A COMPONENT-BASED EVENT-DRIVEN INTERACTIVE VISUALIZATION SOFTWARE ARCHITECTURE

Robert F. Erbacher
Department of Computer Science, LI 67A
University at Albany-SUNY
Albany, NY 12222, USA
erbacher@cs.albany.edu

Keywords: Software Architecture, User Interface, Information Visualization, Component Architecture

Abstract

This paper describes our research to develop an effective visualization environment for real-time intrusion detection and the resultant architecture. The environment requirements necessitate that effective interaction be maintained while performing real-time visualization of system log data. Additionally, the environment design is based on the understanding that potentially different data sources and visualization techniques would be used, requiring the ability to swap in different functional units and components. The architecture relies on an object-oriented platform, standard libraries, and publicly available toolkits. No proprietary tools are used and complete system independence is maintained. Additional benefits are gained from the multithreaded architecture, temporal event management, and guaranteed interaction response time. We show the benefits of this architecture over those of typical graphical environments while maintaining development ease.

1. INTRODUCTION

Graphics architectures have been discussed with respect to graphical hardware [1], general graphical environments [2], visualization specific applications [3, 4], and even the architecture for OpenGL [5]. However, there has been a lack of discussion as to effective software architectures for system-independent visualization environments using standard libraries and toolkits. In this paper, we discuss the software architecture for our intrusion detection environment [6, 7]. This environment acts as a visual front-end to system log files and other intrusion detection systems.

The potential users of the environment include system administrators, network administrators, and security experts. The expected user domain delegates the following needs within the architectural specification:

- The environment must be system independent. We have to expect our user base to employ the environment on both MS Windows and UNIX platforms.
- The system must rely only on standardly available libraries and software tools. This must follow from the previous item since many libraries are often MS Windows specific.
- No commercial libraries or tools should be used. Since many potential users are academic institutions and home users, requiring commercial libraries will not be acceptable and prevent wide adoption of the tools.
- The environment should use the minimal amount of screen real-estate possible when not being interacted with directly. Our tool is designed to work with a system administrator's needs beyond network monitoring, namely system configuration, maintenance, etc. Most of the details of the network activity are visible with a very small window that can be placed in a corner of the screen with the system administrator's other tools. Only when additional analysis is required must the window be enlarged and the user interface fully displayed.
- The environment must be easily expandable to support additional data formats and visualization techniques. As we continue to research and develop the capabilities of the environment, we will add additional visualization capabilities and support for additional database paradigms. These new capabilities must expand the capabilities of the environment without losing other capabilities. Different visualization techniques must be available to system administrators for different needs, as they are developed. New capabilities must consequently be added with the ability to select between them.

Our research resulted in the development of a component-based object-oriented architecture [8] to meet these goals. Section 2 provides a brief overview of the intrusion detection environment followed by a description of the software architecture as well as the features important to the effectiveness of the software architecture in section 3. Section 4 provides insights into issues resolving around maintaining a system-independent architecture. Section 5 discusses multithreading issues. We conclude with details of the system's implementation, conclusions, and future work.

2. THE INTRUSION DETECTION ENVIRONMENT

Figure 1 shows an example of our intrusion detection visualization [6, 7]. The environment shows the system being monitored in the center of the window and connecting systems in concentric circles around the monitored system. The ring in which a node is placed is representative of the difference between the IP addresses of the monitored system and the remote system, providing a representation of locality. A border surrounding the entire display window provides a visual representation of the time of day. The border is black at midnight and white at noon. An additional yellow border is indicative of PM vs. AM, a necessity given the difficulty of determining if intensity is increasing or decreasing when changed in small increments.

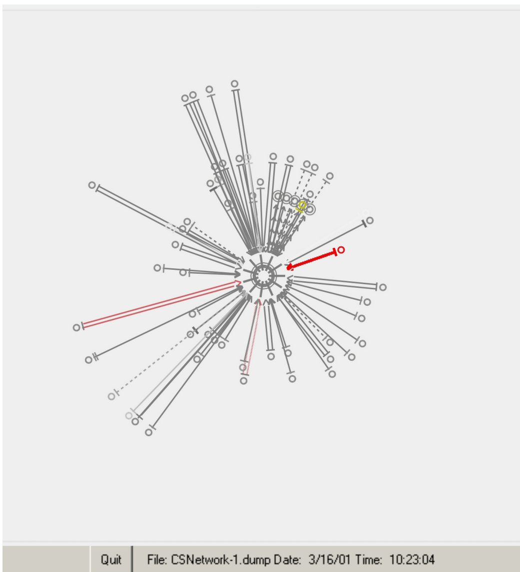

Figure 1: The intrusion detection environment.

The monitored system has additional information attached to its glyph. Each spoke is representative of ten users and the thickness of the inner circle is representative of the system load. The connecting nodes have cross hashes representing the number of different users connecting from that node, represented by individual cross hashes. The number of connections by an individual user is represented by the thickness of the corresponding cross hash.

The directed lines are themselves glyphs showing the direction of the connection, the state of the connection, and the type of connection. A node with two parallel lines is indicative of an unauthenticated connection. If the parallel lines are red then the authentication has failed. Solid lines are telnet or rlogin connections, long dashed lines are privileged FTP connections, and short dashed lines are anonymous FTP connections. Lines with multiple arrows, usually four, are indicative of NFS connections. A lost NFS connection is represented by highlighting the node in yellow. Thick red lines represent port sentry identified attacks.

2.1. The User Interface

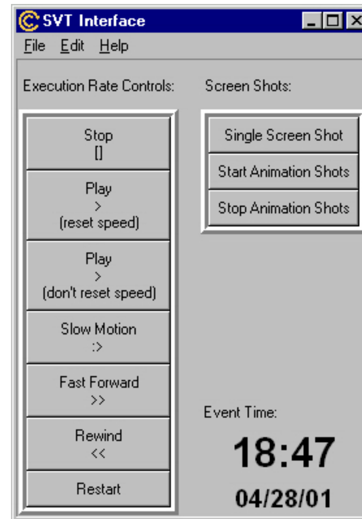

Figure 2: The user interface.

The user interface, figure 2, provides basic capabilities for controlling the environment. In addition to a typical menu the user interface contains convenient buttons for taking snapshots and animations of the environment. This is critical if the system administrator needs to capture information relevant to an attack for future analysis or for law enforcement. The exact time of day is also provided in digital form in the user interface. A representation of the time of day is needed within the visual display to maintain the user's focus on that single display so that no critical events are missed, however, it is often necessary to know the exact time of day, particularly when it is determined that action must be taken and details of an attack forwarded to the offending site. The remainder of the interface is occupied by VCR-like controls that allow the user to control the execution rate of the simulation, speeding it up, slowing it down, pausing it, and rewinding it as necessary.

3. ARCHITECTURAL OVERVIEW

The main environment architecture is shown in figure 3. The architecture is designed to allow easy replacement of components or inclusion of additional components. For example, currently the data retrieval routines acquire data from a Postgres [9] database dump. By ensuring this is a lightweight independent component it can easily be replaced with a new component to directly retrieve the appropriate data elements from the Postgres database itself, rather than from a dump file. Alternatively, it may be necessary to incorporate additional data formats simultaneously. As we are attempting to develop a visual front-end for intrusion detection systems we must be prepared to integrate the ability to read/parse whatever formats other intrusion detection tool's output. Fortunately, most tools will output their results to the standard system log files, our main source of data, requiring no changes. With the advent of IDWG [10] as a potential coordinated format for data distribution between intrusion detection tools we must be able to incorporate the

ability to read this data format in conjunction with data formats already supported and integrated. This type of component-oriented integration is carried through the visualization manager as well. This is critical as we explore additional visualization capabilities and layouts. We wish to reuse as much of the graphics display capabilities already verified to be correct as possible and only re-implement aspects of the code that are actually necessary for the new visualization capabilities.

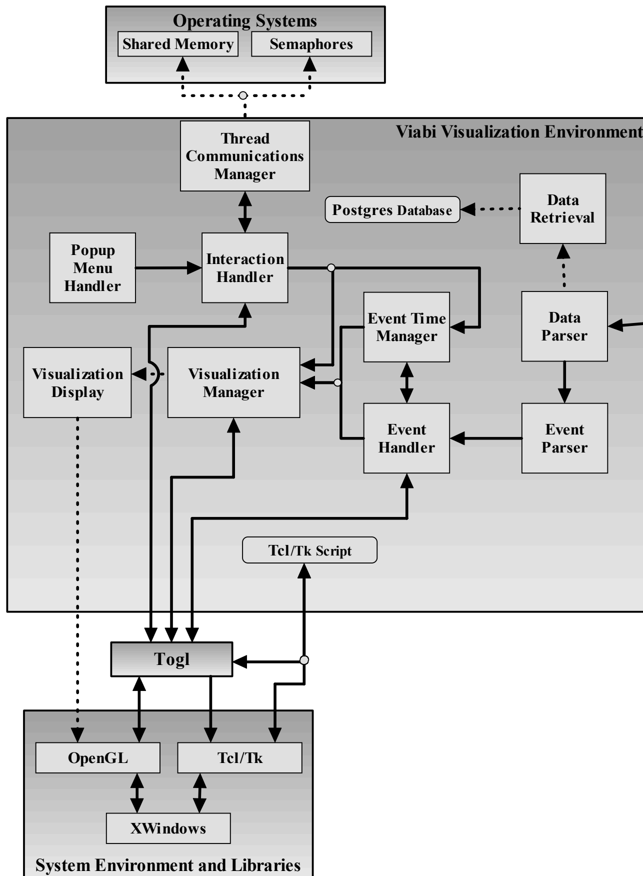

Figure 3: Software architecture for main environment.

The control of the environment is placed on the event handler, which interprets and dispatches events. All interactions, either within the current architecture or within the user interface architecture, are processed by the interaction handler. This is important since some events can be initiated through a popup window within the visual display or through the separate Tcl/Tk interface thread.

3.1. The User Interface

The architecture for the user-interface thread is shown in figure 4. This is in essence a small subset of the architecture shown in figure 3. The consistency in design aids in development and debugging of both environments. The user-interface thread communicates with the main thread through shared memory and semaphores. Aspects of the architectural style are similar to that of a C2 architecture [11] with the GUI being contained within a separate thread and communication being performed through standard inter-process communication paradigms. However, our architecture differs in the scale of the individual threads and the communication paradigm that is not intrinsically message-based, as in the C2 style. These architectural differences are directed by our need for a high performance environment.

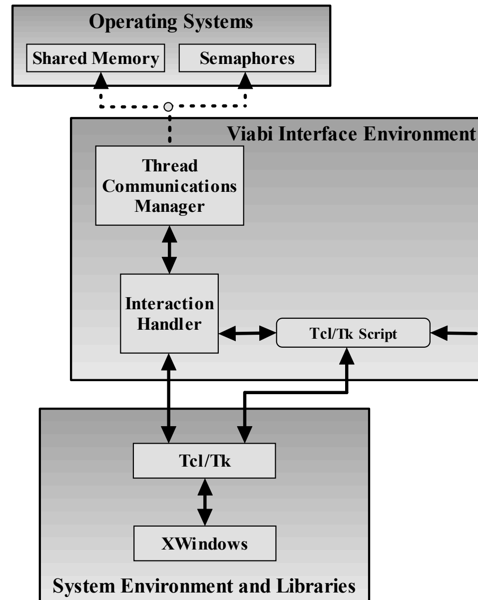

Figure 4: Software architecture for the user interface environment.

As this thread of the environment does not require OpenGL we have not integrated togl. The interaction handler is a ‘C’ component forked and initialized by the main viabi environment that then initializes and executes the Tcl/Tk script. While most of the interface code and subsequent data handling can be incorporated directly into the Tcl/Tk script, the need to interface with the main environment requires a ‘C’ implementation. In particular, the Thread Communications Manager is responsible for storing the appropriate values into the shared memory region and raising the appropriate semaphores. The Interaction Handler acts as an interface between the Tcl/Tk component and the thread communication manager and consists of initialization routines as well as functions registered with Tcl/Tk.

On the main environment side, the Thread Communication Manager retrieves any interactions by the user and passes the necessary parameters to the Interaction Handler. The Interaction Handler collates interactions from the separate user interface, the integrated user interface components, and popup menus. This is necessary to synchronize the inter-

actions and associated data variables while limiting the number of entry points interaction has with various objects. Currently, with the limited interaction provided, all interactions are fed from the Interaction Handler to the Event Time Manager. As the environment expands this will not remain the case and the Interaction Handler will provide data directly to the Visualization Manager and Event Handler.

3.2. Event Deployment

Event deployment is controlled by the event manager. Initial deployment is started by the user through initiation of the play command in the user interface. Once initiated, the event manager must read the next event from the list of events, identify any delay necessary for time coherence, and pass the event and associated data to the visualization manager. The Visualization Manager then updates the display with appropriate changes to the relevant node, updates the remaining nodes with appropriate temporal adjustments, and finally redisplays the visualization.

3.3. Time Management

The event time manager has two primary responsibilities. First, it must keep track of the user's interaction with the interface panel and enact appropriate delays and notify the event handler of the direction that the analysis should be run, necessary for running in reverse. It is also necessary when running the analysis in fast forward, in which case in order to speed up the simulation the actual display of steps is skipped, though the events are incorporated into the appropriate display lists for rendering at intervals. Essentially, this process can be thought of as only visually displaying the accumulated events at predetermined synchronization points, at which time the display is synchronized with the set of processed events. This is effective at increasing execution rate since it is the display of the events that is the greatest performance hinderer.

The second function of the event time manager is to provide temporal interpolation of the display and event processing. The data collection scripts are guaranteed to collect system statistics every five minutes. Other events, such as user logins or system attacks, will be generated at inconsistent intervals. The display will subsequently be updated at these inconsistent points. In order to provide temporal consistency, it is necessary to provide intermediary updates during greater durations of inactivity. The lack of such temporal consistency would otherwise lead to confusion as to the temporal relationship between events.

3.4. Visualization Manager and Display

The visualization manager is a component-based object responsible for transforming dispatched events and associated parameters into the appropriate visual attributes in association with the displayed nodes. The Update Display Node routine is the primary function and has two main tasks. First, it must generate a new node when indicated by an appropriate event. Second, it is responsible for modifying nodes during each iteration of the environment to reflect changes indicative of the passage of time and as necessitated by events. For example, nodes reflecting no longer active connections must fade each frame, node attributes must change to reflect changes in state, such as from unauthenticated to authenticated, and nodes that initiate more than a single connection must have representative connectors.

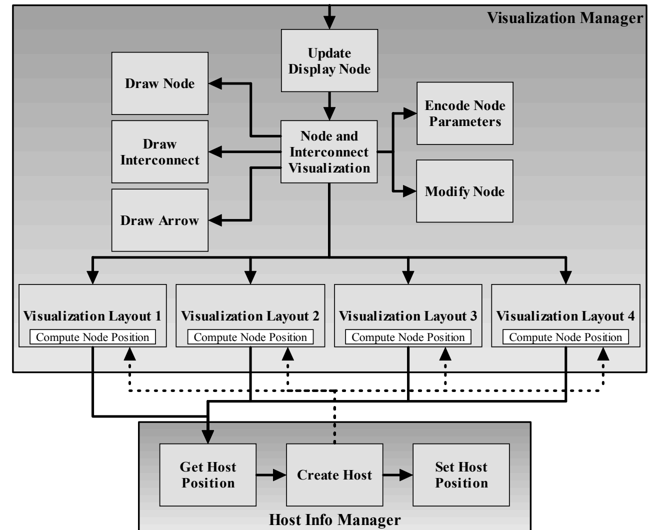

Figure 5: Software architecture for visualization manager and display.

Once the task to be performed for each node is identified, the Node and Interconnect Visualization function is called with the appropriate instructions. Many component functions have been created to assist in the generation and modification of nodes: Draw Node, Draw Interconnect, Draw Arrow, Encode Node Parameters, etc. In addition to these basic functions, there are additional functions for controlling the placement of the nodes based on the current node placement strategy. The components operate individually and provide appropriate algorithms for determining where to place a node. In the example of figure 1, the monitored node is placed in the center of the screen and connecting remote nodes are placed in one of five rings depending on the number of values differing between the IP addresses of the remote and monitored nodes. When monitoring multiple systems, the layout must be changed significantly. We are currently experimenting with different layouts in an attempt to identify a visual layout that is effective and yet efficient in its use of screen real-estate. Providing these functions as selectable components are critical for the selection and comparison of the different layouts.

The node layout functions rely heavily on functionality provided by the HostInfo Manager object. This object maintains a list of remote nodes, both past and present. The node layout functions will first call the GetHost Position function within the HostInfo Manager to determine if the current node has been seen previously. If it has, then we attempt to display the node in the same position as it was seen previously. This is necessary for assisting system administrators in correlating temporal relationships. If the node has not been seen previously then the HostInfo Manager will create a new node and store information related to the new node and determine an appropriate position for this new node by calling an available function within the appropriate Visualization Layout function.

4. SYSTEM INDEPENDENCE SUPPORT

In general, the development of a system-independent application requires the use of the glut library [12] or extensive application of compiler directives and the incorporation of multiple window managing routines for use by the appropriate windowing environment. Glut essentially provides the window management capabilities and correspondingly a system independence layer for window management functionality. While GLUT is an effective window managing library it does not provide the detailed control that is needed for some application environments. Glut has proven to be more than adequate for our needs and combined with Togl, as discussed below, many of the issues incurred from the use of Glut are eliminated.

4.1. UNIX Functionality

Cygwin [13] incorporates library and runtime facilities typical of UNIX environments, thus providing portability of UNIX functionality not provided in standard MS Windows environments. Cygwin, in essence, implements an emulation layer on top of MS Windows. It should be made clear that the emulation layer incorporates no loss of performance and functionality of DLL's are accessible through the compatibility layer. This is critical for incorporating OpenGL with the necessary performance requirements of the environment. Using OpenGL within the Cygwin environment allows full access to hardware acceleration while maintaining UNIX compatible programming conventions. Cygwin additionally incorporates most standard UNIX commands, tools, and libraries, providing a UNIX like look and feel.

4.2. Window Management Independence

When attempting to incorporate a user interface into an OpenGL program, while maintaining system independence, the developer has several principal choices, OpenGL native components, GLUI (GLUT User Interface Library) components [14], and Tcl/Tk components [15]. While many potential implementations of user interfaces for use in conjunction with OpenGL have been and are being developed, these represent the most widely accepted choices. OpenGL native components are currently too limited as interface development is not a strength of OpenGL. GLUI is a relatively new library with extensive interface component capabilities. However, being relatively new we have found GLUI to be relatively buggy. In addition, interface development with GLUI can be confusing and time confusing. Tcl/Tk provides a robust interface development environment refined over years of use. The simplicity of Tcl/Tk interface development is undeniable. In addition, Tcl/Tk provides an unparalleled diversity of interface components. Combined with the availability of a wide assortment of extensions, Tcl/Tk provides an unparalleled environment.

Tcl/Tk does have one disadvantage in that it is a self-contained callback based platform. Consequently, Tcl/Tk essentially takes over control of the executing program, returning control to appropriate functions only when triggered to do so by an event. This conflicts with the behavior of glut which is also a self-contained callback based platform. This conflict means that once control is passed off to either Tcl/Tk or GLUT it is no longer possible to initialize the other platform. Togl eliminates this conflict by providing an interface between glut and Tcl/Tk through a native Tk OpenGL widget. This widget intrinsically provides a canvas into which OpenGL graphics can be displayed. It is designed to work with the GLUT library, thus ensuring complete system independence and ease of implementation as compared with a system-specific windowing environment.

Additionally, togl provides an encapsulation layer, improving the data hiding and data passing capabilities of the platform. GLUT in and of itself only passes the basic parameters required by a callback with no mechanism for passing shared data. This results in the need for extensive global variables. Togl resolves much of this problem by providing the ability to pass client data as a component of the callback arguments, greatly improving the data encapsulation and data hiding capabilities of the platform.

The main display window for the environment can be seen in figure 1. The top part of the display shown in the figure provides the OpenGL canvas. The bottom part of the display provides the Tcl/Tk interface components provided with the main display window. While the interface components provided with the main display window are limited and could easily be incorporated using GLUI, the additional features provided by Tcl/Tk and Togl outweigh the benefits of using an OpenGL native interface.

While Togl is a "C" based platform, it is rudimentary to incorporate wrappers that hide the true nature of the passed data elements and incorporate the environment into a "C++" platform.

5. MULTITHREADING

The environment is based principally around a two-thread approach. The visualization display provides the primary thread and is initiated at startup time. The second thread, the user interface environment, is initiated through fork and exec by the visualization display environment. Providing a separate thread of execution for the user interface is critical for providing the user with access to the features of the environment during execution. In particular, the main thread can consume all idle time associated with the process, preventing or delaying the user's interaction from being processed. Since it is critical that the user be able to control the execution rate of the environment, take snapshots of critical points of activity, and review past periods of activity, the user must have unfettered control over the execution of the environment. Providing a separate thread of control for this purpose ensures such control that a single thread of execution cannot.

Currently, all computations for the visualization are performed by a single thread. While we are exploring the applicability of parallelizing this component of the architecture we do not expect to achieve any substantial improvement in performance. This results from the computational simplicity of the algorithms involved and the fact that much of the computation and display process is offloaded to the specialized display processor of the graphics board.

5.1. Thread Communication and Synchronization

With a multithreaded approach it is necessary to effectively communicate information, e.g., state changes, parameter changes, variable updates, etc. There are several options for this communication, including: disk files, pipes, shared memory, message queues, and TCP/IP [16]. With a real-time interactive environment it is critical that we choose an efficient strategy. Disk files are clearly far too slow. TCP/IP incorporates far too much overhead for threads guaranteed to be on the same system. The remaining techniques are specifically designed to provide effective communication among threads (processes) on a single computer system.

We decided upon shared memory as it incurs the least overhead and provides the most natural data representation and exchange mechanism. It is, however, limited in that it does not directly provide a mechanism by which it can notify threads of updates or new messages. This notification of updates is implicit in pipes and message queues as they are intrinsically message based mechanisms. In order to incorporate a message notification capability in conjunction with the shared memory we also incorporated the use of semaphores. By raising a semaphore in one thread and checking the value of the semaphore in non-blocking mode in the other thread we can determine instantly if data in the shared memory region must be read. These aspects of the architecture can be seen in the architectural diagrams of figures 3 and 4. The shared memory region essentially consists of a structure of the following form:

```
struct svt_interface {
  int eventType;            // What type
of event data must be read
  int button;               // Which
button (if any) was pressed
  int dataParameters[4];    // Pass rele-
vant data parameters

  int year, month, day;     // Pass event
date and time details to the interface
  int hour, minute,second;
};
```

As can be envisioned, upon identification of a raised semaphore, the main environment thread first checks the eventType to determine what type of event has been received. If the primary event type is that of a button press then we must apply the appropriate metaphor, exiting the environment or modifying the execution rate where appropriate. Additionally, since we are incorporating the ability to modify visual attributes dynamically, the environment will need to be able to pass the new variable values back to the main environment. The dataParameters variable renders this service. The time and date variables are used to pass the current date and time of the executing event to the user interface thread. This allows the user interface to visually display the current event time. This is particularly critical when reviewing past history within the environment. Since this data is updated for every display update, the time and date are provided as separate variables that can be monitored constantly by the user-interface thread.

6. SYSTEM IMPLEMENTATION

The environment is implemented in C++ using OpenGL and Tcl/Tk. Compilation is done with g++ under UNIX. Compilation on an MS Windows machine is performed using g++ under the Cygwin environment [13], requiring no changes from a UNIX compilation. As the node locations are dependent on the IP address of the system, IP address and hostname pairs are cached and stored in an external file for quick recovery when the environment is restarted at a later time. This occurs for both successfully identified addresses/hosts and hosts that fail to resolve. This greatly improves initial load performance for known hosts, as the time delay incurred for many name resolution calls is substantial.

C/C++ with OpenGL was chosen over a potential Java implementation due to familiarity with the languages and API's, particularly among the students. Additionally, the capabilities and performance advantages of OpenGL,

particularly at the start of the project, inclined the project towards development under C++ with OpenGL.

On A 1.7 Ghz Pentium 4 with a FireGL 2 card the system generates ~100 frames per second. Each frame consists of ~150 active nodes on the system. This is more than sufficient to adequately display all incoming events even on much slower systems as they arrive without any delay. Therefore, the focus of the implementation is towards providing the system administrator with an effective experience as opposed to focusing on speed and efficiency. If the environment runs too quickly or displays an event for too short a period of time then the system administrator may completely miss an event or not fully absorb the impact of the event and its temporal relationship to other events. On the other hand, when attempting to review large periods of time for identification of temporal activity, the performance of the system is sufficient to review vast periods of time.

7. CONCLUSION AND FUTURE WORK

We have described the software architecture for a visualization environment geared towards aiding real users in the exploration and analysis of security data. The current environment, while geared towards event-based security data, can be applied to any event-based database. In particular, we have applied the environment towards a visual representation of system log data. Other suitable data sources would include: event-based network simulators, traffic control simulators, etc.

The architecture incorporates object-oriented and component-based designs to improve future enhancements and modifications as well as provide selectable capabilities. The need to support real users and associated issues greatly impacted the architectural design.

A multithreaded approach is used to guarantee real-time interaction with immediate feedback for the users. The environment configuration and implementation also provides system independence, ensuring compilation and execution on any available platform. These capabilities ensure superior availability of the system. Finally, synchronization and interpolation of temporal events aid in correct correlation of temporal events by the user.

Much work remains to be done in relation to the visualization environment itself to improve the interactive and visualization capabilities of the environment. This will clearly necessitate changes and improvements to the architecture as well. However, the design of the environment is conducive to such changes.

REFERENCES

[1] T. Mitra and Tzi-Cker Chiueh, “Three-dimensional graphics architecture,” *Current Science*, Vol. 78, No. 7, pp. 838-846, April 2000.

[2] M. Hinsolt, “Graphlet: design and implementation of a graph editor,” *Software - Practice and Experience*, Vol. 30, No. 11, pp. 1303-1324, September 2000.

[3] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, P. Hanrahan, “Rivet: a flexible environment for computer systems visualization,” *Computer Graphics*, Vol. 37, No. 1, pp. 68-73, February 2000.

[4] M.S. Marshall, I Herman, G. Melancon, “An object-oriented design for graph visualization,” *Software - Practice and Experience*, Vol. 31, No. 8, pp. 739-756, July 2001.

[5] OpenGL Architecture Review Board, *OpenGL Reference Manual*, Editors: Renate Kempf and Chris Frazier, Addison-Wesley, 1997.

[6] Robert F. Erbacher and Deborah Frincke, “ Visualization in Detection of Intrusions and Misuse in Large Scale Networks,” *Proceedings of the International Conference on Information Visualization ‘2000*, London, UK, July, 2000, pp. 294-299.

[7] Robert F. Erbacher, Kenneth L. Walker, Deborah A. Frincke, “Intrusion and Misuse Detection in Large-Scale Systems,” *Computer Graphics and Applications*, Vol. 22, No. 1, pp. 38-48, January/February 2002.

[8] M. Barrio and P. De La Fuenta, “Software Architecture: Object vs. Process Approach,” *Proceedings of the 17th International Conference of the Chilean Computer Science Society*, IEEE Press, Valparaiso, Chile, Nov. 1997, pp. 9-15.

[9] Bruce Momjian, *PostgreSQL: Introduction and Concepts*, Addison-Wesley, 2000.

[10] B. Feinstein, G. Matthews, J. White, “The Intrusion Detection Exchange Protocol (IDXP),” Network Working Group, Internet Draft, draft-ietf-idwg-beep-idxp-02,

[11] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, “A component and message-based architectural style for GUI software,” *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June 1996, pp. 390-406.

[12] Edward Angel, *OpenGL: A Primer*, Addison-Wesley, 2002.

[13] Brian W. Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999.

[14] Paul Rademacher, “GUI: A GLUT-Based User Interface Library,” http://www.cs.unc.edu/~rademach/glui/, June 1999.

[15] Brent Welch, *Practical Programming in Tcl and Tk*, Second Edition, Prentice-Hall, 1997.

[16] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1993.