. Fielding. The Hypertext Transfer Protocol. National Center for Super Computer Applications, Software Development Group. Chicago, 1992.

[4] J. Ousterhout. An Introduction To Tcl and Tk. Chicago, Addison-Wesley, 1994.

[5] M. Prahlad. A Dynamic Rule Editor for Sentinel: Design and Implementation. Master's thesis, University of Florida, Gainesville, 1997.

[6] S. George. Client/Server: Past, Present and Future. *Client/Server Internet Conference & Exposition*, Chicago, June 12-14, 1996.

[7] J. Gosling and H. McGilton. The Java Language Environment : A White Paper. Sun Microsystems Inc., Mountain View, 1995.

[8] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, University of Florida, Gainesville, 1994.

[9] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, 1994.

[10] D. Mishra. SNOOP: An Event Specification Language for Active Databases. Master's thesis, University of Florida, Gainesville, 1991.

[11] O. Diaz, A. Jaime, N. Paton. DEAR: A Debugger for Active Rules in an Object-Oriented Context. *Proceedings of the First International Workshop on Rules in Database Systems,* Edinburgh, Scotland, 1993, pages 180-193.

[12] A. Jahne, S. Urban. PEARD: A Prototype Environment for Active Rule Debugging. *Processings of the Journal of Intelligent Information Systems*, 7(2):5-23, 1996.

[13] H. Liao. Global Events in Sentinel: Design and Implementation of a Global Event Detector. Master's thesis, University of Florida, Gainesville, 1997.

BIOGRAPHICAL SKETCH

Sang-Woo Han was born on February 21, 1971, in Seoul, Korea. He received the Bachelor of Science degree in computer science from the University of Hawaii at Manoa, Honolulu, in August 1995. He has worked as a research assistant at the Collaborative Software Development Laboratory at the University of Hawaii.

In the Fall of '95, he started his graduate studies with a major in computer and information science and engineering at the University of Florida. He has worked in the Database Systems Research and Development Center of UF. He will receive his Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, in December 1997. His research interests include client/server architectures, network programming, user interface and object-orient languages.