Virtual Window Systems: A New Approach to Supporting Concurrent Heterogeneous Windowing Systems

Rita Pascale, Jeremy Epstein – TRW Systems Division

ABSTRACT

A "virtual window system" (VWS) is a simple model of a window system which can be used to host other more sophisticated window systems. The VWS allows the window systems to share the physical display in a controlled fashion. A VWS is analogous to the virtual machine monitor (VMM) [Madnick74] concept in operating systems, where a single physical computer can run multiple operating systems, each in its own protection domain. Unlike the VMM concept, the window systems supported by the VWS need close cooperation to perform tasks, such as cut and paste between windows of different window systems.

This paper describes the VWS concept, discusses an architecture for a VWS, describes limitations of the VWS concept, discusses some lessons learned from the design and implementation of our prototype, and describes the use of VWSs for various application domains.

Introduction

Users want to run more than one window system (WS) simultaneously on a single platform. The users are a very diverse set whose needs range from debugging to teaching to running a variety of applications. The system developer needs an effective debugging tool for window systems and their applications. The instructor needs a flexible system to teach in as many environments as possible on one machine. The every day user needs to run applications built for more than one window system, a technique which we call "mixed applications".

We propose a general solution to allow many windowing environments to run cooperatively; we refer to it as the "virtual window system" (VWS) concept[1]. A VWS is a simple model of a window system which can be used to host other more sophisticated window systems. The more complicated WSs are considered "guests" on the VWS platform and may be referred to as guest WS. The VWS allows disparate WSs to share the physical display in a controlled fashion and provides a mechanism for communication across WSs.

When compared to similar systems, the VWS is much more versatile. Hybrid WS environments combine two particular systems and provide the ability to run applications from the two specified WSs only. Examples of these hybrid systems are Xvision from VisionWare which combines Microsoft Windows and X; MacX from Macintosh which combines MacOS and X; X-under-NeXTstep from Pencom which combines NeXT and X; and Domain from Apollo which combines Apollo's native WS and X. There are some systems which combine three windowing environments together (i.e., X11/NeWS and SunView), but three seems to be the maximum. By being limited to two or three environments, many of the advantages of the VWS are not possible. The one function these systems fulfill is the ability to run mixed applications and this has been argued as not being a great advantage.

Goals

To meet as many of the various needs of the diverse user group, we need a small, but flexible window system base. To avoid the temptation of building an entire new window system, the size is to be kept small and minimal, using as few primitives as possible. Despite this minimality, we wish to remain flexible. The set of primitives must provide enough functionality to accommodate any window system.

The primitives between the guest WSs and the VWS system can be grouped into connection requests, startup information, input data, and output data. The majority of the primitives deal with changing the display, such as requests to map and unmap windows, update windows, and change the window stacking order. Overall, there are fewer than 20 primitives which is significantly less than the 120 protocol requests in the X server [Scheifler90].

[1]This work is sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA 972-89-C0029.

Details of the Problem

There are a number of problems involved in hosting many environments and protocols on a single platform. The main areas to address are random device accesses and overcoming different protocols.

Access to the keyboard, mouse, console and framebuffer must be regulated. Allowing each WS all input at all times would result in mass confusion since each WS interprets data differently. A mouse click in one environment may pop up a menu, while in another it may cause an application to exit, and in yet another, the data format may not even be valid. Unlimited access to the screen will allow guest WSs[2] to cause chaos by drawing on top of one another's windows, creating a confusing mesh of partial windows.

Another difficulty is cut and paste across WSs. Each system supports a different mechanism through different protocols. A primitive method must be developed that can cut across these various platforms. One drawback of being generic is that unique data formats are not supported. For example, in X [Scheifler90], resource ids (instead of the actual data) can be cut and pasted; this id is only useful within that particular instantiation of the X server and is not meaningful to any other WS process. At the very least, ASCII text can be transferred between all supported WSs, and this is the most common form of data transfer.

[2]By guest window system, we mean a window system environment that is supported and hosted by the virtual window system.

Method

Our solution provides a mechanism to control device access and regulate inter-environment (and intra-environment) communication. The VWS performs these actions through three logical servers: an input manager, an output manager, and a control server. The input manager routes the input to a single designated window system. The output manager displays each window system's output on the screen while handling window overlapping and clipping. The control server is inactive, except for administrative duties. Figure 1 shows the interactions between the VWS and the guest WSs.

Each guest WS must be modified to virtualize its device access. Input is received from the input manager instead of reading the devices directly and drawing is performed in a virtual framebuffer and then sent to the output manager for display. These modifications are necessary since there is no direct access to the hardware devices. The guest WS must also request services of the three VWS servers using minimal primitives. A possible drawback to the virtualized output is that there is no advantage of using intelligent graphics boards unless they can utilize the virtual framebuffers.

In the VWS environment, there is always one active WS which receives all input from the input manager. Any other running system is passive, meaning it can send updates to the screen, but does

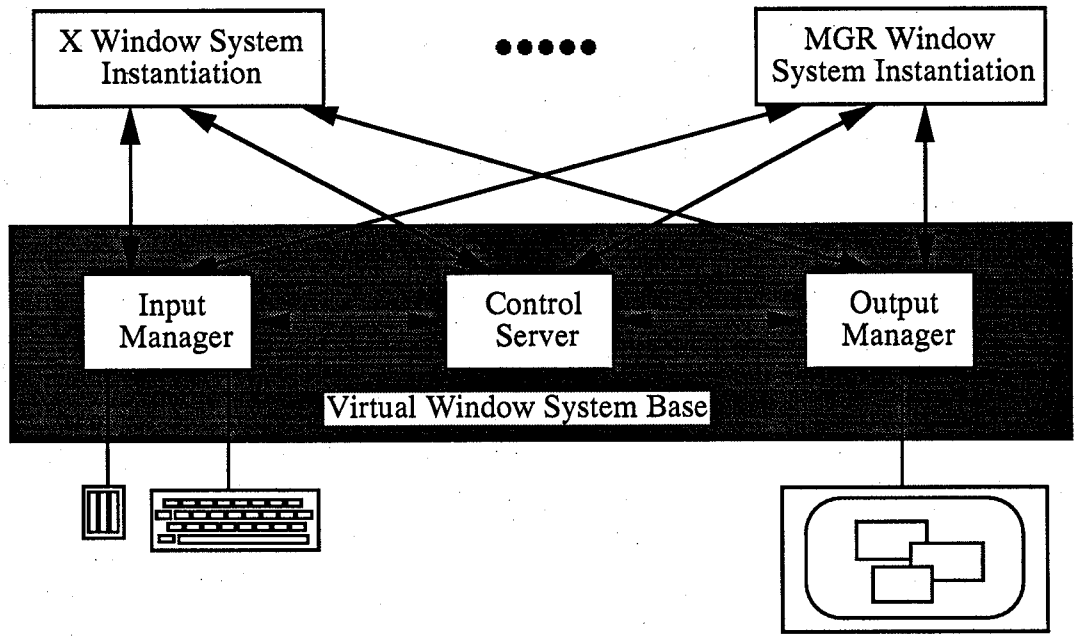

Figure 1: VWS Interactions

not receive any input. Interpretation of the input is the responsibility of the active window system, meaning it is up to the guest WS to send the input events to the appropriate clients and process them as dictated by its own internal protocol. The input manager's only interaction with the input is scanning for the attention sequence which activates the control server. The control server is activated strictly on certain keyboard input, not clicking on an icon. This is because mouse position (at the time of the click) is up to interpretation per guest WS.

The only primitives from the guest WS to the input manager are connection requests and requests to ring the bell[3]. Input manager primitives to the guest WS provide keyboard and mouse input, initialization data, and notification of selection and deselection by the user.

The output server controls the mapping and unmapping of windows from the various WSs as well as handling stacking order, screen updates and cursor imaging. Because no sophisticated graphics operations are provided, the output manager is much simpler than the output component of an ordinary window system. Performance is the cost of this simplicity, but having such base primitives makes the output manager more adoptive of other WSs. A disadvantage of this scheme is that the screen background is not for use by any guest window system. A mechanism is provided to draw helping lines for placing, moving and resizing windows, but other applications that draw directly on the screen background, such as vine and xroach in the X Window System are simply not supported. These applications are generally decorative and were considered expendable.

The following primitives are supported from the guest WS to the output server:

- connection request
- window map and unmap requests
- cursor position change
- cursor image change
- window update
- raise and lower window requests
- draw dashed box request (for placing, moving and resizing windows)
- load a colormap

The output server primitives to the guests WSs provide initialization data and window map acknowledgements; other requests do not require replies.

The control server starts new WSs, switches between WSs, and provides cut and paste operations between WSs. Cut and paste is the only interaction between windowing environments. The control server provides no interpretation of the data to be pasted. At a minimum, guest WSs must support a canonical text format for interoperability. Additional formats may be supported, but their interoperability is less likely. This may be a disadvantage to systems that have unique data forms, such as resource ids in X.

The primitives sent from the guest WS to the control server are connection requests, cut data and paste requests. The control server responds to the paste requests with the most recent cut data that meets the criteria specified by the guest WS.

With minimal changes, existing WSs can be modified to work within the VWS environment. How closely the window system's implementation is tied to the hardware dictates the amount of change. X is very modular and encapsulates its device usage; therefore, it was quite simple to change. We modified the MIT X11R4 server for Sun hardware to accept input from the input server and display output through the output manager with a few hundred lines of code. Modifying the Macintosh WS would be much more complicated because of its close relationship with its hardware, but we believe that even this can be overcome.

Details on VWS Uses

The VWS exhibits great versatility in its ability to handle a wide variety of needs in a minimal amount of code. VWS neither enhances nor detracts from the given graphical user interface; if the guest WS is poor, it will remain that way. Below we further explain the uses of the VWS.

Debugging Tool

The VWS can be applied to paradigms such as debugging WSs and applications. New versions of the same window system can be tested under the VWS environment. For example, X11R4 can be run at the same time as X11R5 and differences in capabilities of both can be monitored simultaneously. In the same light, the same WS can be run while testing different versions of client applications. For example, one could start three X11R4 servers, and test a different window manager on each. We are able to run the OSF/Motif mwm, OPEN LOOK olwm, and MIT twm window managers simultaneously; each is connected to a different instance of the X server.

Another advantage of the VWS testing environment is that resource grabs are limited to the environment that they were initiated in. In X, a client can "grab" exclusive access to any and all devices, including the server itself. In the VWS system, these grabs are limited to the instantiation of the server that requested the lock. This allows the developer an easier way out of a potential deadlock situation.

[3]Bell ringing is of course an output function, but on our Sun system the bell is part of the keyboard, so we put this function in the input manager.

The VWS is an effective way to test graphical user interfaces; however, performance benchmarking results have no meaning in this environment since all processes are ultimately sharing the same CPU and job scheduler. The main testing advantage is varying visual results due to different configurations and avoidance of detrimental server hangs.

Teaching Tool

Because of its ability to run various environments and configurations, the VWS can be used as a teaching tool for all of those environments rather than requiring one machine for each platform. For comparisons, the VWS can display X11R4 and X11R5 servers and their various applications. Differences are manifested in a more memorable way when they are captured on a single display. The same goes for varying window managers.

Currently, only three window systems are running on our prototype VWS. They are X11R4, X11R5 and Bellcore's MGR[4] With the addition of Macintosh windows, Microsoft Windows and Presentation Manager, this system would be a very useful and inexpensive teaching system. Rather than buying three or four machines, one platform would suffice for all requirements.

Mixed Applications

In today's computing environment there are a half-dozen competing "standard" window systems; X, Macintosh, Microsoft Windows, Presentation Manager, and SunView chief among them. VWS allows users to run applications built for more than one window system simultaneously. We can run editres on X11R5, Motif on X11R4, and spot (a pointer tracking program) on MGR. These programs do not run as well (if at all) on the other systems mentioned. X11R4 did not have editres at the time of its release; X11R5 does not support Motif unless it was compiled with the backward compatibility flag; spot is an MGR specific application which does not exist for X11R4 or X11R5.

Again, once other WSs are integrated, the system will become much more useful. For example, with Macintosh and SunView integrated, a software developer could use a document processing program developed for a Macintosh while using development tools designed for the X Window System and SunView system.

Secure Window Systems

Our specific implementation of the VWS is for a highly-secure multi-level window system [Epstein91]. It was developed on a Sun 3/60 running TMach, a prototype trusted operating system based on Mach 2.5. A secure environment is achieved by running multiple instantiations of the X WS and the MGR WS; one at each security level to be displayed on the screen (e.g., one server controls all secret windows on the screen, a different server handles top secret windows). The VWS keeps the workspaces separate and allows cut and paste according to the security policy implemented. This architecture gives us a high degree of trust without relying on the correct functioning of a large and complex window system such as X.

Another advantage of this highly flexible secure VWS base is that the guest WS is entirely untrusted. This means that any window system that can be run on the VWS can be used in a secure environment without having to go through the extensive process of accreditation.

Multi-Processor Environments

Because the VWS consists of several cooperating processes (input manager, output manager, control server, and guest WSs), it is able to use multiple processors without additional investment. For example, the X server could run on one processor while the MGR server runs on a different processor.

Results

The design and implementation of the VWS took two man years. Modification of the X11R4 server required less than four months for a junior programmer to incorporate the necessary changes. Upon the release of X11R5, modifications were adapted in less than a week. We adopted Bellcore's MGR window system to the VWS with less than a month. Using ports and multi-threading, fundamental aspects of Mach-like operating systems, attributed to the bulk of the modified code. Much of the low level communication code in the X WS had to be rewritten to use ports; using an operating system that supported sockets would have saved implementation time.

Runtime improvements can be made by using faster hardware and implementing the VWS in an environment that supported sockets (rather than ports as in TMach) and shared memory. Other benefits to using sockets over ports is that sockets can be prioritized. Without this ability, previously simple processes had to become multi-threaded to force prioritization on port message retrieval. For example, our X server has one thread per client in addition to several control threads. While multi-threading has its benefits in allowing many activities to occur at once, there is a performance cost in context switching, and a general overhead burden. Despite our poor hardware configuration, the system is not intolerable. With some enhancements, including those mentioned above, performance would be greatly improved.

[4]MGR [Uhler88] is a freely available window system. It is far less flexible than X, but it is faster and smaller.

Running benchmarks on the VWS for comparisons with unchanged WSs has proven to be much more difficult than expected. Currently we have some preliminary results from x11perf that show the X11R4 server running under VWS yields 50 to 75 percent of the runtime speeds compared to the X11R4 server running directly on the hardware. Window manipulations (such as raises, lowers, circulates, maps, and unmaps) performed comparably whereas graphics operations did not perform as well. In the graphics area, the VWS system compared best on tests of direct copies rather than stippled patterns[5]. Also, simpler requests like lines and rectangles performed significantly better than more complex shapes like circles and ellipses.

Our VWS implementation is less than 20,000 lines of heavily commented C code (about 6,000 statements). A significant fraction of that is due to security requirements. By contrast, an X11R4 server and Motif window manager total about 400,000 lines of code, including support libraries.

Our implementation can be improved with faster machines, hardware that can handle many framebuffers, tuned operating systems as well as a number of other things. Despite the added overhead, the performance is acceptable for the typical user. The more intensive the load, the more the performance will downgrade.

Conclusion

The architecture and implementation of the VWS system has achieved our goals of minimality and flexibility. As a proof of concept experiment, the notion of a VWS has proven itself useful for building trusted window systems. We feel it is applicable in other problem domains as well, and offers significant advantages over alternate architectures. There are limitations to our implementation, but may be acceptable, given the payoffs of being able to operate in a heterogeneous environment. Also, some of our limitations are specific to security. If shared data between WSs is allowed, as is the case in unsecure systems, the memory usage and performance would improve immensely.

Future work includes research into different hardware, different operating systems and more guest window systems.

References

[Madnick74] *Operating Systems*, Stuart Madnick and John Donovan, McGraw Hill, 1974.

[Scheifler90] *X Window System*, Second Edition, Robert Scheifler and James Gettys, Digital Press, 1990.

[Uhler88] Stephen Uhler, *MGR – C Language Application Interface*, Bell Communications Research, 1988.

[Epstein91] Jeremy Epstein, et. al., "A Prototype B3 Trusted X Window System", published in *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio TX, December 1991.

[5]Stippled patterns are as stencils to indicate where to draw and where not to draw.

Availability

A series of technical papers are available from the authors on our VMS implementation to support a highly trusted version of X. The software itself is not available at this time.

Author Information

Rita Pascale is a Programmer on TRW's Advanced Computing Systems project building a highly trusted version of the X Window System. She holds a B.S. in Computer Science from Virginia Tech. Her U.S. Mail address is 1 Federal Systems Park Drive, Fairfax VA 22033. She can be reached electronically at pascale@trwacs.fp.trw.com.

Jeremy Epstein is the Lead Engineer on TRW's Advanced Computing Systems project building. In his previous life, he was a lead engineer with Addamax developing trusted UNIX systems. Jeremy has been working with UNIX since Version 6, and still refuses to use "csh". He holds a B.S. in Computer Science from New Mexico Tech, M.S. in Computer Sciences from Purdue University, and is working on a Ph.D. in Information Technology at George Mason University. His U.S. Mail address is 1 Federal Systems Park Drive, Fairfax VA 22033. He can be reached electronically at epstein@trwacs.fp.trw.com.