Implementing Remote Display on Commodity Operating Systems

Lei Zhang

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in the Department of Computer Science

COLUMBIA UNIVERSITY

January, 2006

Acknowledgments

I would like to thank my advisor Jason Nieh. His foresight in research and wealth of expertise in computer systems guided me throughout this work. This thesis would not have been possible without his encouragement, wisdom, feedback, and support.

I owe particular thanks to Ricardo Baratto, one of Professor Nieh's PhD students. Ricardo shared with me his experiences in designing and implementing THINC on X/Linux. He also contributed his portable techniques of command queue maintenance and command delivery to my implementation of THINC on Windows. This work would not have succeeded without his help.

I would also like to thank Steve Millman, who provided resources for the memory sharing technique on Windows, and Pinxing Ye, who participated in the development of the client end of THINC on Windows.

Abstract

Remote display provides a number of benefits, including ubiquitous access, thin-client computing, remote assistance and secure central management. In practice, it is crucial that remote display systems effectively support existing unmodified applications on commodity operating systems. However, commodity operating system environments have very different display architectures which impact how remote display can be implemented. This dissertation discusses the details of deploying remote display on commodity operating systems, including Linux(with X Window System) and Microsoft Windows. We focus on the implementation of a remote display system on Microsoft Windows and introduce several techniques, including kernel-user mode communication, a high throughput command channel, and efficient synchronization. With these techniques, we successfully implement an asynchronous remote display system on top of a synchronous local display system. Our work extends THINC, a virtual display architecture for remote display, from an X/Linux-only architecture to one across X/Linux and Windows. We present the detailed algorithm we used for supporting a portable command processing pipeline across multiple commodity operating systems. Our experiment results demonstrate that our approach can achieve similar remote display performance on both X/Linux and Windows.

Contents

Acknowledgments **iii**

Abstract **iv**

List of Figures **iii**

Chapter 1 Introduction **1**

Chapter 2 Background and Framework of Remote Display **3**

2.1 The Display System of Microsoft Windows 4

2.2 The Display System of X Window System 7

2.3 Remote Display Based on Existing Display Systems 8

Chapter 3 Protocol and Architecture **10**

3.1 Remote Display Protocol 10

3.2 Interception and Display Virtualization 11

3.3 Asynchronous vs. Synchronous 12

3.4 Command Delivery 12

3.5 Kernel mode vs. User mode 13

3.6 User Inputs 14

3.7 Putting Them All Together 14

Chapter 4 Intermediate Commands **16**

4.1 Copy on Delivery 17

4.2 Intermediate Command Channel 18

Chapter 5 Signaling and Synchronization **20**

5.1 Signaling and Polling 20

5.2 Synchronization 22

Chapter 6 Implementation Details **24**

6.1 The Virtual Display Driver 24

6.1.1 Display Commands and Intercepting 24

6.1.2 Mirror Driver 25

6.2 Command Queues 26

6.3 The Service Thread On Windows 29

6.4 Memory Sharing 32

Chapter 7 Experimental Results **33**

7.1 Experimental Setup 33

7.2 Application Benchmarks 34

7.3 Measurements 35

Chapter 8 Related Work **39**

Chapter 9 Conclusions **42**

List of Figures

2.1 Windows Display System Architecture 5
2.2 Graphics Driver and GDI Interaction 6
2.3 X Window System Architecture 8

3.1 Remote Display Architecture on Windows 14

7.1 Experimental Testbed 34
7.2 Web Benchmark: Average Page Latency 36
7.3 Web Benchmark: Per Page Command Numbers 37
7.4 Web Benchmark: Average Page Data Transferred 37
7.5 Average Number of Per Page Commands 38

Chapter 1

Introduction

Continuing advances in network speed, cost, and ubiquity are transforming the traditional computer from one whose components are locally isolated in a box to one whose subsystems are physically distributed across a network and decoupled from the local computer. Examples of this phenomenon include the widespread use of network attached storage and storage area networks in lieu of local disks for storing and managing data, the adoption of grid computing to distribute computations across the network and make use of available computing power beyond the local desktop, and the growing popularity of remote display technologies for enabling geographically remote users to interact with the graphical user interface of a computer over the network as though they are sitting in front of the desktop.

A remote display system consists of a server and a client that communicate over a network using a remote display protocol. The protocol allows graphical displays to be virtualized and served across a network to a client device, while application logic is executed on the server. Using the remote display protocol, the client transmits user input to the server, and the server returns screen updates of the user interface of the applications to the client.

Remote display can provide many potential benefits, including enabling remote graphical access to a user's computer from any network device, providing scientists with full graphical access to use computer-controlled scientific instrumentation in remote geographic regions from the convenience of their own offices, supporting screen sharing of the same desktop by multiple users to support online collaboration, giving technical support personnel the exact same view of the computer desktop as the end user to troubleshoot and fix computer problems, and moving application state off the desktop back into the machine room such that clients only need to serve as simple remote display devices, thereby reducing IT management complexity via the centralization benefits that come from a thin-client computing approach.

While many of the concepts behind remote display have been discussed in detail in previous work, little attention has been given to understanding how to actually implement it to function correctly and perform effectively in practice. In particular, for remote display systems to be successfully deployed, it is crucial that they effectively support existing unmodified applications on commodity operating systems. However, commodity operating system environments such as

Linux and Windows have very different display architectures which impact how remote display can be implemented in these systems.

We describe our experiences in implementing a remote display system across both X/Linux and Windows operating system environments. We discuss the differences in the display system architectures of these widely used systems and consider how these differences impact the implementation of remote display. The differences include X versus Windows application display command primitives, kernel versus user-level display subsystems, and synchronous versus asynchronous display mechanisms. The implementation is based on THINC [8], a virtual display architecture for high-performance remote display. Because of its display virtualization approach, THINC can fit into the display architectures of most commodity operating systems without modifying the original systems. The previous implementation of THINC on X/Linux outperforms most commercial remote display systems [8]. In this dissertation, we extend the THINC prototype from an X/Linux-based remote display system to one that crosses X/Linux and Windows, and implement an asynchronous remote display system on top of the synchronous local display system of Microsoft Windows. We introduce techniques, including kernel-user mode communication, the high throughput command channel and efficient synchronization. We also present the details of some other portable techniques, such as off-screen drawing awareness and multiple command queues maintenance. Our experiment results show that remote display systems on X/Linux and Windows can achieve similar performance for web applications, based on the THINC architecture.

This dissertation is organized as follows. Chapter 2 presents background of existing display system and the basic framework of a remote display system. Chapter 3 presents an overview of the system architecture and the remote display protocol. Chapter 4 and 5 discuss the special techniques for the implementation on Windows. Chapter 6 covers the details of implementation and the algorithm for intermediate command processing. Chapter 7 demonstrates experimental results. Finally, we discuss some related work and present concluding remarks.

Chapter 2

Background and Framework of Remote Display

A remote display system should ensure that existing applications working correctly. Large numbers of applications running on each existing operating system. Two of the most widely used commodity operating systems, Microsoft Windows [1] and Linux, have both been developed for a long time with considerably stable graphic APIs that millions of applications rely on. Their users are unwilling to move to other systems if they cannot use the applications that they are familiar with. Therefore, deploying remote display on commodity operating systems should not have any changes to the usage of applications.

To migrate applications to a new system with no changes to application usage, modification to the underlying part of applications is usually required. Once an application is compiled to binaries, even a small change to the system interface or the behalves of APIs can break the application easily. Building a new system from scratch may provide excellent remote display, but it is very difficult to guarantee that the new system works seamlessly with all existing application binaries. It is also impossible to have all applications modified to fit into the new system. Moreover, the development cycle can last for years before a new system can be ready for normal users.

Due to the disadvantage of building a new system, leveraging existing display systems becomes an appealing option in implementing remote display. The display can be intercepted and delivered to clients. This approach eases the implementation of remote display and make no modification to applications become possible.

Modern display systems usually consist of four layers, from top to bottom: applications, graphic/window systems, display drivers and framebuffers. A display driver is simply a set of rendering functions, each of which represents a rendering operation. Graphic libraries, as part to the graphic system, is the interface to applications. Applications call APIs in the graphic libraries to request drawing actions. The graphic system performs these requests in the form of calling

[1] We refer to NT-based Windows, including Windows 2000, Windows XP and Windows Server 2003, when we talk about Microsoft Windows in this dissertation.

rendering functions in the display driver, which then renders on the hardware framebuffer. The pixel data in the framebuffer will eventually go to the monitor. Although this is a common structure of display systems, either a *synchronous model* or an *asynchronous model* can be applied. In addition, interfaces at each layer can be very different between systems. We will look into the display systems of Microsoft Windows and X/Linux before discussing how to implement remote display.

2.1 The Display System of Microsoft Windows

The display system of Microsoft Windows has the four layer structure previously discussed, as show in Figure 2.1. Its graphic system crosses kernel mode and user mode. The user mode part consists of two dynamic-link libraries, Gdi32.dll and User32.dll. These two dynamic-link libraries are loaded by applications during runtime. Applications call GDI (Graphic Device Interface) functions for rendering and call USER functions to create user interface controls, such as windows and buttons, on the display. The kernel mode component of graphic system is a driver called Win32k.sys. It includes the window manager and the kernel mode GDI. Requests from both Gdi32.dll and User32.dll are eventually routed to the kernel mode GDI. The window manager also collects input from keyboard, mouse and other devices and passes user messages up to applications. The kernel mode GDI is also referred as the GDI engine or graphic engine. It interprets application requests for graphic output and sends the requests to display drivers [2]. The user mode GDI library is a standard interface for applications to use various graphics output devices [30].

In Windows, only two interfaces are published in Microsoft's documents. The first one is between applications and GDI/USER libraries. It includes the rendering functions in GDI and windows manipulation functions in USER. The second one is between the GDI engine and display device drivers. The internals of the graphic system, including the user mode libraries and the kernel mode components, are not public.

There are several hundred GDI functions [27], including a rich set of *graphic primitives* and display management functions that relate to *device contexts* or *GDI objects*. The graphic primitives provide the service of drawing lines and curves, filling an area, displaying bitmaps and outputting text. The USER library provides more functions at higher levels for managing windows, user message queues, multiple document views and user interface timers [1]. At this interface, most function calls happen in the same direction. Applications calls functions in the libraries to obtain services. The libraries will not call into the applications unless applications register call back functions.

The interface between the kernel mode GDI engine and graphics drivers[3] is bi-direction. The number of commands at this interface are much fewer than that of those at the first interface. Figure 2.2 shows the graphics driver and GDI interaction from a driver's perspective [21]. GDI communicates with the graphics driver through a set of *graphics device driver interface (GDDI) functions*. These functions are identified by their *Drv* prefix. Information is passed between GDI

[2] In Windows Vista, the display architecture has been changed. A portion of display device driver has been moved to user mode. The model no longer uses services of the GDI engine, but uses services of the Microsoft Direct3D runtime and Microsoft DirectX graphics kernel subsystem. This change is beyond the discussion of this thesis.

[3] In this thesis, both graphic driver and display driver refer to the driver for the display hardware.

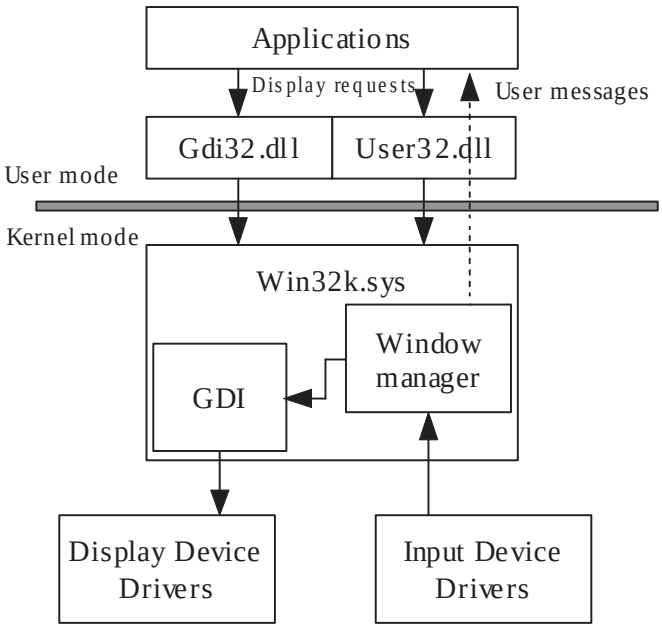

Figure 2.1: **Windows Display System Architecture**

and the driver through the input/output parameters of these functions. The display driver must support certain *DrvXxx* functions for GDI to call. The driver supports GDI's requests by performing the appropriate operations on its associated hardware before returning to GDI. GDI also exports service functions to drivers, including standard rendering functions and limited kernel services. They are identified by their *Eng* prefix.

The GDDI functions include nine required functions, four conditional required functions and thirty optional functions, as shown in Table 2.1. The first two types of GDDI functions are normally needed to be implemented and the third type are mostly for optimization purpose [22]. Based on our experience, however, some optional GDDI functions must be implemented to get a complete screen image. They are *DrvAlphaBlend, DrvGradientFill, DrvStretchBlt, DrvLineTo* and *DrvTransparentBlt*.

The display driver is stateless. Although the state of current display device are created or changed by the driver, they are kept track of by the GDI engine. For example, the GDI engine will call the *DrvEnablePDEV* function to create a structure that describes the state of display device. The *DrvEnablePDEV* function returns the pointer to this structure to the GDI engine and this pointer will be pass from the GDI engine to a driver function everything when GDI engine requests a drawing operations on this display device.

The display system of Windows uses a synchronous model and the graphic system is a static component of the operating system. When an application calls a graphic API, it calls through the graphic system to the driver functions. The driver rendering functions are executed synchronously in an application's process context. The graphic system usually translates a graphic API into multiple functions, because library APIs are at a higher level than that of driver functions. The

Type	GDDI function names
Required Functions	*DrvEnableDriver, DrvAssertMode, DrvCompletePDEV, DrvDisableDriver, DrvDisablePDEV, DrvDisableSurface, DrvEnablePDEV, DrvEnableSurface, DrvGetModes*
Conditional Required Functions	*DrvCopyBits, DrvStrokePath, DrvTextOut, DrvSetPalette*
Optional Functions	**Bitmap Management Functions**: *DrvCreateDeviceBitmap and DrvDeleteDeviceBitmap* **Drawing Functions**: *DrvAlphaBlend, DrvBitBlt, DrvDitherColor, DrvFillPath, DrvGradientFill, DrvLineTo, DrvPlgBlt, DrvRealizeBrush, DrvStretchBlt, DrvStretchBltROP, DrvStrokeAndFillPath, DrvTransparentBlt* **Image Color Management Functions**: *DrvIcmCheckBitmapBits, DrvIcmCreateColorTransform, DrvIcmDeleteColorTransform, DrvIcmSetDeviceGammaRamp* **Pointer and Window Management Functions**: *DrvDescribePixelFormat, DrvMovePointer, DrvSaveScreenBits, DrvSetPixelFormat, DrvSetPointerShape* **Miscellaneous Functions**: *DrvDestroyFont, DrvDrawEscape, DrvEscape, DrvFree, DrvNotify, DrvSynchronize, DrvSynchronizeSurface*

Table 2.1: GDDI functions

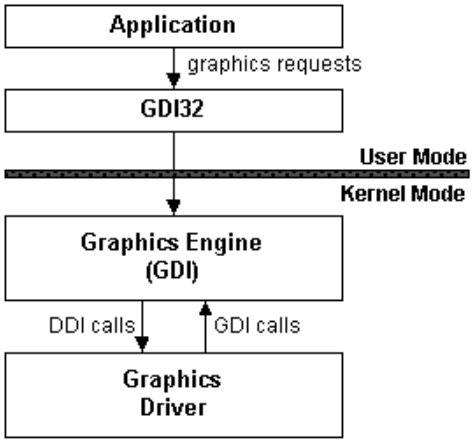

Figure 2.2: **Graphics Driver and GDI Interaction**

number of driver functions depends on the status of the current desktop and other applications. The application is blocked[4] until all these driver functions are finished. This synchronous model improves the performance of local display because it eliminates the overhead of batching and sending drawing requests. Context switch between applications and the dedicated drawing processes is also avoided.

[4]Actually, only the thread that calls the API get blocked.

2.2 The Display System of X Window System

With the identical four layer stack, X presents a different architecture compared to the display system of Microsoft Windows. In Windows, requests from applications are routed from graphic libraries to the graphic system by in the form of function calls. In X, however, requests are sent as packets through sockets. The graphic system is a dynamic component – the *X Server* process.

As showed in Figure 2.3, the X server is a process which acts as part of the graphic system of X. Applications are referred to as *X clients*. The graphic system exposes itself to applications by the X library. As documented in [2], there are more than one thousand library functions. When an application calls the X library functions, the X library sends requests to the X server through sockets. On receiving those requests, the X server will call the rendering functions in the display driver to perform the drawing. On the other hand, the X server also collects user input from input devices and sends them over to X applications through the same sockets. If a X server and its applications are on different machine, the sockets are normal network sockets. If they are one the same machine, the socket connections are just buffers in the local memory. Unlike Windows, the window manager of X is an "application" that controls the placement and appearance of windows under the X Window System. When a window manager is running, some kind of interaction between the X server and its clients is redirected through the window manager. In particular, whenever an attempt to show a new window is made, this request is redirected to the window manager, which decides the initial position of the window [36]. Displaying windows and clipping overlapped windows are still done by the X server.

The X server consists of four pieces: Device Independent (DIX) layer, Operating System (OS) layer, Device Dependent (DDX) layer and Extension Interface [6]. In an implementation of X server, the DIX layer should be identical to all operating systems and devices. The OS layer is different for each operating system but is shared among all graphics devices for this operating system. The code for DDX layer is different for each combination of operating systems and graphic devices. These layers are usually unchanged while the OS layer and DDX layer provides the interface for device drivers to hook it routines.

The X server is typically a single thread process. The heard of the DIX layer is a loop called *dispatch loop*. Each time X server goes around the loop, it sends over accumulated input events to applications and processes requests from applications. The dispatch loop creates the illusion of multitasking among applications. If there is too much requests from the one application, the X server serves only part of them before shifting to another applications.

Whenever there are no inputs from users, requests from applications or new connections from new applications, the X server will turn into a waiting state. It happens at each iteration of the dispatch loop. Before it starts waiting, the OS layer calls the per-screen[5] *BlockHandler*. The per-screen *WaitupHandler* is called after X server terminates the waiting state. These two routines are registered by display drivers. Besides, display drivers also register routines of drawing, window management and graphic context manipulation with the DDX layer. The drivers can either implement these routines by

[5] X server supports multiple screens. Each screen registers its own handler functions.

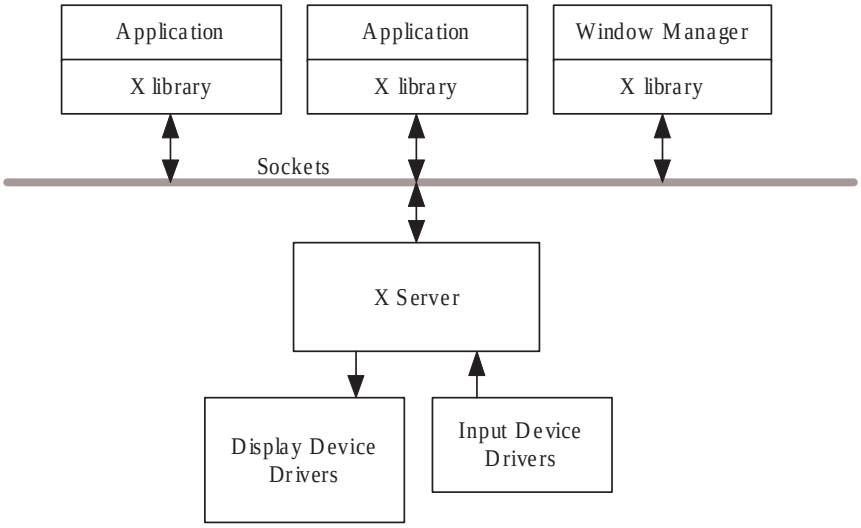

Figure 2.3: **X Window System Architecture**

themselves or calling standard functions provided by X.

Because of the X server and applications are separated in different processes, X Window employs an *asynchronous model*, where, in most scenario, applications do not wait for the X server to finish a request unless the X server cannot process requests fast enough and the buffer is full. Once a request is sent over the socket, the application is unblocked. If the application and the X server locate on the same machine, the socket connection become a local buffer in the system memory. Applications will not be suffered from any network delay.

2.3 Remote Display Based on Existing Display Systems

In order to leverage the existing display systems, we must embed our system into a certain place of the four layer stack, where we are able to intercept display information. That should be a layer where sufficient semantic information of drawing is preserved and the changes at this layer does not lead to any modification to application logic or binaries. Because the intercepted display information will be processed and delivered to clients. We must guarantee the efficiency of processing and delivery, and minimize the overhead that could potentially impact the application and system performance.

The graphic system is too complicated to be changed. It is unrealistic to replace the graphic system or the graphic library with our own pieces. That costs the re-implementation of the sophisticated graphic system. Therefore, the display must be decoupled at the layer lower than graphic systems. The framebuffer is the place where raw pixel data is stored. Reading pixel data from the framebuffer does not affect applications at all and is easy to implement. However, no semantic information of drawing is preserved in the framebuffer and delivering raw pixel is inefficient [24]. The display driver layer is well-abstracted and open to third party. In both Microsoft Windows and X/Linux, the number of driver functions are

much smaller than that of library functions. Rendering operations at this layer has been optimized by graphic systems. Semantic information at this layer is sufficient to implement efficient remote display. Writing a new display driver involves much less work than re-implementing the graphic system. Moreover, the graphic system separates the driver from applications, which make no modification to applications become possible. Therefore, the display driver layer is the appropriate place where we can intercept display from applications.

Processing and delivering the intercepted display information should be performed at the right time and in the right place. In most applications, the computation logic and the drawing operations are tightly coupled. Intercepting has to be in display drivers, or in particular, in rendering functions. However, it is more flexible for display processing and delivery. They should interfere the application execution as less as possible. More details about these are discussed in later chapters.

Remote display protocol is another important issue. Clients on different platforms provide ubiquitous access to the servers. The command set in the protocol should be natively supported on various system so that we can achieve a good performance with a simple client. These commands should be able to be translated from the drawing actions on different platform, without much difficulty. We will discuss the protocol in Chapter 3.

We are also able to maintain the same code base of command processing and delivery on X/Linux and Windows. Display intercepting that occurs in the display driver has to be platform dependent but it works seamless with the portable part of the system by taking the advantage of intermediate commands. We do not sacrifice performance for the ability of cross-platform. Instead, the intermediate commands introduces several benefits to improve the system. In this thesis, we continue to discuss every details of implementing these frameworks in practice. More issues are involved, such as delivering timing, kernel mode vs. user mode, synchronization problems and etc. They are presented in the following chapters.

Chapter 3

Protocol and Architecture

To deploy our remote display system on Windows without changing the system, we employ THINC [8], a high performance remote display architecture which requires no modification to operating systems and applications. It leverages the virtual display technique to intercept semantic drawing information from original display systems.

Several efficient techniques that have been developed on X/Linux, including off-screen drawing awareness and multiple command queues maintenance, can be easily ported to the Windows platform. Moreover, by introducing new techniques for the Windows platform, we extend the THINC prototype from an X/Linux-based system to one that crosses X/Linux and Windows, which have very different display systems. This is a strong evidence that THINC is a portable architecture across platforms.

Besides the interaction with the local display systems, the remote display protocol is also a critical part of the system. It should not cause much overhead in translating rendering operations on different platforms into our remote display commands. In addition, these commands should be ubiquitously supported, simple to implement, and easily portable to a range of client environments. THINC protocol fulfills this requirement. It eases the implemenation on both the server and the client sides.

3.1 Remote Display Protocol

THINC uses a small set of low-level display commands for encoding display updates [8], inspired by the core commands originally used in Sun Ray [33]. The five commands used in the protocol are listed in Table 3.1. They mimic operations commonly found in client display hardware and represent a subset of operations accelerated by most graphics subsystems. Graphics acceleration interfaces such as XAA and KAA for X and Microsoft Windows' GDI/GDI+ use a set of operations which can be synthesized using these commands. In this manner, clients need only translate protocol commands into hardware calls, and servers avoid the need to do full translation to actual pixel data, reducing display processing latency.

Command	Description
RAW	Display raw pixel data at a given location
COPY	Copy frame buffer area to specified coordinates
SFILL	Fill an area with a given pixel color value
PFILL	Tile an area with a given pixel pattern
BITMAP	Fill a region using a bitmap image

Table 3.1: Remote Display Commands

These display commands are as follows. RAW is used to transmit unencoded pixel data to be displayed verbatim on a region of the screen. This command is invoked as a last resort if the server is unable to employ any other command, and it is the only command that may be compressed to mitigate its impact on the network. COPY instructs the client to copy a region of the screen from its local framebuffer to another location. This command improves the user experience by accelerating scrolling and opaque window movement without having to resend screen data from the server. SFILL, PFILL, and BITMAP are commands that paint a fixed-size region on the screen. They are useful for accelerating the display of solid window backgrounds, desktop patterns, backgrounds of web pages, text drawing, and certain operations in graphics manipulation programs. SFILL fills a sizable region on the screen with a single color. PFILL replicates a tile over a screen region. BITMAP performs a fill using a bitmap of ones and zeros as a stipple to apply a foreground and background color.

3.2 Interception and Display Virtualization

Intercepting display at the driver layer implies the demand of spying on a normal display driver or implementing a special driver. However, spying is difficult unless existing systems provide interfaces to monitor the display. Normal display drivers are highly hardware specified and it is hard to interpose on them. Even on X/Linux, it is not an easy job to modify and re-compile open-sourced drivers for all kinds of display devices. As remote display systems direct the display to clients, it is not necessary for servers to display the desktop on their monitors. Therefore, the display devices on the server can be virtualized. The “special driver” can be a virtual display driver. It acts like a normal one to the graphic system, but it never talks to any hardware. The graphic system is unaware of this and thinks it is still rendering on the hardware. In the virtual display driver, normal rendering functions are replaced with virtual rendering functions, inside which the display are intercepted. This approach is called *display virtualization*. It simplifies the implementation of the remote display system because the virtual display driver gets rid of all hardware operations and hence completely executes in the memory at a higher speed. In addition, both Windows display system and X Window System provide the mechanism that display drivers can call the standard rendering functions in the graphic system to perform the drawing.

3.3 Asynchronous vs. Synchronous

As we discussed in Chapter 2, the execution of applications and rendering functions in the driver can be asynchronous or synchronous. In most applications, the computation logic and the drawing operations are tightly coupled. Any delay in the drawing operation will slow down the applications. In Windows, which employs the synchronous model, this delay is roughly the running time of rendering functions in the display driver. This delay is very short because the rendering functions are fast. In X/Linux, where the asynchronous applies, this delay is the time to send out the request over network or buffer the request locally. If the applications and the X server local on different machines within a high-speed LAN or on the same machines. This delay is very short too.

Display interception has to be done in the virtual rendering functions. With such a virtual display driver, the applications in the asynchronous model does not get any impact. However, in the synchronous model, the applications can potentially get slowed down. Therefore, the virtual driver translates the drawing operations into intermediate commands as quickly as possible and defer them for later processing. Because the translation is fast enough that the impact to application performance is negligible. This lead to the asynchronous relationship between application execution and intermediate command processing and delivery. On X/Linux, this relationship already exists even if the intermediate commands are processed and delivered within the virtual rendering functions. In fact, the command delivery on X/Linux is also deferred in our implementation.

3.4 Command Delivery

The display on the server ultimately becomes a stream of remote display commands that are transferred over the network. Although it is possible on some operating systems, such as X/Linux, to process commands in virtual rendering functions, we do not do this for command delivery on either X/Linux or Windows.

Each rendering function at the driver layer represents a platform dependent drawing action. They have been optimized by the graphic system for local display, but not for remote display. A rendering function might change only one pixel. For example, on Windows, drawing a dotted line on the screen will becomes a large number of actions, each of which just renders one pixel on the screen. This would hurt remote display performance if we deliver one command for one action, because the overhead in command headers and network operations. In this case, these native actions must be aggregated into one command before being sent. In addition, local display system does not worry about overwriting, which usually happens when a window or a web page is drawn. However, we do not want to deliver pixel data that will be immediately overwritten and should clip the overwritten part in commands.

These two optimization cannot be applied to a single command. Preceding or succeeding commands are required. Therefore, the system needs to cache the commands for a while so that optimizations can be applied to related commands at the same time. As a result, remote display commands are not delivered immediately after they are generated inside

rendering functions. The optimizations are applied to intermediate commands in the phase of command processing. A later command can lead to changes to previous commands, as long as those commands have not been delivered.

However, commands must be delivered without significant delay, because users are waiting for the display updates. Command delivery should be triggered at a proper time. We either hook the delivery into the local display system or launch a thread to deliver commands.

The X Server is single threaded program which runs an infinite loop at most of the time. In the loop, it serves the requests from applications by calling proper rendering functions in the driver. At the end of each iteration, it calls a per-screen BlockHandler, and then goes to sleep, waiting for the new requests. This BlockHandler is usually for deferred work or wrapping up operations. When requests from applications are in large numbers, the X server calls the BlockHandler before it finishes all requests and restart the loop again. Therefore, the BlockHandler is the right place to deliver commands because the X Server does not delay the deferred or wrapping up drawing actions for too long. This delay is for local display and therefore will not be long for remote display.

Windows, however, does not provide this feature. It completes each request from applications synchronously. Therefore, we launch a dedicated service thread to deliver commands periodically. As we discussed above, our command processing on Windows is deferred. This service thread is also the right one to process deferred commands. It sleeps for a very short time and wakes up to check whether there are new commands. If not, it goes back to sleep. Otherwise, it processes commands, delivers them to clients and then go to sleep again. We discuss more details about this service thread in later chapters.

3.5 Kernel mode vs. User mode

There is a good reason to implement the remote display on X/Linux in the approach of display virtualization. The X Server is a user mode process and the 2-D portion of the display driver is in user mode too. As a result, the remote display system implemented as a display driver of the X Server [8] resides in user mode and can naturally use the socket network interface.

On Windows, however, the entire display driver and part of the graphic system is located in kernel mode. The service thread we mentioned above can either be a system thread in kernel mode or a normal thread in a user mode process. If it runs as a system thread in kernel mode, the time of kernel/user mode transition and memory copying can be saved. However, Windows does not provide a compatible network socket interface in kernel mode, but a more complex one called *Transport Data Interface (TDI)* [1] and, using which will leads to unportable code. Therefore, we deploy the service thread in user mode and introduces an additional component – the *intermediate command channel* that connects the driver and the service thread. We employ user/kernel memory sharing to eliminate copying data from kernel mode to user mode, and introduce several novel techniques such as *copy on delivery*, *parallel reading/writing*, and *unfair back-off synchronization*, to reduce the overhead of kernel/user mode transition and context switch.

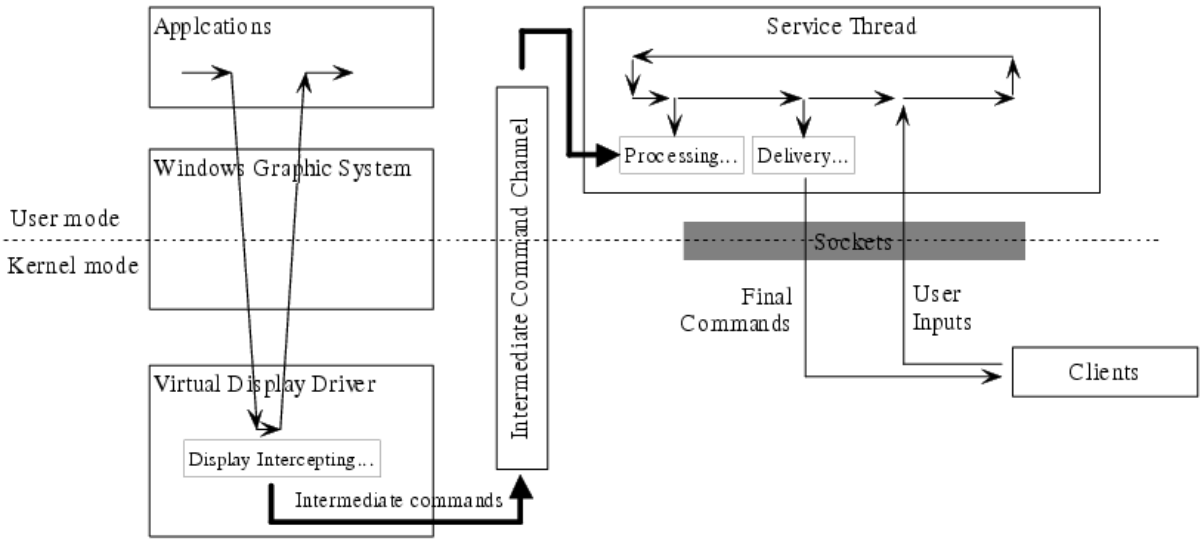

Figure 3.1: **Remote Display Architecture on Windows**

3.6 User Inputs

The clients of remote display systems gather input from user and send them to the server. As applications are still running on the "local display system", the remote display system should convert the user input from remote users to regular local inputs, which can be accepted by applications.

On X/Linux, each application registers on the X server as a socket connection. Once there are inputs from these sockets, the X Server will be woken up and the loop is started. Given that each clients connect to the remote display system through sockets, we register the socket of each connection on the X Server. Therefore, the X Server can also be woken up by the user inputs and then calls the per-screen WaitUpHandler, which will dispatch the input to a *virtual mouse driver* or a *virtual keyboard driver*. The drivers process inputs and propagate them to X Server through regular interfaces. The X Server considers them as local input and deliver them to corresponding applications.

On Windows, we bypass the mouse/keyboard driver and send input to the current desktop by calling certain Windows APIs. The service thread we mentioned above also wakes up when there is user inputs. This is implemented by the WSASelectEvent() and WaitForMultipleObjects() APIs [1]. The service thread dispatches the user inputs to an input service thread which then calls keybd_event() or mouse_event() to deliver user input to the current desktop.

3.7 Putting Them All Together

Display virtualization is a cross-platform approach to construct a remote display system on commodity operation systems. It leverages well-developed existing graphic systems by providing a virtual display driver, which virtualizes a display

device to the graphic system. On X/Linux, the remote display system can be implemented as a virtual display driver of the X server. It intercepts and processes display commands in regular rendering functions and deliver them in the BlockHandler. On Microsoft Windows, we also provide a virtual display driver, but it exists in kernel mode. Display commands are intercepted in virtual rendering functions and quickly translated into intermediate display commands. A user mode service thread processes deferred commands and delivers them to clients. An intermediate command channel transfers data from kernel mode to user mode. The architecture is showed in Figure 3.1. The implementation on both X/Linux and Windows share the some code base for command processing and delivery. They achieve similar performance for the same application on these two different platform, which are showed on Chapter 7. The remaining part of this dissertation will focus on the implementation of the remote display system on Windows.

Chapter 4

Intermediate Commands

Drawing information intercepted from local display systems can be translated into either a certain internal form for further processing or final commands which are ready for immediate delivery. The second option can simplify the implementation of the system. However, much information that can be utilized for optimization is lost. For example, we will not be able to preserve the semantic information in off-screen drawing information and leverage the the command queue techniques as discussed in Section 6.2. Therefore, we introduce *intermediate commands* as the objects that are processed in the remote display system. They are generated in the virtual rendering functions and passed to the next step. Intermediate commands are particularly important in our Windows implementation, where most of the management and delivery work is performed by the service thread in user mode and the drawing information is intercepted in kernel mode. In our implementation, intermediate commands are created in kernel mode by the virtualized rendering function and passed on to the service thread through an *intermediate command channel*.

The intermediate command set can be similar with either the highly abstracted actions in the display driver or the final remote display protocol. There are two disadvantages for the first option. First, on Windows, they may require process context dependent information which other parts of the remote display system cannot access. For example, an action in the display driver may refer to some pixel data that resides in the application's address space, and the data may become unavailable once the rendering function is finished. The service thread is unable to refer to data of this sort. Second, their richness and complexity will largely increase the difficulties in command processing and optimization.

Because of this, we define an intermediate command set that closely resembles the final remote display protocol [8]. This approach simplifies the command processing in the service thread. In addition, they are completely self-contained units, carrying all information required to generate the appropriate protocol update. For example, in Windows, the intermediate commands have no process context related information associated with them, allowing the service thread to fully access and manipulate them.

The intermediate commands augment the final remote display protocol to include off-screen management and

Operations	Use	Commands
Unary	Render on a single surface (either onscreen or offscreen)	RAW, SFILL, BITMAP, PFILL
Binary	Operate between two surface (onscreen or offscreen)	COPY
Surface	Create or delete an offscreen surface	CREATE_SURFACE, DELETE_SURFACE

Table 4.1: Intermediate Commands

drawing, as show in Table 4.1. The additional commands work as follows. When a surface[1] is created, the virtual display driver assigns it an integer surface ID and pushes a CREATE_SURFACE command into the intermediate command channel. If later there is a unary or binary operation that operates on this surface, the virtual driver attachs the surface ID to this command before pushing the command into the channel. When the driver is instructed to delete an off-screen region, it pushes a DELETE_SURFACE command into the channel with the corresponding ID. The framebuffer is created when the graphics system is initialized and never gets deleted.

The current state of the display, including framebuffer and offscreen surface, can be represented using two intermediate commands for each surface. In the case of Windows, where the service thread may be created after the system (and the virtual device driver) has been running for some time, when the service thread starts, it notifies the virtual display driver which provides a short sequence of intermediate commands that allow the service thread to build the current framebuffer state along with all off-screen surfaces. Each existing surface can be represented using a CREATE_SURFACE command and a RAW command which cover the entire surface.

4.1 Copy on Delivery

A RAW command consists of a destination region[2] and the pixel data that will be filled into this region. When an original display command is translated into a RAW command, it is required to copy the pixel data from the framebuffer or an offscreen surface and attach the data to the RAW command. However, it is normal that a later opaque command partially or completely overwrites the destination region of an earlier RAW command. For example, a web page may draw a background picture and then put some buttons or similar decoration on top of it. The pixel data of the overwritten region should not be delivered to clients to save bandwidth usage . Therefore, we introduce a *copy on delivery* mechanism, which defers the pixel data copying for each RAW command to the time right before delivery. When the display driver generates an intermediate RAW command, it does not attach the pixel data to it. Instead, the service thread reads pixel data from the framebuffer only when it is going to deliver it over the network. If the destination region of this RAW command has been partially or completely overwritten by a later command, less data needs to be copied. The correct image is guaranteed on the client, because the later command will reach the client within a short time. The image remains the same after the later command has been applied on the client.

The only exception is the COPY command when both the destination and source regions are on the framebuffer.

[1] A surface is either the framebuffer or an offscreen bitmap. We stick to this terminology in Chapter 6.
[2] A region is a surface or part of the surface.

If a command ahead of COPY delivers a region of image that has been changed by another command that comes later than the COPY command and this region overlaps with the source region of the COPY command, the destination region of the COPY will get an incorrect image when the COPY command is executed on the client before the later command. To prevent this from happening, we scan all commands that have not been delivered once the virtual driver gets an on-screen to on-screen COPY command. For each RAW command, the service thread reads its related pixel data from the framebuffer. These data are then stored into a temporary buffer before any later command writes to the source region of the COPY command. In this way, the correct image on the client is guaranteed. Even in this case, the total amount of pixels copied is still not more than that making an extra copy of pixel data for each RAW command.

4.2 Intermediate Command Channel

The intermediate command channel in our Windows implementation connects the virtual driver in kernel space and the service thread in user space. It simply consists of a *circular command buffer*, a *read_offset* and a *write_offset*. The command buffer and the places that store these two offsets locate in the non-paged memory pool, which will not be paged out. They are mapped from the kernel-mode address space into the service thread's process address space. Two offsets both starts at zero. The *read_offset* points to the next available command while the *write_offset* points to the next available bytes in the buffer. They are modified when an intermediate command is read from or written into the circular buffer, respectively.

To make the communication as efficient as possible, our remote display system leverages the atomic load/store CPU instructions on x86 [4] so that read/write operations on the channel can be done in parallel. As a result, neither the service thread nor the driver need to worry whether the command channel is being accessed by the other side. No locking is required. Therefore, our system saves the overhead in synchronization context switch. This is extremely helpful on multi-processor machines where the service thread and applications can run on different processors.

The *read_offset* and *write_offset* can be read by both the driver and the service thread, but they are changed by one of them, not both. When the driver pushes a command into the channel, it does the following:

copy *new_command* to *buffer*[*write_offset mod buffer_size*]

write_offset $\Leftarrow$ *write_offset* + *sizeof(new_command)*

The service thread reads a command from the channel by doing:

process *command* at *buffer*[*read_offset mod buffer_size*]

read_offset $\Leftarrow$ *read_offset* + *sizeof(command)*

The service thread decides whether there are new commands in the channel by comparing the read and write offsets. If these two offsets are the same, the channel is empty. When the driver is going to push a new command into the channel, it also checks these two offsets. The channel does not have enough space for the new command if

write_offset + *sizeof(new_command)* - *read_offset* $>$ *buffer_size*

Because loading/storing a 32-bit integer is atomic, neither the driver or the service thread needs locking. They do not get "wrong" values of the offsets, although they can get an "old" value. As the *read_offset* is changed only by the service thread, the service thread may get an old value of *write_offset* in determining whether there are new commands if the driver is meanwhile pushing a command into the channel. Similarly, the driver may get an old value of *read_offset* when the service thread is finishing processing a command. In the first case, the new command will be read next time and the processing is delayed for just a very short time. The second case usually does not prevent the driver from pushing a new command into the channel. Only in rare situation when the buffer gets full, the driver will block the application for a while and let the service thread take the commands out of the buffer. We have not seen these situations in our experiments.

Chapter 5

Signaling and Synchronization

On Windows, the display driver and the service thread are separated in kernel mode and user mode. Therefore, synchronization becomes a crucial part of the remote display system. We present several key mechanisms which have been effective in our Windows remote display system implementation.

5.1 Signaling and Polling

To respond to clients quickly, the service thread must be aware if the virtual display driver pushes a new command into the channel. Intuitively, an asynchronous model is efficient. The service thread starts with a waiting state. When the virtual display driver pushes a new command into the channel, it signals the service thread. The service thread then reads one command from the channel and waits for the next signal. This asynchronous mode works well when there are new commands occasionally. However, it faces several problems.

First, the service thread must be running at a priority higher than normal. The remote display system must respond to clients as fast as possible. If the service thread runs at a normal priority, the response speed will be slow down as the server gets heavier loaded, because its chances of being scheduled to run gets smaller. If the service thread running at a priority lower than normal, it will even starve in a heavily loaded server.

However, in the signal model the service thread with high priority will be scheduled to run once it is signaled, according to the preemptive scheduling policy in Windows [30]. As we discussed above, Windows executes rendering functions synchronously in an application's process context. When the virtual driver signals the service thread, it is the application's thread that gives off the signal. On a uniprocessor machine, the service thread will preempt the execution of this thread and thus leads to a context switch, because the application and the service thread are in different processes. On a multi-processor machine, the service thread may preempt either this application's thread or other thread with a priority lower than the service thread's. Frequent context switch hurts the performance of the system. When a browser displays

a web page, for example, the rendering functions are called hundreds of times, which generates hundreds of intermediate commands. The overhead of context switching will slowdown the entire systems.

The frequent context switching can be avoided because it is not necessary to process each display command as soon as it is issued by the application. On a local display system, the refresh rate for widely used LCD monitors is usually 60Hz. That means users cannot feel the delay of 15ms or event longer between their input and display output on the screen. For the burst of display commands within such a short time, there is no difference for users between getting each command instantly and getting all commands in one time at the end of this short time interval. Many commands are in small size and can be aggregated. Aggregating small commands of the same type can reduce the number of times that the clients do drawing actions, because the client is usually a thin one and does not do any more optimizations. Moreover, commands that draw on an offscreen surface will not be delivered until the offscreen surface is copied to framebuffer. They do not have to be processed immediately either. Therefore, we should let applications finish as many display commands as possible before the service thread starts processing and delivering them. The service thread can be waken up after each short time interval so that the frequent overhead of context switching is avoid.

Based on the above argument, we introduce a polling model in the service thread. The service thread wakes up after a certain time interval. It decides whether there are new commands in the channel. If not, it yields the CPU. Otherwise, it reads and processes those commands. In the meantime, there might be new commands added to the queue, but the service thread will not process them until the next iteration. These commands are skipped because some applications with heavy graphic operations may keep pushing new commands into the channel. The processing of user inputs will otherwise be delayed. When the system is idle, the service thread is sleeping. To give fast response to users, the service thread also wakes up immediately on receiving a user input. The polling model is as follows:

```
while (true)
{
  Sleep until
      1) There are inputs from users, or
      2) The sleep time interval expires
  If there are inputs from users then
     Process user inputs;
  If there are new commands in the channel
     Take a snap shot of the value of read_offset and write_offset;
     Read and process commands between read_offset and write_offset;
     Deliver commands;
}
```

This minimal time unit on Windows is a *clock interval*. A clock interval is roughly 10ms on uniprocessor machines

and 15ms on multi-processor machines as determined by the granularity of the timing interrupt supported by Windows [30]. We set this minimal time unit as time interval in the polling model to guarantee a refreshing rate of at least 60Hz. It processes user input before processing commands because a user input, such as a mouse clicking or a keyboard stroke, will probably triggers a number of display commands. This polling model is efficient in dealing with a burst of display commands, any command will not wait for longer than an iteration time to be processed. The service thread is running at a priority hider than normal, which again guarantees that each command can be processed in a timely manner.

The intermediate command channel supports parallel reading and writing. The driver keeps pushing command into the channel without worrying about service thread, while the service thread just reads command from the channel without considering any signal from the driver. It maximizes the parallel execution of applications and the service thread. In the situation with graphic intensive applications, the polling model cost less context switching overhead than the signaling model.

5.2 Synchronization

Because of the copy on delivery mechanism, we have to make sure that the service thread and the driver do not access the framebuffer at the same time. Most operating systems provide semaphore mechanisms to synchronize different processes in accessing the same resource; so does Windows. Implementations of semaphore mechanism usually have a built-in FCFS queue that guarantees the fairness between different processes. However, this fairness may hurt the performance in our case. Normally, drawing does not occur continuously. A burst of display commands is usually triggered by a user input event such as clicking a link on the web page or starts an application. These commands come from the same process and probably the same thread. As he Windows display system executes display commands synchronously in the application process/thread's context, this thread will access the framebuffer many times within a short period. On the other hand, the service thread accesses the framebuffer when it is about to deliver a RAW command to the client, because of the copy on delivery mechanism discussed in Session 4.1. As RAW commands are frequently used, the service thread may also access the framebuffer many times within a short period. The semaphore mechanism in this case will lead to a situation that the execution of applications and the service are interlaced. The overhead of frequent context switching will slow down both applications and the service thread. In addition, once a graphics application is blocked, the entire graphics system is blocked because Windows display system does not support parallel rendering. It is necessary to give the application a higher priority in accessing the framebuffer.

Therefore, we introduce an *unfair back-off* mechanism. A flag in the memory indicates whether the framebuffer is being accessed. Either the driver or the service thread will test this flag before accessing the framebuffer. If the flag is true, it yields the CPU and tries again after back-off time. If the flag is not, it sets the flag, accesses the framebuffer and resets the flag when it has done. Again, we use the atomic *TestAndSet* CPU instruction of the x86 architecture to

avoid race condition. We say this mechanism is unfair because the back off time for the driver is shorter than that of the service thread. The driver then has a higher priority in accessing the framebuffer and the application can quickly finish all drawing operations without being interrupted by the service thread. Windows provide users with an option which set the time quantum[1] when to 2 clock intervals which optimizes[2] graphic application performance [30]. It implies that 2 time clock intervals is a sufficient time for an application to finish the burst of display commands. Thus, we set the back off time for the service thread as twice clock interval and the driver will back off for one clock interval. Because the service thread runs at a priority which is usually higher than the priority of the application, it will get scheduled to run after the second clock interval in most cases. We also prevent the service thread from starving by temporarily increasing the back off time for the driver if the service thread cannot get a chance to access the framebuffer, but this rarely happens.

[1] Time quantum is the amount of time a thread gets to run before any context switch can possible happen.
[2] On Windows Server systems, by default, a thread runs for 12 clock intervals to optimize the performance of services

Chapter 6

Implementation Details

We have discussed most key issues of implementing a remote display system on commodity operating systems in previous section. [8] also presents several platform independent techniques based on THINC. In this section, we will discuss how these techniques are applied in remote display system of Windows.

6.1 The Virtual Display Driver

As a driver on Windows, the virtual display driver is loaded when the system boots up. This is before the user mode part of the remote system can be started. When the driver is loaded, it creates and initials the intermediate command channel. The driver does not write any commands into the channel until the user mode service thread is started. It stops writing when the service thread is terminated.

6.1.1 Display Commands and Intercepting

The virtual display driver intercepts display commands by creating the framebuffer and offscreen surfaces in device dependent format and implementing necessary *DrvXxx* functions in the virtual display driver. Without doing any real rendering operation by itself, it is able to maintain a up-to-date framebuffer and all offscreen surfaces by leveraging the GDI service functions provided by the GDI engine. Each *DrvXxx* function calls its corresponding *EngXxx* function. This considerably simplifies the implementation of the virtual display driver.

Surfaces are created in a certain format, which can be in either device dependent or standard bitmap format. Framebuffer or offscreen surfaces in device dependent format are maintained by display drivers only, while surfaces in standard bitmap format are maintained by both display drivers and the GDI engine. The virtual display driver creates the framebuffer and all offscreen surfaces in device dependent format so that it is able to monitor all drawing operations that

operate on them. Surfaces are stored in the memory, because the virtual display driver does not operates on a real hardware.

The virtual display driver must maintain the correct images of the framebuffer and offscreen surfaces, to gain the benefits from the copy on delivery mechanism. When a RAW command is about to be delivered, its corresponding data will be read from the framebuffer. In addition, when the client connects to the server, the entire framebuffer must be delivered to the client. Therefore, image of the framebuffer must be correct. It is also necessary to maintain correct images of the offscreen surface, because the framebuffer is sometimes updated by an offscreen-to-onscreen copy.

Each rendering function, i.e. a *DrvXxx* function, represents a drawing operation. In Windows, these drawing operations includes a few low level ones such as bit-block-transfer and text-output and some high level operations, such as gradient-fill and stretch-bit-block-transfer. A display commands corresponds to a *DrvXxx* function in the display driver. The *DrvXxx* functions for low level operations are required to be implemented by each display driver. The high-level ones are optional. They are usually implemented when the hardware provide acceleration.

For each *DrvXxx* function, there is a corresponding GDI service function with the same name except the *Eng* prefix. An *EngXxx* function finishes exactly the same operation as its *DrvXxx* function should do. When a *DrvXxx* function is called, a normal display driver can either finish the drawing operation by itself or punt it back to the GDI engine by calling the corresponding GDI service function with the same parameters. For example, the *DrvBitBlt* function can either do the bit block transfer itself or call *EngBitBlt* to do the same thing. The virtual display driver always take the second option. This significantly simplifies the implementation. Each *DrvXxx* function generates intermediate commands, based on its expected behaviors and the parameters that GDI engine passes to it, usually before calling the corresponding *EngXxx* function with exactly the same parameters. Some *DrvXxx* functions call *EngXxx* functions before generating intermediate commands, because they use post-render pixel data in the framebuffer or offscreen surface to construct intermediate commands.

Although the GDI engine passes rich drawing information to each *DrvXxxx* function in parameters, the virtual display driver utilizes only a small portion of the provided information to generate intermediate commands. For example, high level rendering operations, such as *DrvAlphaBlend, DrvGradientFill, DrvStretchBlt* and *DrvTransparentBlt*, are translated into RAW commands, each of which updates a region with raw pixel data.

Table 6.1 shows the *DrvXxx* functions (GDDI functions) from which we extract semantic drawing information. In addition to normal drawing, the virtual driver also monitors the cursor drawing in the *DrvSetPointerShape* function. Once Windows changes the cursor's shape, the data is captured and forwarded to the client.

6.1.2 Mirror Driver

In the display virtualization approach, we provide a virtual display driver to Windows. Instead of replacing the primary display driver, however, we use a *mirror driver*. A mirror driver is the driver for a mirror device that mirrors the drawing operations of physical display devices. A virtual primary display driver can intercept all drawing information while a mirror driver cannot monitor the Direct3D and DirectDraw operations. However, developing a primary display driver is

	GDDI functions			
Command	*DrvBitBlt, DrvCopyBits*	*DrvTextOut*	*DrvLineTo*	*DrvAlphaBlend, DrvGradientFill, DrvStretchBlt, DrvTransparentBlt*
RAW	X	X	X[1]	X
COPY	X			
SFILL	X	X	X	
PFILL	X			
BITMAP		X		

Table 6.1: Intercepts Display Commands from GDDI functions

much more difficult, because Windows will not work at all if the primary display driver cannot function correctly. The system cannot boot if the driver is not loaded. In addition, with the virtual primary display driver, Windows generates no output to the monitor. These tremendously increase the difficulties in development and debugging. With a regular primary display driver, we can still administer the system when the remote display system goes wrong for certain reasons. Although the mirror driver does not support Direct3D and DirectDraw, it is still able to intercept most of the regular graphic applications, except video players and computer games. However, when a mirror driver is later converted into a virtual primary display driver for further development, all rendering functions can remain the same. Most changes will be in the functions that initialize the display driver, such as *DrvEnableDriver*.

A mirror driver is implemented and behaves like another display driver. GDI supports a virtual desktop and provides the ability to replicate a portion of the virtual desktop on a mirror device. GDI implements the virtual desktop as a graphics layer above the physical display driver layer. All drawing operations start in this virtual desktop space; GDI clips and renders them on the appropriate physical display devices that exist in the virtual desktop [21].

A mirror device can specify an arbitrary clip region in the virtual desktop, typically covering the primary display device. GDI then sends the mirror device all drawing operations that intersect this clip region. If we set the clip region that exactly matches a particular display device, it can effectively mirror the device.

6.2 Command Queues

The command queue mechanism is the core of command processing. Our remote display system applies optimizations to intermediate commands in the approach of maintaining command queues. It not only maintains a command queue for onscreen drawing, but also one queue for each offscreen surface and therefore preserves semantic information of offscreen drawing. The concept of a command queue is discussed in [8] and we will present its detail algorithm in this section.

There are different ways to deal with unary and binary intermediate commands. Unary intermediate commands that operates on one surface will be demultiplex from the global command channel to the command queue associated with

[1] We convert line-to command into raw if it does not draw horizontally or vertically

its destination surface. The system tries to merge it only with the last command in the queue. The chances of merging non-subsequential commands are much lower, because they usually draw in different regions and for different contents. Overwriting/clipping is also tried. Below is the pseudo code of adding a command to a queue.

```
ADD_TO_QUEUE(new_cmd, cmd_queue)
{
    if (new_cmd is non-opaque){
        APPEND(new_cmd, cmd_queue)
        return;
    }
    /* If the new_cmd is opaque, try merge it with the last command in the queue. */
    if (cmd_queue is not empty){
        last_cmd ⇐ last command in the queue;
        if (MERGE(new_cmd, last_cmd) is successful)
            return;
    }
    /* If the command cannot be merged, do overwriting through the queue.*/
    new_cmd_region ⇐ region that new_cmd will affect;

    for each cmd in cmd_queue do{
        cmd_region ⇐ region that cmd will affect;
        intersect_region ⇐ new_cmd_region ∩ cmd_region;
        UPDATE_COMMAND_REGION(cmd, cmd_region − intersect_region);
        /*clip the overwritten part*/
    }
}
```

Because the command queue technique preserves semantic information of offscreen drawing, when the driver copies a region from an offscreen surface to another offscreen surface, the command queue of a surface is appended to the queue of another surface. Clipping is involved in this process. To simplify our discussion, let's take the most common case as an example, where the entire offscreen surface is copied to a certain position of another surface. A natural approach is to append each individual command in the source queue to the end of destination queue. When we append a command, we scan through the destination queue. For each command in the destination queue, we clip the region that will be overwritten by the appended command. It takes $O(n_1 n_2 + n_1^2/2)$ time to merge two queues, where n_1 and n_2 are the numbers of

commands in the source and destination queues respectively.

However, we take an optimized approach and reduce the running time to $O(n_1 + n_2)$. Because each command in the destination queue may overwrite a certain region of the destination surface, the union of these regions is a opaque region that commands in the source command queue will eventually overwrite. We first compute this opaque region by scanning through the entire source queue, and then go through the destination queue with this opaque region. If the rendering region of a command in the destination queue is completely or partially covered by the opaque region, we clip the covered region from the rendering region of this command. Both command queues are scanned for one time during the merge. Below is the pseudo code.

COPY_QUEUES($src\_cmd\_queue, dst\_cmd\_queue, src\_region, dst\_left, dst\_upper$)
{ /* First get the list of command that will affect the destination region. */
$src\_affect\_cmd\_list \Leftarrow$ empty;

/* src_region may not be completely opaque. To get the opaque part of src_region,
* we first compute the non_opaque source region */
$src\_non\_opaque\_region \Leftarrow src\_region$;

for each cmd in src_cmd_queue do {
$cmd\_region \Leftarrow$ region that cmd will affect;
if ($cmd\_region \cap src\_region$ is not empty){
/* cmd will affect the src_region, so add cmd to the end of the list */
APPEND_TO_LIST($cmd, src\_affect\_cmd\_list$);
if (cmd is opaque){
$cmd\_region \Leftarrow$ region that cmd will affect;
$src\_non\_opaque\_region \Leftarrow src\_non\_opaque\_region - cmd\_region$;
}
}
}
/* compute the opaque source region*/
$src\_opaque\_region \Leftarrow src\_region - src\_non\_opaque\_region$;
/* Add the offset to src_opaque_region because the source surface will be copied to a certain position.*/
ADD_OFFSET($src\_opaque\_region, dst\_left, dst\_upper$);
for each cmd in dst_cmd_queue {
$cmd\_region \Leftarrow$ region that cmd will affect;

$intersect\_region \Leftarrow src\_opaque\_region \cap cmd\_region$;
/*clip the overwritten part*/
UPDATE_COMMAND_REGION($cmd, cmd\_region - intersect\_region$);
}
/* Append the entire list at the tail of the queue*/
APPEND_LIST_TO_QUEUE($src\_affect\_cmd\_list, dst\_cmd\_queue$);
}

6.3 The Service Thread On Windows

The service thread on Windows reads commands from the channel and dispatches them to corresponding command queues. It also processes input from clients during this process. The service thread maintains the command queues for the framebuffer and each offscreen surface. After notifying the driver upon startup, it starts accessing the command channel and framebuffer by mapping their memory from kernel mode space to user mode space. Below is the pseudo code.

/* When the server starts up*/
Trap into the display driver by calling GDI escape.

⇓

The following steps are executed by the virtual driver
1) Map framebuffer, command queue into service thread's address space.
2) Read the cursor data to the service thread's address space.
3) For each existed offscreen region, push a CREATE_REGION
and RAW commandinto the channel

⇓

CREATE_COMMAND_QUEUE_TABLE(cmd_queue_table[]);
CREATE_COMMAND_QUEUE($framebuffer\_cmd\_queue$);
$cmd\_queue\_table[1] \Leftarrow framebuffer\_cmd\_queue$;
while (server is not termiated) {
Wait until
1) There is client input, or
2) There is an incoming connection to the server, or

```
    3) Sleeping time expires
if (A new client is connecting to the server){
    /*Create and initial a per-client structure, including a network buffer*/
    CREATE_PER_CLIENT_STRUCTURE(new_client);
} else if (there is a client input) {
    Forward the input to the virtual mouse/keyboard thread.
} else if (sleeping time expires) {
    num ⇐ number of command currently in the channel
    for i=1 to num do {
        command ⇐ FIRST_COMMAND_IN_CHANNEL;
        switch (type of command) {
            case CREATE_SURFACE:
                id ⇐ surface ID in command;
                CREATE_COMMAND_QUEUE(new_cmd_queue);
                cmd_queue_table[id] ⇐ new_cmd_queue;
                break;
            case DESTROY_SURFACE:
                id ⇐ surface ID in the command;
                cmd_queue ⇐ cmd_queue_table[id];
                DESTROY_COMMAND_QUEUE(cmd_queue);
                break;
            case SFILL/PFILL/BITMAP/RAW/PIXMAP:
                id ⇐ surface ID of destination surface;
                cmd_queue ⇐ cmd_queue_table[id];
                /* cmd_queue_table[1] is the framebuffer_cmd_queue. */
                ADD_TO_QUEUE(command, cmd_queue);
                break;
            case CURSOR_CHANGE:
                ADD_TO_QUEUE(command, framebuffer_cmd_quue);
                break;
            case COPY:
                if (COPY is from onscreen to offscreen) {
                    dst_id ⇐ surface ID of destination surface;
```

```
                    dst_cmd_queue ⇐ cmd_queue_table[dst_id];
                    ADD_TO_QUEUE(RAW, dst_cmd_queue);
                    /* We reduce it to RAW because early commands in the framebuffer_cmd_queue has been deleted.*/
                } else {
                    dst_id ⇐ surface ID of destination surface;
                    src_id ⇐ surface ID of source surface;
                    src_cmd_queue ⇐ cmd_queue_table[src_id];
                    dst_cmd_queue ⇐ cmd_queue_table[dst_id];
                    COPY_QUEUES(src_cmd_queue, dst_cmd_queue, src_region, dst_left, dst_upper);
                    /* (dst_left, dst_upper) is the left and upper point of the destination that COPY will write to. */

                    if (COPY is from onscreen to onscreen) {
                        /* Driver block itself after pushing an onscreen-to-onscreen COPY command
                         * into the intermediate command channel. At this point, we copy pixel data for
                         * RAW commands that have not been delivered.*/
                        for each raw_cmd in framebuffer_cmd_queue
                            COPY_PIXEL_DATA(raw_cmd);
                        UNBLOCK_DISPLAY_DRIVER;
                    }
                }
            }
            REMOVE_FIRST_COMMAND_FROM_CHANNEL;
            /*get ready for the next command.*/
        }
        FLUSH_TO_ALL_CLIENTS(framebuffer_cmd_queue);
        /* This is an non-blocking operation. If some commands cannot be delivered at this moment, they will be
         * stored in a per-client buffer and transferred next time */
    }
}
```

6.4 Memory Sharing

On Windows, the display driver is located in kernel space and the service thread is in user space. Many intermediate commands, such as RAW, carry pixel data. It will waste lots of CPU cycles if large amounts of data are frequently copied from kernel space to user space. therefore, memory sharing must be used. Although Windows provides the techniques for sharing memory between kernel mode and user mode [3], it does not allow a display driver to use any kernel API directly for security and stability purposes. The memory sharing between display drivers and applications is then naturally forbidden. We use some tricks to bypass this restriction. We encapsulate some necessary kernel APIs in the kernel mode DLL file, but the source code of this DLL file is compiled into a kernel mode driver file, i.e. a SYS file. After that, we change the name of this file to be a DLL file and place it under %SystemRoot%\system32\ directory. The display driver calls *EngLoadImage* to load this DLL file and calls *EngFindImageProcAddress* to get function pointers to the functions in the DLL file. By doing this, the display driver is able to utilize all kernel APIs and share memory between itself and applications.

Chapter 7

Experimental Results

The experiment results are based on the prototype system of THINC on X/Linux and Windows. We setup a THINC server on X/Linux, a THINC server on Windows, a X/Linux client and a Windows client. We tested four combinations of servers and clients and compare their performance with applications running on local machine. Section 7.1 describes our experimental setup. Section 7.2 describes the application benchmarks used for our studies. Section 7.3 presents our measurement results. The effectiveness of THINC architecture has been demonstrated in [8], in which a direct comparison with a number of state-of-the-art and widely used thin-client platforms are conducted. The following experiments focus on its cross-platform performance.

7.1 Experimental Setup

We compared the performance of various combination of THINC servers and clients of using an isolated network test bed. As shown in Figure 7.1, our test bed consisted of five computers connected on a switched FastEthernet network: one thin clients, a packet monitor, a network emulator for emulating various network environments, a thin-client server, and a web server used for testing web applications. Except for the thin clients, all computers were IBM Netfinity 4500R servers, with dual 933 MHz Pentium III processors and 512 MB of RAM. The client computers were a 450 MHz Pentium II computer with 128 MB of RAM. During each test, only one client/server pair was active at a time. The web server used was Apache 1.3.27, the network emulator was NISTNet 2.0.12, and the packet monitor was Ethereal 0.10.9.

To provide a fair comparison, we standardized on common hardware and operating systems whenever possible. For THINC on Windows, we ran Windows Server 2003 on the server and Windows XP Professional on the client. For THINC on Linux, we ran the Debian Linux Unstable distribution with the Linux 2.6.10 kernel on both server and client. We enable the RC4 connections for each combinations of THINC server and THINC client.

We considered two different network configurations: *LAN Desktop* and *WAN Desktop*. LAN Desktop represents

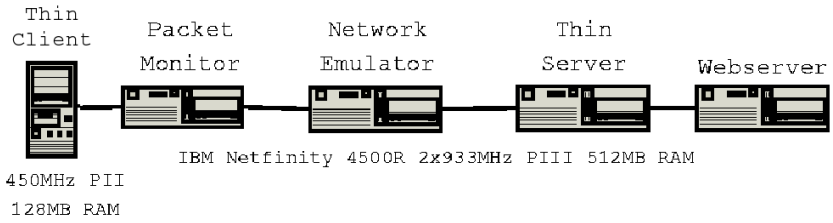

Figure 7.1: **Experimental Testbed**

a client with a 1024 x 768 display resolution and a 100 Mbps LAN network. WAN Desktop represents a client with a 1024 x 768 display resolution and a 100 Mbps WAN network with a 66 ms RTT, which emulates Internet2 connectivity to a US cross-country remote server [18]. We conducted our WAN experiments using the kind of high-bandwidth network environment that is becoming increasingly available in public settings [5].

Similar with many thin-client systems tested, THINC uses TCP as the underlying transport protocol, we were careful to consider the impact of TCP window sizing on performance in WAN environments. Since TCP windows should be adjusted to at least the bandwidth delay product size to maximize bandwidth utilization, we used a 1 MB TCP window size in our test bed WAN environment to take full advantage of the network bandwidth capacity available.

7.2 Application Benchmarks

We evaluated THINC on web browsing, one of the most dominant applications used on the desktop. Web browsing performance was measured by running a benchmark based on the Web Page Load test i-Bench benchmark suite [16]. The benchmark consists of a sequence of 54 web pages containing a mix of text and graphics. Once a page has been downloaded, a link is available on the page that can be clicked to download the next page in the sequence. This mouse clicking operation was done using a mechanical device we built to press the mouse button in a precisely timed fashion. The mechanical device enabled us to better simulate a user browsing experience and ensure that the test could be easily repeated on different thin-client systems without introducing human timing errors. We used the Mozilla 1.6 browser set to full-screen resolution for all experiments to minimize the impact of different window frames on Windows and X/Linux.

Our primary measure of web browsing performance is page download latency. Using slow-motion benchmarking [24, 19], we captured network traffic and measured page latency as the time from when the first packet of mouse input is sent to the server until the last packet of web page data is sent to the client. We ensured that a long enough delay was present between successive page downloads so that separate pages could be disambiguated in the network packet capture. The latency we get from this measurement is the network latency plus the server process latency. However, this measure

does not fully account for client processing time. To account for client processing time, we instrumented the client or the client window system to measure the time between the initial mouse input and the processing of the last graphical update for each page.

7.3 Measurements

Figures 7.2 and 7.4 show web browsing performance results. Figure 7.2 shows that client processing time is a dominant factor for local PC web browsing performance since the web browser needs to process the HTML on a slow client. In WAN, the time that a THINC client spent on getting display information of a web page from the THINC server is just a little bit longer than that a local PC web browser fetches the web page from the web server. There are two reasons for this. First, the client leverages the fast server to download and process the web page. Second, the data size is still small compared to the network bandwidth.

In Figure 7.2, different combinations of server and client in the same environment have different performance under the same benchmark. This is because of platform dependent factors. The graphic systems of X/Linux and Windows vary and the implementation of THINC on Windows and X/Linux are not completely the same. However, the variation in performance is not large. The standard deviation of the latency for the same environment and same measurement are not larger than 0.07 milliseconds. This implies that the display virtualization approach, including the command queue and offscreen-awareness technique, can achieve close performance for similar applications[1] on different commodity operating systems.

Because of the parallel reading/writing of the command channel, we save a lot of synchronization operations. In addition, the unfair back-off techniques limits the overhead of synchronization operations that can be added to our implementation on Windows. To measure the overhead, we disable synchronization operations and run the test on a Windows THINC server against a Windows THINC client. It gives us a incorrect image, but we got a latency which is only 0.02 second less than the one with all synchronization. The data size is different too, as shown in Figure 7.4, because disable the locking leads an incorrect image on the client end. Synchronization operations does not bring significant overhead to our system.

Figure 7.3 shows, the average number of intermediate commands per page translated from original Windows display commands, numbers of commands that are merged, commands that are completely or partially overwritten and final commands sent over the network. We did not show the onscreen-to-onscreen COPY and onscreen-to-offscreen COPY because they do not occurs in our benchmarking. Onscreen-to-onscreen happens when users scroll the web page. Onscreen-to-offscreen COPY usually occurs when user minimize a Window. As a command queue is copied from an offscreen region to another offscreen region, commands that will affect the destination region are duplicated. Therefore, the number of commands in all queues can be more than the number of command translated from original display commands. That's the

[1] We said similar applications because Mozilla on Windows and X/Linux do not do exactly the same thing. They at least using different APIs.

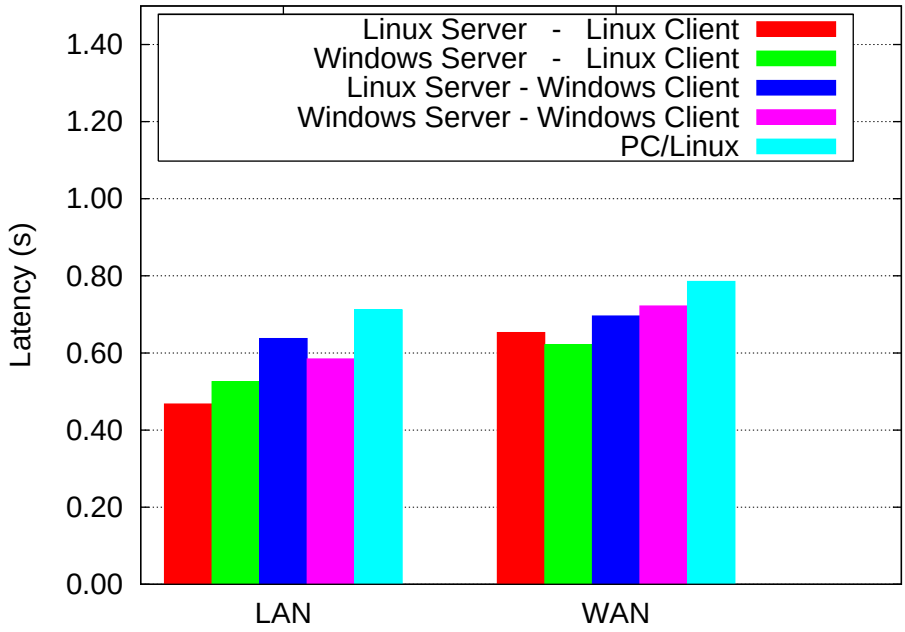

Figure 7.2: **Web Benchmark: Average Page Latency**

reason why merged PFILL commands are more than those translated. If a command is only partially overwritten, it might be overwritten again later, so the same command may be counted multiple times in our statistic results. If we compare the number of commands translated from Windows display command and the number of commands sent, we can see that the command merging and overwriting greatly reduced the number of commands that are ultimately delivered over the network, especially for the SFILL command.

It is worth mentioning that THINC provides fidelity display for clients with 24-bit colors images. It keeps all kinds of effects including anti-alias texts. Despite of all of these, it did well in both LAN and WAN on both X/Linux and Windows. As shown in Figure 7.2, all four combinations in both LAN and WAN environments, having latencies below the 0.72 second threshold for users to have an uninterrupted browsing experience [25].

Figure 7.5 shows the average number of commands generated per web page by Windows after a single click and the average number of commands processed by THINC as time goes. There are hundreds of commands generated within one second and they are from the same application - the Mozilla web browser. The asynchronous command processing of THINC saves us a lot of overhead in context switch. If the command processing and delivery is not deferred on Windows, the services thread and applications will context switch back and forth, which will greatly slow then the entire system. The two lines in Figure 7.5 are very close to each other, which implies that commands generated by Windows are still processed within very short time even thought they are deferred.

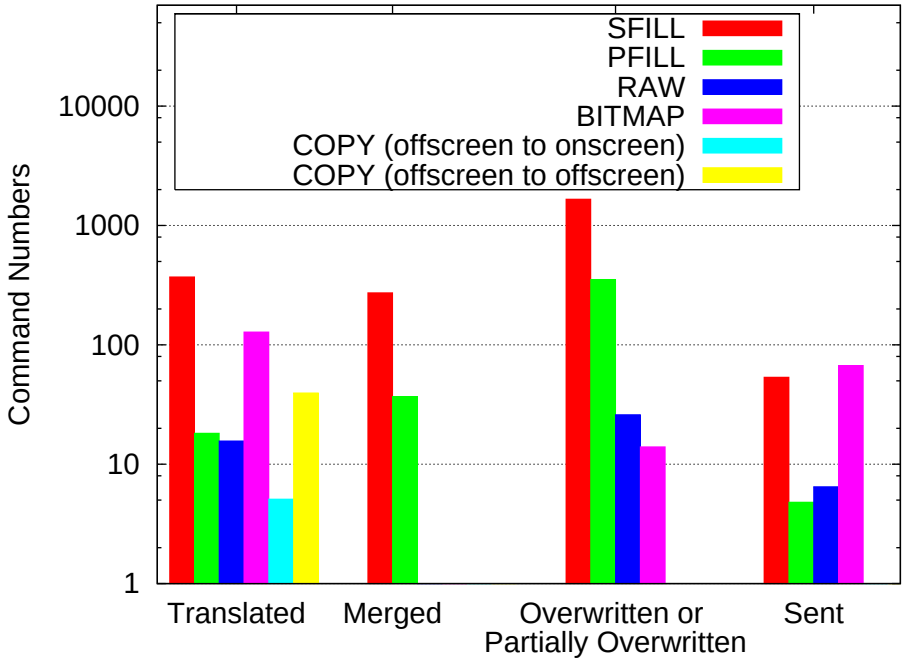

Figure 7.3: **Web Benchmark: Per Page Command Numbers**

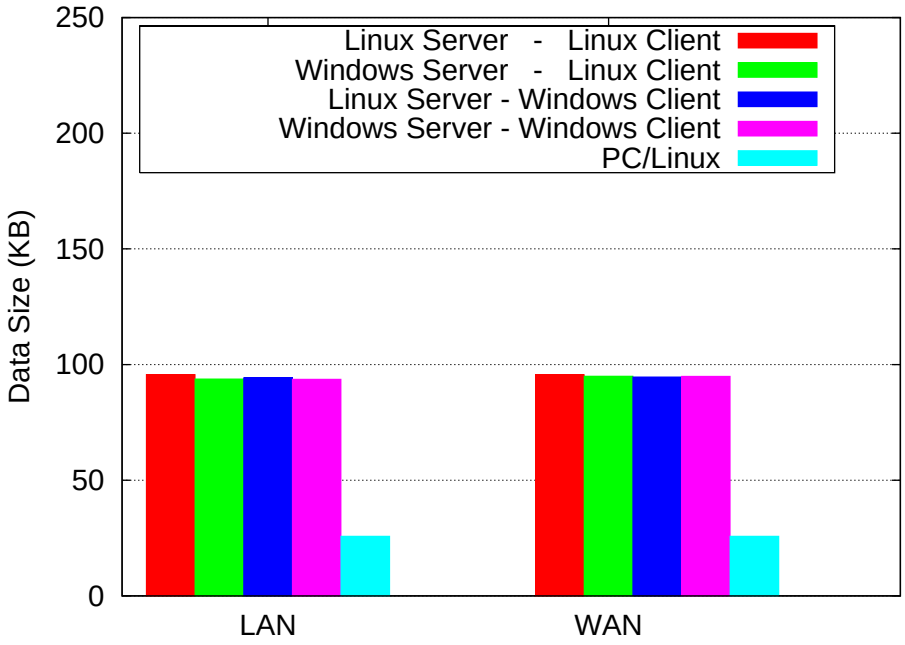

Figure 7.4: **Web Benchmark: Average Page Data Transferred**

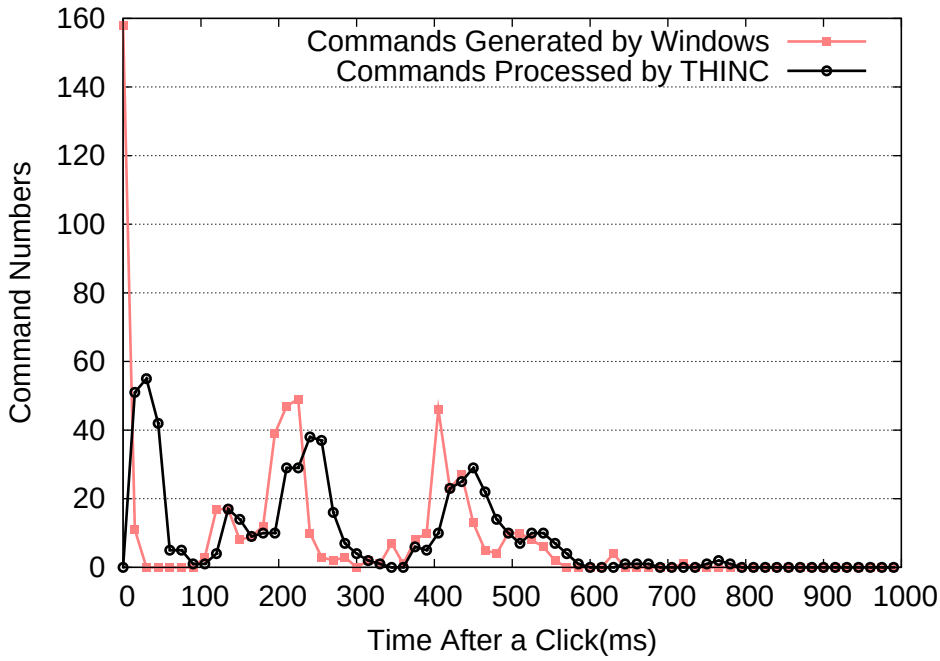

Figure 7.5: **Average Number of Per Page Commands**

Chapter 8

Related Work

Many alternative designs for developing effective remote display systems have been proposed. The existing remote display system can be loosely classified based on two related design choices: (1) How are the four layer stacks split between the server and the client, and (2) what display primitives are used for sending display updates over the network. We discuss them and compare some similar designs with our system.

Older remote display systems such as Plan 9 [28] and X [32] provide remote display functionality by separates the applications and the windows system on different machines. This division of work is more apparent in X, where, X applications perform graphics operations by calling library functions in charge of forwarding application-level display commands over the network to the machine where the window system of X resists. X commands present a high-level model of the overall characteristics of the display system, including descriptions of the operation and management of windows, graphics state, input mechanisms, and display capabilities of the system. By running the user interface on the client, user interface interactions that do not involve application logic can be processed locally without incurring network latencies. The use of high-level application display commands for sending updates over the network is also widely thought to be bandwidth efficient.

However, there are several important drawbacks to this approach. First, since application user interfaces and application logic are usually tightly coupled, running the user interface on the client and application logic on the server often results in a need for continuous synchronization between client and server. In high-latency WAN environments, this kind of synchronization causes substantial interactive performance degradation [18]. Second, the use of high-level application display commands, such as those used by X, in practice turns out to be not very bandwidth efficient [18, 33]. Third, the window server software used to process application user interfaces is large and complex. Maintaining that software on the client requires frequent updates and software maintenance costs contrary to the zero administration goals of thin clients. Fourth, as application user interfaces become richer, they impose more complex processing requirements. Running the user interface on the client necessitates replacing clients at the same high frequency as traditional desktop PCs

in order to run the latest applications effectively. Otherwise, they grow outdated quickly as X-based terminals did when web browsers came out in the early 1990s. Finally, storing and managing all display state at the client makes it difficult to support seamless user mobility across locations.

Proxy extensions such as low-bandwidth X (LBX) [38] and NoMachine's NX [26] have been developed to address some of these problems and improve X performance. LBX has been shown to have poor performance [17] compared to other thin-client systems [37]. NX is a more recent development that provides good X protocol compression and reduces the need for network round trips to improve X performance in WAN environments. Neither of these systems address the maintenance costs associated with executing a complete window system on the client.

More recent remote display systems such as Citrix MetaFrame [12], Microsoft Remote Desktop [13] which comes standard with Windows, Sun Ray [33] and VNC [29] avoid the need to maintain complex window system at the client and leave it on the server side as it originally locates. This approach is the same as the one that we present in the thesis. The client functions simply as an input-output device. It maintains a local copy of the framebuffer state used to refresh its display and forwards all user input directly to the server for processing. When applications generate display commands, the server processes those commands and sends screen updates over the network to the client to update the client's local framebuffer. The server maintains the true application and display state, while the client only contains transient soft state.

This approach provides several important benefits. First, synchronization overhead across the network between the user interface and applications can be eliminated since both components run on the server. Second, no window system needs to run on the client, allowing for less complex client implementation. Third, client processing requirements can scale with display size instead of graphical user interface complexity, enabling clients to be designed as fixed-function devices for a given display resolution. Fourth, since all persistent state resides on the server, mobile users can easily obtain the same persistent and consistent computing environment by connecting to the server from any client.

Achieving these benefits with good system performance remains a key challenge. Citrix MetaFrame, Tarantella [31], and Microsoft Remote Desktop decouple the display from applications at the interface between the window system and display devices. Display commands from windows systems are translated into a rich set of low-level graphics commands and forwarded to clients before they reaches the display device. However, performance studies [18, 39] of these systems indicate that using a richer set of display primitives does not necessarily provide substantial gains in bandwidth efficiency. Furthermore, the added overhead of supporting a complex set of display primitives results in slower responsiveness and degraded performance in WAN environment. Compared to the development of these systems, we implement our prototype system on Windows without access to Windows sources code. We do not use any unpublished interface of Windows.

Sun Ray uses simpler 2D drawing primitives for sending updates over the network. The original command set developed was simple and easy to implement, and was thoroughly evaluated [33], which motivated the use of a simple command set for our system. Sun Ray has since evolved and is now in its third major product version. However, it lacks

efficient and transparent mechanisms to translate application display commands into its command set. For example, some application commands need to be reduced to pixel data then sampled to determine which drawing primitives to use. This can be difficult to do effectively, and processing overhead can adversely affect overall performance. Applications which generate display commands that Sun Ray cannot efficiently translate need to be explicitly modified to deliver adequate performance. In particular, Sun Ray lacks transparent support for video playback. Sun Ray intercepts application commands using a customized X server, which causes difficulty in keeping up with continuing advances in more widely supported window server implementations, such as XFree86 and X.org.

VNC [35] and GoToMyPC [15] do not separate the four layer stack of display system. Instead, they read pixel data from the updated region in the framebuffer, encode or compressing it and deliver it to clients. This process sometimes called screen scraping. Other similar systems include Laplink [20] and PC Anywhere [34], which have been previously shown to perform poorly [23]. Screen scraping is a simple process, and decouples the processing of application display commands from the generation of display updates sent to the client. Servers do the full translation from application display commands to actual pixel data, while clients can be very simple and stateless, allowing for maximum portability of the system across client platforms. However, display commands consisting of raw pixels alone are typically too bandwidth-intensive. As a result, the raw pixel data must be compressed. Many compression techniques have been developed for this purpose, including FABD [14], PWC [7], and TCC [10, 11, 9]. Generating display updates in this manner is computationally intensive since the original application display semantics are lost and cannot be used in the process. However, these inefficiencies may be less important in the context of providing a user with remote access to an otherwise idle PC [15].

Chapter 9

Conclusions

We successfully built an asynchronous remote display system on top of the synchronous Windows display system using a virtual display architecture. Our approach works transparently without any modifications to applications, window systems, or operating systems. Our work extends previous work on using a virtual display architecture called THINC for remote display on X/Linux to operating on Windows, demonstrating that this approach works portably and transparently across different window systems and operating systems. We introduced several novel techniques for portability and performance, including a high-throughput intermediate command channel between kernel and user mode, a copy-on-delivery mechanism, and unfair backoff synchronization. In addition, we discuss the details of other portable implementation techniques, including command queues and off-screen awareness.

To demonstrate the effectiveness of our approach across heterogeneous systems, we measured remote display performance of our system across different combinations of X/Linux and Windows clients and servers. Our experimental results demonstrate that our display virtualization approach can achieve similar remote display performance on platforms with vastly different display architectures. Our results also show that the techniques we introduced for Windows do not result in significant performance overhead. Our implementation and measurement results on real applications provide strong evidence that our display virtualization approach using THINC provides an effective and portable architecture across different platforms.

Bibliography

[1] MSDN Library. http://msdn.microsoft.com/library.

[2] X Library Functions. http://xorg.freedesktop.org/releases/X11R7.0/doc/html/manindex3.html.

[3] A Common Topic Explained - Sharing Memory Between Drivers and Applications, 2000. http://www.osronline.com/article.cfm?id=39.

[4] IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide, September 2005.

[5] 100x100 Project. http://100x100network.org/.

[6] Susan Angebranndt, Raymond Drewry, Philip Karlton, Todd Newman, Bob Scheifler, Keith Packard, and David P. Wiggins. Definition of the Porting Layer for the X v11 Sample Server, 1994.

[7] P.J. Ausbeck. A Streaming Piecewise-constant Model. In *Proceedings of the Data Compression Conference (DCC)*, March 1999.

[8] Ricardo Baratto, Leonard Kim, and Jason Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the* 20^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[9] B. O. Christiansen and K. E. Schauser. Fast Motion Detection for Thin Client Compression. In *Proceedings of the Data Compression Conference (DCC)*, April 2002.

[10] B. O. Christiansen, K. E. Schauser, and M. Munke. A Novel Codec for Thin Client Computing. In *Proceedings of the Data Compression Conference (DCC)*, March 2000.

[11] B. O. Christiansen, K. E. Schauser, and M. Munke. Streaming Thin Client Compression. In *Proceedings of the Data Compression Conference (DCC)*, March 2001.

[12] Citrix Metaframe. http://www.citrix.com.

[13] B. C. Cumberland, G. Carius, and A. Muir. *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*. Microsoft Press, Redmond, WA, 1999.

[14] J. M. Gilbert and R. W. Brodersen. A Lossless 2-D Image Compression Technique for Synthetic Discrete-Tone Images. In *Proceedings of the Data Compression Conference (DCC)*, March - April 1998.

[15] GoToMyPC. `http://www.gotomypc.com/`.

[16] i-Bench version 1.5. `http://etestinglabs.com/benchmarks/i-bench/i-bench.asp`.

[17] Keith Packard. An LBX Postmortem. `http://keithp.com/~keithp/talks/lbxpost/paper.html`.

[18] Albert Lai and Jason Nieh. Limits of Wide-Area Thin-Client Computing. In *Proceedings of the International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, June 2002.

[19] Albert Lai, Jason Nieh, Bhagyashree Bohra, Vijayarka Nandikonda, Abhishek P. Surana, and Suchita Varshneya. Improving Web Browsing on Wireless PDAs Using Thin-Client Computing. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, May 2004.

[20] LapLink, Bothell, WA. *LapLink 2000 User's Guide*, 1999.

[21] MSDN Library: Display Devices: Windows DDK. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Display_d/hh/Display_d/GRDesGde_adffbe1d-e845-4e5a-b840-f878f159e401.xml.asp`.

[22] MSDN Windows 2000 Display Driver Model. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Display_d/hh/Display_d/DisplayDriverModel_GuideIntro_W2KModel_4d6b9e83-d6d7-4323-8f1e-6ab27c160927.xml.asp`.

[23] Jason Nieh, S. Jae Yang, and Naomi Novik. A Comparison of Thin-Client Computing Architectures. Technical Report CUCS-022-00, Department of Computer Science, Columbia University, November 2000.

[24] Jason Nieh, S. Jae Yang, and Naomi Novik. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. *ACM Trans. Computer Systems*, 21(1):87–115, February 2003.

[25] J. Nielsen. *Designing Web Usability*. New Riders Publishing, Indianapolis, IN, 2000.

[26] NoMachine NX. `http://www.nomachine.com`.

[27] Charles Petzold. *Programming Windows, 5th Edition*. Microsoft Press, Redmond, WA, 1998.

[28] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[29] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), January/February 1998.

[30] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, Redmond, WA, 2005.

[31] Tarantella Web-Enabling Software: The Adaptive Internet Protocol. SCO Technical White Paper, December 1998.

[32] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–106, April 1986.

[33] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture. In *Proceedings of the* 17^{th} *ACM Symposium on Operating System Principles (SOSP)*, December 1999.

[34] PC Anywhere. `http://www.pcanywhere.com`.

[35] Virtual Network Computing. `http://www.realvnc.com/`.

[36] X Window Manager. `http://en.wikipedia.org/wiki/Window_manager`.

[37] A. Y. Wong and M. Seltzer. Operating System Support for Multi-User, Remote, Graphical Interaction. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[38] X Web FAQ. `http://www.broadwayinfo.com/bwfaq.htm`.

[39] S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari. The Performance of Remote Display Mechanisms for Thin-Client Computing. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.