THINC: A Virtual and Remote Display Architecture for Desktop Computing and Mobile Devices

Ricardo A. Baratto

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2011

©2011

Ricardo A. Baratto

This work may be used in accordance with Creative Commons,
Attribution-NonCommercial-NoDerivs License. For more information about that license,
see http://creativecommons.org/licenses/by-nc-nd/3.0/. For other uses, please contact the
author.

ABSTRACT

THINC: A Virtual and Remote Display Architecture for Desktop Computing and Mobile Devices

Ricardo A. Baratto

THINC is a new virtual and remote display architecture for desktop computing. It has been designed to address the limitations and performance shortcomings of existing remote display technology, and to provide a building block around which novel desktop architectures can be built.

THINC is architected around the notion of a virtual display device driver, a software-only component that behaves like a traditional device driver, but instead of managing specific hardware, enables desktop input and output to be intercepted, manipulated, and redirected at will. On top of this architecture, THINC introduces a simple, low-level, device-independent representation of display changes, and a number of novel optimizations and techniques to perform efficient interception and redirection of display output.

This dissertation presents the design and implementation of THINC. It also introduces a number of novel systems which build upon THINC's architecture to provide new and improved desktop computing services. The contributions of this dissertation are as follows:

- A high performance remote display system for LAN and WAN environments. This system differs from existing remote display technologies in that it focuses

on the architecture of the system as a mechanism to improve performance, and not just on the remote display protocol and compression techniques.

- A novel mechanism to natively support multimedia content in a remote display system in a way that is both transparent to applications and format independent.

- pTHINC, a system to deliver improved remote display support for mobile devices, both in terms of performance and usability, and provide a competitive, and in some cases superior, alternative to native mobile applications.

- MobiDesk, a desktop utility computing infrastructure that enables service providers to host desktop sessions in fully virtualized environments. Hosted sessions can be remotely accessed using THINC, they can be migrated across computers to provide high-availability, and can be effectively and efficiently protected from denial of service attacks.

- Moving beyond remote display, we show how THINC's architecture can be used to provide continuous, low overhead recording of a desktop. Alongside, we introduce a novel way to leverage desktop accessibility services to allow users to search their recording based on captured text content.

We have implemented prototypes for these systems, and evaluated their performance in a number of scenarios, and compared it to representative alternatives whenever possible. Our results demonstrate that THINC can provide superior remote display performance, and can be successfully used as a fundamental building block for new and improved desktop applications and services.

Contents

Contents i

List of Figures vi

List of Tables x

List of Algorithms xi

Acknowledgments xii

1 Introduction 1

1.1 Contributions . . . 11

1.2 Dissertation Roadmap . . . 12

2 THINC Architecture 13

2.1 Remote Display Design . . . 13

2.2 Display Virtualization . . . 18

2.3 Remote Display Protocol . . . 25

2.4 Display Update Translation . . . 29

2.4.1 Offscreen Drawing . . . 33

2.5 Display Update Delivery . . . 35

2.6 Implementation 39
2.6.1 Back End 40
2.6.1.1 Creating Commands 40
2.6.1.2 Adding and Manipulating Commands 41
2.6.1.3 Abstracting Command Destinations 47
2.6.1.4 Delivering Commands 49
2.6.2 Front End 51
2.6.3 Remote Display Implementation 54
2.7 Experimental Results 56
2.7.1 Web Browsing Benchmark 60
2.7.2 Results 61
2.8 Summary 66

3 Multimedia **68**
3.1 Video Support 70
3.2 Audio Support 73
3.3 Media Synchronization 77
3.4 Implementation Details 81
3.5 Experimental Results 82
3.5.1 Experimental Setup and Benchmarks 82
3.5.2 Results 85
3.6 Summary 95

4 Mobile Devices **96**
4.1 pTHINC Usage Model 99

4.2 pTHINC System Architecture 104
4.2.1 Display Management 106
4.2.2 Video Playback 107
4.3 Experimental Results 109
4.3.1 Experimental Testbed 110
4.3.2 Application Benchmarks 112
4.3.3 Qualitative Results 114
4.3.4 Quantitative Results 117
4.4 Summary 126

5 Desktop Virtualization 128

5.1 MobiDesk: Mobile Virtual Desktop
Computing 130
5.1.1 Display Virtualization 135
5.1.2 Operating System Virtualization 138
5.1.3 Network Virtualization 142
5.2 A^{2}M: Access-Assured Mobile Desktop Computing 146
5.2.1 System Operation 150
5.3 Experimental Results 151
5.3.1 MobiDesk Virtualization Overhead 153
5.3.2 MobiDesk Application Performance 159
5.3.3 A^{2}M Performance Evaluation 164
5.3.3.1 Overall Performance 166
5.3.3.2 Interactive Applications 172
5.3.3.3 Wireless 174

5.4 Summary 175

6 Display Recording and Text Capture 178

6.1 Display Recording 179

6.2 Text Capture 182

6.3 Playback 185

6.4 Search 186

6.5 DejaView 188

6.6 Experimental Results 190

6.7 Summary 198

7 Related Work 199

7.1 Remote Display and Thin-Client Computing 199

7.2 Multimedia Support 206

7.3 Support for Mobile Devices 209

7.4 Display Recording and Text Capture 211

8 Conclusions and Future Work 213

8.1 Future Work 215

Bibliography 220

A THINC Protocol specification 237

A.1 Packet Format 237

A.1.1 Message Type and Flags 238

A.1.2 Unused Field (mbz) 238

A.2 Handshake Protocol 239

A.2.1 Version verification 239
A.2.2 Security Handshake 239
A.2.3 Parameter Negotiation 241
A.2.3.1 Packet Format of Server Replies 242
A.2.4 Summary of Handshake Messages 244
A.3 Remote Display Protocol 245
A.3.1 Server Messages 246
A.3.2 Client Messages 254

List of Figures

1.1 Remote Display Architecture 2

1.2 Audio/Video playback performance of popular remote display systems 6

2.1 Standard display architecture 16

2.2 Standard display pipeline 19

2.3 THINC virtual display architecture 22

2.4 THINC architecture components 23

2.5 Experimental Testbed 57

2.6 Web Benchmark: Average Page Latency 62

2.7 Web Benchmark: Average Data Transferred per Web Page 62

2.8 Web Benchmark: THINC Average Page Latency Using Remote Sites 64

3.1 Audio Playback 75

3.2 Audio Capture 76

3.3 Experimental Testbed for Audio Capture/Playback Benchmark . . . 84

3.4 A/V Benchmark: A/V Quality 86

3.5 A/V Benchmark: Total Data Transferred 87

3.6 A/V Benchmark: THINC A/V Quality Using Remote Sites 88

3.7 Timestamp Deltas: MPEG-1 352x240 89
3.8 Timestamp Deltas: MPEG-1 480x260 90
3.9 Timestamp Deltas: QuickTime 480x360 90
3.10 Distribution of Timestamp Deltas: MPEG-1 352x240 91
3.11 Distribution of Timestamp Deltas : MPEG-1 480x260 91
3.12 Distribution of Timestamp Deltas: QuickTime 480x360 92
3.13 Mouth-to-ear latency overhead for VoIP applications 93

4.1 pTHINC shortcut keys 102
4.2 PDA Experimental Testbed 110
4.3 pTHINC Web Screenshot: BBC News 115
4.4 Native IE Screenshot: BBC News 115
4.5 pTHINC Application Screenshot: Quicken 115
4.6 Native Application Screenshot: Pocket Quicken 115
4.7 PDA Browsing Benchmark: Average Page Latency 118
4.8 PDA Browsing Benchmark: Average Page Data Transferred 119
4.9 PDA Browser Screenshot: RDP 640x480 121
4.10 PDA Browser Screenshot: VNC 1024x768 121
4.11 PDA Browser Screenshot: ICA Resized 1024x768 122
4.12 PDA Browser Screenshot: pTHINC Resized 1024x768 122
4.13 PDA Video Benchmark: Fullscreen Video Quality 123
4.14 PDA Video Benchmark: Fullscreen Video Data 124
4.15 PDA Video Screenshot: RDP 640x480 126
4.16 PDA Video Screenshot: VNC 1024x768 126
4.17 PDA Video Screenshot: ICA Resized 1024x768 126

4.18 PDA Video Screenshot: pTHINC Resized 1024x768 126

5.1 MobiDesk Architecture 133

5.2 Problems of Migrating Connections 143

5.3 MobiDesk Network Virtualization 144

5.4 A^2M Architecture 147

5.5 MobiDesk Evaluation Experimental Testbed 152

5.6 MobiDesk Operating System Virtualization Overhead 155

5.7 MobiDesk Network Virtualization Throughput Overhead 156

5.8 MobiDesk Network Virtualization Latency Overhead 157

5.9 MobiDesk TCP Connection Setup Overhead 158

5.10 MobiDesk Average Per Web Page latency 161

5.11 MobiDesk Video Quality 162

5.12 A^2M web latency *vs.* packet replication 167

5.13 A^2M video quality *vs.* packet replication 168

5.14 A^2M average per-page data transfer *vs.* packet replication 169

5.15 A^2M total video data transmitted *vs.* packet replication 169

5.16 A^2M web latency under DDoS attack 171

5.17 A^2M video quality under DDoS attack 171

5.18 A^2M interactive performance for the echo test 173

5.19 A^2M interactive performance for minimize/maximize window test . . 173

5.20 A^2M Interactive performance for the scroll test 173

5.21 A^2M interactive performance for the move window test 173

5.22 A^2M video quality under DDoS attack in the wireless scenario 174

6.1 Recording runtime overhead 192
6.2 Recording storage growth 194
6.3 Browse and search latency 195
6.4 Playback speedup 196

7.1 Standard display architecture 200

A.1 THINC packet format 238
A.2 Security capabilities packet 239

List of Tables

2.1 THINC Protocol Display Commands 27

2.2 Remote Sites for WAN Experiments 59

3.1 THINC Video Commands 72

3.2 THINC Audio Commands 74

4.1 PDA Testbed Configuration Settings 111

5.1 MobiDesk Application Benchmarks 154

5.2 MobiDesk Migrated KDE Desktop Computing Session 163

6.1 Recording application benchmark scenarios 192

7.1 Remote Display Systems Comparison 205

A.1 List of client requests 241

A.2 List of server replies 242

A.3 List of handshake protocol messages 244

A.4 List of protocol messages 245

List of Algorithms

2.1 QueueCommand . . . 42

2.2 OverwriteCommands . . . 43

2.3 TryMerge . . . 44

2.4 X server main loop . . . 51

Acknowledgments

Having the opportunity to acknowledge everyone who made this work possible has been the best part of writing my dissertation. Countless individuals have been along for the ride providing mentoring, counseling, friendship, or even just a joke at the right moment, and I will never be able to thank them enough for this.

My advisor, Jason Nieh, without whom none of this work would have been possible. It has been a long adventure, and even after all these years he still manages to surprise me with his broad knowledge, thoroughness, sharpness, and pursuit of excellence. I am both a better computer scientist and person because of him.

Countless people made my tenure at Columbia a memorable experience. My friends and fellow PhD-ers Stelios Sidiroglou, Dan Phung, and Oren Laadan, thank you for your technical insight, hard work, patience, and all those arguments, about nothing, anything, and everything. Leo Kim was my teammate through most of the research that became THINC, and made large technical contributions to this work. He also taught me how not to manage people (by allowing me to be his "boss"), how to balance work and life, and how to appreciate a glass of single malt. Thanks to all the members of the NCL for the many days and nights of work, talk, and no-sleep we spent to the hum of aria, takamine, and the cluster machines: Shaya

Potter, Ravi Ghadia, Chris Vail, David Olshefski, Alex Sherman, Lei Zhang, Joeng Kim, Angelos Stavrou, Nikhil Tiwari, and Jae Yang. I was fortunate to work in many projects with Angelos Keromytis and Kenneth Ocheltree, and I thank them for their invaluable insight and guidance. Gail Kaiser and Henning Schulzrinne agreed to be part of my committee, and provided great research discussions and valuable feedback all through this process. I would also like to acknowledge the administrative staff at the Computer Science department, including Alice Cueba, Twinkle Edwards, Pat Hervey, and Remiko Moss, who helped me navigate the intricate bureaucracy that Columbia can be, always with a smile on their face.

Thanks to my parents, Maria Mercedes and Alvaro Germán, who taught me, pushed me, and supported me through everything, and without whom I would not be here. My sister Paola and my brother Germán for always being there. And the rest of my family for your support and for being who you are.

Finally, thank you to all the friends that pushed me through the end of this process: Liz, Alison, Nat, Julie, Carolyn, Dennis, Michelle and Pat. Without your looks of disbelief upon hearing my story, the subsequent heckling and worrying, and most of all your friendship, this dissertation may have been left collecting dust forever. Special thanks to Ing. Dustin Byford for providing beer when it was most needed. And thanks to Jet for keeping me honest.

This work was supported in part by a DOE Early Career Award, AMD, Google, Sun Microsystems, an IBM SUR Award, NSF grants CNS-0717544, CNS-0914845, CNS-0905246, CNS-0714277, CNS-0426623, CCR-0219943, and ANI-0240525, and AFOSR MURI grant FA9550-07-1-0527.

A Nicolás y Daniel Julián,

estrellas del presente, dueños del futuro

Chapter 1

Introduction

The advent of networking technology has enabled the transformation of our computing world from one of complete isolation, where physical data movement provides the only link between computers, to a world where computers and devices of all kinds are interconnected to one another. Furthermore, continuing advances in network capacity, performance, and ubiquity have enabled the proliferation of technologies that extend our computing environment beyond the boundaries of a single computer, a phenomenon many have denominated the *dis-integration of the computer*.

In this world, network connections replace what once were internal communication paths in the computer. For example, using network storage, our data can be spread out across multiple computers, while permitting us to maintain a single namespace view, and, more importantly, ubiquitous access to it. Similarly, computational clusters and grid computing are able to harness the power of discrete machines in disparate geographical connections to work as a single entity with massive computational power.

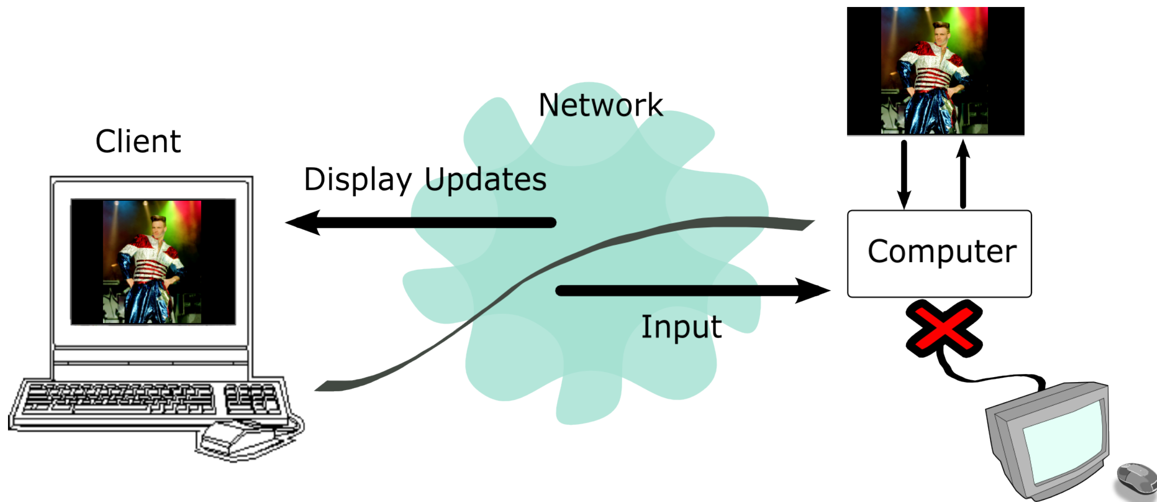


Figure 1.1 – Remote Display Architecture. Display output from the computer's desktop that would normally be delivered to locally attached devices, is redirected over the network to a client to be displayed. In response to the user interacting with the desktop, input events are generated and forwarded from the client back to the computer.

Another example of this dis-integration is *remote display*. Remote display is a client/server technology that decouples a computer from the devices used to access and interact with it, in particular its monitor, keyboard, and mouse, and uses the network to provide a communication channel between these devices and the computer. Figure 1.1 shows the architecture and mode of operation. Graphical output from the computer's desktop that would normally be sent to the local video hardware, is instead intercepted and redirected over the network in the form of a remote display protocol to a client to be displayed. Similarly, in response to the user interacting with the desktop, input events are generated and forwarded from the client back to the server.

While simple in theory, this architecture can be used in powerful ways to provide a number of benefits. For example:

- *Ubiquitous access*. Remote display enables ubiquitous access to complete desktop environments or individual applications, only requiring a network path to connect the client to the target computer. Since the required client function-

ality is so basic, it can be provided almost everywhere, using anything from a simple viewer application that can be embedded in web browsers [150], to simple devices like cellphones or PDAs [46, 106], or specialized lightweight terminals [141, 168].

- *Remote collaboration.* As the display output is redirected over the network, it can also be replicated and forwarded to multiple clients simultaneously. In this manner, groups of globally distributed users can collaborate by sharing access to a single computer and the applications and instruments on it. It also enables more efficient sharing and utilization of specialized equipment that can be centrally located and time-shared over the network.

- *Online help.* Remote display creates a new paradigm for providing live computer help and technical support. Sharing the user desktop provides the perfect environment for either showing somebody else how to perform a computer task, or helping them troubleshoot a problem [24, 36, 71, 130, 162].

- *Virtual displays.* Using remote display technology, it is possible to enable multiple displays from disparate computers to behave as belonging to one single computer and be controlled by either one or multiple users [29, 156]. It also enables remote displays to be used as extensions of the local display [158], for example to control display walls or external equipment.

- *Smart displays.* Stand alone displays, such as public large screen monitors, TVs, and projectors, can become live components of the network that can be accessed and manipulated remotely and on-demand [96]. For example, screens distributed across an office can be used to automatically display a user's desktop when they detect the owner's presence nearby [11, 26].

- *Thin-client computing.* Remote display is the core enabling technology for thin clients. Thin-client computing offers a solution to the rising management complexity and security hazards of the current desktop computing model to return to a more centralized computing strategy. A thin-client system functions by moving all application processing and data to centralized servers and secure server rooms, and using a remote display system to provide access to these servers from *thin* client devices. In this model, the edges of the network, where management is more costly, are composed of simple devices that require little or no maintenance. These devices are low-power, produce very little heat and noise, and in case of failure can be simply discarded and replaced with new ones. In addition, they are stateless and do not store any sensitive data that can be lost or stolen, and do not need to be backed up or restored. Furthermore, server resources can be physically secured in protected data centers and centrally administered, with all the attendant benefits of easier maintenance and cheaper upgrades. Finally, computing resources can be consolidated and shared across many users, resulting in more effective utilization of hardware.

- *Desktop virtualization.* By decoupling applications from the underlying display hardware, remote display enables desktop environments to be completely encapsulated, and possibly moved across computers. For example, combining remote display and operating system virtualization technologies, users can carry their desktop on a portable storage device, allowing them to maintain a consistent desktop environment even as they move across computers [112, 113].

- *On-demand application access.* Remote display can be used to provide remote access to a centralized pool of application servers, enabling more cost-effective application licensing, and better utilization of both applications and

resources[22, 86]. It also enables specific tailoring of computing resources, since different applications can be assigned to specific servers according to the application's profile and needs (e.g. I/O intensive, compute intensive, requiring specialized hardware).

Given these benefits it is not surprising the sheer number of remote display systems available today [17, 23, 36, 46, 47, 71, 75, 82, 95, 106, 120, 141, 142, 144, 150, 163, 170]. In addition, the market has been and is expected to continue to grow substantially [31, 37, 143, 165], as seen in the increasing number of startup companies [16, 28, 79, 108, 116, 128, 131], and the ongoing standards work in the area [96].

However, remote display systems face a number of technical challenges before achieving mass acceptance. The most salient of these is the need to provide a high fidelity visual and interactive experience for end users across the vast spectrum of graphical and multimedia applications commonly found on traditional desktop computers. For example, as Figure 1.2 shows, many of the most popular remote display systems are unable to provide desktop-like audio/video playback performance, even in the presence of optimal network conditions.

Most of these systems have focused on supporting office productivity tools in LAN environments, and reducing data transfer for low bandwidth links such as ISDN and modem lines. This focus has resulted in the majority of the work done in this area to be centered on the remote display protocol, either towards augmenting it to support higher-level primitives which can better represent certain application requests [83, 84], or in developing better compression algorithms [19, 20]. While these approaches have resulted in improvements for the scenarios mentioned above, they have also resulted in many systems being unable to effectively support more display-intensive applications which have become an integral part of today's desktop environments, or to operate

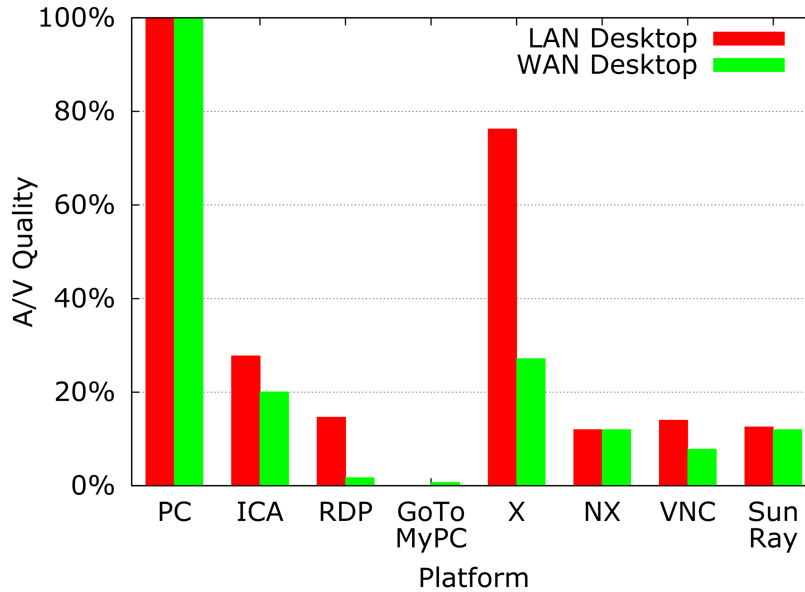


Figure 1.2 – Audio/Video playback performance of popular remote display systems. None of the systems is able to provide 100 % audio/video playback quality, even in the most favorable network conditions

effectively in the increasingly ubiquitous higher latency wide-area network (WAN) environments. WAN performance is particularly important given the growing number of application service providers attempting to provide desktop computing services over the Internet [46, 131].

In this context, we introduce THINC (*THin-client InterNet Computing*), a virtual and remote display architecture for desktop computing. THINC has been designed to address the limitations and performance shortcomings of all previous remote display systems, and to provide a building block around which new and improved desktop architectures and services can be built.

In designing THINC we departed from the mainstream view of tying remote display performance to the design of the remote display protocol. Instead, we argue that, while having a suitable remote display protocol is important, the architecture of the system is just as important to the performance of the system. To guide the

process of designing and implementing the architecture of THINC we followed these goals:

- *Responsiveness:* Our measure of success, and the focus of our efforts was low user-perceived latency. THINC should provide an experience as close as possible to existing desktop computers.
- *Transparency:* THINC should not require any modifications to existing operating systems, window systems, and applications.
- *Client simplicity:* THINC clients should be simple to implement, and be able to run across a large number of hardware and software architectures.
- *User mobility:* THINC users must be able to seamlessly connect and disconnect from many clients, without losing any session state or data.

THINC is architected around the notion of a virtual display device driver, a software-only component that behaves like a traditional device driver. Instead of managing a specific video hardware instance, the virtual device driver creates an abstraction of the computer's display hardware, and enables display output to be seamlessly intercepted, manipulated, and redirected. Reusing the device driver interface enables THINC to be completely transparent to existing applications, window systems, and operating systems, while allowing it to leverage existing display systems functionality and have access to a wealth of display-related semantic information. THINC's virtual device driver also completely encapsulates the state of the display, enabling applications and the desktop environment to become independent from both the underlying display hardware, and the characteristics of the client device being used to interact with them.

On top of this virtual device architecture, THINC introduces a simple, low-level, device-independent format to represent changes to the display. On a remote display context, this representation becomes a simple protocol that closely mimics a common set of operations natively supported by most commodity display hardware. In this manner, clients become simple, stateless entities that do little more than receive updates from the network and redirect them to their underlying hardware. Outside of remote display, this representation can also be used in many other scenarios, for example to provide a compact and easily reproducible representation of desktop display changes for archival purposes.

The virtual device and protocol are brought together with a number of novel optimizations and techniques to perform efficient translation from application requests received by the virtual device driver to THINC's protocol, and then to efficiently deliver the resulting protocol commands to clients across the network. In Chapter 2 we show how this core architecture can provide superior remote display performance, and can efficiently cope with high-latency network environments.

However, simply improving basic remote display performance is no longer enough to fulfill the needs and expectations of today's desktop users. As previously mentioned, a key feature missing from previous remote display solutions is support for display intensive applications, and in particular, multimedia applications. These solutions suffer from their inability to distinguish multimedia content, and their attempts to apply ineffective and expensive compression algorithms on the rapidly changing video data.

THINC addresses these shortcomings by leveraging its virtual device architecture to provide a virtual "bridge" between the remote client hardware and desktop applications, allowing these to transparently use the hardware capabilities of the client to perform multimedia operations across the network. As Chapter 3 describes, this

is accomplished by extending the virtual display device to provide video playback acceleration. Alongside, THINC introduces a virtual sound device which can capture and forward audio onto the client, and can receive audio data captured by the client and forward it to applications. These two mechanisms are brought together using a client-side synchronization mechanism which uses timing information generated by the drivers to fully synchronize multimedia content. As our experimental results show, this mechanism enables THINC to provide native, format-independent, and seamless support for display intensive multimedia applications.

In designing THINC, one of our goals has been to provide seamless desktop access across a multitude of devices. As wireless networks have become pervasive, small, mobile personal devices such as PDAs and cellphones have become an integral part of our computing environment. While native applications exist for these devices which attempt to provide functionality found on their counterpart desktop applications, a resource-constrained environment, coupled with differences in hardware and software environments, oftentimes result in limited feature sets and subpar performance.

To address these problems, we present pTHINC, an alternative solution for enabling remote desktop access, and delivering application services on mobile handheld devices by using thin-client computing. In this model, handheld devices become simple clients which communicate over a network with a server hosting desktop applications. This approach enables unmodified desktop applications to be used in mobile devices, leveraging server resources to run all complex logic without taxing the constrained PDA resources, and provides stateless and secure access to these applications and data associated with them.

In Chapter 4 we show how pTHINC leverages THINC's virtualization to decouple applications from the particulars of the client device, and seamlessly adapt the display output to the best mode for the device being used. In particular for mobile

devices, pTHINC is able to resize updates on the fly, enabling the user to zoom in and out of the desktop as needed. pTHINC couples this with a number of user interface optimizations tailored specifically to the characteristics of mobile devices. The resulting solution is able to outperform and provide better usability than both previously available solutions and native applications.

THINC can also be leveraged to build desktop computing systems beyond remote display. Chapters 5 and 6 describe two examples of these.

In Chapter 5, we introduce MobiDesk [14], an architecture that leverages THINC display virtualization and remote display architecture to create a desktop utility computing infrastructure. MobiDesk transparently virtualizes a user's computing session and decouples it from any particular end-user device, allowing all application logic to be moved to hosting providers. MobiDesk's virtualization layer also decouples a user's computing session from the underlying display hardware, operating system, and server instance, enabling high-availability service by transparently migrating sessions from one server to another during server maintenance or upgrades. We also present A^2M [135], a mechanism to protect MobiDesk's hosting infrastructure from distributed denial of service attacks [30]. A^2M combines a stateless and secure communication protocol, an indirection-based network (IBN) and THINC's remote display architecture to provide continuous access to hosted desktop sessions. A^2M takes advantage of THINC's low-latency remote display mechanisms and asymmetric traffic characteristics by using multi-path routing to send a small number of replicas of each packet transmitted from client to server.

Chapter 6 presents how THINC has been integrated into DejaView [67], a personal virtual computer recorder that provides a complete recording of a user's desktop computing. THINC's virtualization is leveraged by DejaView to provide efficient and transparent recording of all display output of a desktop session. Combined with

automatic, application-independent text capture and indexing, and user-generated annotation capabilities, it provides a novel mechanism for users to gain visual access to all information they have come in contact with on their desktop. Similar in functionality to a PVR, users of this system can seamlessly playback, browse, and search the recorded data. Furthermore, DejaView uses application and file system virtualization and checkpointing, so that users can not only search and view their recorded data, but also interact with it by reviving the state of their desktop at any point in the past.

1.1 Contributions

The contributions of this dissertation include:

1. The architecture of THINC, especially
   - Its virtual device driver which transparently intercepts display output in an application and OS agnostic manner;
   - Its efficient translation mechanism from high-level application display requests to low-level protocol primitives;
   - Its delivery architecture which prioritizes latency-sensitive updates, can discard stale updates, and operates without blocking applications, or the operating system;
2. A novel approach to efficient remoting of multimedia applications;
3. A system (pTHINC) to support remote display to mobile devices, using automatic display resizing, rotation, and efficient network usage;

4. THINC's use in MobiDesk, a personal desktop hosting infrastructure, and A^2M, a way to protect hosted desktops from distributed denial of service attacks;

5. THINC's use in DejaView, a personal virtual computer recorder that provides a complete recording of a desktop computing experience;

6. An extensive experimental evaluation of the performance of THINC;

7. A software implementation of THINC.

The current implementation of THINC is available for download from `http://www.ncl.cs.columbia.edu/research/thinc/download/`

1.2 Dissertation Roadmap

The rest of this dissertation is organized as follows. Chapter 2 presents the core remote display architecture of THINC. Chapter 3 discusses the native multimedia support in THINC. Chapter 4 introduces pTHINC, a remote display system for mobile devices. Chapter 5 presents MobiDesk, a system which can be used to centrally host desktop sessions and A^2M, a way to protect MobiDesk-type desktop hosting infrastructures from DDoS attacks. Chapter 6 presents DejaView, a personal virtual computer recorder. Chapter 7 discusses related work, and Chapter 8 presents conclusions and future work.

Chapter 2

THINC Architecture

This chapter introduces the remote display and virtualization architecture of THINC. It begins by describing how remote display systems work, and the design choices which need to be made to construct one. Building on this discussion it describes the design of THINC, its architecture, and the different mechanisms THINC uses to provide efficient remote display. It follows with a discussion of implementation details, and finishes with an experimental evaluation of the performance of THINC compared to existing remote display systems.

2.1 Remote Display Design

A remote display system decouples a desktop computer from the devices used to interact with it. In particular, the monitor, keyboard, and mouse no longer need to be directly attached to the physical ports in the computer in order to interact with it. Instead, a network connection is used to provide a communication channel between these devices and the computer. Graphical output from the computer's desktop, that would normally be sent to the local video hardware, is instead intercepted and

redirected over the network to a client to be displayed. Similarly, in response to the user interacting with the desktop at the client, input events are generated and sent back to the server. The client and server use a remote display protocol for this back and forth communication.

As discussed in the introduction to this dissertation, this decoupling has a number of benefits and applications. For example, providing ubiquitous access to desktop computers, fostering remote collaboration, and enabling more cost-effective ways to manage desktop computers. As a result, remote display systems have become widely popular, which in turn has led to the development of many different systems. This has also made remote display performance a topic of large interest.

While existing systems differ in many aspects, most of the work done to improve their performance has focused on the remote protocol: exploring what primitives should be used to represent display changes on the desktop, and designing new compression algorithms better suited for remote display traffic.

In this context, THINC was developed to address the limitations and performance shortcomings of existing remote display systems, and to provide a building block around which new and improved desktop architectures and services can be built. With THINC we departed from the mainstream view of focusing on the protocol, and argue that, while having a suitable remote display protocol is important, the architecture of the system is just as important to the performance of the system. To guide the process of designing and implementing THINC we followed these goals:

- *Responsiveness:* Our measure of success, and the focus of our efforts was low user-perceived latency. THINC should provide an experience as close as possible to existing desktop computers.

- *Transparency:* THINC should not require any modifications to existing operat-

ing systems, window systems, and applications.

- *Client simplicity:* THINC clients should be simple to implement, and be able to run across a large number of hardware and software architectures.

- *User mobility:* THINC users must be able to seamlessly connect and disconnect from many clients, without losing any session state or data.

A significant part of the design process of THINC consisted of examining existing systems, and evaluating the relationships between the architectural design choices made by their authors and the final system. Thus, the first step in presenting the design and architecture of THINC must be to discuss these. Before delving into this discussion, it is beneficial to discuss the display architecture of a typical desktop computer on which all remote display systems are based.

As Figure 2.1 shows, a computer's display system works as a pipeline, with desktop applications on one end, and the framebuffer and input devices at the other. The purpose of this architecture is to allow applications to generate visual output to users, and in turn to receive input events generated by the users as they interact with applications.

To illustrate how this works, we will show the process triggered by a user clicking on a hyperlink on a web page displayed by a web browser. The hyperlink points to an image which gets loaded on the screen in response to the user clicking on it.

1. The user clicks on the link, and an interrupt is generated by the mouse, caught by the operating system kernel and passed to the mouse device driver. Along with this interrupt, the mouse generates a data packet describing the input event. In our case, two packets will be generated. One describing that the left mouse button was pressed, one describing that the left mouse button was depressed.

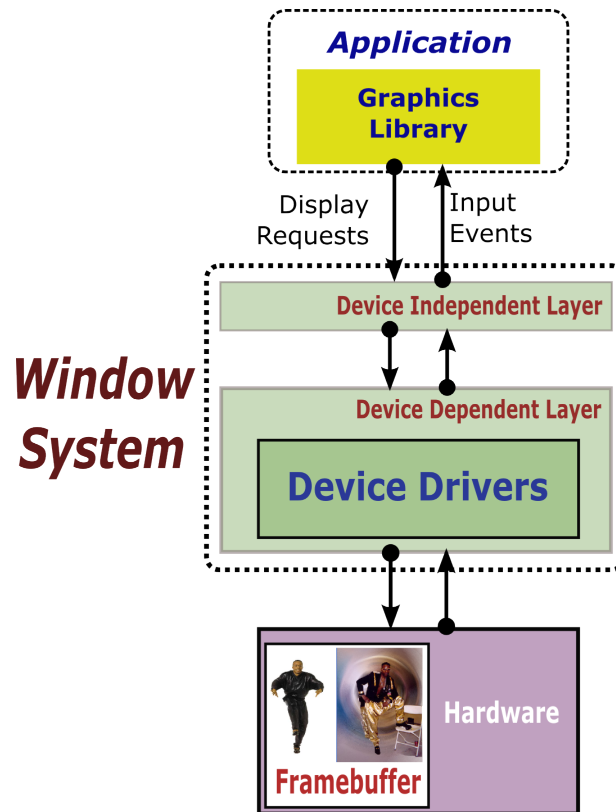


Figure 2.1 – Standard display architecture

2. The driver reads and decodes the data packet, translates it to a device-independent format the window system can understand, and passes it on.

3. The window system reads the input information from the device driver, and from the information generates an input event for the application:

   ```
   Left mouse button (de)pressed at coordinates X,Y
   ```

   It will also inform the application on which of its windows the click occurred. Since user-perceived latency is critical, this event is passed to the application in such a way to guarantee the speediest delivery possible, normally through some asynchronous notification mechanism.

4. The application is woken up, receives and processes the input event. In our example, the web browser will receive both the mouse pressed and depressed events, realize they occurred over a hyperlink, and follow the link.

5. The image is read from the network, decoded, and converted to a collection of pixels by the web browser. These pixels are ordered in a format that is understood (and perhaps negotiated at initialization time) by both application and window system. Then, a request is sent to the window system to replace the current contents of the application's window with the rendered image:

   ```
   Draw image of size WxH at coordinates Xi, Yi
   Read the image data from buffer B
   ```

   For performance reasons, in particular when dealing with large sizes, it is common for images to be transferred from the application to the window system using some shared memory mechanism, and for this shared memory to be exposed all the way down to the driver. We will assume this is the case here.

6. The window system receives the request, performs any necessary checks related to window management, and converts the high level application request to a low-level request to the video device driver. In our simplified example, this high-level to low-level mapping is one-to-one, but many other application requests have to be broken down into simpler device driver requests.

7. Finally, the device driver receives the request, reads the image data, and passes it to the video card. The video card takes care of adding the image data to the current contents of the framebuffer and sending the updated framebuffer contents to the computer's display. At this point the user sees the image on the screen.

With this architecture in mind, we can begin discussing the design of THINC by exploring the decisions made for other systems, their advantages and disadvantages, and the choices we made for THINC. The following discussion will examine these choices with a focus on driving the discussion of the architecture of THINC. Chapter 7 has a more in depth discussion of the specific design choices and architecture of the most popular remote display systems in use today.

In general, three major choices need to be made when designing a remote display system:

1. *where* in the graphics pipeline display commands from applications are intercepted,

2. *what* display primitives are used for sending updates over the network, and

3. *how* these commands are translated and sent from server to client.

The following sections discuss each of these in detail.

2.2 Display Virtualization

The first choice to be made when designing a remote display system is where in the graphics pipeline display updates are intercepted. Given the display architecture described above, there are three possible layers at which a remote display system may intercept graphical output and redirect it to the client: (1) the graphics library layer, (2) the framebuffer layer, and (3) the display driver layer. Figure 2.2 shows these interception points.

The *graphics library layer* sits between the applications and the window system proper. Interception at this layer is performed by providing replacement graphics

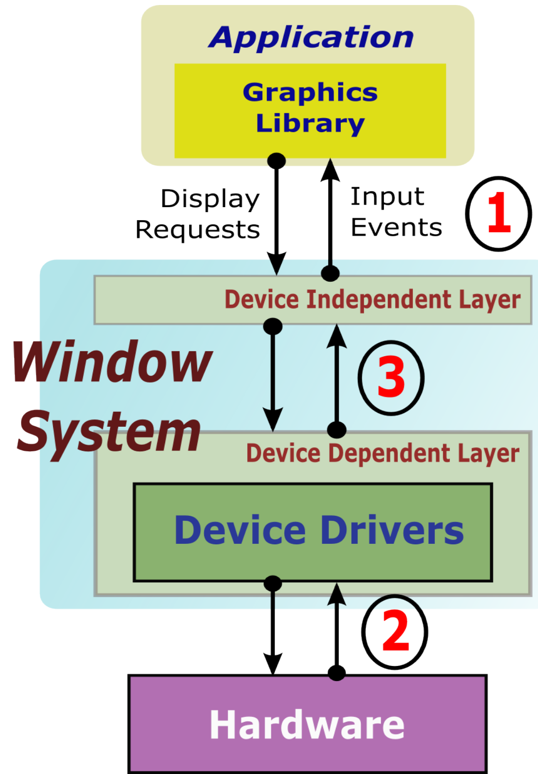


Figure 2.2 – Standard display pipeline. The possible interception layers are shown: 1. Graphics Library. 2. Hardware Framebuffer. 3. Device Driver. THINC uses the device driver layer for its virtualization architecture.

libraries, or in systems with client/server display architectures, a proxy server can provide the desired interception and redirection. In this model, everything below and including the window system is executed on the client.

Intercepting at this level has the advantage of giving the remote display system complete knowledge and control, since not only drawing requests, but also window and general management requests, are intercepted. Unfortunately it also has two drawbacks. First, executing the window system on the client results in the client holding a large amount of state (some of it possibly critical to the functioning of the desktop), which directly affects the mobility and ubiquitous access benefits of using a remote display system, and goes counter to one of our design goals. Second, since application logic and its user interface are typically tightly coupled, running the user

interface on the client and the logic on the server may result in a need for continuous synchronization over the network. In high-latency wide area network environments this kind of synchronization leads to substantial performance degradation.

The *framebuffer layer* sits at the bottom of the pipeline, and contains the finished rendered contents of the screen as they will be transmitted to the computer's monitor. When intercepting at this layer all display updates are reduced to raw pixel values. The resulting framebuffer data is read back, encoded, and compressed, a process called screen scraping.

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 must do the full translation from application display commands to actual pixel data, but clients can be made extremely simple and stateless, since all they are required to do is transfer pixel data from the network to the screen. Unfortunately, display updates consisting of raw pixels alone are typically too bandwidth-intensive. For example, using them to encode display updates for a video player displaying at 30 frames per second (fps) full-screen video clip on a typical 1024x768 24-bit resolution screen would require over 0.5 Gbps of network bandwidth.

While compressing the raw pixel data will alleviate the bandwidth consumption, generating display updates in this manner is fundamentally inefficient since the original display semantics are lost and cannot be used in the process. For example, an application request that fills the screen with one color, would result in the following process:

1. The framebuffer is filled with the requested color,

2. The complete screen is marked as having changed,

3. The remote display system reads back the full framebuffer contents and tries to

compress it, and

4. The compression algorithm realizes the screen is filled with one color and sends this simple update to the client.

If the server had some a-priori notion of what was drawn it could have done a much more efficient translation from the color fill to what it sent to the client.

Finally, the *device driver layer* sits below the window server proper and above the framebuffer. This is a well-defined, low-level, device-dependent layer that presents the video hardware and its capabilities to the display system above. In a typical desktop system, this layer is used to implement hardware-specific display drivers that enable the use of a particular video card.

Intercepting at this layer provides a best of both worlds approach when compared with the first two layers. There's enough state and information for the remote display to optimize the translation from graphics updates to remote display commands. Very little state is kept within the clients, allowing users to be very mobile. Clients also have the potential to be very simple in the case where the protocol mimics the device driver interface. Its main drawbacks come from the fact that its neither intercepting with full knowledge, as the graphics library layer does, nor does it have the straightforward simplicity of only having to deal with raw pixel values as intercepting at the framebuffer does. It also has some implementation challenges as some of the abstractions of the device driver layer may have strong assumptions about the underlying hardware, and in some cases may desire or require direct access to it. This latter issue is more prevalent when dealing with 3D applications.

As Figure 2.3 shows, THINC's architecture is based on intercepting at this layer. Instead of providing a driver for a particular piece of display hardware, THINC virtualizes the display by introducing a simple display driver that intercepts drawing

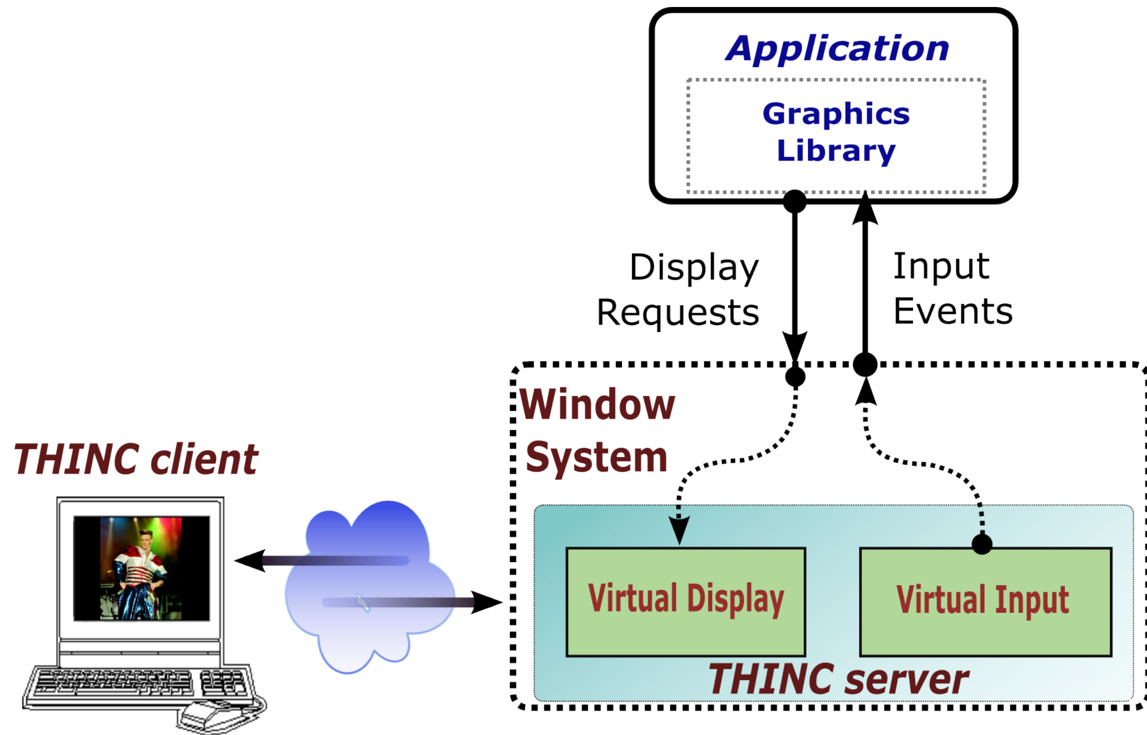


Figure 2.3 – THINC virtual display architecture

commands, packetizes them, and sends them over the network to a client device to display. We call this a virtual display driver because once THINC runs, the window system and the applications continue to believe they have access to a physical video card, with its associated framebuffer and video memory, and use it the same way as they would a traditional graphics card. The virtual display driver is also able to abstract any differences that various client hardware may have, and provide a consistent view as users move from one client to another. Similarly, THINC uses virtual input device drivers in order to handle input events coming from input devices in the client, such as its mouse and keyboard. The input drivers take care of transparently passing events back to the system and applications. In addition, they provide consistency as users connect from different clients, with potentially different input devices.

THINC's virtual device approach provides several important benefits directly re-

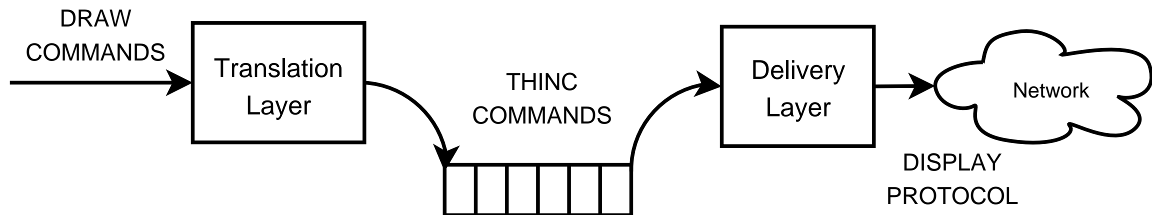


Figure 2.4 – THINC architecture components

lated to our design goals. First, because the device layer sits below the window server proper, THINC avoids re-implementing display system functionality already available, resulting in a simpler system that can leverage existing investments in window server technology. Second, using a standard interface enables THINC to work seamlessly with existing unmodified applications, window systems, and operating systems. Third, THINC can support new video hardware features with at most the same amount of work necessary to support them in traditional hardware-specific display drivers, allowing it to keep pace with continuous developments in desktop graphics. Fourth, since the video device driver layer still provides semantic information regarding application display commands, THINC can utilize those semantics to encode application commands and transmit them from the server to the client in a manner that is both computationally and bandwidth efficient.

The virtual device driver encapsulates THINC's three architectural components: (1) the device driver interface which exposes the drawing functionality to the window system using the standard display driver interface, (2) the translation layer which is in charge of taking display driver commands and converting them to THINC protocol primitives, and (3) the delivery layer which takes the generated protocol primitives and delivers them to the client.

Figure 2.4 shows these components, and how they interact with each other. The process to generate display updates as applications draw to the screen is as follows:

- A handler for each supported draw operation is exported from THINC's virtual device driver to the window system. As application draw requests are received, the window system calls the appropriate handler to service the request.

- Each handler knows the corresponding set of THINC commands which should be generated in response to its respective draw operation. When the handler is called, it examines the request (its arguments and any context or state associated with the request) and decides which command should be generated. The handler now passes a request to the translation layer to generate a new command of the appropriate type.

- The translation layer takes all necessary information, and creates a new command. In the simplest scenario, this command is then immediately passed to the delivery layer to be sent to the client. However, as will be discussed later on, in many cases the command may not affect the visible parts of the screen, and instead may be recorded for future usage.

- Once the delivery layer receives the command, it may decide to send it immediately, or, in the more common case, buffer it to be sent at a later time. Buffering allows THINC to employ a number of mechanisms that improve the performance of the system and its interactive response.

- Eventually, the delivery layer decides to send the command to the client, at which point it will modify the command to fit the current characteristics of the client, and then generate the network representation of the command, and send it over the network.

Choosing the interception point in the graphics pipeline provides the basic underpinning for the architecture of THINC. Our choice of intercepting at the device

driver layer provides the right set of characteristics and benefits for our design goals. It also helps drive the choices made for the rest of the system. The following sections will focus on the remaining design choices: what primitives to use to send updates over the network, and how application commands are translated to these primitives and delivered to the client.

2.3 Remote Display Protocol

The second choice to be made when designing a remote display system is what primitives are used for sending updates over the network to the client. These display primitives are collectively called the remote display protocol. As discussed before, this is the area where most display systems have focused their attention as they seek to improve the performance of their systems. In many cases, the choice of display primitives is tightly coupled with the choice of where to intercept in the graphics pipeline. For example, systems intercepting at the graphics layer tend to use high level primitives that closely resemble the requests they are intercepting. Similarly, systems intercepting at the framebuffer layer tend to choose to have a single primitive which carries compressed pixel values that they extract from the framebuffer.

The choice of display primitives is an important one because it has a direct correlation to bandwidth consumption and the complexity of the clients. High level primitives are very efficient at encoding complex drawing operations. For example, sending an order to the client to draw a line on the screen is many times more compact than sending all the pixels affected once the line is drawn on the screen. At the same time, these high level primitives require the client to have large complex logic to be able to execute the orders. Continuing with the line drawing example, the client will need to be able to draw different types of lines, and perform anti-aliasing when

drawing diagonal lines. High level primitives can also have large state associated with them. This state needs to be kept at the client so it can execute the drawing operation correctly, but it also needs to be replicated on the server so clients can disconnect and reconnect.

As expected, systems at the other end of the spectrum which employ only one primitive have very simple and stateless clients: all they have to do is read packets from the network, decode the pixel values, and put them on the screen. Some of the compression algorithms used by these systems can be decoded directly in hardware (dedicated or in more recent times using the computational facilities of modern video cards), which leads to very specialized, compact, and straightforward clients.

The main drawback to this approach comes from having to use the same strategy for any possible change in the screen. This one size fits all approach needs to deal with disparate situations like how to encode a simple update which fills the screen with one color, more complex discrete graphics like high resolution text displayed by document processing applications, and fast changing, or continuous color graphics displayed by video playback and photo processing applications. Designers for these systems are normally faced with the choice of optimizing their algorithms for the most common case at the expense of other scenarios, or providing an algorithm that can deliver average performance for all cases. Traditionally the choice has been to provide a good experience for office-centric document processing applications, the most common scenario where remote display systems were used. However, with an increasing user base with much broader application needs, and the advent of richer user interfaces, these systems have been forced to develop more general algorithms with higher computational needs or dedicated offloading hardware[85, 108, 153].

THINC follows the pattern of having its protocol be tied to the decision of where graphics updates are intercepted. It uses a small set of low-level display commands

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 2.1 – THINC Protocol Display Commands. See Appendix A for a complete description

that mirror a subset of the video display driver interface. The five commands used in THINC's display protocol are listed in Table 2.1. These commands were chosen because they are ubiquitously supported, simple to implement, and easily portable to a range of environments. They mimic operations commonly found in display hardware found on client devices, and represent a subset of operations accelerated by most graphics subsystems. Graphics acceleration interfaces for all major operating systems use a set of operations which can be synthesized using THINC's 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, greatly reducing display processing latency.

THINC 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 frame-buffer 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 series of fixed-size regions 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, optionally, a background color.

For high fidelity display, all THINC commands are designed to support full 24-bit color as well as an alpha channel, a feature not supported by remote display systems that execute the graphical user interface of applications on the server. The alpha channel enables THINC to support graphics compositing operations [111] and work with more advanced window system features that depend on these operations, such as anti-aliased text. Although graphics compositing operations have been used in the 3D graphics world for some time, only recently have they been used in the context of 2D desktop graphics. As a result, there is currently a dearth of support for hardware acceleration of these operations, particularly with low-end 2D only cards commonly used in more modest machines. In particular, low-end 2D-only cards, which provide the perfect platform for remote display systems, often lack this type of support.

THINC provides support for graphics composition by leveraging available client hardware acceleration support only when present. In its absence, THINC's virtual device driver approach allows it to transparently fall back to the software implementation provided by the window system precisely for video cards lacking hardware support. By doing so, THINC guarantees the simplicity of the client while utilizing the faster server CPU to perform the software rendering.

The choice of primitives used by the THINC protocol attempts to strike a balance, where typical screen operations can be represented directly and delivered efficiently, while still allowing clients to be simple and stateless. For more complex drawing operations it can fall back to generic, raw pixel values, where it can take advantage of advances in compression algorithms, or, as we will discuss in Chapter 3, leveraging hardware acceleration interfaces to use more scenario-specific primitives, for example for video playback.

2.4 Display Update Translation

The final design choice for developing a remote display system is the process by which intercepted graphics commands from applications are translated to protocol primitives and delivered to the client. Given our focus on the architecture of the system as the core mechanism for providing good performance, this part of THINC is the most important one, and the one where we spent most of our effort. We believe that by providing an efficient and smart translation, leveraging semantic information available at our interception point, and appropriately choosing when to send commands, and which commands to send, THINC can provide superior performance and meet our design goals.

For this reason, we have split this discussion in two parts. This section describes the translation process, how it leverages semantic information provided at the time we intercept updates, and the data structures and optimizations we developed to make the process efficient. In the following section we will discuss the delivery process, its interface with the translation component, and how the system decides when and which updates to send.

The key aspect behind THINC's translation mechanism is how it utilizes the

virtual display approach to transparently intercept application display commands and translate them efficiently into THINC commands. There are three important principles in how the translation is performed:

- First, as the window server processes application requests, THINC intercepts display commands and translates the result into its own commands. By translating at the time the application display commands are processed, THINC can use the semantic information available about the command (and which is lost once processing is finished), to identify which command or commands should be used. In particular, THINC can know precisely what display primitives are used at the display device layer instead of attempting to infer those primitives after the fact. THINC translation in many cases becomes a simple one-to-one mapping to the respective THINC command. For example, a fill operation to color a region of the screen a given color is easily mapped to a `SFILL` command.

- Second, THINC decouples the processing of application display commands and their network transmission. This allows THINC to aggregate small display updates into larger ones before they are sent to the client, and is helpful in many situations. For example, sending a display update for rendering a single character can result in high overhead when there are many small display updates being generated. Similarly, some application display commands can result in many small display primitives being generated at the display device layer. Rasterizing a large image is often done by rendering individual scan lines. The cost of individually processing and sending scan lines can degrade system performance when an application does extensive image manipulation.

- Third, THINC preserves command semantics throughout the processing of application display requests and manipulation of the resulting commands. Since

THINC commands are not immediately dispatched as they are generated by the server, it is important to ensure that they are correctly queued and their semantic information preserved throughout the command's lifetime. For example, it is not uncommon for regions of display data to be copied and manipulated. If copying from one display region to another is done by simply copying the raw pixel values, the original command semantics will be lost in the copied region. If THINC commands were reduced to raw pixels at any time, semantic information regarding those commands would be lost making it difficult to revert back to the original commands to efficiently transmit them over the network.

THINC's translation layer builds on these three principles by utilizing two basic objects: the *protocol command object*, and the *command queue object*. Protocol command objects, or just *command objects*, are implemented in an object-oriented fashion. They are based on a generic interface that allows the THINC server to operate on the commands, without having to know each command's specific details. On top of this generic interface, each protocol command provides its own concrete implementation.

As previously mentioned, translated commands are not instantly dispatched to the client. Instead, depending on where drawing occurs and current conditions in the system, commands normally need to be stored and groups of commands may need to be manipulated as a single entity. To handle command processing, THINC introduces the notion of a *command queue*. A command queue is a queue where commands drawing to a particular region are ordered according to their arrival time. The command queue keeps track of commands affecting its draw region, and guarantees that only those commands relevant to the current contents of the region are in the queue. As application drawing occurs, the contents of the region may be overwritten.

In the same manner, as commands are generated in response to these new draw operations, they may overwrite existing commands either partially or fully. As commands are overwritten they may become irrelevant, and thus are evicted from the queue. Command queues provide a powerful mechanism for THINC to manage groups of commands as a single entity. For example, queues can be merged and the resulting queue will maintain the queue properties automatically.

To guarantee correct drawing as commands are overwritten, the queue distinguishes among three types of commands based on how they overwrite and are overwritten by other commands.

- *Partial* commands are opaque commands which can be partially or completely overwritten by other commands. For example, a `RAW` command representing an image to be displayed inside a window could have parts of it obscured by another window, or completely overwritten before it is sent over the network. In the case where the *partial* command is partially overwritten it is more efficient to clip it than to send the original unmodified command.

- *Complete* commands are opaque commands that can only be completely overwritten. The distinction between *complete* and *partial* commands is made for performance reasons. Sometimes it is more expensive to break up a command and send the partially overwritten result, than to transmit the original command. Both `SFILL` and `PFILL` are examples of complete commands. To help illustrate this point, let us consider the case of the SFILL command. This command has the following structure:

  ```
  | fill color | list of rectangles to fill |
  ```

  Now lets examine the case of a typical web page with a solid white background,

and some text and images. The way this page will be rendered will roughly follow this process:

1. Fill the browser window with white color.
2. Add images on top of the white background.
3. Render text on top of the white background.

If the SFILL command resulting from step 1 was allowed to be overwritten by the subsequent commands generated by steps 2 and 3, the list of rectangles to fill would grow from a single rectangle, to possibly tens or hundreds, depending on the complexity of the additional web page elements. Breaking up the SFILL command in this manner would result in additional computational complexity at the server to properly manipulate the command, increased bandwidth use dominated by the extra number of rectangles (and very possibly negating the advantages of using the SFILL command in the first place), and a performance impact on the client which would need to fill each single rectangle in turn.

- Finally, *Transparent* commands are commands that depend on commands previously generated and do not overwrite commands already in the queue.

The command queue guarantees that the overlap properties of each command type are preserved at all times.

2.4.1 Offscreen Drawing

Today's graphic applications use a drawing model where the user interface is prepared using offscreen video memory; that is, the interface is computed offscreen and copied onscreen only when it is ready to present to the user. This idea is similar to the double-

and triple-buffering methods used in video and 3D-intensive applications. Although this practice provides the user with a more pleasant experience on a regular local desktop client, it can pose a serious performance problem for remote display systems. Remote display systems typically ignore all offscreen commands since they do not directly result in any visible change to the framebuffer. Only when offscreen data are copied onscreen does the remote display server send a corresponding display update to the client. However, all semantic information regarding the offscreen data has been lost at this point and the server must resort to using raw pixel drawing commands for the onscreen display update. This can be very bandwidth-intensive if there are many offscreen operations that result in large onscreen updates. Even if the updates can be successfully compressed, this process can be computationally expensive and would impose additional load on the server.

To deliver effective performance for applications that use offscreen drawing operations, THINC provides a translation optimization that tracks drawing commands as they occur in offscreen memory. The server then sends only those commands that affect the display when offscreen data are copied onscreen. THINC implements this by keeping a command queue for each offscreen region where drawing occurs. When a draw command is received by THINC with an offscreen destination, a THINC protocol command object is generated and added to the command queue associated with the destination offscreen region. The command queue guarantees that only relevant commands are stored for each offscreen region, while allowing new commands to be merged with existing commands of the same kind that draw next to each other.

THINC's offscreen awareness mechanism also accounts for applications that create a hierarchy of offscreen regions to help them manage the drawing of their graphical interfaces. Smaller offscreen regions are used to draw simple elements, which are then combined with larger offscreen regions to form more complex elements. This is

accomplished by copying the contents of one offscreen region to another. To preserve display content semantics across these copy operations, THINC mimics the process by copying the group of commands that draw on the source region to the destination region's queue and modifying them to reflect their new location. Note that the commands cannot simply be moved from one queue to the other since an offscreen region may be used multiple times as source for a copy.

When offscreen data are copied onscreen, THINC executes the queue of display commands associated with the respective offscreen region. Because the display primitives in the queue are already encoded as THINC commands, THINC's execution stage normally entails little more than extracting the relevant data from the command's structure and passing it to the functions in charge of formatting and outputting THINC protocol commands to be sent to the client. The simplicity of this stage is crucial to the performance of the offscreen mechanism since it should behave equivalently to a local desktop client that transfers pixel data from offscreen to onscreen memory.

In monitoring offscreen operations, THINC incurs some tracking and translation overhead compared to systems that completely ignore offscreen operations. However, the dominant cost of offscreen operations is the actual drawing that occurs, which is the same regardless of whether the operations are tracked or ignored. As a result, THINC's offscreen awareness imposes negligible overhead and yields substantial improvements in overall system performance, as demonstrated in Section 2.7.

2.5 Display Update Delivery

THINC schedules commands to be sent from server to client with interactive responsiveness and latency tolerance as a top priority. THINC maintains a per-client com-

mand buffer based on the command queue structure described in Section 2.4 to keep track of commands that need to be sent to the client. While the client buffer maintains command ordering based on arrival time, THINC does not necessarily follow this ordering when delivering commands over the network. Instead, alongside the client buffer THINC provides a multi-queue *Shortest-Remaining-Size-First (SRSF)* preemptive scheduler, analogous to Shortest-Remaining-Processing-Time (SRPT). SRPT is known to be optimal for minimizing mean response time, a primary goal in improving the interactivity of a system [12]. The size of a command refers to its size in bytes, not its size in terms of the number of pixels it updates. THINC uses remaining size instead of the command's original size to shorten the delay between delivery of segments of a display update and to minimize artifacts due to partially sent commands. Commands are sorted in multiple queues in increasing order with respect to the amount of data needed to deliver them to the client. Each queue represents a size range, and commands within the queue are ordered by arrival time. The current implementation uses ten queues with powers of two representing queue size boundaries. When a command is added to the client's command buffer, the scheduler chooses the appropriate queue to store it. The commands are then flushed in increasing queue order.

Reordering of commands is possible with guaranteed correct final output as long as any dependencies between a command and commands issued before it are handled correctly. To demonstrate how THINC's scheduler guarantees correct drawing, we distinguish between opaque and transparent commands, and between the two classes of opaque commands, partial and complete.

- Opaque commands completely overwrite their destination region. Therefore, dependency problems can arise after reordering only if an earlier-queued com-

mand can draw over the output of a later-queued command. However, this situation cannot occur for partial commands because the command queue guarantees that no overlap exists among these types of commands. Furthermore, since complete commands are typical of various types of fills such as solid fills, their size is constantly small and they are guaranteed to end up in the first scheduler queue. Since each queue is ordered by arrival time, it is not possible for these commands to overwrite later similar commands.

- On the other hand, transparent commands need to be handled more carefully because they explicitly depend on the output of commands drawn before them. To guarantee efficient scheduling, THINC schedules a transparent command C using a two step process.

  1. Dependencies are found by computing the overlap between the output region of C and the output region of existing buffered commands. C will depend on all those commands with which it overlaps.
  2. From the set of dependencies, the largest command L is chosen, and the new command is added to the back of the queue where L currently resides.

  In this way, as queues are flushed in increasing order, THINC's approach guarantees that all commands upon which C depends will have been completely drawn before C itself is sent to the client. Although more sophisticated approaches could be used to allow the reordering of transparent commands, we found that their additional complexity outweighed any potential benefits to the performance of the system.

In addition to the queues for normal commands, the scheduler has a *real-time* queue for commands with high interactivity needs. Commands in the real-time queue

take priority and preempt commands in the normal queues. Real-time commands are small to medium-sized and are issued in direct response to user interaction with the applications. For example, when the user clicks on a button or enters keyboard input, she expects immediate feedback from the system in the form of a pressed button image. Because a video driver does not have a notion of a button or other high-level primitives, THINC defines a small-sized region around the location of the last received input event. By marking updates which overlap these regions as real-time and delivering them sooner as opposed to later, THINC improves the user-perceived responsiveness of the system.

THINC sends commands to the client using a *server-push* architecture, where display updates are *pushed* to the client as soon as they are generated. In contrast to the *client-pull* model used by popular systems such as VNC [150] and GoToMyPC [46], server-push maximizes display response time by obviating the need for a round trip delay on every update. This is particularly important for display-intensive applications such as video playback since updates are generated faster than the rate at which the client can send update requests back to the server. Furthermore, a server-push model minimizes the impact of network latency on the responsiveness of the system because it requires no client-server synchronization, whereas a client-driven system has an update delay of at least half the round-trip time in the network.

Although a push mechanism can outperform client-pull systems, a server blindly pushing data to clients can quickly overwhelm slow or congested networks and slowly responding clients. In this situation, the server may have to block or buffer updates. If updates are not buffered carefully and the state of the display continues to change, outdated content is sent to the client before relevant updates can be delivered.

Blocking can have potentially worse effects. Display systems are commonly built around a monolithic server core which manages display and input events, and where

display drivers are integrated. If the video device driver blocks, the core display server also blocks. As a result, the system becomes unresponsive since neither application requests nor user input events can be serviced. In display systems where applications send requests to the window system using IPC mechanisms, blocking may eventually cause applications to also block after the IPC buffers are filled.

The THINC server guarantees correct buffering and low overhead display update management by using its command queue-based client buffer. The client buffer ensures that outdated commands are automatically evicted. THINC periodically attempts to flush the buffer using its SRSF scheduler in a two stage process. First, each command in the buffer's queue is committed to the network layer by using the command's flush handler. Since the server can detect if it will block when attempting to write to a socket, it can postpone the command until the next flush period. Second, to protect the server from blocking on large updates, a command's flush handler is required to guarantee non-blocking operation during the commit by breaking large commands into smaller updates. When the handler detects that it cannot continue without blocking, it reformats the command to reflect the portion that was committed and informs the server to stop flushing the buffer. Commands are not broken up in advance to minimize overhead and allow the system to adapt to changing conditions.

2.6 Implementation

THINC's remote display architecture has been implemented for both the X Window System [122] in Linux, and for Microsoft Windows. Most of the implementation effort on THINC has focused on performance and portability. This section discusses the most important details of this effort. We focus on the implementation details for the X/Linux version of THINC. For a detailed description of the Microsoft Windows

implementation effort, the reader is referred to [174].

The THINC server consists of about 25,000 lines of C code. To maximize its portability, the system is divided in two parts. The *front end*, which interfaces directly with the native window system by implementing its device driver API, and the *back end*, which implements the core THINC display architecture, and is completely window and operating system independent. The back end is described first, as it provides the core functionality around which the front end is built. At the end of the section, some implementation details specific to providing remote display are discussed.

2.6.1 Back End

The back end encapsulates the core THINC functionality, and provides three interface points for the front end. First, an API for converting display updates into THINC protocol commands. Second, an interface for buffering commands, either in off-screen regions or as on-screen protocol updates. And third, the back end provides the necessary infrastructure for delivering buffered commands to any number of connected clients. In rough terms, the first and second interfaces correspond to the translation layer, as described in Section 2.4, and the third interface corresponds to the delivery layer, as described in Section 2.5.

2.6.1.1 Creating Commands

Converting display updates for the front end is accomplished by creating objects that represent the respective THINC protocol command. The purpose of these objects is to encapsulate all information necessary to send the display update to the client, while allowing the server to manipulate it before delivery.

An object oriented approach is used, with a generic `Command` superclass, and

derived subclasses for each type of protocol message (`RAW, COPY, SFILL, PFILL, and BITMAP`). The generic command object stores the type of the command, a set of flags, the bounding box where the command draws, and a reference count. Subclasses extend this class to store information specific to the type of command. For example, the `PFILL` subclass stores the dimensions, the size, and the pixel data of the tile used to perform the fill.

The base command class also defines a generic interface that must be implemented by all subclasses. It consists of the following functions: `Create`, `Destroy`, `Copy`, `Modify`, `GetInfo`, `Execute`, and `Flush`. As their names imply, the first three functions create, destroy, and copy a command object, respectively. `Modify` is used to transform the command in some specific way. Available modifications are `clip`, `move/translate`, and `merge` the command with another command of the same type. `GetInfo` is used to obtain specific information about the command, most commonly information which needs to be computed dynamically. For example, its size (used for scheduling purposes) and the specific region where it draws (not just its bounding box). The final two functions represent the final stages in the lifetime of a command. `Execute` is used to move a command from an off-screen area to on-screen, therefore buffering it for delivery. `Flush` is used to deliver a command object to the client. Both of these processes are described in more detail below.

2.6.1.2 Adding and Manipulating Commands

Once a command object has been created, it needs to be injected into the system. This is accomplished by adding it into a client's buffer or in the queue for an off-screen area. In both cases the process is the same, as both use the same *command queue* structure described in Section 2.4. This process is shown in Algorithms 2.1, 2.2, and 2.3.

Algorithm 2.1: QueueCommand(cmd, queue)

```
1 if cmd not transparent then
2     OverwriteCommands(cmd, queue) ;           /* See Algorithm 2.2 */
3 end
4 merged ← TryMerge(cmd, queue) ;               /* See Algorithm 2.3 */
5 if merged is TRUE then
6     Destroy(cmd)
7 else
8     add cmd to tail of queue
9 end
```

Algorithm 2.1 shows `QueueCommand`, the main function used to add a command object to a queue. QueueCommand's processing is divided in three stages. First, it guarantees that only relevant commands are in the queue taking into account the new command. Second, it tries to minimize the length of the queue by merging commands whenever possible. Third, if merging is not possible, it adds the command to the *end* of queue, to guarantee correctness once the commands in the queue are delivered to the client.

As shown in line 1, `QueueCommand` only needs to check for irrelevant commands if the new command is opaque[1]. If the command is transparent, the queue's existing contents are still relevant and `QueueCommand` goes on to the next step. If the command is opaque, the auxiliary function `OverwriteCommands` is called.

Algorithm 2.2 details `OverwriteCommands`. In the current implementation, command queues are implemented as doubly linked-lists, and `OverwriteCommands` walks the list, checking each command in turn to see if it is overwritten by the new addition, either partially (10) or completely (7). Notice how in the partial overwrite case the command's Modify function is called, allowing the overwritten command to handle clipping itself. For example, in the case of *complete* opaque commands, the command

[1]See Section 2.4 for details on opaque and transparent commands

Algorithm 2.2: OverwriteCommands(cmd, queue)

```
1  reg ← GetInfo(cmd, REGION)
2  foreach oldcmd in queue do
3      oldbox ← BoundingBox(oldcmd)
4      if oldbox is Outside(reg) then
5          continue
6      else if oldbox is Inside(reg) then
7          delete oldcmd from queue
8          Destroy(oldcmd)
9      else
10         Modify(oldcmd, CLIP, reg)
11     end
12 end
```

will just ignore the CLIP request and return immediately. In the case of complete overwrite, the existing command is removed from the queue and destroyed.

After `OverwriteCommands` returns, the new command needs to be added to the queue. In some situations, it is possible to reduce overhead and improve performance by merging multiple commands into one. Combining multiple commands reduces the length of the queue, in turn reducing the time complexity of adding new commands to the queue in the future. Also, merging will result in bandwidth savings. The most common scenario where merging can be exploited is when displaying large compressed images, for example, in web browsers. These images are typically line-encoded, resulting in them being delivered to the display system one line at a time (i.e. each line is displayed right after it is decompressed). Without merging, a single large image would have to be wastefully delivered to the client as a multitude of RAW updates.

Algorithm 2.3 shows `TryMerge` which provides the basic merging functionality for THINC. By default, it will only try to merge with the last command in the queue. This is a sensible tradeoff between maximizing merging opportunities, and minimizing the performance hit of looking for these opportunities. The most important detail

Algorithm 2.3: TryMerge(cmd, queue)

```
1  if IsEmpty(queue) then
2      return FALSE
3  end
4  last ← LastElement(queue)
5  if type(last) not equal type(cmd) then
6      return FALSE
7  end
8  return Modify(last, MERGE, cmd)
   // Tries to merge command two into command one
9  RawTryMerge (one, two)
10 begin
11     oneb ←BoundingBox(one)
12     twob ←BoundingBox(two)
13     if (twob.x1 = oneb.x1) and (twob.x2 = oneb.x2) then
14         if twob.y2 = oneb.y1 then
               // one is below two, merge them
15             oneb.y1 := twob.y1
16             return TRUE
17         else if oneb.y2 = twob.y1 then
               // one is on top of two, merge them
18             oneb.y2 := twob.y2
19             return TRUE
20         else
               // Cannot merge
21             return FALSE
22         end
23     else if (twob.y1 = oneb.y1) and (twob.y2 = oneb.y2) then
24         if oneb.x2 = twob.x1 then
               // one is left of two, merge them
25             oneb.x2 := twob.x2
26             return TRUE
27         else if twob.x2 = oneb.x1 then
               // one is right of two, merge them
28             oneb.x1 := twob.x1
29             return TRUE
30         else
               // Cannot merge
31             return FALSE
32         end
33     else
           // Cannot merge
34         return FALSE
35     end
36 end
```

to note about `TryMerge` is that the majority of the merging logic is delegated to the type-specific methods. This is by design. The possibility of merging and performing the actual operation is highly dependent on the structure and characteristics of the command.

After checking that the commands can indeed be merged (i.e. they have the same type), `TryMerge` delegates to the type-specific function the actual merge operation. A simplified RAW merge function is shown in Algorithm 2.3 starting at line 9. It handles the basic case of merging two commands with a simple, one rectangle draw region, by extending the existing command's draw region to encompass the region of the new command.

As was just discussed, the command queue is currently implemented as a doubly linked list, which provides constant time insertions and removals, and a simple implementation. However, the time complexity of adding new commands is dominated by the cost of `OverwriteCommands`, which, given a linked list, is *O(n)*: each command in the queue needs to be checked for possible overwrite by the new command. In large off-screen areas or when a large part of the screen is changed this may lead to degraded performance. To avoid this penalty, we can envision using a more advanced data structure, for example a Quadtree [35], that can scale better to bigger screens, and more gracefully handle large bursts of commands.

After commands are added to a queue, they are either moved on-screen, or delivered over the network. The first process is called *Executing* the queue, the second is called *Flushing* it. Queue execution is normally performed in response to a request to copy some of the contents of an off-screen area to the visible part of the screen. For example:

```
CopyArea (srcx:50, srcy:100, width:650, height:400, dstx:150, dsty:200)
```

In this case, the contents of the off-screen rectangle with coordinates `(50,100)` $\rightarrow$ `(700,500)` are to be copied to the screen region with coordinates `(150, 200)` $\rightarrow$ `(800, 600)`. The process is exported by the back end as the `CopyQueue` function, and works as follows:

1. Given the source and destination regions, compute the horizontal and vertical delta. Commands will need to be translated by these amounts before being added to the destination queue.

2. Use the destination region to overwrite commands in the destination queue, in the same manner as described for `OverwriteCommands` before. It is important to note that overwrite will always happen. Although there may be transparent commands in the source queue, an opaque command is guaranteed to exist which draws in the same region as each of the transparent commands.

3. Use the source region to find the list of commands that should be copied to the destination queue.

4. For each command in this list, call its `Execute` function, passing translation parameters, and the destination region. The command is supposed to create a translated and clipped copy, then add the copy to the destination queue. The copy is needed because executing a queue (and in higher level terms, copying an off-screen area) does not entail removing the commands from the off-screen queue. The same offscreen region (and thus its commands) may well be used multiple times to draw to the screen.

2.6.1.3 Abstracting Command Destinations

Once commands are ready to be delivered by the system, they need to be serialized and written out to their destination. Before describing this process, we first describe how THINC abstracts the details of where commands are being redirected, allowing it to redirect its output anywhere.

The back end does not make any assumptions about the destination of the commands, and treats all destinations in the same manner. This is accomplished using a technique similar to that used for manipulating commands: all details of the destination are abstracted by a generic `Client` interface that all destinations must implement, and through which THINC controls and uses each destination. In the description that follows, we refer to the implementation of a particular destination as a *client*.

In addition to masking the intricacies of a particular client, this approach also allows many different types of client to be used simultaneously. For example, one client may be used for remote display over the network, while at the same time another client is recording all output to disk. THINC allows multiple connected clients by keeping an instance of the `Client` class for each of them in an internal linked list, to which clients can be added and removed dynamically.

The `Client` interface consists of the following methods:

- `Buffer`. This function is called to schedule a command for delivery. In most cases, clients are assumed to have an output buffer implemented using a command queue. They may also have a scheduler that prioritizes updates in the buffer. This function allows clients to implement this functionality. In the normal scenario, a client will receive the command to be delivered, add it to its buffer using `QueueCommand`, and schedule it using the SRSF scheduler described

in Section 2.5. Both functions are exported to the clients by the back end, since they are independent of the client's details.

- `Flush`. This function performs the actual serialization and writing to destination of commands. As opposed to `buffer` which operates on one command at a time, this operation is meant to operate on all of the commands buffered in the client. The client walks its buffer, and for each command it calls the command's `Flush` method. If delivery is successful, the command should be removed from the client's buffer. The only case where `flush` may not operate on a set of commands is the case where the client performs no buffering. In this case, `buffer` and `flush` may be merged into a single operation, by delivering the command inside `buffer`, and converting `flush` to a `NoOp`.

  The order in which commands are delivered depends on the details of the client. However, for performance reasons this ordering is expected to be determined during the `buffer` operation. For example, a client which uses the SRSF scheduler will have the scheduler determine the queue in which the command should be buffered at insertion time, and, at delivery time, all it needs to do is flush each scheduler queue in turn.

  When `flush` is called, the client is expected to attempt to deliver all pending commands. If it is unable to do so, for example, in the case of a remote display client if the network is congested, the client's `flush` function should inform the back end of this fact. The back end will in turn set up a timer, which once expired, will retry the operation. This functionality is a crucial component of our X Window System implementation. The X server, inside which THINC resides, is a single threaded application. If a client were to block trying to deliver all buffered commands, the X server itself would also block waiting for

this operation to continue. As a result, the whole desktop would freeze since no application requests or input events could be processed.

- `Read` and `Write`. These two methods provide a lower level interface to the client's destination. They allow clients to leverage generic functionality from the back end (for example, the output stack described in the next section), while still abstracting the details of how data is delivered to the destination. They also allow THINC to directly send or read data from the client, for example during the initial handshake.

- `Close`. As its name implies, this method simply shutdowns the client, and frees all resources associated with it.

2.6.1.4 Delivering Commands

The actual serialization of a command is performed by its corresponding `Flush` function. To make this process extensible, maintainable, and allow it to be changed dynamically, THINC uses a stackable pipeline model based on the architecture of the Click modular router [64]. In this model, functionally independent modules are stacked and connected such that the output of one is passed as input to the next one.

This model has a number of benefits. Modules can be added and removed dynamically from the stack. In this way, changes in the client that require a different serialization process can be dealt with simply by adding or removing modules from the stack. In addition, since each module is functionally separate from the rest of the process, new modules can be easily prototyped and tested, which greatly helped us in the development of THINC.

Each type of command has a set of modules that make up its output stack, depending on the needs of its serialization process. The simplest types (e.g. `SFILL`)

will have a couple of modules that collect statistics, and a module where the command data its actually sent to the client (using the client's `write` function). On the other hand, a type like `RAW` has modules to deal with a number of situations, for example, extracting the command's data from the framebuffer, caching this data to minimize the amount of data sent to the client, compressing the update, and finally sending it to the client.

The output stack structure is implemented as a linked list of modules. Each module consists of a name, the module's function, and a priority. Modules in the list are ordered by increasing priority order. For simplicity reasons, priorities must be assigned manually when the modules are first declared and added to the stack, and they cannot change while the module is inside the stack. In addition, a handle object is used to transfer data from one module to another. The handle is specific to each type's implementation, and it is treated as an opaque object by the output stack code. To output a command, the stack is *executed* by traversing the list and calling each module's function in turn, passing it the corresponding handle. At any point a function may signal the stack to stop execution, either temporarily or permanently. In the first case, the last module called is saved, and the next time execution is requested, the process will resume from that saved module. This is particularly useful for cases where a response is required before executing the next module, for example, during the handshake process executed when new clients connect to THINC. In the second case, no state is saved. Subsequent executions will result in the first module on the list to be called.

Once all the modules in the stack have been called, the command is assumed to have been delivered to the client and the `Flush` operation is finished.

2.6.2 Front End

As previously mentioned, the front end is in charge of interfacing with a particular window system by implementing its device driver API, and leveraging the back end functionality to provide a virtualized display.

The X Window System provides display output using a single-threaded server process, the “X server”, to which applications connect and pass requests, and which in turn passes these requests down to the underlying video hardware through device drivers. Application requests are processed by the Device Independent Layer (DIX) of the X server. In turn, device drivers are part of the Device Dependent Layer (DDX). The DDX consists of glue code that interfaces with the DIX, plus a dynamic module loader that inserts driver code into the X server according to the underlying hardware. Our front end implementation leverages this particular architecture by implementing a device driver that supports an “imaginary” THINC video card, that can be loaded into the X server using a couple of configuration directives. This way, the front end can take advantage of the X server infrastructure and functionality, while encapsulating all of THINC's functionality.

Algorithm 2.4: Simplified X server main loop, showing THINC interception points

```
1 while True do
2     ...
3     BlockHandler()
4     select();              /* sleep until activity is detected */
5     WakeupHandler()
6     ...
7     handle display requests
8     ...
9 end
```

The front end provides the functionality required for THINC's operation by us-

ing the DDX driver interface to hook into the X server's main loop. As shown in Algorithm 2.4 THINC intercepts in three different places:

- The `WakeupHandler()` hook (on line 5) is called when the server wakes up from `select()` [127], in response to a received event (e.g. from input devices, applications, etc.). The front end uses the X server's `select` call to also monitor for events of interest to THINC, by adding any active file descriptors to the list monitored by `select`. Events of interest to THINC will be automatically reported to the front end. Three types of events are currently implemented:

  1. New client connections. When a new client attempts to connect to the THINC server, an event is received by the X server on THINC's main socket. In this case, the new connection is accepted, and the handshake process is initialized (described in more detail in 2.6.3).
  2. Messages from existing clients. These events consist mostly of input events, and control messages. The front end uses virtual mouse and keyboard drivers to inject these events into the X server. In this manner, they are interpreted by the X server as events coming from actual input hardware.
  3. Client closed connection. When a client disconnects, an event is received on the client's corresponding file descriptor. As previously described, the client's `close` method is called to free any internal resources associated with it. Since the client is stateless, no actual shutdown process is required before the client is allowed to disconnect.

- The `BlockHandler` hook (on line 3) is called when the X server is ready to call `select()` again, and go to sleep until a new event is received. At this moment, the front end calls into the back end to flush all buffered commands.

As previously mentioned, if trying to send updates would result on the server blocking, a timer is set up and the `BlockHandler` returns, allowing the server to go to sleep. Once the timer goes off, THINC tries to flush updates again. This process continues until all updates are flushed, or new ones are generated which overwrite existing ones.

- The final interception point occurs while handling draw requests from applications. These requests are received by the X server, then decoded and transformed into device driver requests. The front end intercepts these requests by setting up a number of function pointers, one for each available request. A draw function in the front end will receive the request, along with any extra parameters and decide which THINC command type to generate in response. Using the back end, the new command object will be created, then depending on whether the request is to an on-screen or off-screen area, the command will be buffered into clients, or added to the off-screen area's command queue, respectively. Once the command is passed to the back end, the front end device driver function is finished and returns control to the X server. As we just discussed, once the X server has finished serving all application requests, the `BlockHandler` will be called, and all new commands flushed to clients.

The front end is also in charge of initializing and maintain the virtual framebuffer for the X server. This framebuffer is created once the device driver is loaded, and the X server passes the characteristics of the display to the front end: width, height, and color depth. The front end takes these parameters, computes the appropriate size, and allocates enough system memory to fit the framebuffer. Finally, a pointer to the framebuffer is passed to the X server and to the back end.

The framebuffer is passed to the X server because the front end depends on it to

perform most of the rendering of application requests. The X server has a software implementation of all possible draw requests, allowing device drivers to only accelerate those operations supported by the underlying hardware. Unsupported operations simply use the generic implementation, which sometimes may take advantage of lower level operations which can be accelerated by the hardware. Since THINC has no underlying hardware, it must do all rendering in software. However, this can be easily accomplished with the virtual display driver approach, which allows the front end to simply leverage the X server's implementation to perform all drawing.

2.6.3 Remote Display Implementation

Finally, we explore some of the implementation details specific to the implementation of a remote display system based on our virtual display architecture. Our discussion is focused on the implementation details of managing remote clients.

One of the most important aspects of managing remote display clients is providing a secure service for accessing the desktop. The security model for THINC is divided in two components. First, an encrypted channel is set up before any other communication occurs, using TLS and the RC4 encryption algorithm (both provided by the OpenSSL [99] library). Second, all remote clients have to be authenticated before they are allowed to access the desktop. Currently, THINC only supports username/password authentication. For a username to be authenticated successfully, two conditions must be met: (1) the username needs to be a valid account on the server, and (2) the username must be the owner of the desktop session. THINC performs all authentication by leveraging standard Unix authentication mechanisms through the PAM authentication library [104].

Once a client has set up an encrypted channel, and it has been authenticated

by the server, a handshake process is initiated in which the client and the server negotiate and exchange all of the parameters needed to successfully initiate a remote display session. The handshake process has been designed to be easily extensible, allowing for new parameters to be added transparently, while maintaining backwards compatibility.

The handshake is based on a model where the client continuously asks questions to the server to find out the parameters, and the server replies with the appropriate values. It is influenced in large part on the standard SSH protocol [173]. The protocol only specifies the set of valid questions and answers. It does not specify (nor impose) the order in which questions have to be asked. This is by design, since it allows the protocol to be easily changed. An example question/answer pair follows:

Client	Display parameters?		
Server	Width	Height	Bits per pixel

The process is completely client-driven to allow the server to continue operating without blocking in the handshake process. The server implements the handshake process using an output stack (as described in 2.6.1.4), with a main module that handles all client requests. Other modules handle the security stages of the handshake, and the final stage of the process. The main module uses the stack's ability to temporarily stop execution to guarantee that it keeps getting called for each client question. The handshake finishes once the client sends a special `Done` message. At this moment, the handshake output stack is destroyed, and the client starts receiving desktop display updates.

If the client asks a question which the server does not know how to answer (i.e. it does not support that particular feature), the server simply replies with an `Unknown` message. The server may also reply with a `Reject` message if the client asks an invalid

question, or the server decides the client cannot be allowed to connect. Some parameters may require multiple client questions to be resolved, for example, for cases where the client first needs to find out if the server supports a feature, before asking for specific values. While the handshake protocol only specifies single question-answer pairs, and does not have a specific mechanism to group multiple questions, these groups are expected to be logically enforced by the server and client implementations. Appendix A contains the THINC protocol specification, including a list of all question/answer pairs currently defined.

2.7 Experimental Results

To demonstrate the effectiveness of THINC's remote display architecture, a direct comparison was conducted with a number of state-of-the-art and widely used remote display platforms, including Citrix MetaFrameXP [23], Microsoft Remote Desktop [25, 82], GoToMyPC [46], X [123], NX [95], Sun Ray [124, 141], and VNC [118, 150]. We follow common practice and refer to Citrix MetaFrameXP and Microsoft Remote Desktop by their respective remote display protocols, ICA (Independent Computing Architecture) and RDP (Remote Desktop Protocol).

For these experiments, we measured the performance of the systems on web applications in LAN and WAN environments. We also used a PC running the benchmark locally as a baseline representing today's prevalent desktop computer model.

We compared the performance of various remote display systems using an isolated network testbed, and we measured wide-area THINC performance using PlanetLab [21] nodes and other remote sites located around the world. As shown in Figure 2.5, our testbed consisted of six computers connected on a switched FastEthernet network: two clients, a packet monitor, a network emulator for emulating various

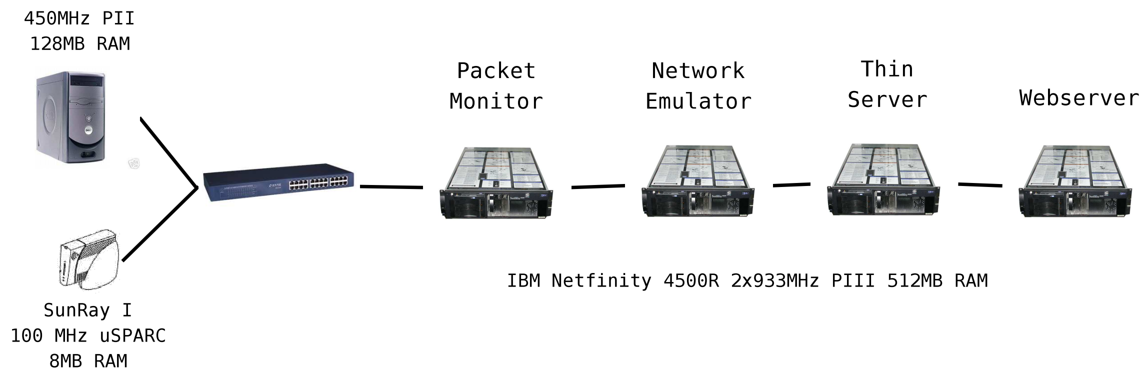


Figure 2.5 – Experimental Testbed

network environments, a remote display server, and a web server used for testing web applications. Except for the 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, and a Sun Ray I with a 100 MHz μSPARC processor and 8 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. All of the remote display systems used the PC as the client, except Sun Ray, for we which we used a Sun Ray I hardware thin client. All of the systems used the Netfinity server as the remote display server. For the three systems designed for Windows (ICA, RDP, and GoToMyPC), we ran Windows 2003 Server on the server and Windows XP Professional on the client. For the systems designed for X-based environments, we ran the Debian Unstable Linux distribution with the Linux 2.6.10 kernel on both server and client, except for Sun Ray, where we encountered a problem with audio playback that required us to revert to a 2.4.27 kernel. We used the latest remote display system versions available on each platform, namely Citrix MetaFrame XP Server for Windows Feature Release 3, Microsoft Re-

mote Desktop built into Windows XP and Windows 2003 using RDP 5.2, GoToMyPC 4.1, VNC 4.0, NX 1.4, Sun Ray 3.0, and XFree86 4.3.0 on Debian.

To minimize application environment differences, we used common remote display configuration options whenever possible. Client display was set to 24-bit color except for GoToMyPC which is limited to 8-bit color. To mimic realistic usage of the systems over public and insecure networks, we enabled RC4 encryption with 128-bit keys on all platforms which supported it. For those which did not, namely X and VNC, we used ssh to provide a secure tunnel through which all traffic was forwarded. The ssh tunnel was configured to use RC4. Following common practice, we configured X's ssh tunnel to also compress all traffic [41]. Any remaining remote display configuration settings were set to their defaults for a particular network environment. ICA, RDP, and NX were set to LAN settings when used in the LAN and WAN settings when used in the WAN. Some remote display systems used a persistent disk cache in addition to a per-session cache. To minimize variability, we left the persistent cache turned on but cleared it before every test was run.

We considered two different client display resolution and network configurations: *LAN Desktop* and *WAN Desktop*. LAN Desktop represents 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 [69]. We conducted our WAN experiments using the kind of high-bandwidth network environment that is becoming increasingly available in public settings [1].

GoToMyPC is only offered as an Internet service that connects the client and server using an intermediate hosted server through which all traffic is routed. As a result, we were unable to fully control the network configuration used. Our measurements show a 70 ms RTT between the intermediate GoToMyPC server used and our

Name	PlanetLab	Location	Distance
NY	yes	New York, NY, USA	5 miles
PA	yes	Philadelphia, PA, USA	78 miles
MA	yes	Cambridge, MA, USA	188 miles
MN	yes	St. Paul, MN, USA	1015 miles
NM	no	Albuquerque, NM, USA	1816 miles
CA	no	Stanford, CA, USA	2571 miles
CAN	yes	Waterloo, Canada	388 miles
IE	no	Maynooth, Ireland	3185 miles
PR	no	San Juan, Puerto Rico	1603 miles
FI	no	Helsinki, Finland	4123 miles
KR	yes	Seoul, Korea	6885 miles

Table 2.2 – Remote Sites for WAN Experiments

testbed, resulting in similar network latencies as our emulated WAN environment. We measured GoToMyPC performance without network emulation and referred to it as WAN Desktop.

We also measured remote display performance in WAN environments by running the server in our local testbed, but running the client on PlanetLab [21] nodes and other remote sites located around the world. Table 2.2 lists the sites used. Since the PlanetLab machines run User-Mode Linux, we were unable to run X-based remote display servers on these machines, and the use of Linux precluded any testing of Windows-based remote display systems. We were also prohibited from making significant modifications to the Linux installations at the non-PlanetLab sites. To measure THINC performance, we developed an instrumented headless version of the THINC client that could process all display data but did not output the result to any display hardware. We deployed this client on the remote sites and ran the same experiments as the WAN configuration.

Since most of the remote display systems tested used 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 testbed WAN environment and with remote sites whenever possible to take full advantage of the network bandwidth capacity available. However, PlanetLab nodes were limited to a window size of 256 KB due to their preconfigured system limits.

2.7.1 Web Browsing Benchmark

Web browsing performance was measured by running a benchmark based on the Web Page Load test i-Bench benchmark suite [54]. 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 remote display systems without introducing human timing errors. For remote site experiments with THINC, the headless client read a script of timed mouse coordinates and clicks to run the web benchmark. We used the Mozilla 1.6 browser set to full-screen resolution for all experiments to minimize application differences across platforms.

Since many of the remote display systems are closed and proprietary, we measured their performance in a noninvasive manner by capturing network traffic with a packet monitor and using a variant of slow-motion benchmarking [93, 70]. Our primary measure of web browsing performance is page download latency. Using slow-motion benchmarking, 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. However, this measure does not fully account for client processing time. To account for client processing time, we also instrumented the client window system to measure the time between the initial mouse input and the processing of the last graphical update for each page. We could only do this for X, VNC, NX, and THINC as we did not have access to client window system code for the other systems. Thus, our results provide a conservative comparison with Windows-based thin clients and Sun Ray for which we cannot fully account for client processing time.

2.7.2 Results

Figures 2.6 to 2.8 show web browsing performance results. Figure 2.6 shows the average latency per web page for each platform. For platforms in which we instrumented the window system to measure client processing time, the solid color bars show latency measured using network traffic, while the cross-hatched bars show a more complete measure by including client processing time. For example, Figure 2.6 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 the client. As shown in Figure 2.6, most of the systems did well in both LAN and WAN environments, having latencies below the one second threshold for users to have an uninterrupted browsing experience [94].

Figure 2.6 shows that THINC provides the fastest web page download latencies of all systems. THINC is up to 1.7 times faster in the LAN and up to 4.8 times

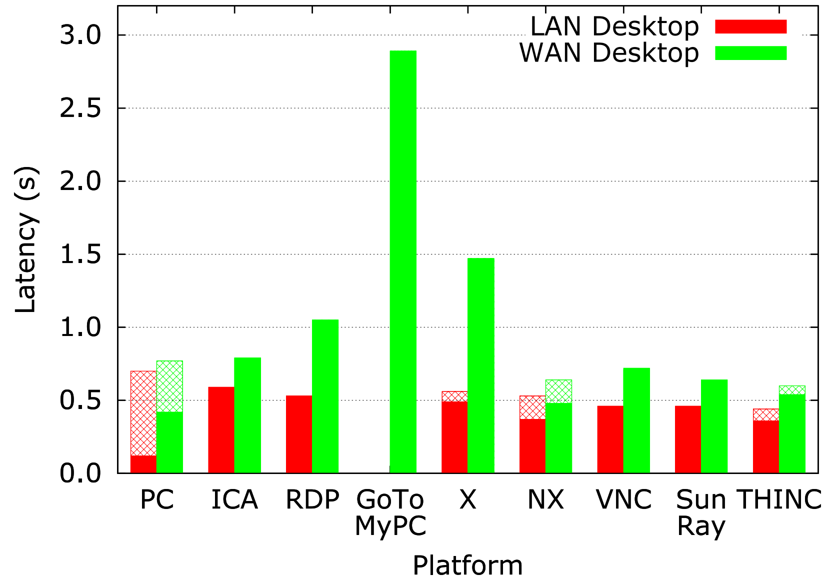


Figure 2.6 – Web Benchmark: Average Page Latency. Solid color bars show latency measured using network traffic, while the cross-hatched bars show a more complete measure by including client processing time.

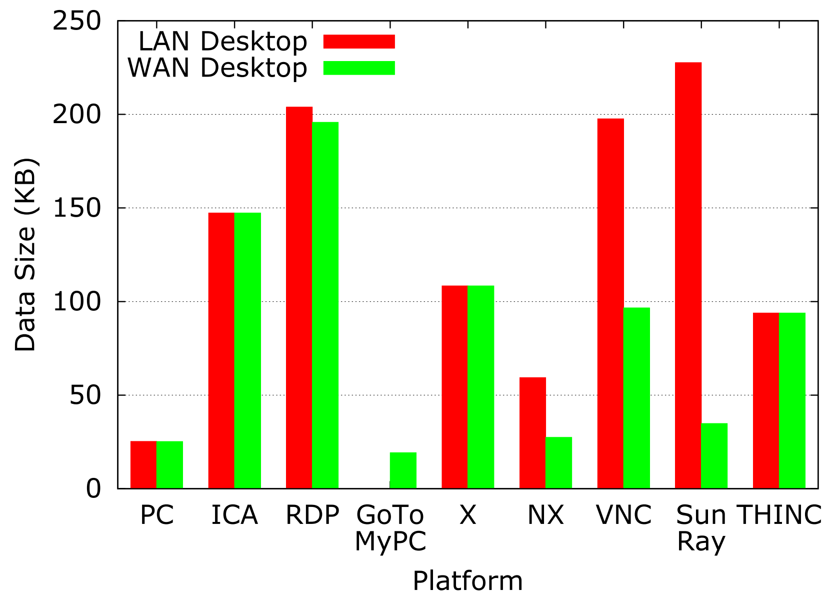


Figure 2.7 – Web Benchmark: Average Data Transferred per Web Page

faster in the WAN versus other systems. THINC outperforms the local PC by more than 60% because it leverages the faster server to process web pages more quickly than the web browser running on the slower client. Figures 2.6 shows that THINC does not suffer much performance degradation going from LAN to WAN, where it still outperforms all other platforms. In contrast, a higher-level approach such as X experiences the largest slowdown, performing about two and a half times worse due to the tight coupling required between applications on the server and the user interface on the client. While still slower than THINC, NX is much faster than X, indicating that some of these problems can be mitigated through careful X proxy design. Figure 2.6 shows that even though we excluded client processing time for ICA, RDP, GoToMyPC, and Sun Ray, THINC including client processing time is faster than all of them. GoToMyPC takes almost three seconds on average to download web pages. Figure 2.7 shows that GoToMyPC's slow performance is not due to its data requirements as it sends the least amount of data. The measurements suggest that GoToMyPC employs complex compression algorithms to reduce its data requirements at the expense of high server utilization and longer latencies. GoToMyPC's use of an intermediate server most likely also affects its performance, but enables it to provide ubiquitous service even in the presence of NATs and firewalls.

Figure 2.8 shows results using remote PlanetLab nodes and other sites as THINC clients, demonstrating that THINC maintains its fast performance under real network conditions even when client and server are located thousands of miles apart. THINC provides sub-second web page download times for all sites except for when the client is running in Korea, which is almost seven thousand miles away from the server in New York. Figure 2.8 shows that THINC's web page download latencies increased by less than 2.5 times in going from running the client in the local LAN testbed to running the client in Finland while the corresponding network RTTs increased by

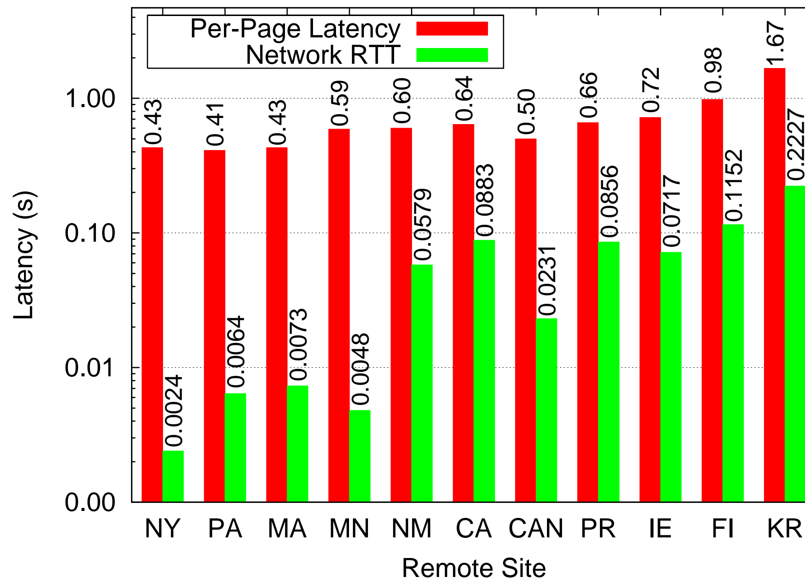


Figure 2.8 – Web Benchmark: THINC Average Page Latency Using Remote Sites. The bars correlate order of magnitude increases in network latency (RTT) to modest per-page latency increases in THINC's performance

more than two orders of magnitude. These measurements show for the first time a thin client that can provide excellent web browsing performance even when clients are located on another continent.

Figure 2.7 shows the average data transferred for each web page and demonstrates that THINC achieves fast performance with only modest data transfer requirements. The local PC is the most bandwidth efficient platform for web browsing, but THINC is better than all other thin clients for LAN Desktop except NX. Surprisingly, GoToMyPC had the smallest data transfer requirements of the thin clients for WAN Desktop despite its low-level pixel-based display approach. While this is an unfair comparison since GoToMyPC only supports 8-bit color, it demonstrates that compression algorithms can be effective at reducing raw pixel data at great computational expense. A number of systems show significant reductions in data size when going from the LAN to the WAN environment. NX has specific user settings for this type of

environment which causes it to use more aggressive data compression techniques. Sun Ray and VNC use adaptive compression schemes which change its encoding settings according to the characteristics of the link. This adaptive mechanism also accounts for the significant decrease in Sun Ray's data requirements, as more complex and cpu-intensive compression schemes are used.

Comparing Sun Ray and THINC provides a measure of the effectiveness of THINC's translation architecture, as both systems use a similar low-level protocol. Although we could not instrument the Sun Ray hardware client to measure client processing time, we can use the network measurements as a basis of comparison between the systems. Both systems perform well, but THINC outperforms Sun Ray by 22% and 16% in the LAN and WAN environments, respectively. Sun Ray incurs higher overhead because it lacks THINC's translation mechanisms, especially offscreen drawing which is used heavily by Mozilla. As a result, it lacks semantic information originally present in the application display commands and must attempt to translate back into its protocol from raw pixel data. Similarly, comparing VNC and THINC provides a measure of the efficiency of THINC's encoding approach versus VNC's pixel data compression approach. THINC is faster than VNC for the LAN Desktop while sending almost half the data. This suggests that THINC's small set of command primitives and translation layer provides significant performance efficiency compared to relying on a single compression strategy for all types of display data. These results show the importance of an effective translation layer, not just a good command set.

Comparing these systems as well as NX on a page-by-page basis provides further insight based on how different web page content contributes to performance differences. Except for THINC, Sun Ray, VNC, and NX were the fastest systems. Compared with these systems, THINC was faster on all web pages except those that primarily consisted of a single large image. For those pages, THINC resorted primarily

to its RAW encoding strategy combined with simple, off-the-shelf compression, given the lack of additional semantic information. In the LAN, Sun Ray's lack of compression and VNC's simple compression strategy both sent more data but provided faster processing of those pages compared to THINC. In the WAN, the more advanced compression used in NX and Sun Ray reduced the data size significantly, allowing them to transfer the pages much faster. This breakdown indicates that THINC's performance on pages with mixed web content (text, logos, tables, etc.) was even better than what is shown in Figure 2.6 when compared with these other systems. These results suggest two important observations. First, not only is THINC's low-level translation approach faster than a pixel-level approach as embodied by VNC, but it is also faster than a high-level encoding approach as embodied by NX, even on non-image content. Second, although optimized compression techniques were not a central focus in the current THINC prototype, the results suggest that better compression algorithms such as used in NX and adapting compression based on network performance as used by VNC and Sun Ray can provide useful performance benefits when displaying large image content.

2.8 Summary

This chapter introduced THINC, a new virtual display architecture for high-performance remote desktop computing. THINC is built around a virtual device drive approach that enables it to leverage continuing advances in window server technology and work seamlessly with unmodified applications, window systems, and operating systems. On top of this architecture, THINC introduces novel translation and delivery optimizations that take advantage of semantic information to efficiently convert high-level application requests to a simple low-level protocol command set, and deliver these

protocol commands to simple and stateless remote clients.

We have measured THINC's web browsing performance in a number of network environments and compared it to existing widely used commercial remote display systems. Our experimental results show that THINC can deliver good interactive performance even when using clients located around the world. THINC provides superior web performance over other systems, with up to 4.5 times faster response time in WAN environments. Our results demonstrate how THINC's unique mapping of application level drawing commands to protocol primitives and its command delivery mechanisms significantly improve the overall performance of a remote display system. Going beyond basic remote display and thin-client computing, this dissertation will show how THINC provides a fundamental building block for a broad range of remote display and desktop computing applications.

Chapter 3

Multimedia

From video conferencing and presentations to movie and music entertainment, multimedia applications play an everyday role in desktop computing. However, many remote display platforms have either limited, format-specific, or no support at all for multimedia applications. Multimedia delivery imposes rather high requirements on the underlying remote display architecture, in particular the delivery of video updates. If the video is completely decoded by applications on the server, there is little the remote display server can do to provide a scalable solution. Real-time re-encoding of the video data is computationally expensive as screen sizes get larger, even with modern high end server CPUs. At the same time, delivering 24 or 30 frames per second of uncompressed color data can rapidly overwhelm the capacity of a typical network. On the other hand, if the video is transmitted without decoding, the client has to contain software to decode all possible formats that users will want access to. This additional software significantly increases the complexity of the client, to the point of becoming a management burden. Further hampering the feasibility of this approach are the lack of well-defined application interfaces for multimedia decoding. Most video players use ad-hoc, unique decoding methods and architectures,

and providing support in this environment would most certainly require prohibitive per-application modifications.

THINC addresses these shortcomings by leveraging and extending its virtual device approach to fulfill the needs of multimedia applications. In essence, THINC provides a virtual "bridge" between the remote client hardware and the local applications, allowing applications to transparently use the hardware capabilities of the client to perform multimedia operations across the network. This is accomplished by extending the virtual display device to provide video playback acceleration. Alongside, THINC introduces a virtual sound device which can capture and forward audio onto the client, and can receive audio data captured by the client and forward it to applications.

This approach has a number of benefits:

- First, it allows THINC to support multimedia content in a manner that is completely application transparent, since they utilize the virtual devices as they would real hardware.

- Second, it provides increased playback performance. THINC extends existing hardware acceleration interfaces to provide a virtual "bridge" between the remote client hardware and the local applications, allowing them to take advantage of the remote hardware as if it were local.

- Finally, it is format agnostic, since the hardware interfaces leveraged by THINC are, by design, low-level, and meant to support as many codecs as possible.

The audio and video drivers work in concert to create a session environment specific to each remote user. Nevertheless, they operate in a loosely-coupled fashion. In particular, multimedia content is demultiplexed by applications before it is delivered

to the virtual devices, and each device may use a separate communication channel with the client. As a result, it is possible for the different streams to become out of sync while en route to the client.

In this manner, we recognize that the largest source of variability in the system is the network between the server and the client; consequently, server-side synchronization is of little use. As a result, the final component of the multimedia architecture is meant to address this major shortcoming. THINC uses a simple yet effective mechanism that adds timing information to each stream. This information allows the client to keep the streams synchronized after they are delivered.

The resulting combination of native video, audio, and synchronization mechanisms allow THINC to provide transparent and high-performance remote multimedia support. The following sections discuss each of these components in detail.

3.1 Video Support

While full video decoding in desktop computers is still confined to the realm of software applications, hardware manufacturers have been slowly adding acceleration capabilities for specific stages of the decoding process. For example, the ability to do hardware color space conversion and scaling (the last stage of the decoding process) is present in almost all of today's commodity video cards. To allow applications to take advantage of these advancements, interfaces have been created in display systems that allow video device drivers to expose their hardware capabilities back to the applications. With its virtual device approach, THINC provides a virtual "bridge" between the remote client hardware and the local applications, and allows applications to transparently use the hardware capabilities of the client to perform video playback across the network.

THINC supports the transmission of video data using widely supported YUV pixel formats. A wide range of YUV pixel formats exist that provide efficient encoding of video content. For example, the preferred pixel format in the MPEG decoding process is YV12, which allows normal true color pixels to be represented with only 12 bits. YUV formats are able to efficiently compress RGB data without loss of quality by taking advantage of the human eye's ability to better distinguish differences in brightness than in color. When using YUV, the client can simply transfer the data to its hardware, which automatically does color space conversion and scaling. Hardware scaling decouples the network transfer requirements of the video from the size at which it is viewed. In other words, playing back a video at full screen resolution does not incur any additional overhead over playing it at its original size, because the client hardware transparently transforms the stream to the desired view size.

THINC's video architecture is built around the notion of video stream objects. Each stream object represents a video being displayed. All streams share a common set of characteristics that allow THINC to manipulate them such as timing information, their position on the screen, and the geometry of the video.

Table 3.1 provides an overview of the commands used to manipulate video streams. More explicit details may be found in Appendix A. When an application attempts to display a video, the THINC server sends an `INIT` message to the client that sets up the video stream. The `INIT` message assigns a unique ID to the stream that other video commands will use to identify and modify the stream. The video initialization process is done asynchronously, guaranteeing that video playback starts as soon as possible on the client. If the client is unable to successfully initialize video playback, it will asynchronously inform the server of the failure, and ignore any outstanding updates already sent by the server. Video playback is accomplished using the `NEXT` command. `NEXT` encapsulates the data needed to display the next frame in the video stream,

Command	**Description**
INIT	Initializes a video stream
END	Tears down a video stream
NEXT	Display the next video frame
MOVE	Change the location of the video display
SRCSIZE	Change the source size of the video stream
DSTSIZE	Change the destination size of the video stream

Table 3.1 – THINC Video Commands. See Appendix A for a complete description

and is sent in response to requests from the application. Because applications have complete control over video playback, THINC does not need to separately implement playback control commands like pause, rewind or fast forward.

The `MOVE`, `SRCSIZE`, and `DSTSIZE` commands are used to change the characteristics of the stream after playback has started. `MOVE` changes the location on the screen where the video is displayed, typically in response to movement of the video player's window. `DSTSIZE` changes the display geometry of the stream, such that videos can be displayed at resolutions different from the actual encoded stream, e.g. displaying a normal-sized video at full screen. `SRCSIZE` informs the client that the dimensions of the encoded stream have changed. The server uses this command to modify the video data on the fly to reduce the resource usage of the video. This is particularly useful in situations where the client's viewport size is smaller than the server's framebuffer size. In this situation, the server will automatically resample the video data in proportion to the client's display resolution, thus reducing bandwidth requirements. As our experimental results demonstrate, this technique allows THINC to provide video playback to mobile devices, such as PDAs, over wireless networks.

3.2 Audio Support

THINC enables transparent audio capture and playback support using a simple virtual audio device driver that resides on the server operating system. The audio driver layer was chosen as it represents a common interception point across all applications, regardless of the specifics on how they perform audio manipulation. The seamless integration of audio support in THINC differentiates it from other approaches, such as using networked audio servers, in that no application modifications or special wrapper scripts need to be invoked to enable capture or playback.

As the operating system device driver layer can be quite system specific and not amenable to complex operations, we paired the virtual device driver with a user level audio daemon. The user level daemon acts as the communication intermediary between the driver and the remote client, and offloads most of the functionality required for audio operations from the driver. This separation of roles provides a number of benefits. First, it allows for system-independent functionality to be encapsulated at the user level, maximizing the portability of our approach. For example, the daemon can provide a secure communication channel, recoding of the audio data on the fly, and many other operations in a manner independent of the particular details of how audio data is intercepted from applications. Similarly, by moving most complex functionality to the user level, we simplify the device driver implementation, which is desirable for any component working within the operating system kernel.

One important consideration regarding the division of responsibilities among THINC's audio components is the performance of the communication between the driver and the daemon. Specifically, using an approach where the daemon blindly copies audio data from kernel space to user space, processes it, then sends it back to kernel space again to be transferred over the network would result in added overhead and introduce

Command	**Description**
OPEN	Initializes a new audio stream
CLOSE	Closes audio device
DATA	Encapsulates the data for the next audio frame
VOLUME	Controls playback volume

Table 3.2 – THINC Audio Commands. `Open, Close and Volume` are sent exclusively from the server to the client. `Data` is sent from the server for playback, and by the client to transfer captured audio data

unnecessary latency. Since the timing of audio playback is important for synchronization purposes, to provide optimal performance while maintaining the driver-daemon separation, we implemented two shared memory regions between the driver and the daemon in which audio data is stored, one for playback, and one for capture. The driver gives access to this region to the daemon by using standard operating system interfaces (such as `mmap()`), allowing it to manipulate the audio data without having to create a local copy.

As audio-related information is written to the driver from a multimedia application, the driver creates commands that represent the operations that the application is attempting to execute. The user level daemon receives the commands from the driver, and forwards them to the client. The commands used are shown in Table 3.2. They are designed to be simple and universally supported by any client audio hardware.

As show in Figure 3.1, when an application begins audio playback, the driver extracts the characteristics of the audio data and informs the daemon about the new audio stream. The daemon in turn encapsulates the audio stream information in an `OPEN` command, and sends it to the client. As in the handling of video, this initialization step is done asynchronously, requiring no round-trip delays before playback can start. As each frame of audio data is sent to the driver for playback, a `DATA`

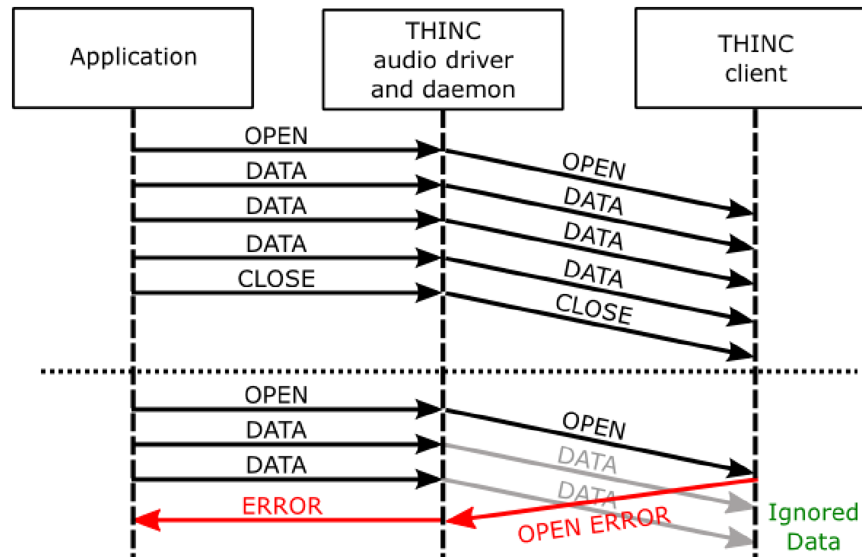


Figure 3.1 – Audio Playback. The top part shows the normal scenario, where the application opens the audio device, plays some data, and then closes it. Notice how the application does not have to wait for the client to open the device before it can start playback. The bottom shows the case where the client cannot open its audio device. It asynchronously informs the driver of the failure, which in turns passes the error to the application. Any in-flight audio data is simply discarded by the client.

command is generated containing the amount of audio data to playback. If the client is unable to do audio playback, it asynchronously sends back an `ERROR` message to the daemon, which passes it on to the driver and the application. In this case, any `DATA` commands already sent are simply discarded by the client upon receipt.

Similarly, as shown in Figure 3.2, when an application requests audio capture, the driver extracts the characteristics of the audio data, and informs the daemon about the new audio stream. The daemon in turn encapsulates the audio stream information in an `OPEN` command, and sends it to the client. In contrast to the playback case, audio capture is not driven by the application. In other words, THINC will not wait for the application to request audio data before the client starts capturing audio. Instead, the client is expected to start transferring audio data upon receipt of the `OPEN` command. The captured data is transferred using the `DATA` command. Once

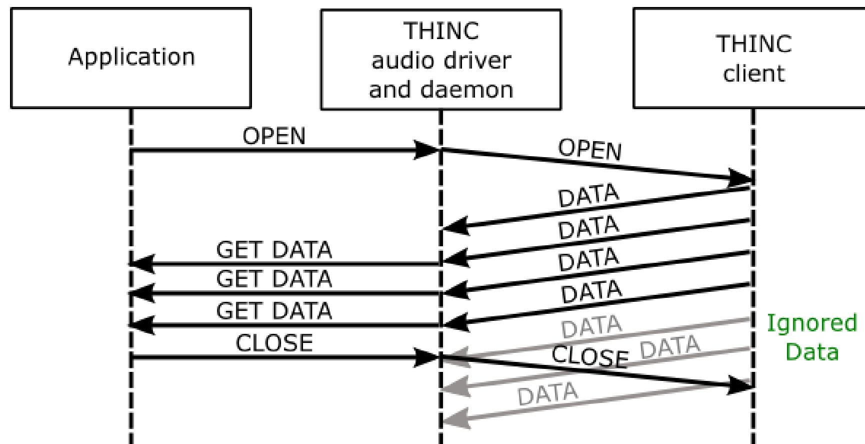


Figure 3.2 – Audio Capture. As soon as the client receives the `OPEN` command, it starts capturing data and transferring it back to the server. In this manner, the time the application has to wait for audio data to become available is minimized.

the daemon receives the data, it copies it to the shared capture buffer, and informs the driver of the new data. Later on, when the application attempts to read audio data, the driver can simply return the data from the buffer. This approach helps to minimize the latency the application perceives from capturing audio data.

However, care must be taken to not capture too much data prematurely, as this would result in the application reading data that was captured too long ago. To minimize this problem, THINC uses a small circular buffer inside the driver where captured data is kept: as new samples are received from the client, old ones are simply overwritten. In this manner, THINC can guarantee a small upper bound on the latency perceived by the application. Ideally, the size of this buffer should be a function of application behavior and network characteristics, in particular latency. For example, in high latency networks, a larger buffer will be needed to avoid the application receiving an underflow error because audio data cannot be delivered fast enough. A low latency network would require exactly the opposite kind of behavior. Being able to dynamically tune this buffer to the requirements of the underlying network conditions and application behavior is a subject of future work.

To maintain portability across a wide range of client audio hardware, THINC uses a lowest common denominator approach, tailored to support everything from high-end audio cards with hardware mixing and multiple channels, to simple, low-powered USB add-on cards supporting only one channel at a time. THINC exposes a single volume control, which can be manipulated using the `VOLUME` command. In addition, it supports only one audio stream in each direction, leveraging traditional application interfaces [5, 80] and similar mechanisms [114] to provide mixing, demultiplexing, and more specialized audio manipulation.

3.3 Media Synchronization

Proper synchronized playback of multimedia streams is an essential component to the desktop user experience. Since synchronization is often taken for granted, users may become upset or annoyed if their media application does not provide proper synchronization. Studies of human perception of inter-stream synchronization show that the tolerance for unsynchronized playback can be exceeded if the playout of audio and video streams differ by as little as ±80 milliseconds [138]. For many remote display systems, the obstacles to providing synchronized playback are larger because of bandwidth constraints and the inability to differentiate between regular display data and actual video data. Because THINC is designed to distinguish between these data types and provide real-time multimedia playback, it can apply a synchronization scheme with minimal additional architectural complexity. Moreover, throughout this process, THINC synchronizes audio and video in a way that is completely transparent to applications.

Because there are several meanings of the term "synchronization" [138], we briefly present some definitions to clarify our terminology. The term *multimedia* refers to

the use of multiple data types, or *media*, such as *continuous* media (audio and video), or *discrete* media (text and graphics). Each of these data types can be individually described as a *media unit*, or *MU*. Continuous media data, or *streams*, are typically integrated, stored, and presented in a way such that a certain relationship must be defined between them in order to preserve their temporal characteristics. *Multimedia synchronization* can be defined as the process of maintaining this temporal order and relationship between integrated media units. Synchronization can occur within a single media stream *(intra-stream)*, between multiple media streams *(inter-stream)*, or between continuous and discrete media *(inter-object)* [167]. For this dissertation, we focus on audio and video support in THINC. Thus, we use the term *multimedia* to refer to audio and video MUs, and *synchronization* to mean *lip synchronization*, or the synchronization of audio and video MUs.

THINC's synchronization mechanism is based on the notion that *time* must be treated as a first-class characteristic of all content delivered over the network. That is, THINC prioritizes playback based on the timing information provided by all media streams, and synchronization relies solely on this timing information which is provided by the low-level driver components. At a high-level, THINC provides multimedia synchronization capabilities during multimedia playback by timestamping audio and video frames as soon as they are received by the corresponding device drivers, and then comparing the timestamps at the client side. This end-to-end approach to synchronization is key to how THINC ensures proper temporal ordering and playout of MUs.

THINC maintains the temporal relationships between MUs based on user interaction with applications during a THINC session. As each media event is generated by an application, the MU associated with the event is given a timestamp before being redirected to the client. In other words, if a video playback application exe-

cuted on the server-side does not have an adequate synchronization mechanism, then THINC does not make any attempt to modify the output of the application. Instead, THINC preserves the timing of media events generated by applications *as they occur*. THINC is able to accomplish this with its virtual device driver design, where it is able to transparently timestamp MUs at the precise moment an application wishes to present them to the user. The timing relationships are maintained on the client side at playout time through a corrective synchronization algorithm.

Thus, timestamping is the most sensitive aspect of the synchronization process. THINC ensures that little latency is introduced since it intercepts data directly after the moment the application sends data to the virtual audio and display device drivers. For multimedia playback, each chunk of video and audio data is associated with a timestamp using microsecond granularity. Once the MUs are received from the applications and timestamped, they are sent immediately to the client. Note that the periodicity of timestamps for each MU is determined by the synchronization mechanisms used by the media playback application. In this manner, THINC is able to represent both the playout time of each MU within each stream and the inter-stream relationship of the MUs using this timestamping mechanism.

Given this timestamping information, the client applies a simple algorithm to ensure synchronized playout. We define the algorithm as follows. Let S_a and S_v represent the current audio and video MU timestamps being processed, respectively, and let Δ_{av} represent the inter-stream difference between S_a and S_v, computed as $S_a - S_v$. Then:

- If $\Delta_{av} > 0$, then $S_a > S_v$, and S_a was generated earlier than S_v. If Δ_{av} exceeds some threshold ϵ_{av}, then the client blocks the processing of audio MUs and reads additional video MUs from the network until $\Delta_{av} < \epsilon_{av}$.

- Likewise, if $-\Delta_{av} > 0$, then $S_v > S_a$, and S_v was generated earlier than S_a. If $-\Delta_{av}$ exceeds ϵ_{av}, then the client blocks the processing of video MUs and reads additional audio MUs from the network until $-\Delta_{av} < \epsilon_{av}$.

Observe that ϵ_{av} determines how aggressively inter-stream synchronization is applied. If this threshold is too loose, meaning that ϵ_{av} is too large, then the possibility of skew or drift is introduced. If this threshold is too strict, the synchronization algorithm may overcorrect even though the timestamps of the MUs are properly representing the inter-stream intervals. THINC uses an inter-stream threshold of 80 milliseconds, which we previously mentioned to be a known human tolerance threshold for inter-stream synchronization.

In THINC's end-to-end synchronization approach, one final design issue to consider is the mechanism used to transfer audio and video data and associated timestamps over the network. In the case of real-time multimedia playback, the use of RTP [126] over UDP is the most widely accepted solution. In this case, delayed, out of order, or lost frames are discarded automatically in order to maintain the application timing constraints. However, in the case of stored media playback such as DVD playback, it may be more desirable for the user to receive all data, even with occasional skips and delays. In this case, TCP may be the method of choice for data transport. To minimize the effects of network variations on TCP and the overall playback, it is customary to implement a *jitter buffer* on the client. The buffer "cushions" any intermittent network changes and guarantees smooth playback.

As the needs of desktop users may vary widely, for example, they could just as well be participating in a video conference, or playing back a DVD, THINC provides support for both UDP and TCP transports, as well as a jitter buffer, leaving it to the user to choose an appropriate method according to what she or he requires.

3.4 Implementation Details

The video playback implementation is part of the remote display implementation described in Section 2.6. In this section we will focus on the implementation of the virtual audio driver.

The audio driver is implemented as a loadable kernel module using the Advanced Linux Sound Architecture (ALSA) [6] driver framework on the Linux platform. ALSA was selected over OSS [102] since OSS is considered a deprecated driver framework and ALSA is its designated replacement. In addition, ALSA provides backward-compatibility with OSS, which allows proper ALSA audio drivers to work with applications written specifically for OSS.

ALSA Applications use a well-defined user-level audio library which communicates with the driver. For commands such as `OPEN`, `CLOSE`, and `VOLUME` which occur with less frequency or require quick response, the driver keeps a separate queue from which the daemon reads using `read()` system calls. The driver awakens the daemon each time the audio driver receives information from an application, and the daemon redirects playback data to the client based on commands generated from this information. These command structures are processed by the daemon in two ways. During normal playback, the driver must awaken the daemon to send audio data to the client via `DATA` commands. Since this represents the most-often recurring command type, the driver partitions the DMA buffer to contain raw audio data as well as `DATA` commands. During audio capture, the daemon gets awaken when it detects activity on its network connection with the client. At this moment it reads the data from the network, copies it to the shared buffer, and using the `write()` system call, informs the driver of how much data it just received from the client.

To communicate with the client, the daemon uses either a pure TCP connection,

or a mixed TCP/UDP strategy. In the first case, all control commands and audio data are transferred using the reliable connection provided by TCP. In the second case, only control commands are transferred using TCP. All data is sent through the unreliable channel. Users are able to select which approach to use according to their needs. For example, for real-time communication, they will most certainly choose UDP.

3.5 Experimental Results

We focused on three different aspects of multimedia performance. First, we measured audio/video playback on THINC and compared it to existing thin-client systems. Second, we measured THINC's ability to maintain synchronized audio and video content. Finally, we measured the performance of THINC's audio capture mechanism, and its impact on VoIP applications.

3.5.1 Experimental Setup and Benchmarks

To measure audio/video playback performance we used the same experimental setup as used for the 2D remote display measurements, as described in Section 2.7. We played a 34.75 s MPEG-1 audio/video clip, with the video being of original size 352x240 pixels and displayed at full-screen resolution. We measured combined audio/video playback performance except for GoToMyPC and VNC for which we only report video playback results since they do not support audio. Although X has no native audio support, various programs have been developed to provide remote audio alongside it. For our experiments, we used aRts 1.3.2, a sound server commonly used for this task. The audio/video (A/V) player used was MPlayer 1.0pre6 for the Unix-based platforms, and Windows Media Player 9 for the Windows-based platforms.

Since many of the thin-client systems are closed and proprietary, we measured their playback performance in a noninvasive manner by capturing network traffic with a packet monitor and using a variant of slow-motion benchmarking tailored for multimedia applications [70, 93]. The benchmark provides a measure of playback quality that accounts for both playback delays and frame drops. For example, 100% quality means that all video frames and audio samples were played at real-time speed. On the other hand, 50% quality could mean that half the data was dropped, or that the clip took twice as long to play even though all of the data was played. We used a combined measure of audio and video playback quality since many of the closed platforms tested transmit both audio and video over the same connection, making it difficult to disambiguate packet captures to determine which data corresponds to each media stream.

To measure synchronization quality, we ran THINC using the same A/V playback setup and collected timestamp logs on the client, comparing the timestamps issued by the server for the audio and video streams over the time that the client received them. We used two versions of the client, one with synchronization enabled and the other with synchronization disabled. To test THINC's format independence as well as its quality of synchronization over time, we used two additional video clips, one 30.2 s QuickTime clip at 480x360 resolution and the other a 148 s MPEG-1 clip at 480x260 resolution, both playing at full-screen and played back using WAN settings. Due to the lack of access to source code from closed proprietary systems and the general difficulty in measuring synchronization quality, we were unable to take similar measurements of the other thin-client systems tested.

Finally, we evaluated audio capture and playback performance by measuring the overhead of using THINC on the mouth-to-ear latency [57] of three Voice-over-IP systems: Skype [132], version 1.4.0.99, WengoPhone [159], version 2.1.1, and Lin-

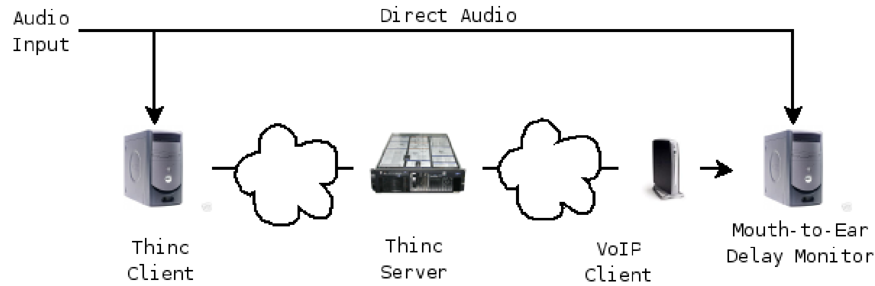


Figure 3.3 – Experimental Testbed for Audio Capture/Playback Benchmark

phone [74], version 1.7.1. As shown in Figure 3.3, our testbed consisted of four computers connected on a private network: one THINC client, one THINC server, one native VoIP client, and a mouth-to-ear delay monitor. The THINC client was a Dell Latitude D420 laptop with a 1.2 GHz Intel Core Duo CPU and 1.5 GB of RAM. The VoIP client was an IBM T30 laptop with a 1.8 GHz Pentium 4 CPU and 1 GB of RAM. The THINC server was a Dell Dimension 5150C desktop with a 3.20 GHz Intel Pentium D CPU and 4 GB of RAM. The mouth-to-ear monitor was a Dell Latitude C400 laptop with a 1.2 GHz Pentium III-M CPU and 512 MB of RAM. All of the computers ran the Debian GNU/Linux distribution, version 3.0. The native VoIP client ran the VoIP systems natively, using the computer's sound card for capture and playback. We ran a second VoIP program instance inside a virtualized desktop on the THINC server which used our virtual audio driver and the THINC client audio hardware for capture and playback. We used UDP to transport all audio data between the THINC server and client.

The benchmark consisted of playing a one minute long sound clip into a connected phone call, then measuring the delay in latency from the time when the sound clip was generated until the time it was heard on the other end. The sound clip consisted of single "beeps" separated by 5 seconds of silence. To measure latency effectively, we captured the audio feeding directly into the microphone input of one of the comput-

ers, and combined it with the audio output of the second computer using an audio mixer. The output of the mixer was then fed into the monitor computer, and the delay measured using the Audacity audio editor for Linux, version 1.3.3. To deliver acceptable VoIP quality, this delay should fall below 400 ms, which user experience studies have shown to be the maximum acceptable mouth-to-ear latency [147].

Capture and playback performance were measured separately by changing the configuration of the testbed. Figure 3.3 shows the configuration used for the capture case. For playback, we switched the position of the VoIP client and the THINC client in the testbed. For each case, we also measured a baseline latency of the native performance of each VoIP system. In this case, we removed the THINC client computer from the testbed, and let the THINC server use its real audio hardware. Finally, we measured performance in both wired and wireless scenarios. For the wired scenario, all computers were connected using a private switched FastEthernet network. For the wireless scenario, we used a 802.11b wireless connection between the THINC client and server, while the rest of the computers were connected using the wired network. For the baseline case, the wireless connection was located between the VoIP client and the THINC server.

3.5.2 Results

Figures 3.4 to 3.6 show A/V playback performance results. Results for VNC and GoToMyPC are for video playback without audio since those platforms do not provide audio support. We also ran the same benchmark on all platforms with video only and no audio. The results were similar to the A/V playback results. For platforms that supported audio, we also ran the same benchmark with audio only and no video. Most of the platforms with audio support provided perfect audio playback quality in

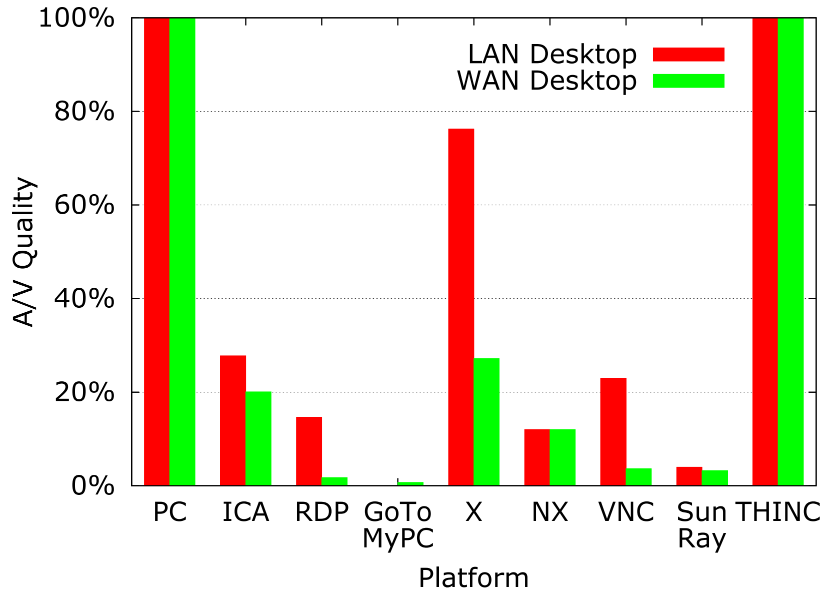


Figure 3.4 – A/V Benchmark: A/V Quality. GoToMyPC and VNC are video only

the absence of video.

Figure 3.4 shows that THINC is the only thin client that provides 100% A/V quality in all network environments and is the only system that provides 100% A/V quality in all configurations. THINC's A/V quality is up to 8 times better than the other systems for LAN Desktop and up to 140 times better for WAN Desktop. Other than THINC, only the local PC provides 100% A/V quality in any of the configurations tested. From a qualitative standpoint, THINC A/V playback was consistently smooth and synchronized and indistinguishable from A/V playback on the local PC. On the other hand, A/V playback was noticeably choppy and jittery for all other thin clients. In particular, playback on RDP and ICA was marked by lower audio fidelity due to compression and frequent drops.

Figure 3.4 shows quantitatively that all other thin clients deliver very poor A/V quality. NX has the worst quality for LAN at only 12%, and GoToMyPC has the worst quality for WAN at less than 2%. These systems suffer from their inability to

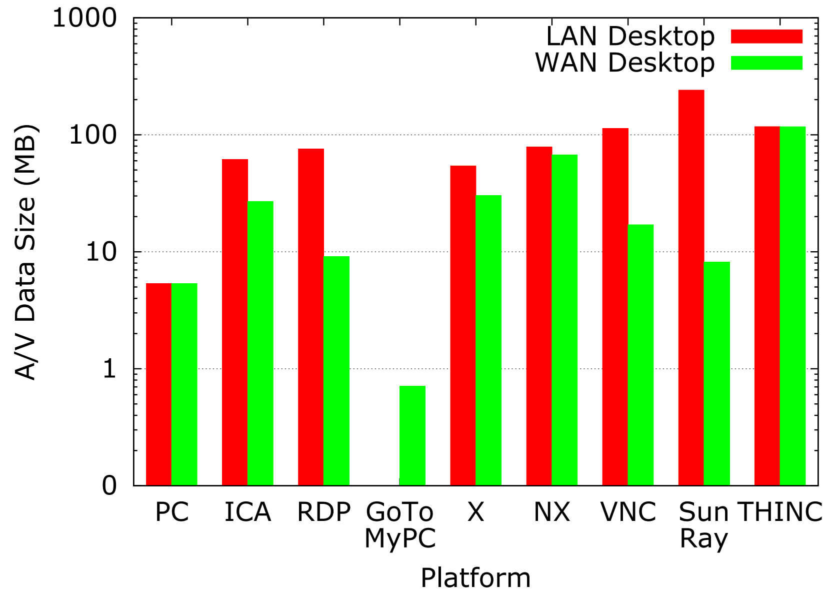


Figure 3.5 – A/V Benchmark: Total Data Transferred. GoToMyPC and VNC are video only

distinguish video from normal display updates, and their attempts to apply ineffective and expensive compression algorithms on the video data. These algorithms are unable to keep up with the stream of updates generated, resulting in dropped frames or extremely long playback times. VNC has poor video performance for these same reasons, and drops quality by half for the WAN Desktop because of its client-pull model. The VNC client needs to request display updates for the server to send them. This is problematic in higher latency WAN environments in which video frames are generated faster than the rate at which the client can send requests to the server. In contrast, THINC's server push model and its native audio/video support provide substantial performance benefits over the other systems.

The effects of ICA's support for native video playback are not reflected in Figures 3.4 to 3.6. Its playback mechanism only supports a limited number of formats, and the widely-used MPEG1 format used for the A/V benchmark is not one of them. We conducted additional experiments with the video clip transcoded to DivX, a supported

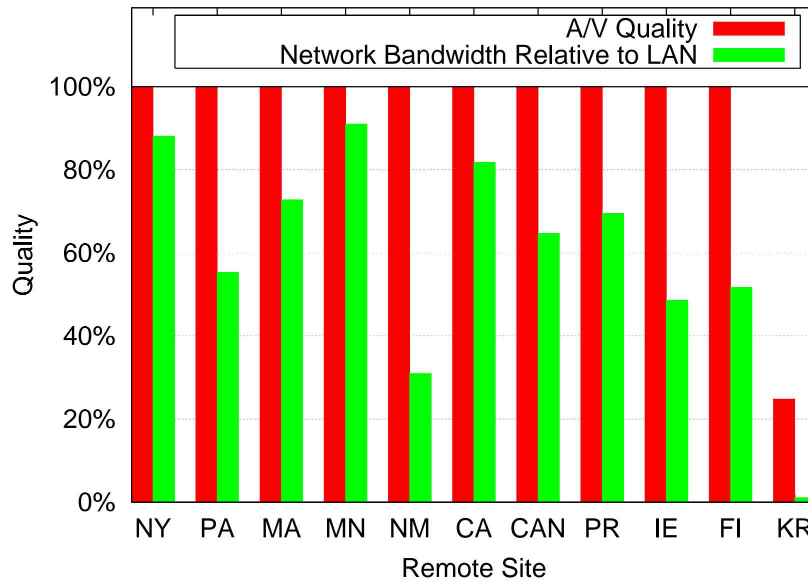


Figure 3.6 – A/V Benchmark: THINC A/V Quality Using Remote Sites

format, and surprisingly found the results to be only slightly better. ICA relies on the Windows Media Player installed on the client to do the video playback, and in turn the player had hardware requirements for this video format beyond what the client could support. The client was unable to keep up with the desired playback rate, resulting in poor video quality.

Figure 3.5 shows the total data transferred during A/V playback for each system. The local PC is the most bandwidth efficient platform for A/V playback, sending less than 6 MB of data, which corresponds to about 1.2 Mbps of bandwidth. THINC's 100% A/V quality requires 117 MB of data for the LAN Desktop and WAN Desktop, which corresponds to bandwidth usage of roughly 24 Mbps. Several other thin clients send less data than THINC, but they do so because they are dropping video data, resulting in degraded A/V quality. For example, GoToMyPC sends the least amount of data but also has the worst A/V quality.

Figure 3.6 shows results using remote PlanetLab nodes and other sites as THINC

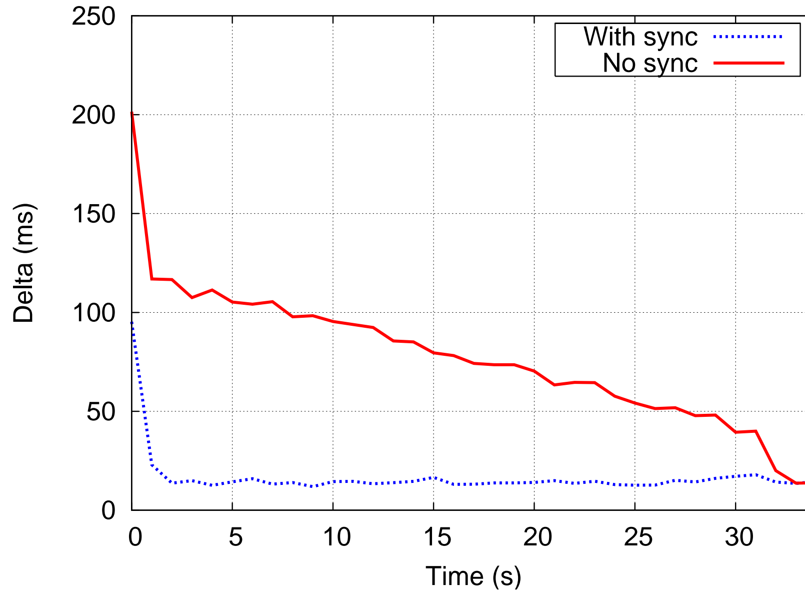


Figure 3.7 – Timestamp Deltas: MPEG-1 352x240

clients, demonstrating that THINC maintains its superior A/V playback performance under real network conditions even when client and server are located thousands of miles apart. THINC provides perfect A/V quality for all remote sites except for Korea. Figure 3.6 also shows the relative bandwidth available from each remote site to the local THINC server compared to the bandwidth available in our local LAN testbed. These measurements were obtained using Iperf. The bandwidth measurements show that THINC does not perform well for Korea due to insufficient bandwidth. The lack of bandwidth in this case was not due to network link itself, but due to the TCP window size configuration of the Korea PlanetLab site, which we were not allowed to change. For other distant non-PlanetLab remote sites such as Puerto Rico, Ireland, and Finland in which a sufficiently-sized TCP window was allowed, Figure 3.6 shows that THINC provides 100% A/V quality.

Figures 3.7 to 3.12 show the effects of THINC's synchronization mechanism versus THINC with no synchronization over the time span of each test A/V clip. We show

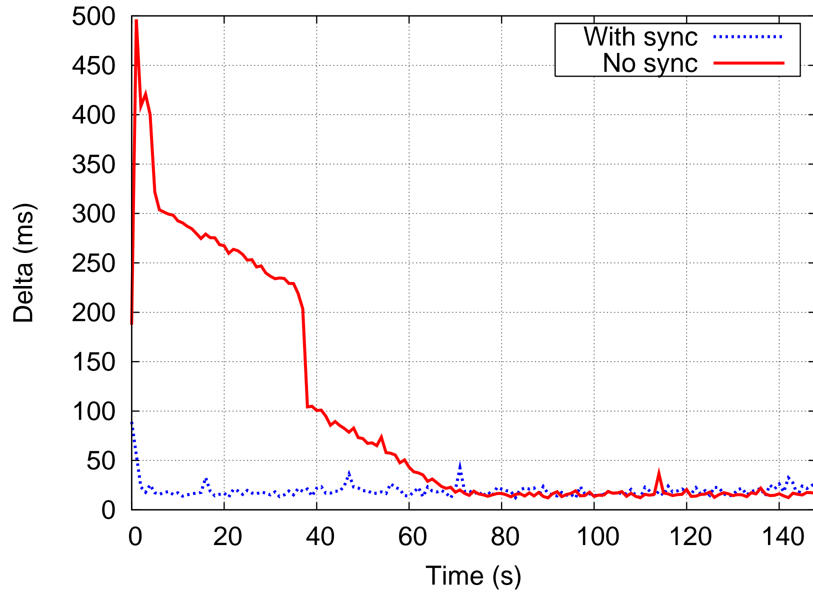


Figure 3.8 – Timestamp Deltas: MPEG-1 480x260

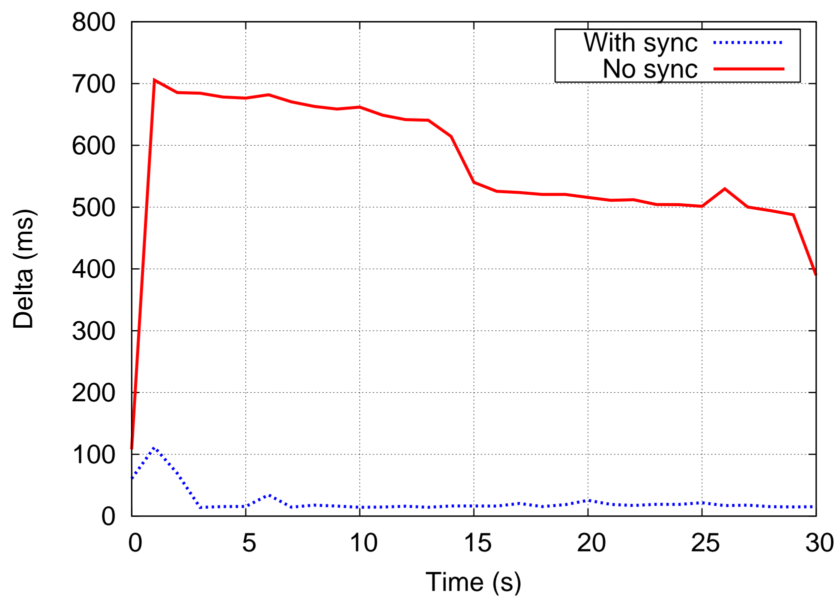


Figure 3.9 – Timestamp Deltas: QuickTime 480x360

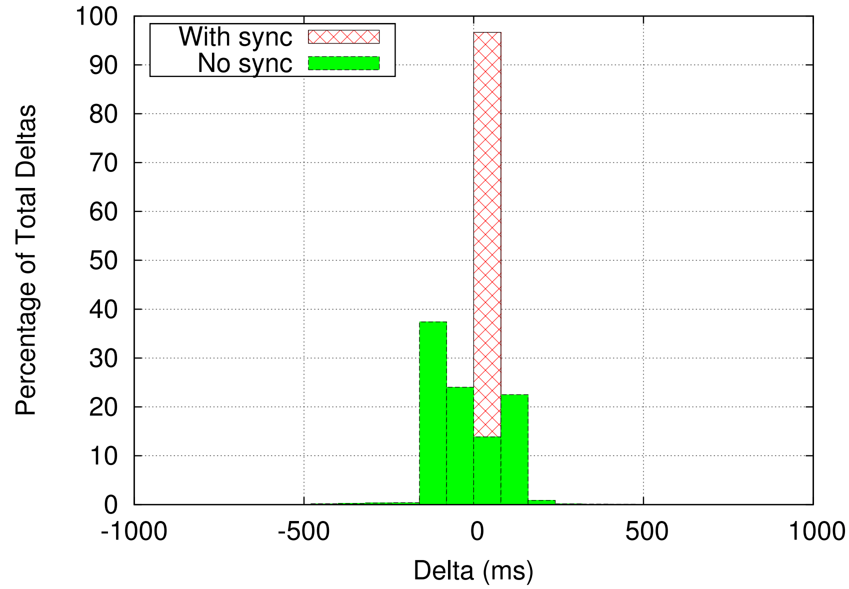


Figure 3.10 – Distribution of Timestamp Deltas: MPEG-1 352x240

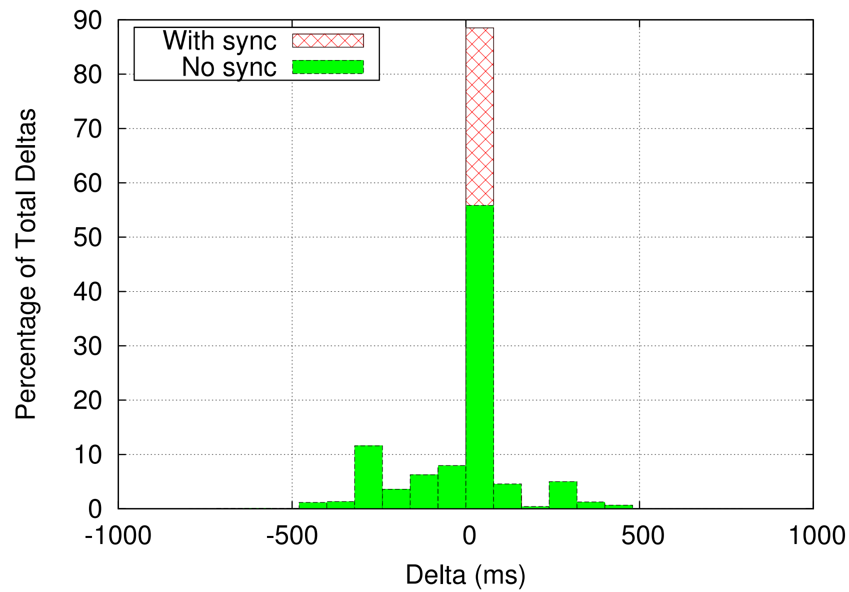


Figure 3.11 – Distribution of Timestamp Deltas : MPEG-1 480x260

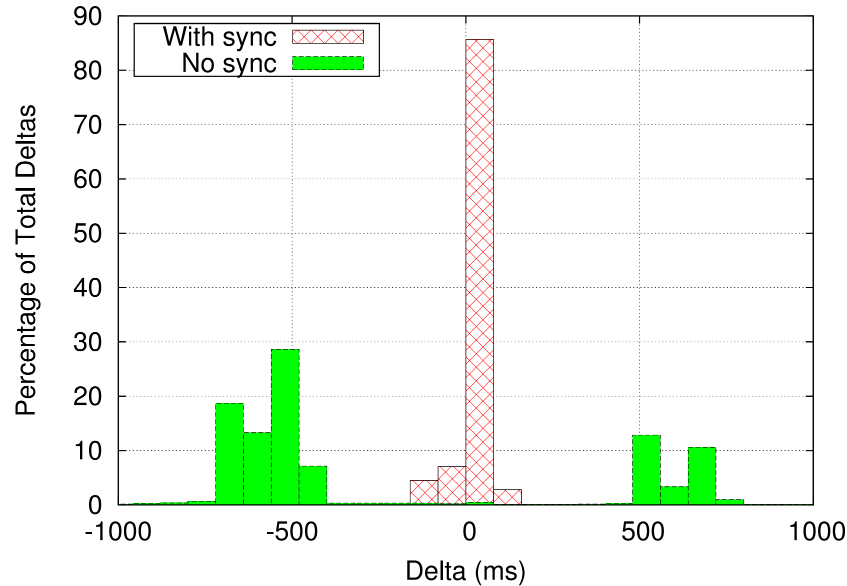


Figure 3.12 – Distribution of Timestamp Deltas: QuickTime 480x360

only the WAN Desktop scenario, as the large network latency provides the most stressful environment for our tests. For Figures 3.7 to 3.9, the lines represent the time difference, or delta, between the audio and video server timestamps at the moment the client received the MU. Since there are thousands of timestamps for each clip, for readability purposes we plotted only the average delta for each one second interval and took its absolute value. THINC's client-side synchronization scheme is able to correct the playout of audio and video quickly and is able to maintain synchronization throughout an extended period. Figures 3.7 and 3.8 show that the non-synchronizing version of the client eventually provides synchronization in smaller sized videos, as the destabilizing effects of network latency are eventually overcome. However, Figure 3.9 shows that as the video frame size increases, synchronization never occurs in the non-synchronized client. This demonstrates that THINC's synchronization mechanisms can handle adverse network environments, and are particularly effective with A/V clips with large frame sizes.

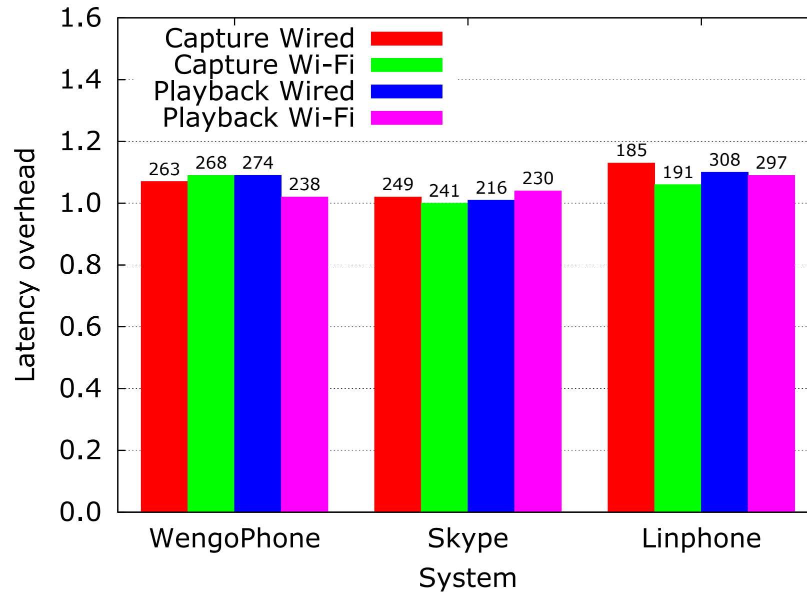


Figure 3.13 – Mouth-to-ear latency overhead for VoIP applications. The playback column shows the overhead when VoIP audio data is played back over THINC. The capture column shows the overhead when VoIP audio data is captured through THINC. Labels on the bars show THINC's total measured latency in milliseconds

Figures 3.10 to 3.12 show the distribution of deltas throughout the playback of each clip. The deltas are grouped using 80 ms bins, which represents the threshold at which synchronization is applied in THINC, as previously discussed. Without applying synchronization, THINC can display synchronized audio and video at most 55% of the time, and, as Figure 3.12 shows, at worst does not provide synchronization at all. With synchronization, we can see in Figures 3.10 through 3.12 that THINC can provide synchronization 97% of the time throughout the A/V clip playout. The distribution of deltas with non-synchronized THINC is also much wider, obscuring the delta bins with synchronized THINC, and almost never converging around the 0 ms range in Figure 3.12.

Finally, Figure 3.13 shows the results of our audio capture and playback benchmark evaluation. Performance is shown normalized to the latency of the native VoIP systems. Labels on top of the bars show THINC's latency in milliseconds. The re-

sults show that THINC can provide good performance for both capture and playback in wireless and wired networks, with low overhead on the mouth-to-ear latency of the system for all cases. The maximum overhead was 13% for the Linphone capture case on the wired network. For all other cases, the overhead was below 10%. The results also show that for all measured scenarios, the mouth-to-ear latency was well below the 400 ms acceptable limit, with the maximum latency being 308 ms for the Linphone playback case on the wired network. In some cases, Linphone being the most pronounced one, capture latencies are lower than playback latencies due to the difference in computational power between the THINC server and the VoIP client. In the playback case, the lower power VoIP client has to do most of the work by compressing the audio data before sending it to the computer running THINC server.

THINC was able to support all three VoIP applications despite the fact that they use different data formats. Skype sends data with 16 bits per sample, 1 audio channel, and 48000 frame rate, for a total bandwidth of 94KB/s. Linphone and WengoPhone use 16 bits per channel, 1 channel, and 16000 frame rate, for a total bandwidth of 32KB/s. This difference in formats is easily supported by THINC since it has no format dependencies of any kind. It simply tunnels the data between the audio hardware of the client and the applications on the server. As long as the client hardware supports the desired format, applications will work flawlessly.

In summary, our results show that THINC can effectively deal with different media characteristics and provide synchronization, full quality playback, and low overhead capture/playback independent of the media format.

3.6 Summary

Efficient multimedia support has always been a major shortcoming of remote display systems. In this chapter we have described THINC's approach to address these deficiencies. In particular, THINC is able to provide native, application transparent, and format independent support for video playback, bidirectional audio, and synchronized audio/video playback. THINC leverages its virtual device architecture to create a virtual "bridge" between the remote client hardware and the local applications. In this manner, applications are able to transparently use the hardware capabilities of the client to perform multimedia operations across the network.

This is accomplished by extending THINC's virtual display device to provide video playback acceleration. Alongside, THINC introduces a virtual sound device which can capture and forward audio onto the client, and can receive audio data captured by the client and forward it to applications. Finally, THINC provides intra-stream synchronization in a manner which is both media format independent and transparent to applications. By treating time as a first-class component, THINC is able to maintain timing characteristics for all media streams as they are delivered over the network.

We have measured THINC's multimedia performance in a number of network environments and compared it to widely used remote display systems. Our experimental results show that THINC is able to deliver full-screen multimedia playback at full frame rate in both LAN and WAN environments, outperforming most popular systems under reasonable network conditions. In addition, THINC is able to apply a simple mechanism that provides effective synchronization performance in WAN settings, independent of media characteristics and format. Finally, our results show that THINC can provide low-latency bidirectional audio support for VoIP applications.

Chapter 4

Mobile Devices

The increasing ubiquity of wireless networks and decreasing cost of hardware is fueling a proliferation of mobile wireless handheld devices, including wireless Personal Digital Assistants (PDA) and integrated PDA/cell phone devices. These devices are enabling new forms of mobile computing and communication. Service providers are leveraging these devices to deliver general application functionality similar to what is found in traditional desktop computing environments, including web browsing, email, video, music, financial planning, and personal information management.

These devices are typically used by running applications locally on them. Although native applications exist for PDAs, many of them deliver subpar performance and have a much smaller feature set and more limited functionality than their desktop counterparts [70]. For example, PDA web browsers are often not able to display web content from web sites that leverage more advanced web technologies to deliver a richer web experience. This fundamental problem arises for two reasons. First, since PDAs have a completely different hardware and software environment from traditional desktop computers, applications need to be rewritten and customized for PDAs if at all possible, duplicating development costs. Because the desktop appli-

cation market is larger and more mature, most development efforts generally end up being spent on desktop applications, resulting in greater functionality and performance than their PDA counterparts. Second, PDAs have a more resource constrained environment than traditional desktop computers to provide a smaller form factor and longer battery life. Desktop applications are large, complex applications that are unable to run on a PDA. Instead, developers are forced to significantly strip down these applications to provide a usable PDA application, thereby crippling PDA application functionality.

To address these problems, we propose an alternative solution for delivering application services on mobile handheld devices by using thin-client computing. In this model, handheld devices communicate over the network with a server using a remote display protocol. This model provides several important benefits. First, standard desktop applications can be used in PDAs without rewriting or adapting them to execute on a PDA, reducing development costs and leveraging existing software investments. Second, complex applications can be executed on powerful servers instead of running stripped down versions on more resource constrained PDAs, providing greater functionality and better performance [70]. Third, applications can take advantage of servers with faster networks and better connectivity, further boosting application performance. Fourth, PDAs can be even simpler devices since they do not need to perform complex application logic, potentially reducing energy consumption and extending battery life. Finally, PDA thin clients can be essentially stateless appliances that do not need to be backed up or restored, require almost no maintenance or upgrades, and do not store any sensitive data that can be lost or stolen. This model provides a viable avenue for medical organizations to comply with HIPAA regulations [50] while embracing mobile handhelds in their day to day operations.

Despite these potential advantages, thin clients have been unable to provide the

full range of these benefits in delivering applications to mobile handheld devices. Existing thin clients were not designed for PDAs and do not account for important usability issues in the context of small form factor devices, resulting in difficulty in navigating displayed content. Furthermore, existing thin clients are ineffective at providing seamless mobility across the heterogeneous mix of device display sizes and resolutions. While existing thin clients can already provide faster performance than native PDA web browsers in delivering HTML web content[70], they do not effectively support more display-intensive applications such as multimedia video, which is increasingly an integral part of the desktop experience.

To harness the full potential of thin-client computing in providing mobile wireless applications on PDAs, we have developed pTHINC (PDA THin-client InterNet Computing) [62, 63]. pTHINC extends THINC's remote display architecture to provide a thin-client architecture for mobile handheld devices. Using THINC's display virtualization, pTHINC resizes the display on the server to efficiently deliver high-fidelity screen updates to a broad range of clients, screen sizes, and screen orientations, including both portrait and landscape viewing modes. This enables pTHINC to provide the same persistent desktop session across different client devices. For example, pTHINC can provide the same web browsing session appropriately scaled for display on a desktop computer and a PDA so that the same cookies, bookmarks, and other meta-data are continuously available on both machines simultaneously. pTHINC also leverages THINC's multimedia support to support display-intensive applications. Given limited display resolution on PDAs, pTHINC maximizes the use of screen real estate for remote display by moving control functionality from the screen to readily available PDA control buttons, improving system usability.

This section presents the design and implementation of pTHINC for the Windows Mobile PDA platform. Quantitative results evaluating pTHINC performance against

local PDA web browsers and other PDA thin-client approaches are also presented. These experimental results demonstrate that pTHINC provides superior performance and is the only PDA thin client that effectively supports crucial display intensive applications such as video playback.

4.1 pTHINC Usage Model

pTHINC is a thin-client system that consists of a simple client viewer application that runs on the PDA and a server that runs on a commodity PC. The server leverages more powerful PCs to store all data and run application logic. The client takes user input from the PDA stylus and the on-screen virtual keyboard and sends them to the server to pass to the applications. Screen updates are then sent back from the server to the client for display to the user.

When the pTHINC PDA client is started, the user is presented with a simple graphical interface where information such as server address and port, user authentication information, and session settings can be provided. pTHINC first attempts to connect to the server and perform the necessary handshaking. Once this process has been completed, pTHINC presents the user with the most recent display contents of her session. If the session does not exist, a new session is created. Existing sessions can be seamlessly continued without changes to the desktop settings or server configuration.

Unlike other thin-client systems, pTHINC provides a user with a persistent session model in which a user can launch a session running any desktop application at the server, then disconnect from that session and reconnect to it again anytime. When a user reconnects to the session, all of the applications continue running where the user left off, allowing the user to continue working as though he or she never disconnected.

The ability to disconnect and reconnect to a session at anytime is an important benefit for mobile wireless PDA users which may have intermittent network connectivity.

pTHINC's persistent session model enables a user to reconnect from devices other than the one on which the session was originally initiated. This provides users with seamless mobility across different devices. For example, if the user's PDA gets lost or stolen, she can easily switch to another PDA to access the session without any loss in functionality or data. Furthermore, pTHINC allows users to connect with non-PDA devices as well, for example, using a desktop computer. In this manner, users have access to the same desktop session from any computer or mobile device.

pTHINC's persistent session model addresses a key problem encountered by mobile users, namely, the lack of a common environment across computers. For example, web browsers often store important information such as bookmarks, cookies, and history, which enable them to function in a much more useful manner. However, when a user moves between multiple computers, this data, which is specific to a web browser installation, cannot move with the user, unless it is explicitly transferred. Transferring this data is a non-trivial process, as demonstrated by the number of programs and services available to address this problem [45, 103, 175]. Furthermore, web browsers often need helper applications to process different media content, and those applications may not be consistently available across all computers. pTHINC addresses this problem by enabling a user to remotely use the exact same desktop environment and applications from any computer. As a result, pTHINC can provide a common, consistent environment for mobile users across different devices without requiring them to attempt to repeatedly synchronize these environments.

To enable a user to access the same session on different devices, pTHINC must provide mechanisms to support different display sizes and resolutions. Toward this end, pTHINC provides a zoom feature that enables a user to zoom in and out of

a display and allows the display to be resized to fit the screen of the device being used. For example, if the server is running at 1024×768 but the client is a PDA with a display resolution of 640×480, pTHINC will resize the desktop display to fit the full display in the smaller screen of the PDA. pTHINC provides the PDA user with the option to increase the size of the display by zooming in to different parts of the display. Users are often familiar with the general layout of commonly visited websites, and are able to leverage this resizing feature to better navigate through web pages. For example, a user can zoom out of the display to view the entire page content and navigate hyperlinks, then zoom in to a region of interest for a better view.

To enable a user to access the same session on different devices, pTHINC must also provide mechanisms to support different display orientations. In a desktop environment, users are typically accustomed to having displays presented in landscape mode where the screen width is larger than its height. However, in a PDA environment, the choice is not always obvious. Some users may prefer having the display in portrait mode, as it is easier to hold the device in their hands, while others may prefer landscape mode in order to minimize the amount of side-scrolling necessary to view the desktop. To accommodate PDA user preferences, pTHINC provides an orientation feature that enables it to seamless rotate the display between landscape and portrait mode. The landscape mode is particularly useful for pTHINC users who frequently access their sessions on both desktop and PDA devices, providing those users with the same familiar landscape setting across different devices.

Because screen space is a relatively scarce resource on PDAs, pTHINC runs in full screen mode to maximize the screen area available to display the session. To be able to use all of the screen on the PDA and still allow the user to control and interact with it, pTHINC reuses the typical shortcut buttons found on PDAs to perform all the control functions available to the user. The buttons used by pTHINC do not

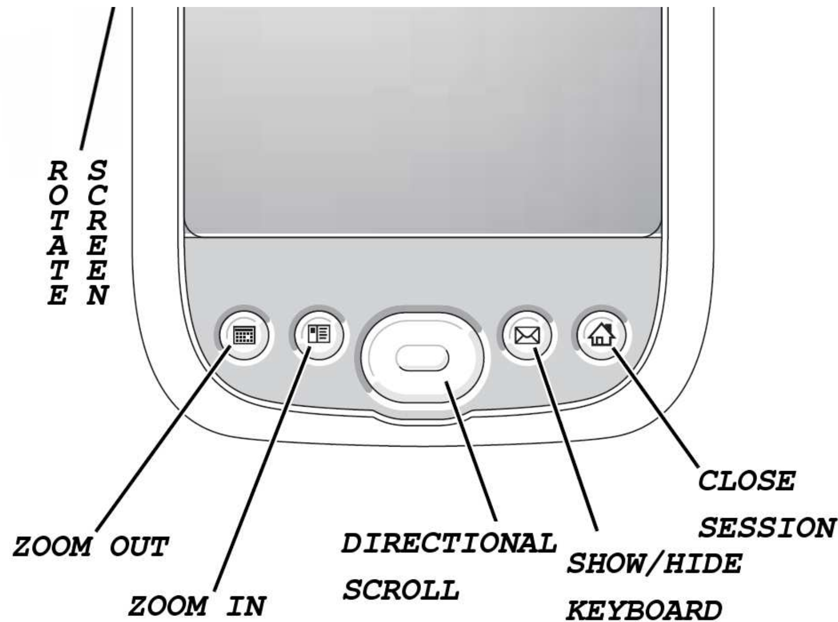


Figure 4.1 – pTHINC shortcut keys

require any OS environment changes; they are simply intercepted by the pTHINC client application when they are pressed. Figure 4.1 shows how pTHINC utilizes the shortcut buttons to provide easy navigation and improve the overall user experience. These buttons are not device specific, and the layout shown is common to widely-used PocketPC devices. pTHINC provides six shortcuts to support its usage model:

- *Rotate Screen*: The record button on the left edge is used to rotate the screen between portrait and landscape mode. Each time the button is pressed, the screen alternate between each mode.

- *Zoom Out*: The leftmost button on the bottom front is used to zoom out the display of the session. This way the user can get a bird's eye view of the desktop, and find the parts of the screen which are of interest.

- *Zoom In*: The second leftmost button on the bottom front is used to zoom in the display of the desktop to more clearly view content of interest.

- *Directional Scroll*: The middle button on the bottom front is used to scroll around the display using a single control button in a way that is already familiar to PDA users. This feature is particularly useful when the user has zoomed in to a region of the display such that only part of the display is visible on the screen.

- *Show/Hide Keyboard*: The second rightmost button on the bottom front is used to bring up a virtual keyboard drawn on the screen for devices which have no physical keyboard. The virtual keyboard uses standard PDA OS mechanisms, providing portability across different PDA environments.

- *Close Session*: The rightmost button on the bottom front is used to disconnect from the pTHINC session.

pTHINC uses the PDA touch screen, stylus, and standard user interface mechanisms to provide a user interface point-and-click metaphor similar to that provided by the mouse in a traditional desktop computing environment. pTHINC does not use a cursor since PDA environments do not provide one. Instead, a user can use the stylus to tap on different sections of the touch screen to indicate input focus. A single tap on the touch screen generates a corresponding single click mouse event. A double tap on the touch screen generates a corresponding double click mouse event. pTHINC provides two-button mouse emulation by using the stylus to press down on the screen for one second to generate a right mouse click. All of these actions are identical to the way users already interact with PDA applications in the PocketPC environment. For example, while web browsing, users can click on hyperlinks and focus on input boxes by simply tapping on the corresponding screen area. Unlike local PDA applications, pTHINC leverages more powerful desktop user interface metaphors to enable users to manipulate multiple open application windows instead

of being limited to a single application window at any given moment. This provides increased flexibility beyond what is currently available on PDA devices. Similar to a desktop environment, browser windows and other application windows can be moved around by pressing down and dragging the stylus.

4.2 pTHINC System Architecture

pTHINC builds on THINC's virtual display architecture to provide a thin-client system for PDAs. While other thin-client approaches intercept display commands at other layers of the display subsystem, pTHINC's display virtualization approach provides some key benefits in efficiently supporting PDA clients. For example, intercepting display commands at a higher layer between applications and the window system as is done by X [123] requires replicating and running a great deal of functionality on the PDA that is traditionally provided by the desktop window system. Given both the size and complexity of traditional window systems, attempting to replicate this functionality in the restricted PDA environment would have proved to be a daunting, and perhaps unfeasible task. Furthermore, applications and the window system often require tight synchronization in their operation and imposing a wireless network between them by running the applications on the server and the window system on the client would significantly degrade performance. On the other hand, intercepting at a lower layer by extracting pixels out of the framebuffer as they are rendered provides a simple solution that requires very little functionality on the PDA client, but can also result in degraded performance. The reason is that by the time the remote display server attempts to send screen updates, it has lost all semantic information that may have helped it encode efficiently, and it must resort to using a generic and expensive encoding mechanism on the server, as well as a potentially expensive de-

coding mechanism on the limited PDA client. In contrast to both the high and low level interception approaches, pTHINC's approach of intercepting at the device driver provides an effective balance between client and server simplicity, and the ability to efficiently encode and decode screen updates.

By using a low-level virtual display approach, pTHINC can efficiently encode application display commands using only a small set of low-level commands, as were described in Section 2.3. In a PDA environment, this set of commands provides a crucial component in maintaining the simplicity of the client in the resource-constrained PDA environment. Using THINC's non-blocking, server push update model, pTHINC obviates the need for clients to explicitly request display updates, thus minimizing the impact that the typical varying network latency of wireless links may have on the responsiveness of the system. Keeping in mind that resource constrained PDAs and wireless networks may not be able to keep up with a fast server generating a large number of updates, pTHINC is able to coalesce, clip, and discard updates automatically if network loss or congestion occurs, or the client cannot keep up with the rate of updates. This type of behavior proves crucial in a web browsing environment, where for example, a page may be redrawn multiple times as it is rendered on the fly by the browser. In this case, the PDA will only receive and render the final result, which clearly is all the user is interested in seeing.

THINC's Shortest-Remaining-Size-First (SRSF) preemptive update scheduler also plays a crucial role in pTHINC's architecture. In a web browsing environment, short jobs are associated with text and basic page layout components such as the page's background, which are critical web content for the user. On the other hand, large jobs are often lower priority "beautifying" elements, or, even worse, web page banners and advertisements, which are of questionable value to the user as he or she is browsing the page. Using SRSF, pTHINC is able to maximize the utilization of the relatively

scarce bandwidth available on the wireless connection between the PDA and the server.

4.2.1 Display Management

To enable users to just as easily access their web browser and helper applications from a desktop computer at home as from a PDA while on the road, pTHINC provides a resize mechanism to zoom in and out of the display of a web session. pTHINC resizing is completely supported by the server, not the client. The server resamples updates to fit within the PDAs viewport before they are transmitted over the network. pTHINC uses Fant's resampling algorithm to resize pixel updates. This provides smooth, visually pleasing updates with properly antialiasing and has only modest computational requirements.

pTHINC's resizing approach has a number of advantages. First, it allows the PDA to leverage the vastly superior computational power of the server to use high quality resampling algorithms and produce higher quality updates for the PDA to display. Second, resizing the screen does not translate into additional resource requirements for the PDA, since it does not need to perform any additional work. Finally, better utilization of the wireless network is attained since rescaling the updates reduces their bandwidth requirements.

To enable users to orient their displays on a PDA to provide a viewing experience that best accommodates user preferences and the layout of web pages or applications, pTHINC provides a display rotation mechanism to switch between landscape and portrait viewing modes. pTHINC display rotation is completely supported by the client, not the server. pTHINC does not explicitly recalculate the geometry of display updates to perform rotation, which would be computationally expensive. In-

stead, pTHINC simply changes the way data is copied into the framebuffer to switch between display modes. When in portrait mode, data is copied along the rows of the framebuffer from left to right. When in landscape mode, data is copied along the columns of the framebuffer from top to bottom. These very fast and simple techniques replace one set of copy operations with another and impose no performance overhead. pTHINC provides its own rotation mechanism to support a wide range of devices without imposing additional feature requirements on the PDA. Although some newer PDA devices provide native support for different orientations, this mechanism is not dynamic and requires the user to rotate the PDA's entire user interface before starting the pTHINC client. Windows Mobile provides native API mechanisms for PDA applications to rotate their UI on the fly, but these mechanisms deliver poor performance and display quality as the rotation is performed naively and is not completely accurate.

4.2.2 Video Playback

Video has gradually become an integral part of the World Wide Web, and its presence will only continue to increase. Web sites today not only use animated graphics and flash to deliver web content in an attractive manner, but also utilize streaming video to enrich the web interface. Users are able to view pre-recorded and live newscasts on CNN, watch sports highlights on ESPN, and even search through large collection of videos on Google Video. To allow applications to provide efficient video playback, interfaces have been created in display systems that allow video device drivers to expose their hardware capabilities back to the applications. pTHINC takes advantage of these interfaces and its virtual device driver approach to provide a virtual bridge between the remote client and its hardware and the applications, and transparently

support video playback.

On top of this architecture, pTHINC uses the YUV colorspace to encode the video content, which provides a number of benefits. First, it has become increasingly common for PDA video hardware to natively support YUV and be able to perform the colorspace conversion and scaling automatically. As a result, pTHINC is able to provide fullscreen video playback without any performance hits. Second, the use of YUV allows for a more efficient representation of RGB data without loss of quality, by taking advantage of the human eye's ability to better distinguish differences in brightness than in color. In particular, pTHINC uses the YV12 format, which allows full color RGB data to be encoded using just 12 bits per pixel. Third, YUV data is produced as one of the last steps of the decoding process of most video codecs, allowing pTHINC to provide video playback in a manner that is format independent. Finally, even if the PDA's video hardware is unable to accelerate playback, the colorspace conversion process is simple enough that it does not impose unreasonable requirements on the PDA.

A more concrete example of how pTHINC leverages the PDA video hardware to support video playback can be seen in our prototype implementation on the popular Dell Axim X51v PDA, which is equipped with the Intel 2700G multimedia accelerator. In this case, pTHINC creates an offscreen buffer in video memory and writes and reads from this memory region data on the YV12 format. When a new video frame arrives, video data is copied from the buffer to an overlay surface in video memory, which is independent of the normal surface used for traditional drawing. As the YV12 data is put onto the overlay, the Intel accelerator automatically performs both colorspace conversion and scaling. By using the overlay surface, pTHINC has no need to redraw the screen once video playback is over since the overlapped surface is unaffected. In addition, specific overlay regions can be manipulated by leveraging

the video hardware, for example to perform hardware linear interpolation to smooth out the frame and display it fullscreen, and to do automatic rotation when the client runs in landscape mode.

Readers may wonder how wireless bandwidth limitation can support fullscreen video. One of the main performance bottlenecks on pTHINC is wireless network capability. While the 802.11b specification allows up to 11 Mbps network bandwidth, previous studies have indicated that 6 Mbps network bandwidth is more typical of what is achievable in practice [161]. We assume that it is impractical to perform video playback which is played on a larger display size than the PDA can support. Therefore, the server sends a resized video data to the client when the clip is too large. Thus always resizing down to the screen size which the PDA can support results in staying under the bandwidth limitation.

4.3 Experimental Results

We have implemented a pTHINC client and server prototype that supports widely-used Windows Mobile-based Pocket PC devices as clients and both Windows and Linux machines as servers. To demonstrate its effectiveness to support mobile wireless devices we present experimental qualitative and quantitative results on different PDA devices for three popular applications, browsing web pages, financial management, and playing video content.

We compared pTHINC against native web applications running locally on the PDA to demonstrate the improvement that pTHINC can provide over the traditional approach. We also compared pTHINC against three of the most widely used thin clients that can run on PDAs, Citrix Meta-FrameXP (ICA), Microsoft Remote Desktop (RDP) and VNC (Virtual Network Computing).

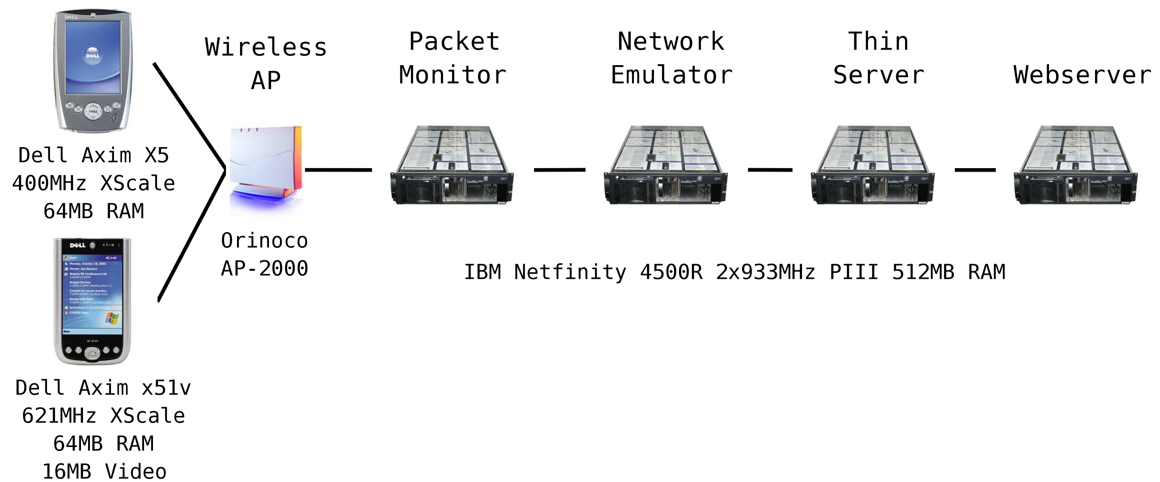


Figure 4.2 – PDA Experimental Testbed

4.3.1 Experimental Testbed

We conducted our experiments using two different wireless Pocket PC PDAs in an isolated Wi-Fi network testbed, as shown in Figure 4.2. The testbed consisted of two PDA client devices, a packet monitor, a thin-client server, and a web server. Except for the PDAs, all of the other machines were the same as those used for our 2D remote display experimental results, described in Section 2.7. The PDA clients connected to the testbed through a 802.11b Lucent Orinoco AP-2000 wireless access point. All experiments using the wireless network were conducted within ten feet of the access point, so we considered the amount of packet loss to be negligible in our experiments.

Two Pocket PC PDAs were used to provide results across both older, less powerful models and newer higher performance models. The older model was a Dell Axim X5 with a 400 MHz Intel XScale PXA255 CPU and 64 MB RAM running Windows Mobile 2003 and a Dell TrueMobile 1180 2.4Ghz CompactFlash card for wireless networking. The newer model was a Dell Axim X51v with a 624 MHz Intel XScale XPA270 CPU and 64 MB RAM running Windows Mobile 5.0 and integrated 802.11b wireless networking. The X51v has an Intel 2700G multimedia accelerator with 16MB

Client	1024×768	640×480	Depth	Resize	Clip
RDP	no	yes	8-bit	no	yes
VNC	yes	yes	16-bit	no	no
ICA	yes	yes	16-bit	yes	no
pTHINC	yes	yes	24-bit	yes	no

Table 4.1 – PDA Testbed Configuration Settings

video memory. Both PDAs are capable of 16-bit color but have different screen sizes and display resolutions. The X5 has a 3.5 inch diagonal screen with 240×320 resolution. The X51v has a 3.7 inch diagonal screen with 480×640.

The four thin clients that we used support different levels of display quality as summarized in Table 4.1. The RDP client only supports a fixed 640×480 display resolution on the server with 8-bit color depth, while other platforms provide higher levels of display quality. To provide a fair comparison across all platforms, we conducted our experiments with thin-client sessions configured for two possible resolutions, 1024×768 and 640×480. Both ICA and VNC were configured to use the native PDA resolution of 16-bit color depth. The current pTHINC prototype uses 24-bit color directly and the client downsamples updates to the 16-bit color depth available on the PDA. RDP was configured using only 8-bit color depth since it does not support any better color depth. Since both pTHINC and ICA provide the ability to view the display resized to fit the screen, we measured both clients with and without the display resized to fit the PDA screen. Each thin client was tested using landscape rather than portrait mode when available. All systems run on the X51v could run in landscape mode because the hardware provides a landscape mode feature. However, the X5 does not provide this functionality. Only pTHINC directly supports landscape mode, so it was the only system that could run in landscape mode on both the X5 and X51v.

To provide a fair comparison, we also standardized on common hardware and

operating systems whenever possible. All of the systems used the Netfinity server as the thin-client server. For the two systems designed for Windows servers, ICA and RDP, we ran Windows 2003 Server on the server. For the other systems which support X-based servers, VNC and pTHINC, we ran the Debian Linux Unstable distribution with the Linux 2.6.10 kernel on the server. We used the latest thin-client server versions available on each platform at the time of our experiments, namely Citrix MetaFrame XP Server for Windows Feature Release 3, Microsoft Remote Desktop built into Windows XP and Windows 2003 using RDP 5.2, and VNC 4.0.

4.3.2 Application Benchmarks

For our qualitative results, we compared two common PDA application scenarios, web browsing and financial management. We present web browsing using pTHINC with a Linux server and financial management using pTHINC with a Windows server to demonstrate the flexibility that pTHINC provides in delivering both Linux and Windows applications. For these tests, we use the Dell Axim X51v PDA exclusively.

For our quantitative results, we used two web application benchmarks for our experiments based on two common application scenarios, browsing web pages and playing video content from the web. Since many thin-client systems including two of the ones tested are closed and proprietary, we measured their performance in a noninvasive manner by capturing network traffic with a packet monitor and using a variant of slow-motion benchmarking [93] previously developed to measure thin-client performance in PDA environments [70]. This measurement methodology accounts for both the display decoupling that can occur between client and server in thin-client systems as well as client processing time, which may be significant in the case of PDAs.

To measure web browsing performance, we used a web browsing benchmark based on the Web Text Page Load Test from the Ziff-Davis i-Bench benchmark suite [54]. The benchmark consists of JavaScript controlled load of 55 pages from the web server. The pages contain both text and graphics with pages varying in size. The graphics are embedded images in GIF and JPEG formats. The original i-Bench benchmark was modified for slow-motion benchmarking by introducing delays of several seconds between the pages using JavaScript. Then two tests were run, one where delays where added between each page, and one where pages where loaded continuously without waiting for them to be displayed on the client. In the first test, delays were sufficiently adjusted in each case to ensure that each page could be received and displayed on the client completely without temporal overlap in transferring the data belonging to two consecutive pages. We used the packet monitor to record the packet traffic for each run of the benchmark, then used the timestamps of the first and last packet in the trace to obtain our latency measures [70]. The packet monitor also recorded the amount of data transmitted between the client and the server. The ratio between the data traffic in the two tests yields a scale factor. This scale factor shows the loss of data between the server and the client due to inability of the client to process the data quickly enough. The product of the scale factor with the latency measurement produces the true latency accounting for client processing time.

To run the web browsing benchmark, we used Mozilla Firefox 1.0.4 running on the thin-client server for the thin clients, and Windows Internet Explorer (IE) Mobile for 2003 and Mobile for 5.0 for the native browsers on the X5 and X51v PDAs, respectively. In all cases, the web browser used was sized to fill the entire display region available.

To measure video playback performance, we used a video benchmark that consisted of playing a 34.75s MPEG-1 video clip containing a mix of news and entertain-

ment programming at full-screen resolution. The video clip is 5.11 MB and consists of 834 352x240 pixel frames with an ideal frame rate of 24 frames/sec. We measured video performance using slow-motion benchmarking by monitoring resulting packet traffic at two playback rates, 1 frames/second (fps) and 24 fps, and comparing the results to determine playback delays and frame drops that occur at 24 fps to measure overall video quality [93]. For example, 100% quality means that all video frames were played at real-time speed. On the other hand, 50% quality could mean that half the video data was dropped, or that the clip took twice as long to play even though all of the video data was displayed.

To run the video benchmark, we used Windows Media Player 9 for Windows-based thin-client servers, MPlayer 1.0 pre 6 for X-based thin-client servers, and Windows Media Player 9 Mobile and 10 Mobile for the native video players running locally on the X5 and X51v PDAs, respectively. In all cases, the video player used was sized to fill the entire display region available.

4.3.3 Qualitative Results

Figures 4.3 and 4.4 show screenshots of web browsing using pTHINC and a full-function Mozilla Firefox web browser versus running Pocket IE natively on the PDA, respectively. Because of the limited resolution of the screenshots, they effectively show the layout differences between different platforms but do not reproduce the actual display quality of the PDA, which is much better than what is shown in these figures.

Both screenshots show the same web page from BBC News [15], but display them very differently. pTHINC provides the user with a wide range of display options, enabling the user to see the entire web page as well as zoom in on different parts

Figure 4.3 – pTHINC Web Screenshot: BBC News

Figure 4.4 – Native IE Screenshot: BBC News

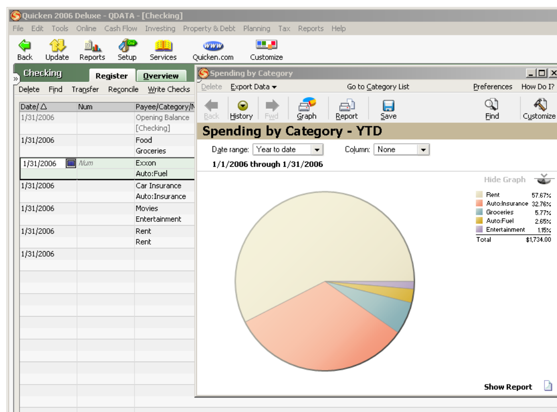


Figure 4.5 – pTHINC Application Screenshot: Quicken

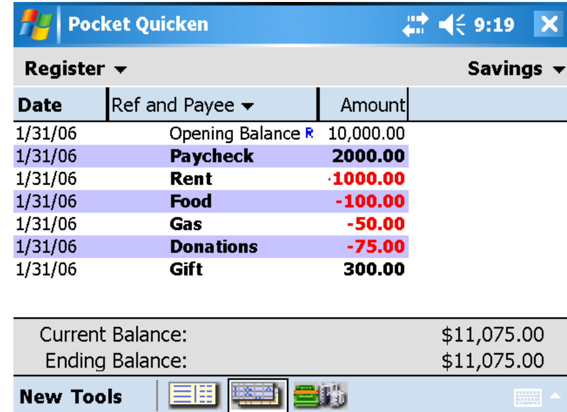


Figure 4.6 – Native Application Screenshot: Pocket Quicken

of the web page. The result is a quality display experience similar to the familiar experience of web browsing on a desktop computer. pTHINC enables the user to use a full-function desktop web browser on the PDA, providing robust support for viewing the same wide range of web sites that are accessible on a desktop computer.

In contrast, running the native PDA application provides the user with a limited viewing experience of only being able to see a small portion of the web page at a time and needing to scroll around the web page frequently to view the content. Because the BBC News web page is not designed for viewing on PDAs, the native PDA web

browser ends up only being able to display the top left corner of the web page when it is initially downloaded. This top left corner primarily consists of the BBC News logo, displaying very little useful content to the user. In addition, the Pocket IE user interface consumes a substantial amount of screen area, particularly the top and bottom menu bars, further reducing the available screen area for displaying useful web content. A bigger problem is that Pocket IE does not correctly parse parts of the BBC News web page depending on the particular web content being displayed. In scrolling around the BBC News web page shown, parts of the page are missing or misaligned. These problems are due to the resource restrictions of the PDA, resulting in the Pocket IE web browser having more limited functionality as a stripped down version of the equivalent Microsoft IE web browser that runs on a desktop computer. A wide range of web sites such as the BBC News web site cannot be displayed properly on the PDA using Pocket IE because of its incomplete support for commonly used web technologies such as JavaScript.

Figures 4.5 and 4.6 show screenshots of running Quicken [115] financial management software using pTHINC and the full-function desktop version versus running Pocket Quicken natively on the PDA, respectively. pTHINC provides the user with a quality display experience similar to the familiar experience of using Quicken on a desktop computer, enabling the user to use the full-function desktop Quicken software on the PDA. As a result, users can access their Quicken data via pTHINC across handheld devices and desktop computers without any need to maintain and attempt to synchronize multiple copies of their data across different devices.

In contrast, running the native PDA Pocket Quicken application provides the user with access to a very limited application compared to the original desktop version. Pocket Quicken is not capable of displaying in-depth financial analysis reports due to display resolution limitations and sub-par processing capabilities. Pocket Quicken is

limited to maintaining short lists of expenses and viewing balances. Because of its limited functionality, Pocket Quicken also requires the desktop version to be installed on another desktop machine and needs to synchronize its data with the desktop version, requiring the user to purchase two versions of the software to provide financial management functionality on the PDA.

4.3.4 Quantitative Results

Figures 4.7 and 4.8 show the results of running the web browsing benchmark. For each platform, we show results for up to four different configurations, two on the X5 and two on the X51v, depending on whether each configuration was supported. However, not all platforms could support all configurations. The local browser only runs at the display resolution of the PDA, 480×680 or less for the X51v and the X5. RDP only runs at 640×480. Neither platform could support 1024×768 display resolution. ICA only ran on the X5 and could not run on the X51v because it did not work on Windows Mobile 5.

Figure 4.7 shows the average latency per web page for each platform. pTHINC provides the lowest average web browsing latency on both PDAs. On the X5, pTHINC performs up to 70 times better than other thin-client systems and 8 times better than the local browser. On the X51v, pTHINC performs up to 80 times better than other thin-client systems and 7 times better than the native browser. In fact, all of the thin clients except VNC outperform the local PDA browser, demonstrating the performance benefits of the thin-client approach. Usability studies have shown that web pages should take less than one second to download for the user to experience an uninterrupted web browsing experience [94]. The measurements show that only the thin clients deliver subsecond web page latencies. In contrast, the local browser requires

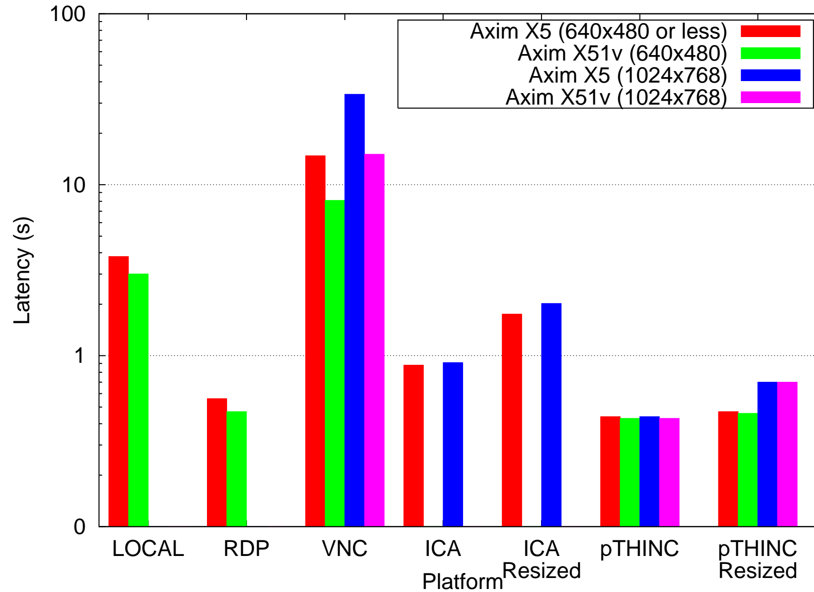


Figure 4.7 – PDA Browsing Benchmark: Average Page Latency

more than 3 seconds on average per web page. The local browser performs worse since it needs to run a more limited web browser to process the HTML, JavaScript, and do all the rendering using the limited capabilities of the PDA. The thin clients can take advantage of faster server hardware and a highly tuned web browser to process the web content much faster.

Figure 4.7 shows that RDP is the next fastest platform after pTHINC. However, RDP is only able to run at a fixed resolution of 640×480 and 8-bit color depth. Furthermore, RDP also clips the display to the size of the PDA screen so that it does not need to send updates that are not visible on the PDA screen. This provides a performance benefit assuming the remaining web content is not viewed, but degrades performance when a user scrolls around the display to view other web content. RDP achieves its performance with significantly lower display quality compared to the other thin clients and with additional display clipping not used by other systems. As a result, RDP performance alone does not provide a complete comparison with the

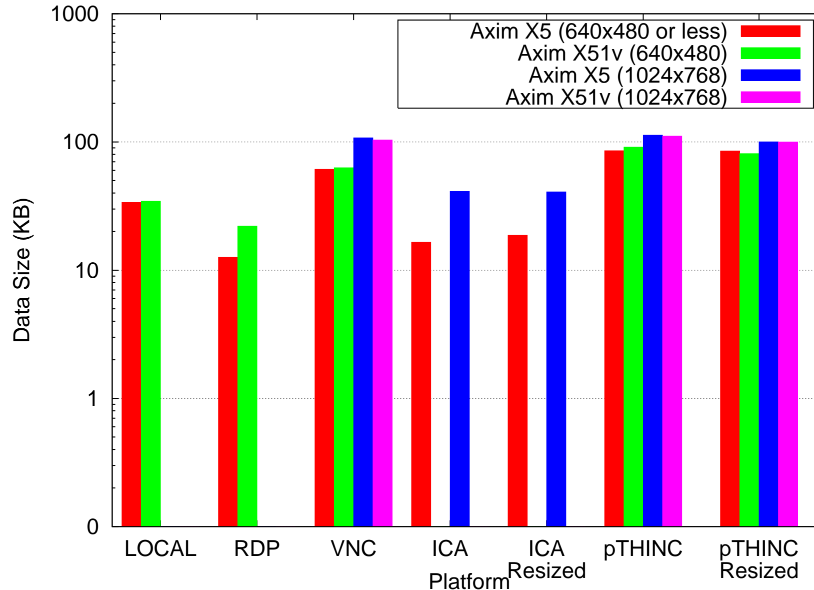


Figure 4.8 – PDA Browsing Benchmark: Average Page Data Transferred

other platforms. In contrast, pTHINC provides the fastest performance while at the same time providing equal or better display quality than the other systems.

Since VNC and ICA provide similar display quality to pTHINC, these systems provide a more fair comparison of different thin-client approaches. ICA performs worse in part because it uses higher-level display primitives that require additional client processing costs. VNC performs worse in part because it loses display data due to its client-pull delivery mechanism and because of the client processing costs in decompressing raw pixel primitives. In both cases, their performance was limited in part because their PDA clients were unable to keep up with the rate at which web pages were being displayed.

Figure 4.7 also shows measurements for those thin clients that support resizing the display to fit the PDA screen, namely ICA and pTHINC. Resizing requires additional processing, which results in slower average web page latencies. The measurements show that the additional delay incurred by ICA when resizing versus not resizing is

much more substantial than for pTHINC. ICA performs resizing on the slower PDA client. In contrast, pTHINC leverage the more powerful server to do resizing, reducing the performance difference between resizing and not resizing. Unlike ICA, pTHINC is able to provide subsecond web page download latencies in both cases.

Figure 4.8 shows the data transferred in KB per page when running the slow-motion version of the tests. All of the platforms have modest data transfer requirements of roughly 100 KB per page or less. This is well within the bandwidth capacity of Wi-Fi networks. The measurements show that the local browser does not transfer the least amount of data. This is surprising as HTML is often considered to be a very compact representation of content. Instead, RDP is the most bandwidth efficient platform, largely as a result of using only 8-bit color depth and screen clipping so that it does not transfer the entire web page to the client. pTHINC overall has the largest data requirements, slightly more than VNC. This is largely a result of the current pTHINC prototype's lack of native support for 16-bit color data in the wire protocol. However, this result also highlights pTHINC's performance as it is faster than all other systems even while transferring more data. Furthermore, as newer PDA models support full 24-bit color, these results indicate that pTHINC will continue to provide good web browsing performance.

Since display usability and quality are as important as performance, Figures 4.9 to 4.12 compare screenshots of the different thin clients when displaying a web page, in this case from the popular BBC news website. Except for ICA, all of the screenshots were taken on the X51v in landscape mode using the maximum display resolution settings for each platform given in Table 4.1. The ICA screenshot was taken on the X5 since ICA does not run on the X51v. While the screenshots lack the visual fidelity of the actual device display, several observations can be made. Figure 4.9 shows that RDP does not support fullscreen mode and wastes lots of screen space for

Figure 4.9 – PDA Browser Screenshot: RDP 640x480

Figure 4.10 – PDA Browser Screenshot: VNC 1024x768

Figure 4.11 – PDA Browser Screenshot: ICA Resized 1024x768

Figure 4.12 – PDA Browser Screenshot: pTHINC Resized 1024x768

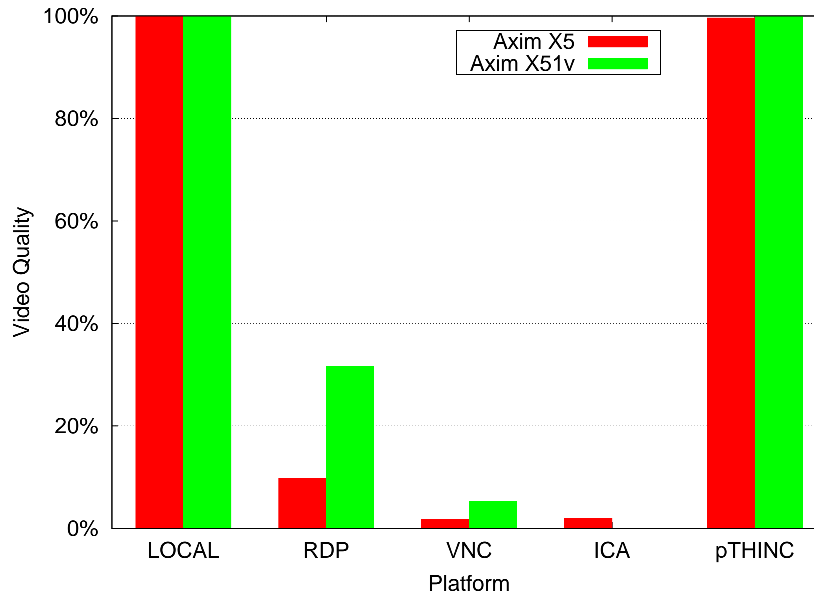


Figure 4.13 – PDA Video Benchmark: Fullscreen Video Quality

controls and UI elements, requiring the user to scroll around in order to access the full contents of the web browsing session. Figure 4.10 shows that VNC makes better use of the screen space and provides better display quality, but still forces the user to scroll around to view the web page due to its lack of resizing support. Figure 4.11 shows ICA's ability to display the full web page given its resizing support, but that its lack of landscape capability and poorer resize algorithm significantly compromise display quality. In contrast, Figure 4.12 shows pTHINC using resizing to provide a high quality fullscreen display of the full width of the web page. pTHINC maximizes the entire viewing region by moving all controls to the PDA buttons. In addition, pTHINC leverages the server computational power to use a high quality resizing algorithm to resize the display to fit the PDA screen without significant overhead.

Figures 4.13 and 4.14 show the results of running the video playback benchmark. For each platform except ICA, we show results for an X5 and X51v configuration. ICA could not run on the X51v as noted earlier. The measurements were done using

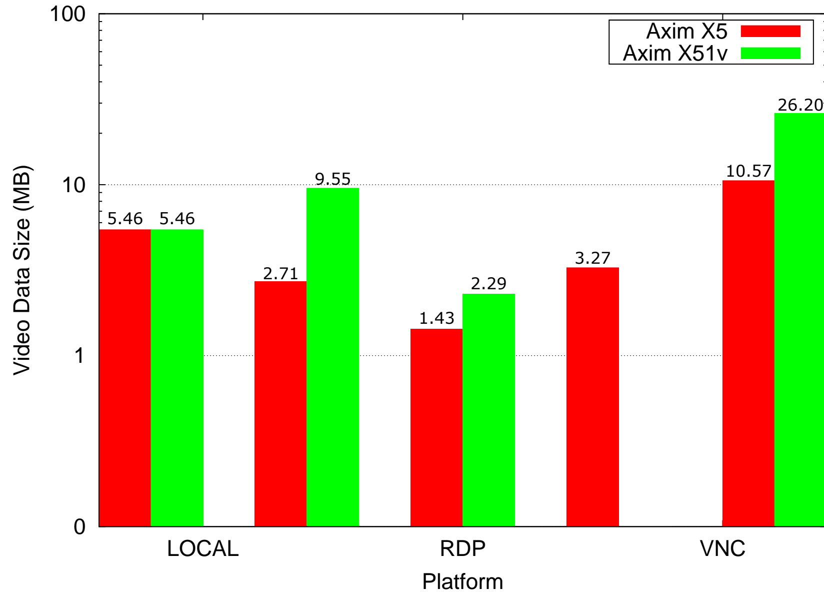


Figure 4.14 – PDA Video Benchmark: Fullscreen Video Data

settings that reflected the environment a user would have to access a web session from both a desktop computer and a PDA. As such, a 1024×768 server display resolution was used whenever possible and the video was shown at fullscreen. RDP was limited to 640×480 display resolution as noted earlier. Since viewing the entire video display is the only really usable option, we resized the display to fit the PDA screen for those platforms that supported this feature, namely ICA and pTHINC.

Figure 4.13 shows the video quality for each platform. pTHINC is the only thin client able to provide perfect video playback quality, similar to the native PDA video player. All of the other thin clients deliver very poor video quality. With the exception of RDP on the X51v which provided unacceptable 35% video quality, none of the other systems were even able to achieve 10% video quality. VNC and ICA have the worst quality at 8% on the X5 device.

pTHINC's native video support enables superior video performance, while other thin clients suffer from their inability to distinguish video from normal display up-

dates. They attempt to apply ineffective and expensive compression algorithms on the video data and are unable to keep up with the stream of updates generated, resulting in dropped frames or long playback times. VNC suffers further from its client-pull update model because video frames are generated faster than the rate at which the client can process and send requests to the server to obtain the next display update. Figure 4.14 shows the total data transferred during video playback for each system. The native player is the most bandwidth efficient platform, sending less than 6 MB of data, which corresponds to about 1.2 Mbps of bandwidth. pTHINC's 100% video quality requires about 25 MB of data which corresponds to a bandwidth usage of less than 6 Mbps. While the other thin clients send less data than pTHINC, they do so because they are dropping video data, resulting in degraded video quality.

Figures 4.15 to 4.18 compare screenshots of the different thin clients when displaying the video clip. Except for ICA, all of the screenshots were taken on the X51v in landscape mode using the maximum display resolution settings for each platform given in Table 4.1. The ICA screenshot was taken on the X5 since ICA does not run on the X51v. Figures 4.15 and 4.16 show that RDP and VNC are unable to display the entire video frame on the PDA screen. RDP wastes screen space for UI elements and VNC only shows the top corner of the video frame on the screen. Figure 4.17 shows that ICA provides resizing to display the entire video frame, but did not proportionally resize the video data, resulting in strange display artifacts. In contrast, Figure 4.18 shows pTHINC using resizing to provide a high quality fullscreen display of the entire video frame. pTHINC provides visually more appealing video display than RDP, VNC, or ICA.

Figure 4.15 – PDA Video Screenshot: RDP 640x480

Figure 4.16 – PDA Video Screenshot: VNC 1024x768

Figure 4.17 – PDA Video Screenshot: ICA Resized 1024x768

Figure 4.18 – PDA Video Screenshot: pTHINC Resized 1024x768

4.4 Summary

This chapter introduced pTHINC, a thin-client architecture for wireless PDAs. pTHINC provides key architectural and usability mechanisms such as server-side screen resizing, client-side screen rotation using simple copy techniques, YUV video playback support, maximizing screen space for display updates, and leveraging existing PDA control buttons for most user interface operations. pTHINC transparently supports

traditional desktop applications on PDA devices and desktop machines, providing mobile users with ubiquitous access to a consistent, personalized, and full-featured computing environment across heterogeneous devices.

We have implemented pTHINC and measured its performance on web applications compared to existing commercial thin-client systems and native web applications. Our results on multiple mobile wireless devices demonstrate that pTHINC delivers web browsing performance up to 80 times better than existing thin-client systems, and 8 times better than a native PDA browser. In addition, pTHINC is the only PDA thin client that transparently provides full-screen, full frame rate video playback, making web sites with multimedia content accessible to mobile web users. Our experiences with the system demonstrate that pTHINC can provide a superior approach for delivering application services to mobile handheld devices.

Chapter 5

Desktop Virtualization

As computers have become more powerful and portable, and broadband networks have become a commodity, ubiquitous computer access has moved beyond being a luxury to a common necessity. However, as the number of computers available to users increases, so does the disparity of desktop environments users must deal with and places where their personal data gets stored.

The complexity of managing these disparate computing environments quickly becomes a burden. At a personal level, users must deal with keeping track of their data, keeping computers in sync, and dealing with the subtle but important differences of each environment. At an organization level, the management problem quickly becomes exacerbated. Each computer needs to be constantly patched and upgraded to protect it, and their data, from the myriad of viruses and other attacks commonplace in today's networks. Furthermore, as mobile users transport their portable computers from one place to another, it is not uncommon for these machines to be damaged or stolen, resulting in the loss of any important data stored on them. Even in the best case, when such data can be recovered from backup, the time consuming process of reconstituting the state of the lost machine on another device, results in a huge

disruption in critical computing service for the user.

THINC's architecture provides a model that enables consistent desktop environments to support the mobility and ubiquitous access needs of users today. THINC's display virtualization allows all display and desktop state to be decoupled and encapsulated from the underlying hardware and operating system. Since the desktop session has no dependencies to its host computer, it can be easily moved from one machine to another by leveraging operating system virtualization and checkpoint/restart mechanisms.

The manner in which the virtualized desktop is accessed provides for different operational modes. In one case, the desktop environment and user data can be encapsulated in a portable storage device and carried by the user as she moves across computers [112, 113]. This model allows the user to directly exploit the characteristics of the computer without being tied in any way to it. To access the desktop session, the user simply connects the storage device to a computer, resumes the checkpointed session, and uses a THINC client to connect to the stored server. All data modifications will be automatically saved to the portable device. Once the user decides to change computers, she simply checkpoints the current state of the session to the storage device, and moves on.

A different model can be provided by combining THINC's display virtualization and remote display architecture to create a desktop utility computing infrastructure. The rest of this chapter discusses this model in more detail. It also presents an exploration into the security implications and vulnerabilities of this kind of infrastructure, and a novel architecture that mitigates the most important of these vulnerabilities, distributed denial of service attacks.

5.1 MobiDesk: Mobile Virtual Desktop Computing

THINC has been integrated into MobiDesk [14], a mobile virtual desktop computing hosting infrastructure. MobiDesk uses the network to decouple a user's desktop from any particular end-user device by moving all application logic to hosting providers. In this manner, end-user devices are simply used to transmit user input and display application output, allowing them to be simple stateless clients. MobiDesk also decouples a user's desktop computing session from the underlying operating system and server instance, allowing a user's entire computing environment to be migrated transparently from one server to another. This enables a server to be brought down for maintenance and upgraded in a timely manner with minimal impact on the availability of a user's computing services. Once the original machine has been updated, the user's computing session can be migrated back and continue to execute even though the underlying operating system may have changed. MobiDesk ensures that any network connections associated with the user's computing session are maintained, even as the session is migrated from one machine to another. MobiDesk provides these benefits without modifying, recompiling, or relinking applications or operating system kernels. MobiDesk requires no changes to clients other than being able to execute a simple user-space application to process and display input and output.

MobiDesk provides a mobile virtual desktop computing environment by introducing a thin virtualization layer between a user's computing environment and the underlying system. MobiDesk focuses on virtualizing three key system resources: display, operating system, and network. MobiDesk virtualizes display resources by leveraging THINC's virtual display driver to decouple all display state from the hosting server, and efficiently intercept, encode and redirect display updates from the server

to an end-user device. MobiDesk virtualizes operating system resources by leveraging ZAP [68, 101] to provide a virtual private namespace for each desktop computing session. The namespace offers a host independent virtualized view of an operating system, enabling the session to be transparently migrated from one server to another. MobiDesk virtualizes network resources by leveraging MOVE [140] to provide virtual address identifiers for connections, and a transport-independent proxy mechanism. Together, they preserve all network connections associated with a user's computing session, even if it is migrated from one server to another inside the MobiDesk server infrastructure.

The MobiDesk hosted desktop computing approach provides a number of important benefits over current computing approaches:

- *High-availability and reliable application services*: Because MobiDesk is designed to work with unmodified legacy applications and commodity operating systems, it offers the potential to bring about more reliable computing without giving up the large investments already made in the existing software base. Furthermore, decoupling from the underlying hardware and operating system allows applications to be moved anywhere, and in particular, migrated off faulty hosts, and before maintenance and upgrades. In contrast to today's long periods of service downtime due to maintenance and upgrades, MobiDesk enables hardware and operating systems to be upgraded in a timely manner with minimal impact on application service availability — by migrating applications to another machine that has already been updated. With MobiDesk, system administrators no longer need to schedule downtime in advance and in cooperation with all the users, thereby closing the vulnerability window of unrepaired systems.

- *Persistence and continuity of business logic*: MobiDesk moves away from the current model of simply backing up file data to secure remote locations, and instead protects entire computing environments by running hosting providers in secure remote locations. This enables academic, business, and government institutions to function much more effectively in times of crisis. Restoring an organization's local computing infrastructure from backup consequent to a crisis is an extremely slow, time consuming process that is increasingly ineffective given the scale of IT infrastructure being deployed today. MobiDesk offers a different, improved model of continuous uptime, especially during a crisis, when infrastructure availability is most crucial.

- *Secure, low-cost global access and transparent user mobility*: MobiDesk client access devices just need to be able to connect to the Internet. They do not need to provide complex computing functionality, making it unnecessary to continuously upgrade to more powerful desktop machines. Simpler, lower-cost, possibly longer battery life client access devices can be made more readily available for such a service. These devices may come in many shapes and sizes, from desktop machines with megapixel displays to handheld devices with pocket sized screens. Furthermore, because all persistent user state is maintained on the servers, users are able to securely access, and freely move among any client access devices and pick up right where they left off.

- *On-demand access to application and computational resources*: By multiplexing a large pool of shared resources among many users, an individual can gain access to substantially more applications and resources than can be afforded on one's local desktop computer. In terms of applications, MobiDesk can provide a wider

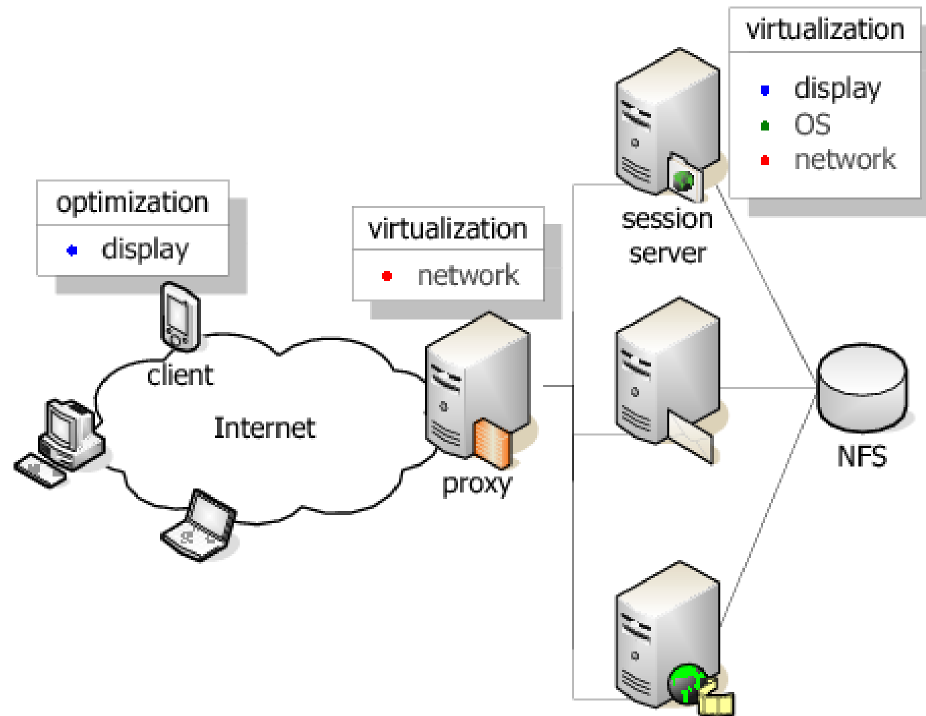


Figure 5.1 – MobiDesk Architecture

range of affordable application services on multiple operating system platforms by amortizing costs over a large number of users. Since not all applications will be in use by all users at one time, statistical multiplexing can serve a larger number of users with fewer software licenses. In terms of resources, a user can be given resource allocations which can be scaled up or down as necessary. Instead of having to throw away their existing local desktop machines every time they need more compute power, users can just ask their service provider to scale up their allocation.

MobiDesk is architected as a proxy-based server cluster system, comparable to systems deployed today by application service providers. The overall architecture of the system is depicted in Figure 5.1. MobiDesk is composed of a proxy, a group of back-end session servers connected in a LAN, a storage server infrastructure, and a

number of external, heterogeneous clients through which users access the system.

The proxy acts as a front-end that admits service requests from clients across the Internet, and dispatches the requests to the appropriate back-end application servers. The proxy, operating at layer 7, exposes a single entry point to the clients, and employs suitable admission and service dispatching policies. The back-end compute servers host completely virtualized environments within which the computing sessions of MobiDesk's users run. The network storage server infrastructure is used for all persistent file storage. The clients are merely *inputing* and *outputting* devices connected to the servers across the Internet.

Users interact with their MobiDesk sessions through a remote display *session viewer*, a simple device or application that relays the user's input and the session's output between the client and the server through a secure channel. Each user in the system is assigned a username and password. Upon the first login, the proxy performs appropriate authentication, and connects the user to a MobiDesk session server. The session server creates a virtual private environment that is populated with a complete set of operating system resources and desktop applications. In contrast to the traditional centralized computing model where users are aware of each others' presence and activities, MobiDesk's sessions are isolated from one another and the underlying server environment. To the user, the session appears no different than a private computer, even though the user's session may coexist with many other sessions on a shared server. When the client disconnects, the session continues to run on the MobiDesk server, unless the user explicitly logs out. On future connections, the session will be in the same state it was when the user last disconnected.

By providing a virtual private environment for each user, MobiDesk is able to dynamically relocate sessions to meet load balancing, system maintenance and/or quality of service requirements. Sessions can be checkpointed and migrated trans-

parently at any point in time. To keep track of the sessions as migration occurs, MobiDesk implements a *session cookie* mechanism. As new sessions are created, the proxy generates a unique cookie that is passed to the hosting servers and associated with the new session. Whenever a session is migrated, the destination server uses the cookie to inform the proxy of the new location. Finally, the next time the user logs in, the proxy will use the cookie to identify the server where the user's session is being hosted.

As mentioned before, MobiDesk provides a mobile virtual desktop computing environment by virtualizing 3 key components: the display, the operating system, and the network. The following sections discuss each of these components in detail.

5.1.1 Display Virtualization

To make MobiDesk a viable replacement to the traditional desktop computing model, it needs to be able to deliver the look and feel of all unmodified desktop applications end-users expect. MobiDesk must work within the framework of existing display systems, intercepting display commands from unmodified applications and redirecting these commands to remote clients. To provide good WAN performance, the virtualization must intercept display commands at an appropriate abstraction layer to provide sufficient information to optimize the processing of display commands in a latency sensitive manner. Furthermore, to support transparent user mobility and eliminate client administration complexity, MobiDesk should support the use of thin, stateless clients, by ensuring that all persistent display state is stored in the server infrastructure.

To achieve these goals MobiDesk leverages THINC's display virtualization and remote display mechanisms, by providing a separate virtual video device for each

computing session. Rather than sending display commands to local display hardware, the virtual video driver packages up display commands associated with a user's computing session and sends them over the network to a remote client, using THINC's low-level, minimum-overhead protocol. The protocol mimics the operations most commonly found in display hardware, allowing clients to do little more than forward protocol commands to their local video hardware, thus reducing the latency of display processing. To provide security, all protocol traffic is encrypted using the standard RC4 [125] stream cipher algorithm.

THINC's virtual and remote display architecture provides a number of crucial benefits to MobiDesk:

- First, THINC enables MobiDesk to take full advantage of existing infrastructure and hardware interfaces, while maximizing client resources and requiring minimal computation on the client. Furthermore, new video hardware features can be supported with at most the same amount of work necessary for supporting them in traditional desktop display drivers. While there is some loss of semantic display information at the low-level video device driver interface, our experiments with desktop applications such as web browsers, indicate that the vast majority of application display commands issued can be mapped directly to standard video hardware primitives.

- Second, THINC enables MobiDesk to maximize client resources to natively and efficiently support important desktop applications, in particular video playback and bidirectional audio. As an example, video support is provided by leveraging alternative YUV video formats natively supported by almost all off-the-shelf video cards available today. In this manner, video data can simply be transferred from the server to the client video hardware, which automatically does

inexpensive, high speed, color space conversion and scaling. THINC also allows MobiDesk to adapt to the many client devices available to connect to the desktop hosting infrastructure. For example, THINC can automatically resize updates to fit within the screen of a small portable device.

- Third, THINC provides two important server-side mechanisms for improving performance when deploying MobiDesk in high latency WAN environments. The first mechanism is the use of a server push model for sending display updates to the client. As soon as display updates are generated on the server, they are delivered to the client. Clients are not required to explicitly request display updates, which add additional network latency to command processing. The second mechanism is the use of display update scheduling to improve the responsiveness of the system, using a *Shortest-Remaining-Size-First (SRSF)* preemptive scheduler. In display applications, short jobs are normally associated with text and general GUI layout components, which are critical to the usability of the system. On the other hand, large jobs are normally lower priority "beautifying" GUI elements, such as image decorations, desktop backgrounds and web page banners.

- Finally, enables MobiDesk to support thin, stateless display clients by storing all session state at the respective session server. Although MobiDesk takes advantage of client resources when available, all client state is considered temporary and destroyed upon disconnect. When a remote client connects to the MobiDesk infrastructure, the server running the user's computing session transfers the current session state to the client. For the duration of the connection, the client forwards input events to the server, which in turn forwards display updates back to the client. The client at no point has an intermediate session

state differing from the server. Furthermore, if allowed by the user, multiple clients can be connected to the same session at the same time, all of them accessing the same centralized view distributed from the server. When a client eventually disconnects, it leaves no state behind in the local computer.

5.1.2 Operating System Virtualization

Using ZAP [68, 101], MobiDesk encapsulates user sessions within a host-independent virtualized view of the operating system. Unlike traditional operating systems, each session is a self contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted. This virtualization operates at a finer granularity than virtual machine approaches, such as VMware [151, 154], which can be used to migrate entire operating system environments. Unlike MobiDesk, virtual machines decouple processes from the underlying hardware, but tie them to an instance of the operating system. As a result, virtual machines cannot migrate processes separate from the operating system, and cannot continue running those processes if the operating system ever goes down, such as during security upgrades. In contrast, MobiDesk decouples process execution from the underlying operating system allowing it to migrate processes to another computer even in the presence of server hardware and operating system maintenance and upgrades.

MobiDesk provides each computing session with its own *virtual private namespace*, that provides the only means for processes to access the underlying operating system. To guarantee correct operation of unmodified legacy applications, this virtualization is done completely transparent. This is accomplished by providing a traditional environment with unchanged application interfaces and access to operating system

services and resources.

MobiDesk's namespace is private in that only processes within the session can see the namespace, and the namespace in turn masks out resources that are not contained in the session. Processes inside the session appear to one another as normal processes, and they are able to communicate using traditional inter-process communication (IPC) mechanisms. On the other hand, no process interaction is possible across the session's boundaries, because outside processes are not part of the private namespace. Processes inside a session and those outside of it are only able to communicate over remote procedure call mechanisms, traditionally used to communicate across computers.

MobiDesk's namespace is virtual in that all operating system resources, including processes, user information, files, and devices, are accessed through virtual identifiers. These virtual identifiers are distinct from the host-dependent, physical resource identifiers used by the operating system. The session's namespace uses the virtual identifiers to provide a host-independent view of the system, which remains consistent throughout a process's and session's lifetime. Since the session's namespace is separate from the underlying namespace, it can preserve naming consistency for its processes, even if the physical namespace changes, as may be the case when sessions are migrated across computers.

Operating system virtualization is accomplished through mechanisms that translate between the session's virtual resource identifiers and the operating system resource identifiers. For every resource accessed by a process in a session, the virtualization layer associates a *virtual name* to an appropriate operating system *physical name*. When an operating system resource is created for a process in a session, the physical name returned by the system is caught, and a corresponding private virtual name created and returned to the process. Similarly, any time a process passes a

virtual name to the operating system, the virtualization layer catches and replaces it with the corresponding physical name. The key virtualization mechanisms used are system call interposition and file system isolation.

Session virtualization uses system call interposition to virtualize operating system resources, including process identifiers, keys and identifiers for IPC mechanisms, and network addresses. System call interposition wraps existing system calls to check and replace arguments that take virtual names with the corresponding physical names, before calling the original system call. Similarly, wrappers are used to capture physical name identifiers that the original system calls return, and return corresponding virtual names to the calling process running inside the session.

MobiDesk employs the `chroot` utility and stackable file systems to provide each session with its own file system namespace. A session's filesystem is composed from remote mounts via a network file system such as NFS, which guarantees that the same files can be made consistently available as a session is migrated from one computer to another. The `chroot` system call is then used to set the centrally mounted filesystem area as the root directory for the session, thereby achieving file system virtualization and isolation with negligible performance overhead. Finally, a simple stackable filesystem is used to address the fact that there are multiples ways to break out of a chrooted environment. The stacked filesystem creates a barrier which takes care of enforcing the chroot environment, and ensures that the session's file system is only accessible to processes within the given session. The barrier is implemented as a directory that prevents processes within the session from traversing it. Since the processes are not allowed to traverse the directory, they are unable to access files outside of the session's file system namespace.

MobiDesk provides the ability to maintain session availability in the presence of server downtime due to operating system and hardware upgrades. This is accom-

plished by leveraging ZAP's checkpoint-restart mechanism which enables sessions to be migrated between computers with different hardware and operating system kernels. MobiDesk is limited to migrating between machines with a common CPU architecture, and where kernel differences are limited to maintenance and security patches. Migration is limited to these instances because major version changes are allowed to break application compatibility, which may cause running processes to break.

To support migration, MobiDesk employs an intermediate format to represent the state that needs to be saved. On checkpoint, the process image is saved and digitally signed to enable the restart process to verify its integrity. Although the internal state that the kernel maintains on behalf of processes can be different across kernels, the high-level properties of the process are much less likely to change. MobiDesk captures the state of a process in terms of this higher-level semantic information rather than the kernel specific data. MobiDesk's intermediate representation format is chosen such that it offers the degree of portability needed for migrating between different kernel minor versions. If the representation of state is too high-level, the checkpoint-restart mechanism could become complicated and impose additional overhead.

MobiDesk leverages high-level native kernel services in order to transform the intermediate representation of the checkpointed image into the complete internal state required by the target kernel. This use of high-level functions helps with general portability when using MobiDesk for migration. Security patches and minor version kernel revisions commonly involve modifying the internal details of the kernel while high-level primitives remain unchanged. As such high-level functions are usually made available to kernel modules through exported kernel symbol interface, the MobiDesk system is able to perform cross-kernel migration without requiring modifications to the kernel.

5.1.3 Network Virtualization

Networking support for MobiDesk sessions must address two issues:

- Multiple sessions on the same server may run the same service, e.g. two sessions may both run the apache server, however, only one of them can listen on port 80.
- Ongoing network connections must be preserved when a session is migrated from one server to another.

When all hosting servers are in the same subnet, the two issues can be addressed relatively easily using existing technologies with minor enhancements from MobiDesk. Each session is assigned a unique IP address from a pool maintained by a DHCP server when it is first created. For example, the servers may occupy IP address range 192.168.1.2 - 192.168.1.50, and the rest of 192.168.1.5 - 192.168.1.254 may be assigned to MobiDesk sessions. The IP address assigned to a session is created as an alias of the hosting server's primary IP address. Multiple aliases, each corresponding to a different session, can be created on a server. MobiDesk privatizes the aliases such that a session only sees its own alias, and cannot interfere with traffic of other sessions on the same server.

Since each session has its own IP address, two sessions on the same server can both listen to port 80, bound to their individual private IP address. When a session is migrated from one server to another, the private IP address of the session remains unchanged; it is simply (re)created as an alias of the new hosting servers primary IP address. ARP resolves the MAC address change at the link layer and the migration is transparent to the network layer and above. Ongoing network connections of the session therefore stay intact.

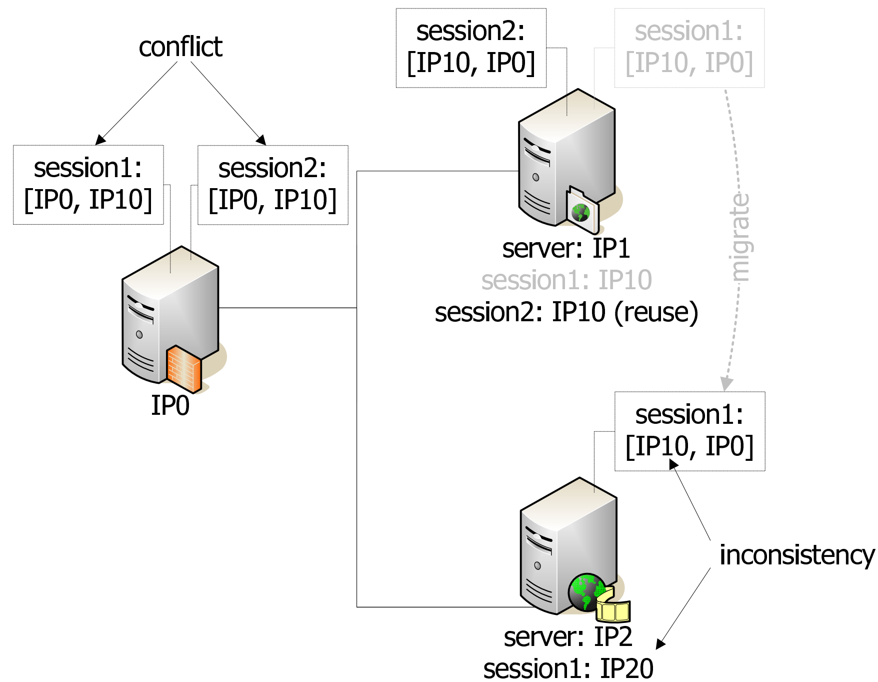


Figure 5.2 – Problems of Migrating Connections

While it is possible to have the entire private network behind the proxy to be in a single subnet, it is often desirable to have separate subnets for scalability and management reasons. In this case, when a session is migrated across subnets, its private IP address can no longer persist, since on the destination subnet the address is no longer valid. As a result, two types of problems can occur, as we illustrate in Figure 5.2. Note that we omit port numbers for simplicity.

We see that when session1 with IP10 migrates from server IP1 to IP2, its transport connection [IP10, IP0] must persist. However, its IP address IP10 cannot persist because IP2 is on a different subnet. In addition, after session1 with IP10 migrates to server IP2, another session2 may reuse IP10 on server IP1 (or another server) and create another connection [IP10, IP0]; a conflict is created since the proxy will see two identical connections [IP0, IP10]. A potential solution is to use MobileIP. However, MobileIP requires assigning each session a permanent home address that cannot be

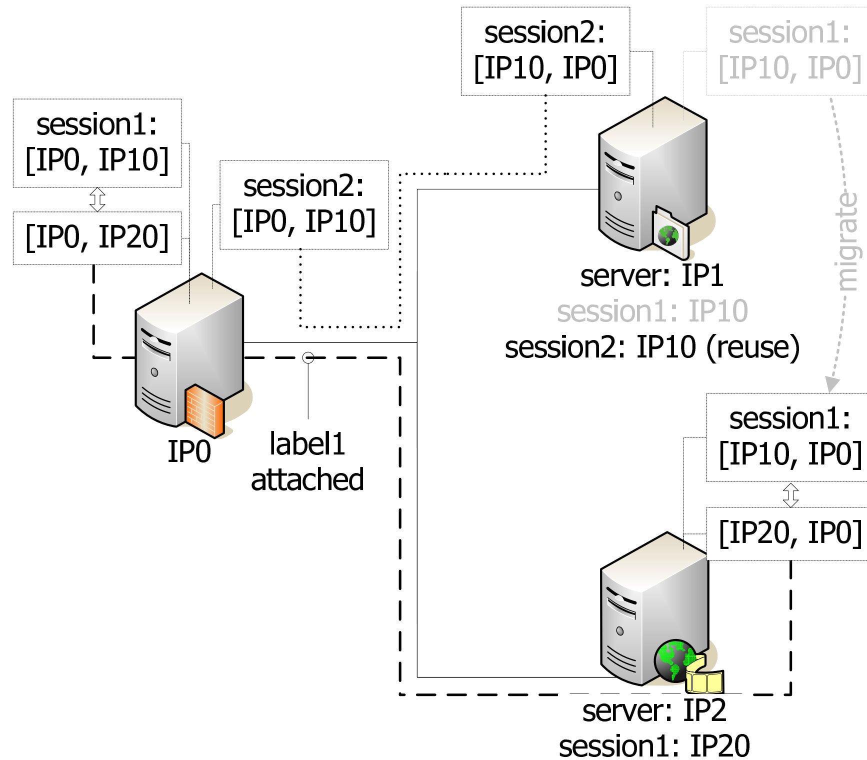


Figure 5.3 – MobiDesk Network Virtualization

reused by other sessions (to avoid conflicts as described later in this section). This is difficult since MobiDesk sessions are dynamically created, volatile entities. One can potentially adopt a solution that takes the initial physical address assigned to a session as its home address. However, this still requires additional management infrastructure to (1) assign dynamic address on a per session basis rather than per host basis, and (2) guarantee that the dynamically assigned home address is never reused by any other sessions, even after it has migrated away from its initial subnet. Using MOVE [140], MobiDesk is able to effectively address these problems without incurring additional management complexity.

To address the inconsistency problem on the MobiDesk server, MOVE associates each session with two IP addresses: a virtual address exposed to the transport layer and above and a physical address seen only at the network layer and below. The vir-

tual address stays constant for the lifetime of the session while the physical address changes whenever the session migrates, to reflect the network settings of the surrounding environment. As sessions migrate across computers, MobiDesk translates the virtual address to the current physical address (and vice versa) for all network traffic. For example, in Figure 5.3, after migration, session1's virtual address IP10 is unchanged while its physical address is assigned by the DHCP server to be IP20 and created as an alias on server IP2. The proxy translates [IP0, IP10] into [IP0, IP20] while the server IP2 translates [IP20, IP0] into [IP10, IP0]. Since the virtual address never changes, the migration is transparent to the transport and above layers, and the applications.

One potential solution to the conflict problem on the MobiDesk proxy is to require that a physical address, once assigned to a session, is never reused until the session finishes, even after the session has migrated to another subnet. However, this results in undesirable dependency of a session on a trail of addresses if it is migrated many times and new connections are opened between each migration. MobiDesk's solution is to privatize virtual addresses, i.e., to associate virtual addresses with separate private virtual network interfaces which provide a per-connection address namespace. Instead of having all connections share the same physical interface, each connection is assigned its own private virtual network interface card (VNIC). A VNIC is simply a software emulation of a NIC at the link layer that appears exactly the same as a NIC to the network and above layers. As a result, two connections using the same virtual IP address due to address reuse can peacefully coexist on the same server, since they are bound to their own private VNIC.

To support per-connection address space, MobiDesk augments the traditional connection tuple with connection labels to identify the VNIC to which a connection is bound. A connection has two labels, independently and uniquely chosen by the Mo-

biDesk proxy and the server at the time the connection is setup. The two sides also exchange their labels at connection setup time. Before a session is migrated, the labels are not used since the tuple alone is enough to identify the connections of the session. After a session is migrated, both sides will attach the peer's label learned at connection setup time for all connections between them. The labels allow the connections to be uniquely identified even when a session's previous physical address is reused.

5.2 A^{2}M: Access-Assured Mobile Desktop Computing

Although the benefits of MobiDesk are manifold, they are predicated on users being able to access the supporting server infrastructure of the respective service providers. A key issue that must be addressed to ensure that users obtain reliable access to hosted computing services is protection of the server infrastructure against denial of service attacks, particularly of the distributed kind (DDoS). DDoS attacks are an increasing occurrence in today's Internet, aiming to deny use of a service to legitimate users [30]. The same increased network connectivity that improves access to a service provider for legitimate mobile users also increases an attacker's ability to launch a DDoS against a service provider, often as part of an extortion scheme [52]. Apart from the pure annoyance factor, such an attack can prove particularly damaging for time- or life-critical services.

Of particular importance to service providers are link congestion attacks, whereby attackers identify "pinch points" in the communication substrate and render them inoperable by flooding them with large volumes of traffic. The usual attack point is

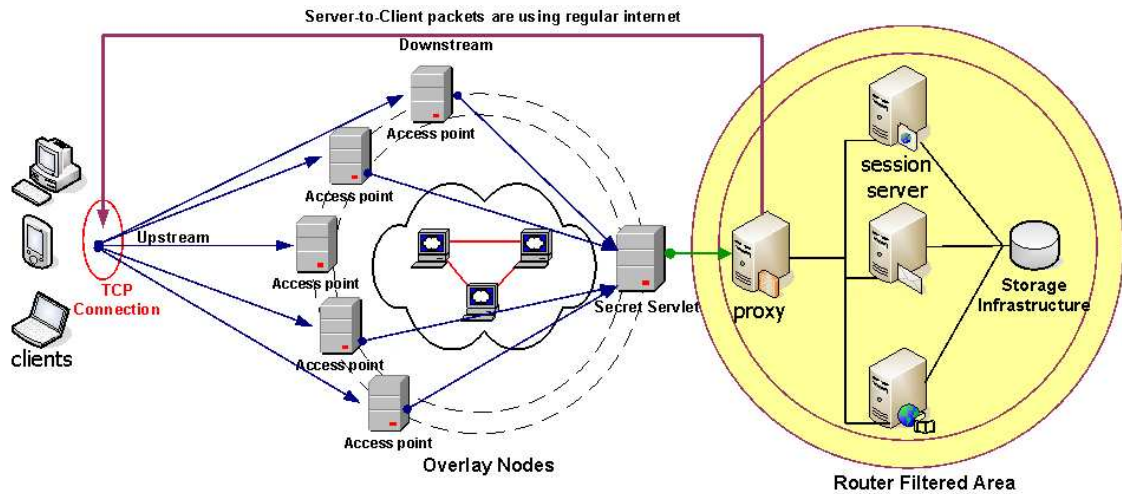


Figure 5.4 – **A^{2}M Architecture**. The two directions of the client-server connection take different paths: the client-to-server direction goes over the indirection-based network, while the server-to-client direction goes directly to the client (not through the infrastructure). Legitimate uplink traffic is spread among the IBN nodes and competes with denial of service traffic for capacity in the links close to the server; allowing legitimate traffic through the indirection system allows us to differentiate the two. Indirection nodes can simultaneously serve as system entry points and secret forwarders, and are dedicated to this task (*i.e.,* they are not end-user-controlled nodes).

the location of the hosting servers, or the routers in their immediate network vicinity. Sending enough attack traffic will cause the links close to the servers to be congested and eventually drop all useful traffic. Clearly, the potential of such an attack to disrupt user access to applications and data poses an important challenge that needs to be addressed before ASPs can achieve mass acceptance.

In this context, we introduce A^{2}M, *Access-Assured Mobile* desktop computing, a hosted computing infrastructure that combines an indirection-based overlay network with a remote display architecture, to provide guaranteed and efficient access to hosted desktop computing environments, even in the presence of denial of service attacks.

As shown in Figure 5.4, A^{2}M's architecture is divided in two major components: the hosting infrastructure and the access infrastructure. The hosting infrastructure

provides an environment for desktop sessions where a user's session is decoupled from any particular end-user access device, by moving all application state to hosting servers. Applications run within these servers, and their display output is redirected over the network to the access device. Redirection is performed by a per-session virtual display driver that translates from application display-draw commands to THINC display protocol commands. The protocol commands are then forwarded to the client device for display. A^2M extends MobiDesk by providing mechanisms that enable continuous access to hosted desktop sessions, even in the presence of distributed denial of service attacks on the hosting servers.

A^2M's access infrastructure provides the connection between users on the network and the applications running on the hosting servers. Users make use of a simple client application that merely forwards input events to the applications running on the server, and processes display updates generated in response to these events. This application model results in a highly asymmetric network traffic pattern. On one side, input events (headed uplink, or upstream toward the server) are very small pieces of information that are generated at a relatively slow, human-dependent rate. On the other hand, display updates (headed down-link, or downstream toward the client) are orders of magnitude larger and are generated as fast bursts of activity. For example, during web browsing, a single user input event (a mouse click on a link) results in a full-screen update having to be displayed (the destination web page).

The traffic asymmetry is made more pronounced when we consider the different roles and importance of input events and display updates. In an interactive system user experience is dictated by the response time, which in turn is determined by how quickly input events are processed and display updates are made visible to the user. If response time is too high, the user will become exasperated and frustrated with the system. Since a single input event triggers the generation of display updates,

guaranteed delivery of each event becomes crucial for the performance of the system. On the other hand, humans are known to be more tolerant to partial updates than to longer response times, because partial updates provide feedback to their actions. Delivery of updates should then be made such that updates can begin to be displayed as soon as possible, even if the complete update takes longer to appear.

The resource centralization around the hosting infrastructure results in a threat model where denial of service attacks on the system will only affect the up-link direction, *i.e.,* the traffic **to** the hosting servers, by saturating the network links and queuing buffers close to the servers or by directly attacking the hosting infrastructure servers. Therefore, it is crucial for A^2M to protect this communications channel from interference, blocking unwanted traffic close to the attacker before it can reach the service providing machines. On the other hand, the down-link direction will for the most part be relatively free of noise, and without any need to be protected.

Taking advantage of both the traffic asymmetry and the threat model, A^2M partitions bi-directional connections between the client and the server into an indirected client-to-server multi-path and a direct server-to-client path. The IBN takes care of routing input events and other client-to-server traffic and protects the hosting infrastructure. Protection is performed by acting as a distributed firewall that conceptually distinguishes between authorized client-generated traffic, and unauthorized and possibly malicious traffic. Traffic permitted to traverse through the IBN is directed to a filtering router close to the hosting servers, whereas all other traffic is dropped or rate-limited providing a distributed "shield" against both network congestion and host directed attacks. The direct server-client path in turn ensures that large and bursty display updates are delivered to the client as quickly as possible, even if parts of them are lost or delayed and need to be retransmitted. A^2M's approach represents a sharp departure from traditional interactive client-server architectures, where a vulnerable

bi-directional direct connection provides the only means of communication between the client and the server.

5.2.1 System Operation

To provide seamless and ubiquitous connectivity, A^2M encapsulates all functionality within a self-contained client application that manages communication with the indirection infrastructure, forwards user events to hosted applications, and displays application output on the local device. To access a desktop session, users must first obtain access to the IBN, which in turn allows them to authenticate with the hosting infrastructure, and then gain access to their session. Users need to be recognized as legitimate in order for the IBN to distinguish their traffic from other unauthorized, possibly malicious traffic. In contrast to traditional service providing infrastructures such as web-content distributors, A^2M requires users to be authenticated and does not allow anonymous users, because only authorized users should be able to connect to the hosting infrastructure. A^2M ties the authentication requirements of the IBN and the hosting infrastructure into a single, seamless process.

When a user attempts to connect to A^2M, the client application first acquires a "ticket" from one of the indirection nodes. This ticket gives it temporary access to the IBN, and allows the client to contact A^2M's authentication service to identify itself as a legitimate user. After being successfully authenticated and authorized, the client receives a longer-term session ticket from the IBN, and a connection to the A^2M server hosting the user's session. The session ticket identifies the client as a legitimate user of the system, and allows it to freely interact with the hosting infrastructure. To avoid stale sessions to be used in an attack, the session ticket needs to be renewed periodically. In the case where a session does not already exist, a new session is created

and populated, before the client is allowed to connect to it. The authentication and connection setup process is done transparently by the client application, and it does not require special support from the underlying devices. This simplicity allows A^{2}M users to access their sessions from almost any number of Internet-enabled devices.

Once the connection to the hosting server is established, the client will be recognized as a legitimate user, and user input events will be allowed to traverse the indirection nodes and be routed to the hosted applications. This process continues until the user disconnects from the session, at which point the client's ticket is revoked and the connections are closed. Since a disconnected client is no longer allowed to use the system, previously legitimate devices cannot be reused as attack tools on the infrastructure.

5.3 Experimental Results

We have developed prototypes of MobiDesk and A^{2}M, and measured their effectiveness at hosting and protecting desktop sessions. This section presents the results of our evaluation. We first evaluate MobiDesk performance, focusing on the overhead of the virtualization environment, and its ability to provide efficient remote access to hosted desktop sessions. Second, we evaluate A^{2}M performance, focusing on quality of service and its ability to protect the desktop hosting infrastructure from denial of service attacks.

We have implemented a prototype MobiDesk system for serving Linux desktop computing environments. On the server-side, our prototype consists of a virtual display driver module for the X Window System and a loadable kernel module for operating system and network virtualization. The display driver runs as part of the display system of the hosting server, and the kernel module is loaded at the

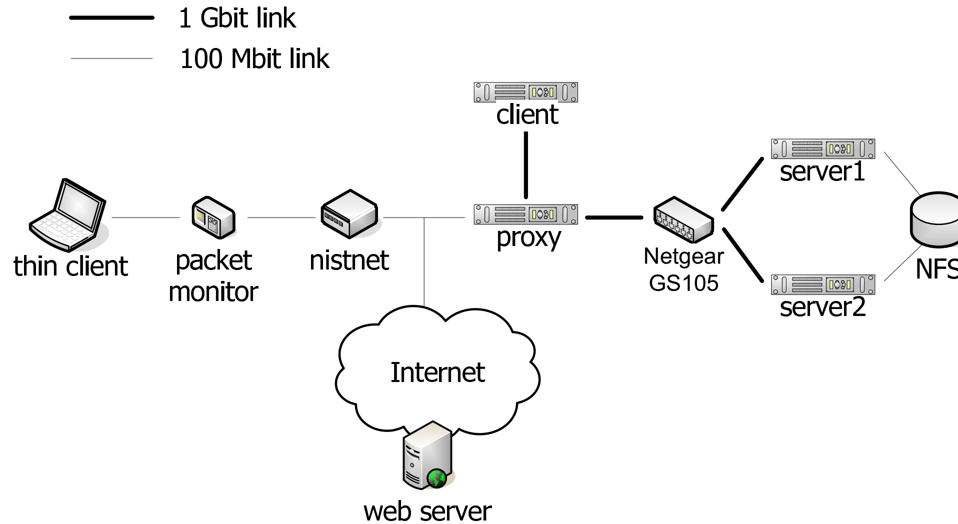


Figure 5.5 – MobiDesk Evaluation Experimental Testbed

hosting server and the proxy. The server-side of our prototype works with unmodified Linux applications and any off-the-shelf Linux 2.4 kernel. On the client-side, our prototype provides a small client application that can be downloaded and run on any unmodified client to provide MobiDesk functionality. We have implemented both Xlib and Java versions of the MobiDesk client application, which can run on both Unix/Linux and Windows clients. We present experimental results using our Linux MobiDesk prototype to quantify its overhead and demonstrate its performance on various desktop computing applications.

Figure 5.5 shows the isolated network testbed we used for our experiments. The testbed consists of eight IBM Netfinity 4500R machines and a Micron desktop PC. The Netfinity machines each had a 933Mhz Intel Pentium-III CPU and 512MB RAM, and all of them were connected via gigabit Ethernet. The Micron desktop PC had a 450Mhz Intel Pentium-II CPU and 128MB RAM, and was used as the MobiDesk client. Four of the machines served as a MobiDesk server infrastructure consisting of one NFS file server, one proxy server running a delegate 8.9.2 [27] general-purpose application level proxy, and two computing session servers. One machine was connected

on the client-side of the MobiDesk proxy and was used as a NISTNet 2.0.12 WAN emulator which could adjust the network characteristics seen by the client. Four machines were connected to the client-side of the WAN emulator, one Micron PC used as a MobiDesk client, a second used as an external web server, a third used as a packet monitor running Ethereal Network Analyzer 0.9.13 for measurement purposes, and the last used as a client for network virtualization overhead measurements. All of the machines ran Debian Linux, with the two computing session servers running Debian Stable with a Linux 2.4.5 kernel and Debian Unstable with a Linux 2.4.18 kernel, respectively. The MobiDesk client machine was installed with a dual boot configuration and also ran Microsoft Windows XP Professional.

5.3.1 MobiDesk Virtualization Overhead

To measure the cost of MobiDesk's operating system virtualization, we used a range of micro benchmarks and real application workloads and measured their performance on our prototype and a vanilla Linux system. Table 5.1 shows the seven microbenchmarks and the four application benchmarks we used to quantify MobiDesk's operating system virtualization overhead, as well as the results for a vanilla Linux system. To obtain accurate measurements, we rebooted the system between measurements. Additionally, the system call microbenchmarks directly used the TSC register to record timestamps at the significant measurement events. The average timestamp event cost was 32 ns. The files for the benchmarks were stored on the NFS server. All of these benchmarks were performed in a chrooted environment on the NFS client machine running Debian Unstable with a Linux 2.4.18 kernel. Figure 5.6 shows the results of running the benchmarks under both configurations, with the vanilla Linux configuration normalized to one. Since all benchmarks measure the time to run the

Name	**Description**	**Linux**
getpid	average `getpid` runtime	350 ns
ioctl	average runtime for the FIONREAD ioctl	427ns
shmget-shmctl	IPC Shared memory segment holding an integer is created and removed	3361 ns
semget-semctl	IPC Semaphore variable is created and removed	1370 ns
fork-exit	process forks and waits for child which calls exit immediately	44.7 us
fork-sh	process forks and waits for child to run `/bin/sh` to run a program that prints "hello world" then exits	3.89 ms
Apache	Runs Apache under load and measures average request time	1.2 ms
Make	Linux Kernel compile with up to 10 process active at one time	224.5s
MySQL	Time per interaction for "TPC-W like" benchmark	8.33s

Table 5.1 – MobiDesk Application Benchmarks

benchmark, a small number is better for all results.

The results in Figure 5.6 show that the operating system virtualization overhead is small. MobiDesk incurs less than 10% overhead for most of the microbenchmarks and less than 4% overhead for the application workloads. The overhead for the simple system call `getpid` benchmark is only 7% compared to vanilla Linux, reflecting the fact that virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup. The most expensive benchmarks for MobiDesk is `semget+semctl` which took 51% longer than vanilla Linux. The cost reflects the fact that our unoptimized MobiDesk prototype needs to allocate memory and do a number of namespace translations. The `ioctl` benchmark also has high overhead, because of the 12 separate assignments it does to protect the call against malicious root processes. This is large compared to the simple FIONREAD `ioctl` that just performs a simple dereference. However, since the `ioctl` is simple, we see that it only

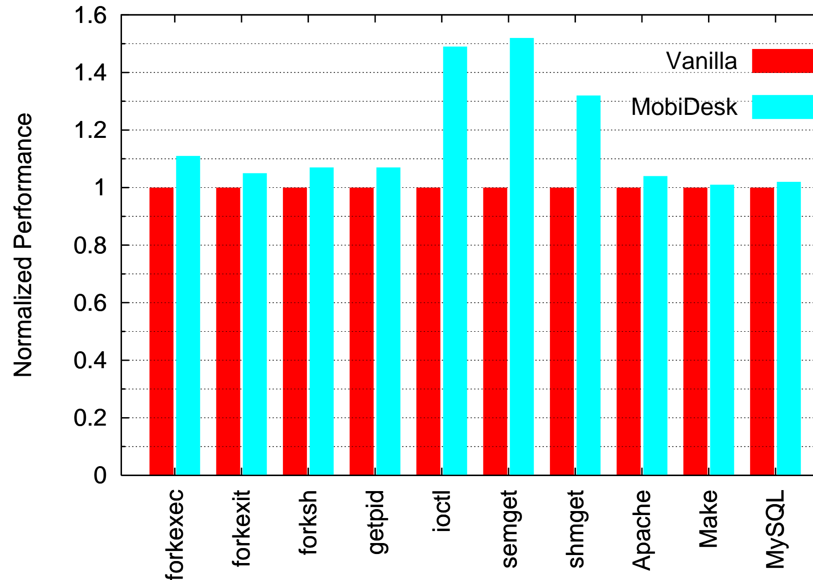


Figure 5.6 – MobiDesk Operating System Virtualization Overhead

adds 200 ns of overhead over any `ioctl`. For real applications, the most overhead was only 4% for the Apache workload, where we used the http_load benchmark [51] to place a parallel fetch load on the server with 30 clients fetching at the same time. Similarly, we tested MySQL as part of a web commerce scenario outlined by TPC-W with a bookstore servlet running on top of Tomcat with a MySQL back-end [145]. The MobiDesk overhead for this scenario was less than 2% versus vanilla Linux.

To measure the cost of MobiDesk's network virtualization, we used netperf 2.2pl4 [90] to measure MobiDesk network I/O overhead versus vanilla Linux in terms of throughput, latency, CPU utilization, and connection setup. We ran the netperf client on the Netfinity client and the netperf server on the MobiDesk session server. We used the Netfinity client for these experiments instead of the MobiDesk client so that all machines used for the network virtualization measurements were connected via gigabit Ethernet. To ensure that we were accurately measuring the performance overheads of our systems as opposed to raw network link performance, we used gigabit Ethernet

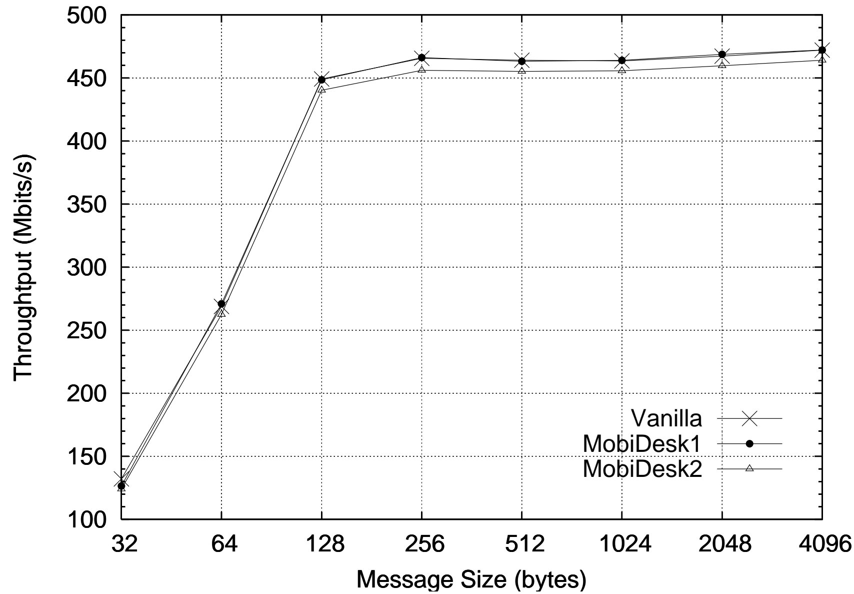


Figure 5.7 – MobiDesk Network Virtualization Throughput Overhead

for our experiments so that the network link capacity could not be saturated easily. All connections from the netperf client to the netperf server were made through the delegate proxy. We compared the performance of three different system configurations: Vanilla, MobiDesk1, and MobiDesk2. The Vanilla system is a stock Linux system without MobiDesk loaded into the kernel. The MobiDesk1 and MobiDesk2 are systems with MobiDesk loaded. On MobiDesk1, no connections are migrated and hence only connection virtualization is performed; on MobiDesk2, all connections are migrated and hence both connection virtualization and virtual-physical mapping are performed.

Figures 5.7 to 5.9 show the results for running the netperf throughput experiment, latency experiment, and connection setup experiment. CPU utilization measurements are omitted due to space constraints, but show similar overhead results. The throughput experiment simply measures the throughput achieved when sending messages of varying sizes as fast as possible from the client to the server. Figure 5.7 shows the

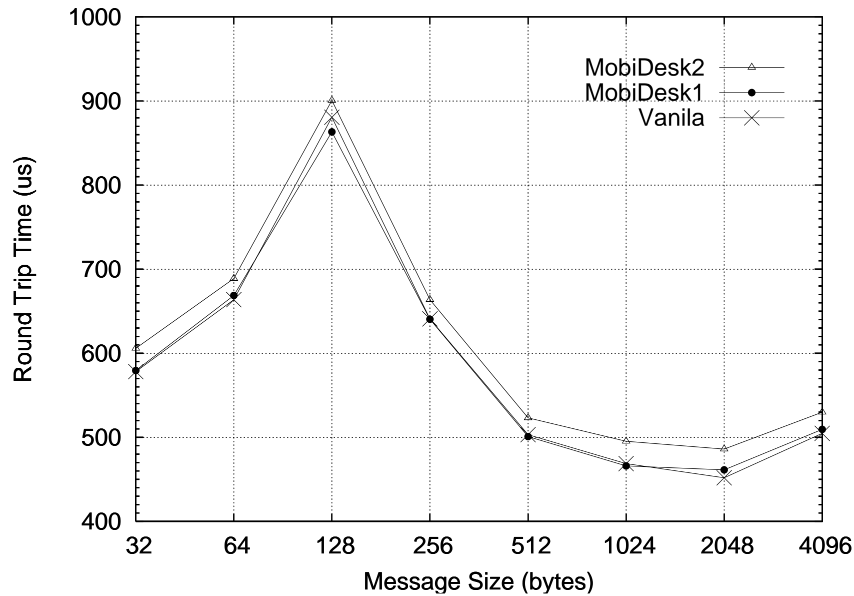


Figure 5.8 – MobiDesk Network Virtualization Latency Overhead

throughput overhead for the three systems we tested. We can see that MobiDesk1 performs very close to Vanilla, with an overhead of about 1.4Mbps. MobiDesk2 shows the throughput overhead due to the virtual-physical mapping, which is around 10Mbps.

The latency experiment measures the inverse of the transaction rate, where a transaction is the exchange of a request message of size 128 bytes and a reply message of varying sizes between the client and the server over a single connection. Figure 5.8 shows the latency overhead for the three systems we tested. The results bear the same characteristic as that for the throughput overhead. Performance difference between Vanilla and MobiDesk1 is about 9.4 microseconds, while latency due to the virtual-physical mapping in MobiDesk2 can be observed to be around 40 microseconds. Note that there is a strange drop of latency above a reply message size of 128 bytes. We determined that this unusual behavior is due to a problem with the Linux device driver for the Intel Pro/1000 network card that was used. While the behavior is unusual, it

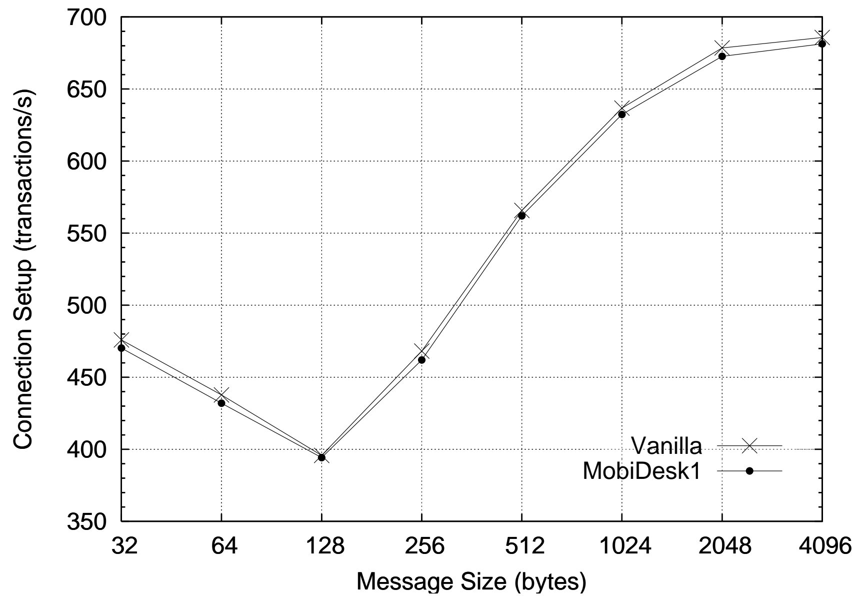


Figure 5.9 – MobiDesk TCP Connection Setup Overhead

does not affect the key result shown, which is the small relative performance difference between using vanilla Linux and MobiDesk.

The TCP connection setup experiment is the same as the latency experiment except that a new connection is used for every request/response transaction. This experiment simulates the interaction between a client and server in which many short-lived connections are opened and closed. Figure 5.9 shows the TCP connection setup overhead for Vanilla and MobiDesk1. Note that since connection setup occurs before migration, there is no virtual-physical mapping overhead associated with connection setup, therefore this measurement is not applicable to MobiDesk2. From the figure we can see that the overhead is fewer than 10 transactions per second. Due to the same Linux driver problem in the latency test, we also see a strange increase of connection rate above reply message size of 128 bytes.

5.3.2 MobiDesk Application Performance

To evaluate MobiDesk performance on real desktop applications, we conducted experiments to measure the display performance of MobiDesk for web and multimedia applications and the migration performance of MobiDesk in moving a user's desktop computing session from one server to another. To measure display performance, we compared MobiDesk against running applications on a local PC. We also compared MobiDesk running with XFree86 4.3.0 against other popular commercial thin-client systems, including Citrix MetaFrame XP for Windows [23], VNC 3.3.7 for Linux [150], and Sun's SunRay 2.0 [141]. All of the thin-client systems, except SunRay, used the Micron PC as the client and a Netfinity server as the server. Since SunRay requires Sun hardware to run, we added a SunRay I hardware thin-client and a Solaris 9 v210 server to our experimental testbed since it does not run with the common hardware/software configuration used by the other systems.

We evaluated display performance using two popular desktop application scenarios, web browsing and video playback. Web browsing performance was measured using a Mozilla 1.4 browser to run a benchmark based on the Web Text Page Load test from the Ziff-Davis iBench benchmark suite [54]. The benchmark consists of a sequence of 54 web pages containing a mix of text and graphics. The browser window was set to 1024x768 for all platforms measured. Video playback performance was measured using a video player to play a 34.75 s video clip of original size 352x240 pixels displayed at 1024x768 full screen resolution. In the Unix platforms we used MPlayer 1.0pre3 as the video player, while for the Windows platforms we used the standard Windows Media Player. We used the packet monitor in our testbed to measure benchmark performance on the thin-client systems using slowmotion benchmarking [93], which allows us to quantify system performance in a non-invasive manner by capturing net-

work traffic. The primary measure of web browsing performance was the average page download latency. The primary measure of video playback performance was video quality [93], which accounts for both playback delays and frame drops that degrade playback quality. For example, 100 percent video quality means that all video frames were displayed at real-time speed. On the other hand, 50 percent video quality could mean that half the video frames were dropped when displayed at real-time speed or that the clip took twice as long to play even though all of the video frames were displayed.

For both benchmarks, we measured all systems in three representative network scenarios: LAN, with an available network bandwidth of 100 Mbps and no introduced network latency (100Mb-0ms), and two WAN scenarios, one with 100 Mbps available network bandwidth and 66 ms round-trip network latency (100Mb-66ms), representative of cross-country and transatlantic latencies [44], and another with 100 Mbps available network bandwidth and 120 ms round-trip network latency (100Mb-120ms), representative of typical transpacific latencies [44]. For the WAN tests we increased the default TCP window size for both server and client. SunRay was unaffected by this since it uses UDP.

Figure 5.10 shows the web browsing performance results in terms of the perceived latency. Figure 5.11 shows the video playback performance results in terms of video quality. As expected, both of these results are in line with the ones presented in Section 2.7, since MobiDesk leverages THINC's display virtualization and remote display architecture, even though the deployment scenario is different. This similarity also demonstrates that operating system and network virtualization overhead do not negatively impact application performance.

Figure 5.10 shows that MobiDesk has the smallest web page download latencies, thus providing the best overall performance. The worst web browsing platform is

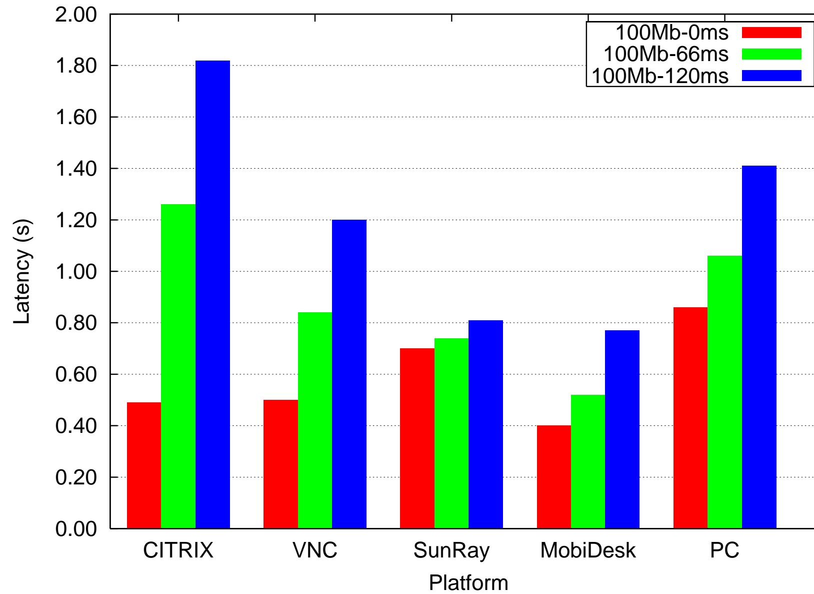


Figure 5.10 – MobiDesk Average Per Web Page latency

Citrix MetaFrame, which adopts a more high-level display approach that results in poor WAN performance because of the tight coupling required between the application running on the server and the Citrix viewer running on the client. VNC has the second worst WAN web browsing performance in part because it relies on a client pull model for sending display updates as opposed to MobiDesk's server-push model, which avoids roundtrip latencies providing better interactive response time. In addition, as a response to the limited WAN network conditions, VNC adaptively uses more efficient compression algorithms, thus reducing its data transfer, but increasing its latency, and worsening its overall web browsing performance.

Our web browsing experiments under WAN conditions show that increased network latency can result in increased web page latencies when using TCP-based thin-client systems. This is due to the fact that TCP implementations reduce the congestion window by half for every roundtrip time that a connection has been idle [49]. As our benchmark mimics traditional web browsing usage by adding delays between

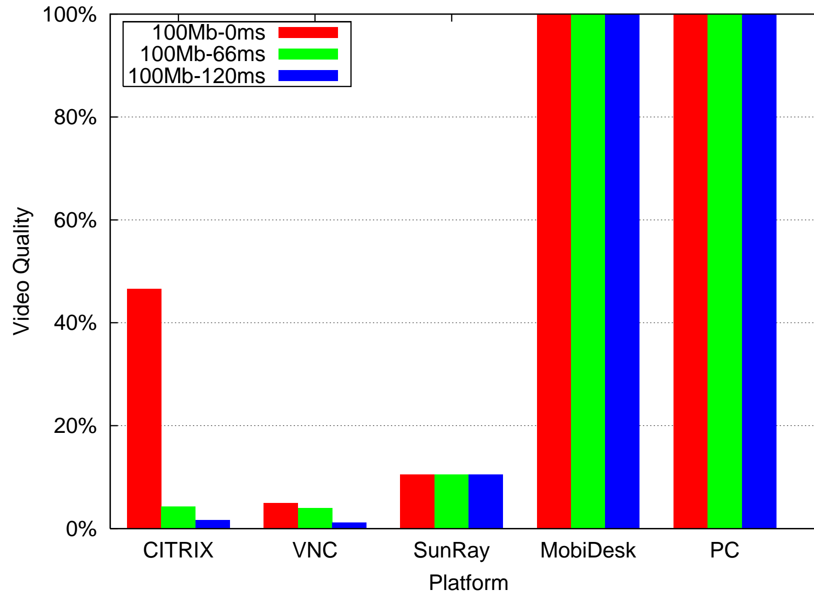


Figure 5.11 – MobiDesk Video Quality

the display of each web page, the thin-client connection ends up going idle and goes through slow-start each time a new page starts downloading. As network latency increases, the TCP connection takes longer to recover from its idle state, thus increasing the time it takes for web pages to load. Like other TCP-based thin-client systems, MobiDesk has higher web page latencies for the web benchmark in the presence of transpacific network latencies. However, Figure 5.10 shows that MobiDesk continues to provide superior sub-second performance over existing systems, even for high latency network connections.

Figure 5.11 shows that MobiDesk provides perfect video quality in the same manner as the traditional desktop PC, and that all of the other platforms deliver very poor video quality. They suffer from an inability to distinguish video data from normal display updates and apply ineffective compression algorithms on the video data, which are unable to keep up with the stream of updates being generated. In contrast, the results show that MobiDesk's ability to leverage local client video hardware

Application	Description
MobiDesk	Remote display server
KDE	Entire KDE 2.2.2 environment, including window manager, panel and utilities
SSH	openssh 3.4p1 client inside a KDE konsole terminal connected to a remote host
Shell	The Bash 2.05a shell running in a konsole terminal
KGhostView	A PDF viewer with a 450k 16 page PDF file loaded
Konqueror	A modern standards compliant web browser that is part of KDE
KOffice	The KDE word processor and spreadsheet programs

Table 5.2 – MobiDesk Migrated KDE Desktop Computing Session

in delivering video using alternative YUV formats provides substantial performance benefits over other thin-client systems. VNC provides the worst overall performance primarily because of its use of a client pull model instead of MobiDesk's server push model. In order to display each video frame, the VNC client needs to send an update request to the server. Clearly, video frames are generated faster than the rate at which the client can send requests to the server.

To measure real application performance in terms of the cost of migration, we migrated a complete KDE [60] desktop computing environment from one MobiDesk server to another. The applications running in the KDE computing session when it was migrated are described in Table 5.2. The KDE session had over 30 different processes running, providing the desktop applications as well as substantial window system infrastructure, in particular, a framework for inter-application sharing. The session also included a rich desktop interface managed by a window manager, and a number of applications running in a panel, such as a clock. To demonstrate the ability to migrate a complete computing session across Linux kernels with different

minor versions, we checkpointed the KDE session on the 2.4.5 kernel client machine and restarted it on the 2.4.18 kernel machine. For this experiment, the workloads were checkpointed to and restarted from local disk. The resulting checkpoint and restart times were less than a second, .85 s and .94 s, respectively. The checkpointed image was only 35 MB for a full desktop computing session, which can be easily compressed using bzip2 down to 8.8 MB. Our results show that MobiDesk can be used to provide fast migration of computing sessions among MobiDesk servers with modest checkpoint state.

5.3.3 A^{2}M Performance Evaluation

To evaluate A^{2}M, we focused on two metrics: the quality of service in terms of latency, as this is perceived by the end user, and the system's resilience when under attack *i.e.,* node failures. We deployed indirection nodes of our prototype across PlanetLab nodes, while having the access client and hosting server reside in our local network. Our architecture spreads all packets across all indirection nodes. PlanetLab provides a realistic network environment for our experiments that stresses the performance of our system because the packets follow different, highly variant paths to reach the protected server. In our experiments, we protected the uplink traffic from the client to the server routing it through the IBN, while the return path followed normal Internet routing (outside the IBN).

Our testbed consisted of a client PC simulating a typical remote-display access device, a laptop used as wireless access device, a server where the benchmark applications executed, and 80 indirection hosts deployed across various PlanetLab locations in the US and Canada. The client computer had a 450Mhz Intel Pentium-II CPU and 128MB RAM running Debian with Linux 2.4.27. Our client PC was chosen to

reflect the type of low-power, thin-client devices which we expect to become A^{2}M's access devices. The laptop PC had a 1.5Ghz Intel Pentium M and 1GB RAM running Debian with Linux 2.6.10. The server was an Intel dual-Xeon 2.80GHz with 1GB of RAM running RedHat 9 with Linux 2.4.20.

We measured the performance of A^{2}M in web, video, and basic interactive tasks as representative applications of typical desktop usage. Our web measurements used the Mozilla 1.6 browser to run a benchmark based on the Web Page Load test from the Ziff-Davis i-Bench benchmark suite. The benchmark consists of a sequence of 54 web pages containing a mix of text and graphics. The browser window was set to full-screen resolution for all platforms measured. Video playback performance was measured using Mplayer 1.0pre3 to play a 34.75 second video clip of original size $352x240$ pixels displayed at full-screen resolution. For our interactive tests we recorded a number of sessions where simple interactive tasks were performed. Recording the sessions allowed us to reliably play back the exact same tasks under different network conditions. The measure of performance for these tests was the latency experienced by a user performing the specific task. The primary measure of web browsing performance was the average page-download latency in response to a mouse-click on a web page link. To minimize any additional overhead from the retrieval of web pages, we used a conservative setup where the web server was directly connected to the hosting server through a LAN connection. The primary measure of video playback performance was video quality [93], which accounts for both playback delays and frame drops that degrade playback quality, as described in our experimental evaluation of THINC's support for multimedia applications in Chapter 3.

5.3.3.1 Overall Performance

We first examined the effects that the basic indirection network and various levels of packet replication had on the overall performance of the system. The levels of replication tested were no replication, 50% (meaning one extra copy of each packet with probability 0.5), 100% replication (one extra copy of each packet) and 200% replication (two extra copies of each packet). We also measured the impact of the IBN size by running our experiments on 8 and 80 nodes participating in the IBN. We ran a baseline test where we used a direct LAN connection between the client and the server. Since the indirection nodes were deployed over a wide area with varying network latency, this test provided us with a very conservative measurement of the indirection overhead. In a realistic A^2M deployment, the client and server will typically reside at different, topologically distant locations. In that case, it is entirely possible for the indirection path to provide better connectivity characteristics than a direct connection due to the multi-path effect, which allows the packets originating from the client to follow a route with lower latency towards the end server [4, 8, 48, 139]. Although not shown in our results for ease of viewing, we also compared the performance of A^2M to that of MobiDesk and found it to be the same on the direct connection case.

Figure 5.12 illustrates the end-to-end average web latency results as perceived by the client. We can see that even for the worst-case scenario, an 80-node IBN without packet replication, the overhead from the indirection results in a latency increase of only 2 (*i.e.*, twice the latency of the baseline direct connection). When 50% packet replication is used (*i.e.*, replicating a packet with probability 0.5), the overhead drops significantly to 40% for the 80-node IBN. The drop in the overhead is due to the variant path latency of nodes participating in the IBN. TCP does not behave optimally when packets appear to have high variance when arriving at the

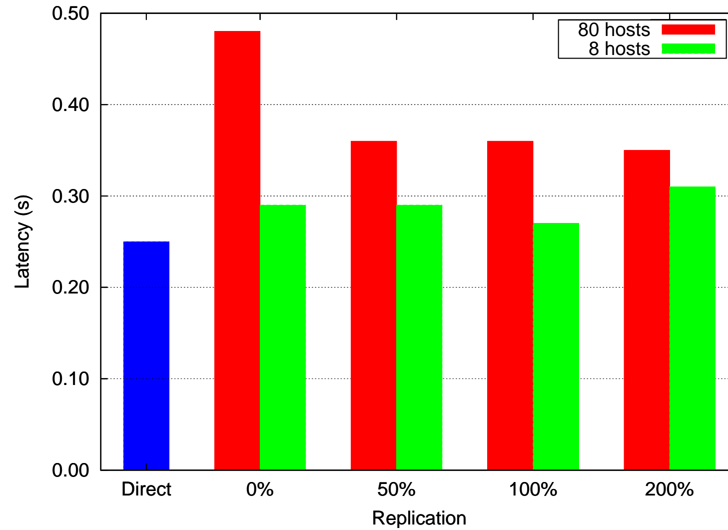


Figure 5.12 – A^{2}M Web latency *vs.* packet replication when measured close to the client and for 8 and 80 nodes participating in the IBN. The direct bar shows the latency when we fetch the page directly from the server locally using a LAN and without protection. The latency overhead drops to 40% at 50% packet replication (*i.e.,* duplicating a packet with probability 0.5).

end server out of order. Adding packet replication decreases this variance, as the same packet follows different paths, each with different latency and the end server uses the one that arrives first. Boosting the replication beyond 50% follows the law of diminishing returns, as each additional increase in replication gives us less latency improvements. Care must be taken however, as too much packet replication can cause performance degradation, since bandwidth is "wasted" on duplicate packets. This is better exemplified by the results on the 8-node network using 200% replication. The 80-node network does not exhibit the same adverse affect because its average path latency is higher, allowing the secret gateway enough time to process the encapsulated packets received by the IBN. Moreover, for the 8-node network, we amplified this effect by lowering the average latency, using PlanetLab nodes that were "close" to the protected server.

To measure our system with an application that could generate more upstream

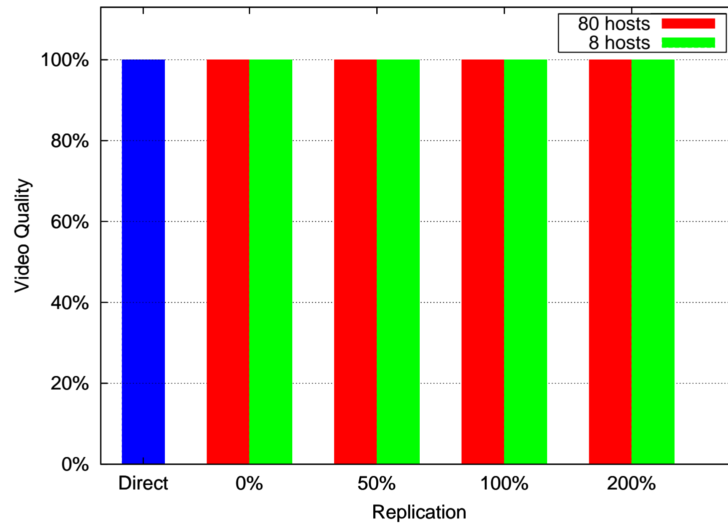


Figure 5.13 – Video quality *vs.* packet replication. Video quality remains 100% under all test scenarios even for a 80-node IBN with no packet replication, despite the use of indirection.

traffic and required the system to maintain its quality of service above a threshold for latency, we used video playback. Figure 5.13 shows the results for video quality as measured at the client side. We can clearly see that A^{2}M performs optimally under all test scenarios, providing the same perfect video quality as the direct LAN connection scenario, even for the worst-case scenario of the 80-node IBN deployed over a WAN with no packet replication.

The average per-page data transfer during the web benchmark in both directions for various packet replication settings is shown in Figure 5.14. The results demonstrate that since the upstream channel carries only input events (in this particular case, mouse clicks) and data ACKs, packet replication has very low overhead (~2.3KB to ~8.5KB) even for large pages. Similarly, Figure 5.15 shows the amount of data transferred during video playback. Although the upstream data size is significantly larger when compared to the web benchmark (from around 900KB to 2.2MB for 200% replication), it is still only a fraction of the traffic generated in the downstream chan-

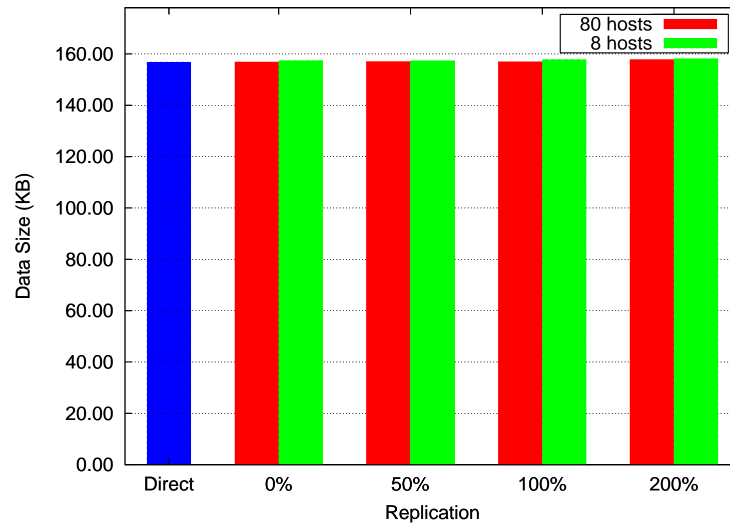


Figure 5.14 – Average per-page data transfer *vs.* packet replication for an 8-node and an 80-node IBN. Notice that the data replication does not show up in the graph, since the upstream link is only used to send input events, which are a small fraction of the total data transmitted.

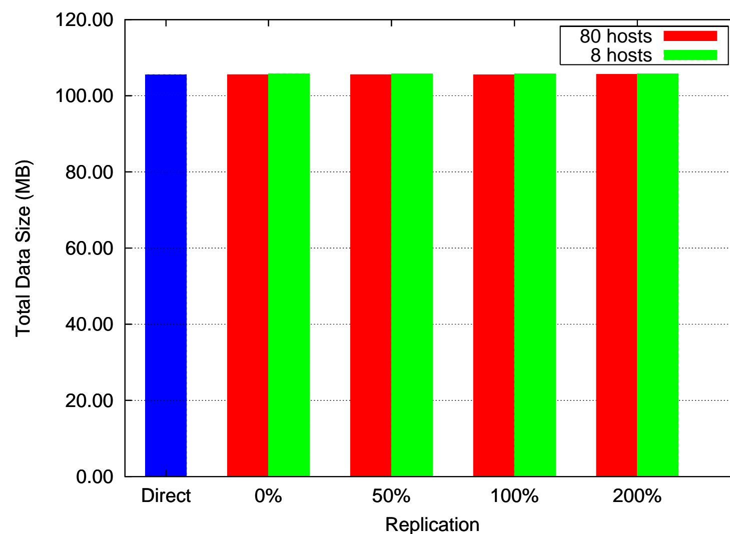


Figure 5.15 – Total video data transmitted *vs.* packet replication for 8 and 80-node testbeds. The upstream is only used for TCP ACKs, which are a tiny portion of the actual downstream video data.

nel, and confirms the large asymmetry of remote display traffic, and the low overhead of A^2M's access infrastructure.

To examine the behavior of the overall system when under attack, we measured its resilience to a simulated denial of service attack that targeted the IBN itself. Our threat model assumes that the attacker can render a fraction of the nodes participating in the IBN unresponsive, thus inducing packet loss in the TCP connection of a user connected to the hosting server. All resilience tests were run on the 80-node IBN network. When attacked, a node stops forwarding packets from the client to the end host, acting as a mute node. Since there is no immediate feedback, clients do not know which A^2M nodes are operating and which are suppressed by the attacker. Figure 5.16 illustrates the effects on the average web page latency as we increase the percentage of node failure, and demonstrates both the resilience of A^2M and the advantages of packet replication. Without packet replication, latency quickly degrades to twice that of the direct connection when we have 15% of node failures, and reaches three times for 20% node failure. On the other hand, employing packet replication allows A^2M to maintain an almost constant latency that is very close to the direct connection, even under 50% A^2M node failure, in the case of 200% replication. These results are reinforced when we consider the video playback measurements. Figure 5.17 demonstrates that excellent video quality can be maintained even after a substantial percentage of nodes become unresponsive. As we increase packet replication, the threshold can be drastically increased, to the point that A^2M is able to provide perfect video playback for up to 30% node failure, and very good (80%) video quality with 50% node failure.

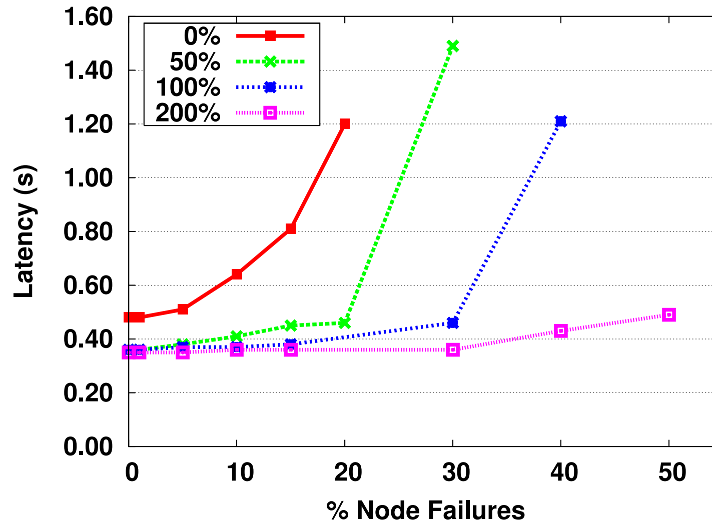


Figure 5.16 – Web latency under DDoS attack. Latency increases in response to increased nodes failure. Allowing packet replication, higher resilience is achieved, while maintaining almost constant latency even in the presence of large node failures.

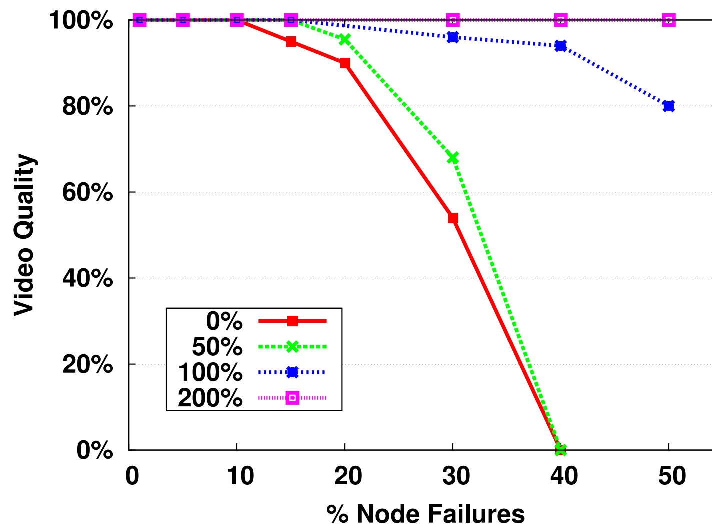


Figure 5.17 – Video quality under DDoS attack. Video quality drops only after a substantial percentage of nodes become unresponsive. At 200% replication, latency does not increase even with 50% node failures.

5.3.3.2 Interactive Applications

Although video streaming and web browsing are both representative and demanding applications, we felt that we needed to include another set of experiments that require a high level of synchronization between the upstream and the downstream channel. We performed four different tests, each representing typical interactive operations on a desktop environment. The tests were performed by first recording a session of a user performing the appropriate operation, and then playing back the session in a number of different experimental scenarios. Our measure of performance was the user-perceived latency in response to the interactive operations. The four tests performed were: echo, minimize/maximize window, scroll, and move window. The echo test measured the time it takes for a line of text to appear on the screen after the user has pressed and depressed a key. The minimize/maximize window tests measures the time it takes to maximize a window after the user has pressed the maximize button, and then (after the window has been maximized) to minimize it after the user has pressed the minimize button. The scroll test measures the time it takes to scroll down a full-screen web page in response to a single `Page Down` key-press, and then the time it takes to scroll back to the top by leaving the `Arrow Up` key pressed. Finally, the move window test measures the time it takes to move a window across the screen. The window's size is about one fifth of the screen's size, and it is moved by dragging the window while the left-mouse button is pressed. The window operation is opaque, *i.e.,* the contents of the window are continuously redrawn as the user performs the move operation.

The end-to-end latency the end users experience for these operations is shown in Figures 5.18 to 5.21. These measurements show that without using packet replication, and for attacks up to 20% of the indirection nodes, the client's end-to-end latency

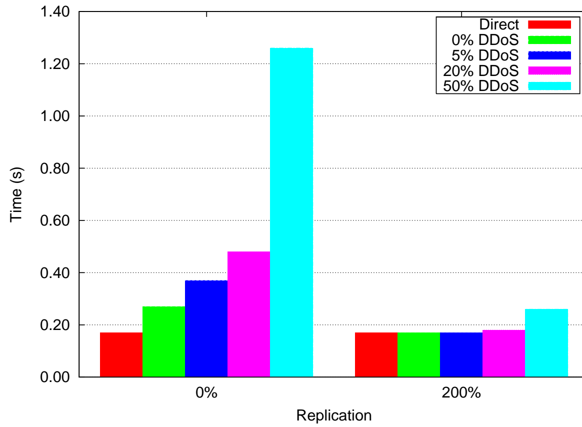


Figure 5.18 – Interactive performance for the echo test. Even without replication and with attacks affecting up to 20% of the IBN nodes, the client's end-to-end latency increases only by a factor of 2.5 when compared to the direct, non-protected case. With packet replication, latency rises only after 50% of the nodes become unresponsive.

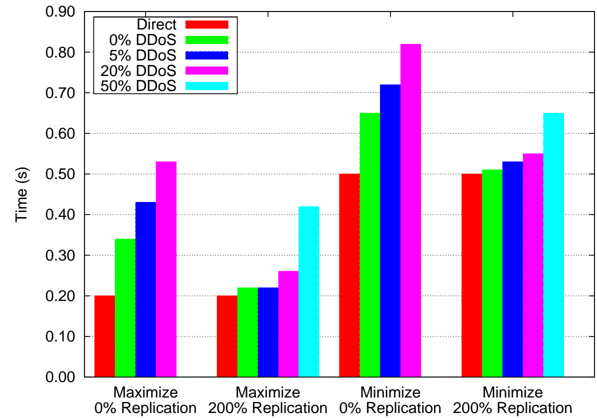


Figure 5.19 – Interactive performance for minimize/maximize window test. Without replication and for attacks affecting up to 20% of the IBN nodes, the client's end-to-end latency increases only by a factor of 2. (Over 20%, the tests could not complete.) With replication, attacks on up to 50% of the IBN nodes had no impact on latency.

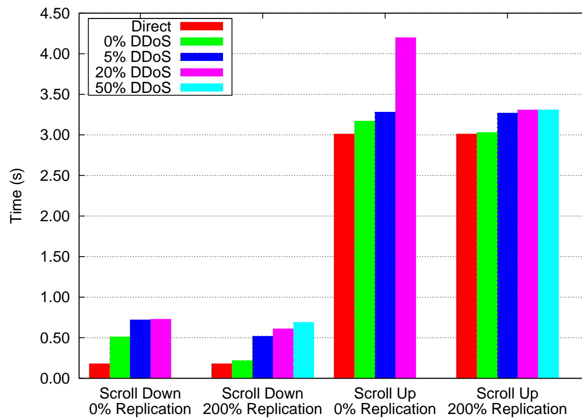


Figure 5.20 – Interactive performance for the scroll test. With packet replication, latency is close to a direct connection even when under severe attack. Without packet replication, latency increases by less than a factor of 3, *vs.* the direct case.

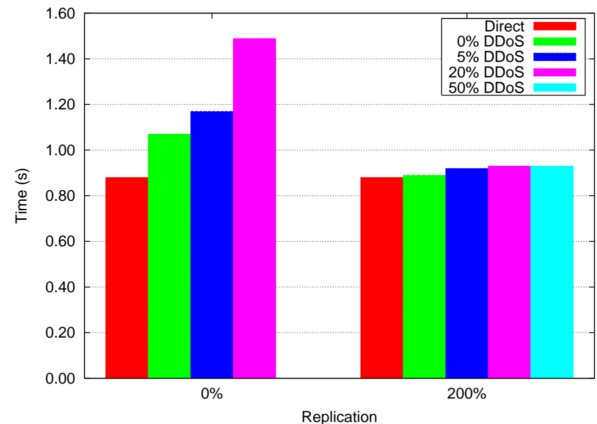


Figure 5.21 – Interactive performance for the move window test. Latency increases significantly only after 20% of IBN nodes are attacked, with no replication. With 200% packet replication, latency does not increase even for attack intensities of 50%.

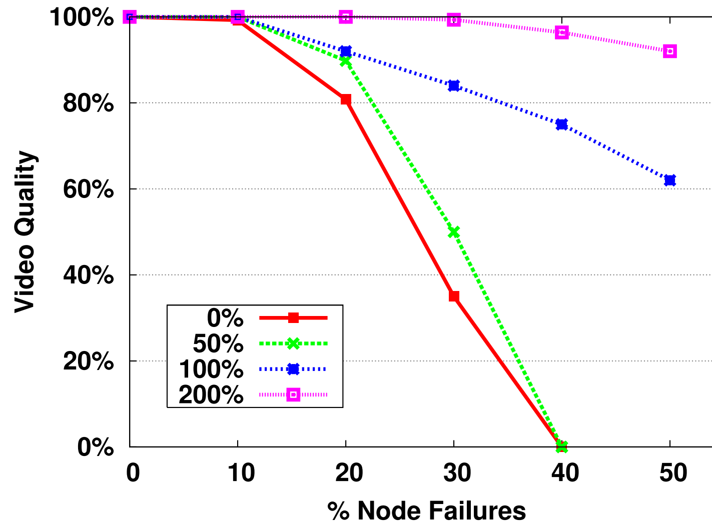


Figure 5.22 – Video quality under DDoS attack in the wireless scenario. Video quality suffers only after a large portion of the indirection nodes are attacked, allowing correct system operation even when 40% of the nodes have been attacked, for 200% packet replication.

increases only by a factor of 2.5 when compared to the direct, non-protected case. On the other hand, if we permit packet replication, we notice an increase in latency only after 50% of the indirection nodes become unresponsive. In some cases, for attack intensities that exceeded 20% of the indirection nodes and without replication the network conditions were too adverse for the test to complete.

5.3.3.3 Wireless

Our next step was to replicate some of the experiments we had for the wired network using a mobile client with a wireless connection. With the wireless tests we want to explore the performance of both traditional laptop computers, and more common mobile access devices, such as PDAs. To this end, we restricted ourselves to using an 802.11b network (as opposed to a fast 802.11a or 802.11g). Furthermore, to realistically show the performance on an expected A^{2}M deployment, we conducted all the tests over our university's public wireless network. Since wireless connections

introduce an additional source of error to our system, we chose to show experiments using video playback, because this type of application stresses the bandwidth and latency requirements of the overall system. In order to simulate a PDA connection, we reduced the client's window size to a resolution of $320x240$ pixels. Similar to our wired tests, we obtained a baseline case where we played back the video using only the wireless connection. Once again, A^2M's baseline case performance is the same as MobiDesk's. We then added the Planetlab-based indirection network to carry the upstream packets from the wireless client back to the server, while continuing to use the direct path for the downstream traffic. Figure 5.22 shows video quality as a function of node failure in the indirection network and for different percentages of packet replication, while using a wireless network. We see the same trend as in our wired-network attack scenario: video quality suffers only after a large portion of the IBN nodes are attacked, reaching 40% node failure for 200% replication.

All the previous results demonstrate that A^2M can significantly increase the attack resilience of the desktop hosting infrastructure with minimum overhead, providing web browsing latency comparable to traditional (and vulnerable) direct LAN connections, and perfect video quality even in the presence of severe attacks on the access infrastructure.

5.4 Summary

This chapter has shown how THINC can be used as a key component of desktop virtualization. We have focused our attention on how THINC can be used to deploy centralized desktop hosting infrastructures, which can provide efficient and seamless remote desktop computing services. Furthermore, we have shown how THINC's unique characteristics enable these type of services to be protected from distributed

denial of service attacks, without severely affecting the performance of hosted desktop sessions.

First, we introduced MobiDesk [14], an architecture for centralized hosting of desktop computing sessions. MobiDesk hosts computing sessions within virtualized private environments by abstracting three key resources: display, using THINC, operating system, using ZAP [101], and network, using MOVE [140]. Display virtualization allows MobiDesk to provide fast remote access to sessions across LAN and WAN environments. Operating system virtualization allows MobiDesk to migrate sessions among hosting servers to provide high-availability computing in the presence of server maintenance and upgrades. Network virtualization allows MobiDesk to transparently maintain persistent connections to unmodified outside hosts, even as a session migrates from one server to another.

We have implemented and evaluated the performance of a MobiDesk prototype in Linux. Our implementation demonstrates that MobiDesk can support unmodified applications in hosted computing sessions without any changes to operating system kernels, network infrastructure, or network protocols. Our experimental results with real applications and hosted desktop computing sessions show that MobiDesk has low virtualization overhead, can migrate computing sessions with subsecond checkpoint/restart times, and provides superior display performance over other remote display systems. MobiDesk is unique in its ability to offer a complete desktop experience remotely with full-motion video support. Given its performance and centralized hosting model, MobiDesk provides the foundation for a utility computing infrastructure that can dramatically reduce the management complexity and costs of desktop computing.

Second, we introduced A^{2}M, an attack-resilient and latency efficient mechanism for protecting MobiDesk-type infrastructures from distributed denial of service attacks

(DDoS). A^{2}M exploits multi-path routing, packet replication, and the high asymmetry inherent to interactive display traffic, to assure access to remote desktop sessions, even in the presence of high-volume DoS attacks. In a departure from traditional client-server systems, A^{2}M provides an asymmetric client-server connection consisting of an indirected client-to-server multi-path, and a direct server-to-client connection. A^{2}M's indirection-based overlay acts both as a first-level distributed firewall and as a routing mechanism for performance-critical user input-events going from the client device to the hosting servers. In turn, the direct server-to-client connection provides quick delivery of display updates, to guarantee quick response time and good user experience.

We have implemented an A^{2}M prototype in Linux and evaluated its performance on PlanetLab. Our experimental results show that, as opposed to existing DDoS protection mechanisms, A^{2}M has minimum latency overhead and can provide good interactive performance for web, video, and general interactive applications. Furthermore, our experimental results show that A^{2}M significantly increases the attack resilience of MobiDesk-type hosting infrastructures, being able to provide perfect video playback and low-latency web browsing and GUI interactions even in the presence of large attacks on the infrastructure. A^{2}M maintains 100% video quality in a number of remote video display scenarios, despite the use of overlay routing. Furthermore, end-to-end latency increases by less than 5% even when 40% of nodes have been rendered unusable by an attacker. Given its performance and resilience to DoS attacks, A^{2}M represents a step forward towards realizing the vision of computer utilities that provide ubiquitous, secure, and assured-access desktop computing.

Chapter 6

Display Recording and Text Capture

Continuing improvements in processing, storage, and network technologies have resulted in an exponential increase in the amount of data users come in contact with everyday. Keeping track of this massive amount of data, and being able to access it when it is not explicitly saved (either as a file, bookmark, or note) is proving to be a major challenge. It is not uncommon for users to realize the importance of something they saw earlier as part of their computer use, and not be able to find it or gain access to it.

Recognizing the importance of providing solutions for this growing problem, we have extended THINC's remote display architecture to provide seamless recording of all display output, and capture of all text displayed on the screen. Display recording provides a continuous log of all visual information users have had access to through their computers, while captured text provides an indexing mechanism to the log. The combined recording can be played back, and arbitrarily browsed. Furthermore, text searches can be performed as a novel mechanism to gain access to the recorded

content.

This section describes THINC's display recording and text capture architecture, as well as the mechanisms available to access the recorded information. We also discuss THINC's integration into DejaView, a personal virtual recorder for desktop computers. Finally, we present experimental results evaluating the performance of the system, and demonstrating its effectiveness.

6.1 Display Recording

THINC's virtual display architecture enables visual output to be redirected anywhere, making recording and playback simple. In particular, THINC allows visual output to be shared so that multiple clients can view the same display and collaborate using screen sharing. THINC takes advantage of this mechanism to record and display simultaneously. As visual output is generated, the virtual display driver multiplexes the output into commands for display by the viewer, and commands for logging to persistent storage. The set of display protocol commands used for both scenarios is the same, enabling both efficient storage and quick playback. Since display records are just display commands, the display record can be easily replayed either locally or over the network using a simple application similar to the normal client.

The virtual display architecture allows THINC to easily adjust the recording quality in terms of both the resolution and frequency of display updates without affecting the output to the user. THINC uses the screen scaling functionality described in Chapter 4 to allow the display to be resized to accommodate a wide range of resolutions. For example, the display can be resized to fit the screen of a PDA even though the original resolution is that of a full desktop screen. The recorded commands are resized independently, so a user can have the recorder save display output

at full screen resolution even if it is currently viewing at a reduced resolution to accommodate a smaller access device. The user can then go back and view the display record at full resolution to see detailed information that may not have been visible when viewed on the smaller device. Similarly, a user can reduce the resolution of the display commands being recorded to reduce its storage requirements. It can also limit the frequency at which updates are recorded, for example to only 30 times a second, by taking advantage of THINC's ability to queue and merge display commands so that only the result of the last update is logged.

THINC records display output as an append-only log of commands, where recorded commands specify a particular operation to be performed on the current contents of the screen. THINC also periodically saves full screenshots of the display for the following two reasons. First, it needs a screenshot to provide the initial state of the display which subsequent recorded commands modify. Second, if a user wants to display a particular point in the time line, THINC can start with the closest prior screenshot and only replay a limited number of commands. THINC records display output in a manner similar to an MPEG movie where screenshots represent self-contained independent frames from which playback can start, and commands in the log represent dependent frames which encode a change relative to the current state of the display. Since screenshots consume significant more space, and they are only required as a starting point for playback, THINC only takes screenshots at long intervals (e.g. every 10 minutes) and only if enough of the screen has changed since the last one.

By using display protocol commands for recording, THINC ensures that only those parts of the screen that change are recorded, thus ensuring that the amount of display state recorded only scales with the amount of display activity. If the screen does not change, no display commands are generated and nothing is recorded. The virtual

display driver knows not only which parts change, but also how they are changed. For example, if the desktop background is filled with a solid color, THINC can efficiently represent this in the record as a simple `solid fill` command. In contrast, regularly taking snapshots of the full screen would waste significant processing and storage resources as even the smallest of changes, such as the clock moving to the next second, would trigger a new screenshot. It could be argued that the screenshots could be compressed on the fly using a standard video codec, which could convert a series of full screenshots into a series of smaller differential changes. However, this additional computation significantly increases the overhead of the system and may not provide a desirable tradeoff between storage and display quality for the synthetic content of desktop screens. In contrast, THINC's approach knows a priori what has changed, what needs to be saved, and the best representation to use when saving it.

THINC uses three types of files to store the recorded display output: *timeline*, *screenshot*, and *command*. All three types of files are written to in an append-only manner, ensuring that the records are always ordered by time. This organization speeds up both recording and playback. While recording, THINC does not incur any seeking overhead. During playback, binary search can be used on the index file to quickly locate the records of interest.

A *timeline* file contain all the meta information required for playback. This file is a collection of tuples of the form `[time, screenshot, command]` where each tuple represents a point in the timeline where a screenshot was taken, and can be used to start playback. The command component represents the next command that was recorded after the screenshot was taken. Both `screenshot` and `command` are tuples of the form `[filename, file_position]`, and represent pointers to where the actual data for the screenshot and command is stored: the filename of the appropriate file, and the offset within that file where the information is stored.

Screenshot files hold the actual screenshot data. They are organized as a contiguous set of records, where each record is a tuple of the form `[type, time, size, dimensions, data]`. `type` specifies whether the record is a screenshot or a reference to another screenshot file. `time` specifies the time at which the screenshot was recorded. `size` specifies the data size of the screenshot. `dimensions` specifies the dimensions of the screenshot, to allow for changes of the geometry of the display to be appropriately recorded. `data` is the actual screenshot data.

Command files contain the stream of display commands. In the same manner as screenshot files, each command is stored as a serialized record of the form `[type, time, size, data]`. `type` specifies the type of THINC display command. `time` specifies the time at which the command was recorded. `size` specifies the data size of the command. `data` is the actual command data.

THINC allows for multiple screenshot and command files to be used if needed or desired, for example for systems with maximum file sizes which could be exceeded by long-running desktop recording sessions. At the end of each file, a special record is appended that points to the next file on the stream. The record has the same format as other records. It uses the `type` field to mark itself as an end-of-file/next-file marker, and the `data` component to store the next filename. As playback occurs, this record is read just like any other record, but causes the playback program to start reading from the next file and continue its operation.

6.2 Text Capture

In addition to visual output, THINC records contextual information by capturing all text that is displayed on the screen, and using it as an index to the display record. Because there are a wide array of application-specific mechanisms used for rendering

text, capturing textual information from display commands is often not possible. We considered using optical character recognition (OCR) on display records, but found currently available OCR technology to be quite slow and inaccurate for typical desktop screen contents. Instead, THINC leverages ubiquitous accessibility mechanisms provided by most modern desktop environments and widely used by screen readers to provide desktop access for visually-impaired users [40]. These mechanisms are typically incorporated into standard GUI toolkits, making it easy for applications to provide basic accessibility functionality. THINC uses this infrastructure to obtain both the text displayed on the screen and useful context including the name and type of the application that generated the text, window focus and mouse input, selected menu items and HTML links, and keyboard and mouse input. By using a mechanism natively supported by applications, THINC has maximum access to textual information without any application or desktop environment modifications.

THINC uses a daemon to collect the text on the desktop and index it in a database augmented with a text search engine. At the most basic level, the daemon behaves very similarly to a screen reader, as both programs have similar functional requirements. At startup time, the daemon registers with the desktop environment and asks it to deliver events when new text is displayed or existing text on the screen changes. As events are received, the daemon wakes up, collects the new text and state from the application, and inserts this information into the database. However, THINC's daemon needs to be mindful of any overhead it creates on the interactive performance of the desktop. In particular, two aspects of the accessibility mechanism need to be handled with care. First, events are delivered synchronously, meaning that applications block until event delivery is finished. Second, the accessible components of applications are stored as trees. These trees can grow as UI complexity increases, and are extremely expensive to traverse, as only one component in the tree can be

accessed at any point in time, and accessing each component requires continuous context switching between the daemon and the application.

THINC's daemon is designed to minimize both event processing time and the number of queries to applications, by keeping a number of data structures that exactly mirror the accessible state of the desktop. At startup, the daemon traverses all the applications, and builds its own mirror tree. This tree is used to keep an exact replica of the state of the desktop, which can be traversed at a tiny fraction of the cost of traversing the real accessible tree; the latter can take a couple seconds and destroy interactive responsiveness. To minimize event processing time, a hash table maps accessible components to nodes in the mirror tree. This way, as event are received the daemon can quickly look up the corresponding node and figure out which parts of the tree need to be updated.

Keeping an exact replica of the state of the desktop is a crucial mechanism to offer useful searching capabilities to the recorded content. As events are generated, the tree is updated, and its full contents indexed into the database. This way, THINC is able to maintain the temporal relationships of all displayed text. To understand how important this is, consider, for example, a user that is looking for the time when she started reading a paper, but all she recalls is that a particular web page was open at the same time. If text was only indexed when it first appeared on the screen, this temporal relationship between the web page and the paper would never have been recorded, and the user would be unable to access the content of interest. THINC's indexing strategy also allows it to infer text persistence information that can be used as a valuable ranking tool. For example, a user could be less interested in those parts of the record when certain text was always visible, and more interested in the records where the text appeared only briefly.

A limitation of our approach is that not every application may provide an accessi-

bility interface. For example, while THINC can capture text information from PDF documents that are opened using the current version of Adobe Acrobat Reader, other PDF viewers used in Linux do not yet provide an accessibility interface. However, our experience has been that most applications do not suffer from this problem, and there is an enormous impetus to get accessibility interfaces into all desktop applications to provide universal access. The needs of visually impaired users will continue to be a driving force in ensuring that applications increasingly provide accessibility interfaces, enabling THINC to extract textual information from them.

6.3 Playback

Visual playback and search are performed by the THINC client. Various time-shifting operations are supported, such as skipping to a particular time in the display record, and fast forward or rewind from one point to another. To skip to any time T in the display record, THINC goes to its timeline file and finds the tuple with the maximum time less than or equal to T. It then reads the tuple's screenshot information and accesses the screenshot at the specified file position, which is used as the starting point for playback. THINC then reads the tuple's command information and accesses the command at the specified file position. Staring with that command, a two step process is performed that guarantees that only those commands relevant at time T are processed, thus minimizing the time spent in the playback operation. First, THINC builds a list of commands that are relevant to the contents of the screen, discarding those that are overwritten by newer ones. This process goes on, until it reaches a command with time greater than T. The list is ordered chronologically to guarantee correct display output. Second, each command on the list is retrieved from the corresponding files, and displayed.

To play the display record from the current time until time T, THINC simply plays the commands in the command file until it reaches a command with time greater than T. THINC keeps track of the time of each command and sleeps between commands as needed to provide playback at the same rate at which the session was originally recorded. THINC can also playback faster or slower by scaling the time interval between display commands. For example, it can provide playback at twice the normal rate by only allowing half as much time as specified to elapse between commands. At the fastest playback rate possible, THINC simply ignores the command times and processes them as quickly as it can. Except for the accounting of time, the THINC playback application functions in a similar manner to the THINC viewer in processing and displaying the output of commands.

To fast forward from the current display to time T, THINC reads the screenshot file and plays each screenshot in turn until it reaches a screenshot with time greater than T. It then finds the tuple in the timeline file with the maximum time less than or equal to T, which corresponds with the last played screenshot, and uses the tuple to find the corresponding next display command in the command file. Starting with that command, THINC plays all subsequent commands until it reaches a command with time greater than T. Rewind is done in a similar manner except going backwards in time through the screenshots.

6.4 Search

In addition to standard PVR-like functionality, THINC provides a mechanism that allows users to quickly and effectively search and access recorded display output. THINC search uses the index built from captured text and contextual information to find and return relevant results. At the most basic level, THINC allows users to

perform simple boolean keyword searches, which will locate the times in the display record in which the query is satisfied. More advanced queries can be performed by specifying extra contextual information. A useful query users have at their disposal is the ability to tie keywords to applications they have used or the whole desktop. For example, a user may look for a particular set of words limited to just those times when they were displayed inside a Firefox window, and further narrow the search by adding the constraint that a different set of words be visible somewhere else on the desktop or on another application. Users can also limit their searches to particular ranges of time or to particular actions. For example, a user may search for results only on a given day and only for text in applications that had window focus. Due to space constraints, a full discussion of how contextual information can be used for search is beyond the scope of this paper.

A final search mechanism is provided by the user through annotations. At the most basic level, annotations can be created by the user by typing text in some visible part of the screen, since the indexing daemon will automatically add it to the record stream. While this approach is extremely simple, the user may have to provide some unique text that will allow the annotation to stand out from the rest of the recorded text. To help users in this case, THINC provides an additional mechanism which takes further advantage of the accessibility infrastructure. To explicitly create an annotation, the user can write the text, then using the mouse, select it and press a combination key which will message the indexing daemon to associate the selected text with an attribute of *annotation*. The indexing daemon is able to provide this functionality transparently, since both text selection and key strokes events can be delivered by the accessibility infrastructure.

Search results are presented to the user in the form of a series of text snippets and screenshots, ordered according to several user-defined criteria. These include

chronological ordering, persistence information (i.e.. how long the text was on the screen), number of times the words appear, and so on. The search is conducted by first passing a query into the database that results in a series of timestamps where the query is satisfied. These timestamps are then used as indices into the display stream to generate screenshots of the user's desktop. The operation is very similar to the visual playback described before, with the difference that the log playback is done completely offscreen, which helps speed up the operation. THINC also caches screenshots for search results, using a LRU scheme, where the cache size is tunable. This provides significant speedup in cases where the user has to continuously go back to specific points in time in the record.

Each screenshot generated is a portal through which users can either quickly glance at the information they were looking for, or, by simply pressing a button, revive their desktop session as it was at that particular point in time. In addition, when the query is satisfied over a contiguous period of time, the result is displayed in the form of a first-last screenshot, which, borrowing a term from Lifestreams [39], represents a *substream* in the display record. *Substreams* behave like a typical recording, where all the PVR functionality is available, but restricted only to that portion of time.

6.5 DejaView

THINC's display record and indexing functionality has been integrated into DejaView [66, 67], a personal virtual computer recorder that provides a complete WYSIWYS (What You Search Is What You've Seen) record of a desktop computing experience. DejaView enables users to playback, browse, search, and revive their past computing experiences, making it easier to retrieve information they have seen before. DejaView leverages continued exponential improvements in storage capacity [105] to

provide a record of what a user has seen; information is recorded as it was displayed with the same personal context and display layout. All viewed information is recorded, be it an email, web page, document, program debugger output, or instant messaging session. The information is automatically indexed based on displayed text captured in the same context as the recorded information.

DejaView enables a user to playback and browse records for information using functions similar to personal video recorders (PVR) such as pause, rewind, fast forward, and play. DejaView enables a user to search records for specific information to generate a set of matching screenshots, which act as portals for the user to gain full access to recorded information. DejaView enables a user to select a given point in time in the record from which to revive a live computing session that corresponds to the desktop state at that time. The user can time travel back and forth through what she has seen, and manipulate the information in the record using the original applications and computing environment.

To supports its personal virtual computer recorder usage model, DejaView needs to record both the display and execution of a user's desktop computing environment such that the desktop recording can be played and manipulated at a later time. DejaView must provide this functionality in a manner that is transparent, has minimal impact on interactive performance, can preserve visual display fidelity, and is space efficient. DejaView achieves this by using a virtualization architecture that consists of two main components, a virtual display provided by THINC, and a virtual execution environment based on ZAP [68, 101]. These components leverage existing system interfaces to provide transparent operation without modifying, recompiling, or relinking applications, window systems, or operating system kernels.

DejaView's virtual execution environment decouples the user's desktop computing environment from the underlying OS, enabling an entire live desktop session to

be continuously checkpointed, and later revived from any checkpoint. Building on Zap, DejaView leverages the standard interface between applications and the OS to transparently encapsulate a user's desktop computing session in a private virtual namespace. This namespace is essential to support DejaView's ability to revive checkpointed sessions. By providing a virtual namespace, revived sessions can appear to access the same OS resources as before, even if they are mapped to different underlying resources upon revival. By providing a private namespace, revived sessions from different points in time can run concurrently and appear to use the same operating system resources inside their respective namespaces, and yet not have any conflicts among each other. This lightweight virtualization mechanism imposes low overhead as it operates above the operating system instance to encapsulate only the user's desktop computing session, not an entire machine instance. By using a virtual display and running its virtual display server inside its virtual execution environment, DejaView ensures that all display state is encapsulated in the virtual execution environment so that it is correctly saved at each checkpoint. Furthermore, revived sessions can then operate concurrently without any conflict for display resources since each has its own display state. DejaView combines logging [65]and unioning file system mechanisms [166] to capture the file system state at each checkpoint. This ensures that applications revived from a checkpoint are given a consistent file system view corresponding to the time at which the checkpoint was taken.

6.6 Experimental Results

We have implemented THINC's display recording and text capturing architecture as an extension to THINC's Linux/X based implementation. This section presents experimental results that quantify its performance when running a variety of common

desktop applications. We present results for both application benchmarks and real user desktop usage. The focus of our experiments is on quantifying the storage requirements and performance overhead in terms of the cost of continuously recording display and indexing text. For both the application benchmark and the real desktop usage experiments, we do full fidelity display recording.

We used the desktop application scenarios listed in Table 6.1. We considered several individual application scenarios running in a full desktop environment, including scenarios that created lots of display data (web, video, untar, make, cat) as well as those that did not and were more compute intensive (gzip, octave). These scenarios measure THINC performance only during periods of busy application activity, providing a conservative measure of performance since real interactive desktop usage typically consists of many periods in which the computer is not fully utilized. For example, our web scenario downloads a series of web pages in rapid succession, instead of having delays between web page downloads for user think time. To provide a more representative measure of performance, we measured real user desktop usage (labeled as desktop in the graphs) by aggregating data from multiple graduate students using our prototype for all their computer work over many hours.

For all our experiments the THINC client and server ran together on a Dell Dimension 5150C with a 3.20 GHz Intel Pentium D CPU, 4 GB RAM, a 500 GB SATA hard drive and connected through a 1000 Mbps ethernet card to a public switched Fast Ethernet network. The machine ran the Debian Linux distribution with kernel version 2.6.11.10 using X.org 7.1 as the window system, and GNOME 2.14 as the desktop environment. The display resolution was 1024x768 for the application benchmarks and 1280x1024 for real desktop usage measurements. For our web application scenario, we also used an IBM Netfinity 4500R server with dual 933 MHz Pentium III CPUs and 512 MB RAM as the web server, running Linux kernel version 2.6.10

Name	Description
web	Firefox 2.0.0.1 running iBench web browsing benchmark to download 54 web pages
video	MPlayer 1.0rc1-4.1.2 playing *Life of David Gale* MPEG2 movie trailer at full-screen resolution
untar	Verbose untar of 2.6.16.3 Linux kernel source tree
gzip	Compress a 1.8 GB Apache access log file
make	Build the 2.6.16.3 Linux kernel
octave	Octave 2.1.73 (MATLAB 4 clone) running Octave 2 numerical benchmark
cat	`cat` a 17 MB system log file
desktop	16 hr of desktop usage by multiple users, including Firefox 2.0.0.1, GAIM 1.5, OpenOffice 2.0.1, Adobe Acrobat Reader 7.0, etc.

Table 6.1 – Recording application benchmark scenarios

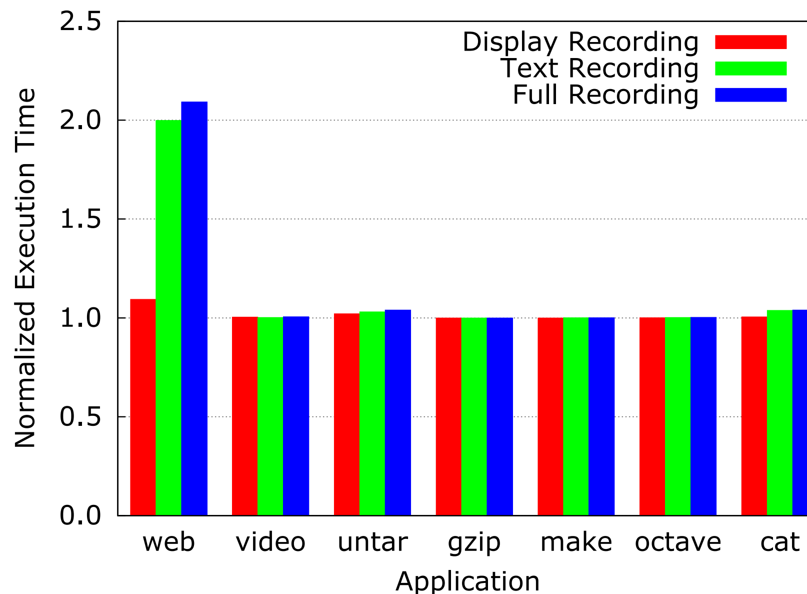


Figure 6.1 – Recording runtime overhead

and Apache 1.3.34.

Figure 6.1 shows the performance overhead for each application scenario. We ran each scenario without recording, with each of display recording and indexing components enabled at a time, and with full recording. Performance is shown normalized

to the execution time without any recording. The results show that there is some overhead for full recording, but there are no visible interruptions in the interactive desktop experience and real-time interaction is not affected in practice. Full recording overhead is small in almost all scenarios, including those that are quite display intensive such as cat and full-screen video playback. In all cases other than web browsing, the overhead was less than 5%. For video, the most time-critical application scenario, the overhead is less than 1% and does not cause any of the video frames to be dropped during display. For web browsing, full recording overhead was about 110% because the average download latency per web page was a little more than half a second with indexing while it was .28 seconds without recording. We discuss the reasons for this overhead below. However, real users do not download web pages in rapid succession as the benchmark does, and the page download latencies with recording are well below the one second threshold for users to have an uninterrupted browsing experience [94], and is fast enough in practice for interactive web browsing. We did not measure the performance overhead of the desktop usage scenario given the lack of precise repeatability.

Figure 6.1 shows how the recording components individually affect performance. The largest display recording overhead is 9% for the rapid fire web page download, which changes almost all of the screen continuously and causes the web browser and THINC server and viewer to compete for CPU and I/O resources. The display overhead for all other cases is less than 2%. As expected, gzip and octave have essentially zero display recording overhead since they produce little visual output. Interestingly, video has one of the smallest display recording overheads of essentially zero. Even though it changes the entire display for each video frame, it requires only one command for each video frame, resulting in 24 commands per second, a relatively modest rate of processing.

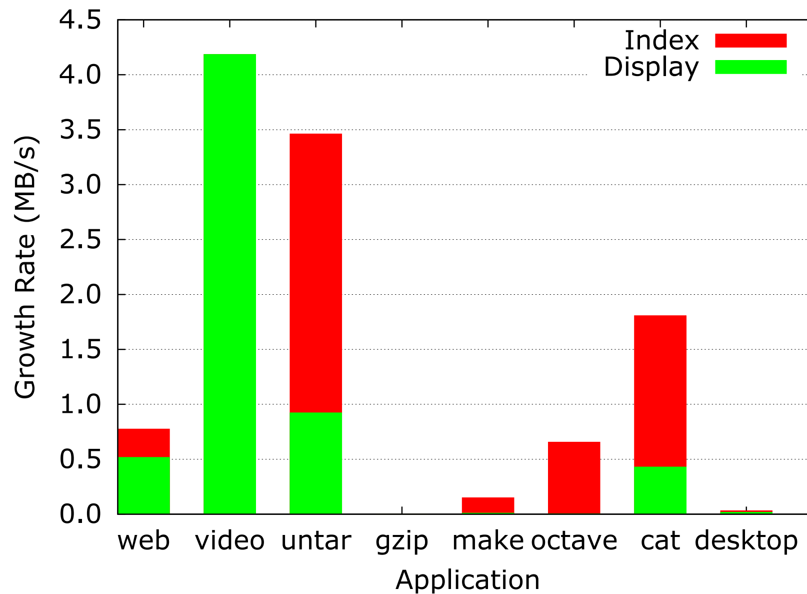


Figure 6.2 – Recording storage growth

Figure 6.1 also shows the index recording overhead, which is small in all scenarios except for the web benchmark. The overhead is less than 4% for all cases except for the web benchmark. For the web benchmark, the indexing overhead is 99%, which accounts for almost all of the overhead of full recording. Unlike other applications, the Firefox web browser creates its accessibility information on demand instead of as part of normal operation. This dynamic generation of accessibility information coupled with weakness in the current Firefox accessibility implementation results in much higher overhead when capturing text. We expect that this overhead will decrease over time as its accessibility features improve [87].

Figure 6.2 shows the storage space growth rate THINC experiences for each of the application scenarios. The results are decomposed into the amount of increased storage THINC imposes for display state and text capture. We do this by measuring the size of the files created to store their respective information. Figure 6.2 shows that for all of the application scenarios except video, untar and cat, storage growth rate

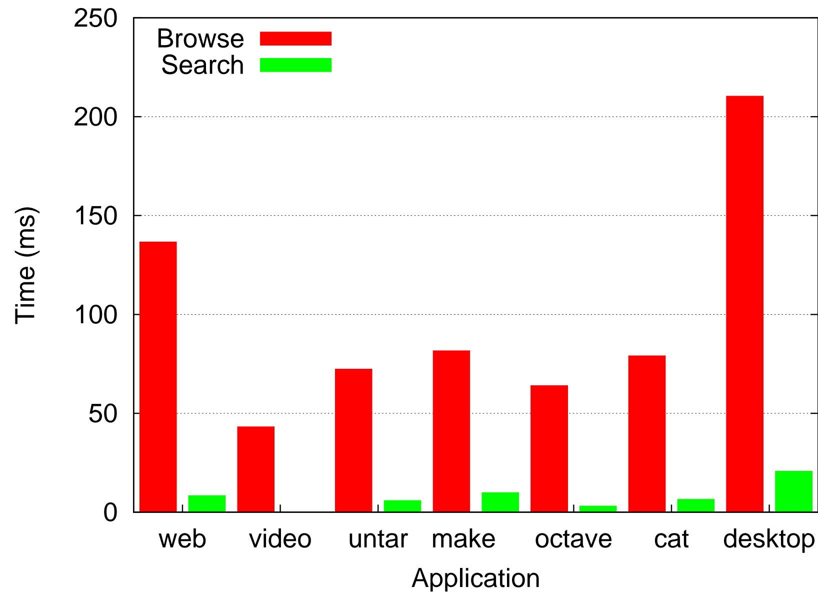


Figure 6.3 – Browse and search latency

is quite low. Video requires more extensive display storage since each event changes the entire display, even though it does not create a high rate of events. Untar and cat have a large indexing growth rate since they both output large amounts of text on the screen.

More importantly, typical usage does not have as high of a growth rate, resulting in much lower storage requirements in practice. As shown in Figure 6.2, the storage space growth rate for real user desktop usage is much more modest at only 0.3 MB/s. In comparison, HDTV PVRs require roughly 9 GB of storage per hour of recording, or 2.5 MB/s. While THINC's storage requirements can be greater than HDTV PVRs during periods of intense application activity, the desktop scenario results indicate that in practice they will be much smaller. Also, disk storage densities continue to double each year and as multi-terabyte drives become commonplace in PCs [105], the storage requirements of THINC will become increasingly practical for many users.

We also conducted experiments that show THINC's effectiveness at providing ac-

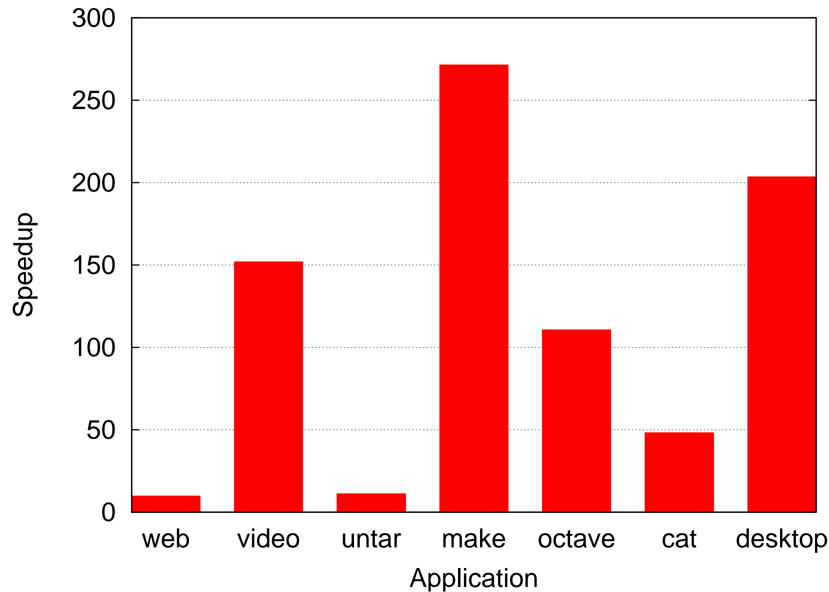


Figure 6.4 – Playback speedup

cess to recorded content, by measuring its search, browse, and playback performance. We measured THINC search performance by first indexing all displayed text for our application tests and desktop usage, each in its own respective database, then issuing various queries. For each application benchmark, we report the average query time for five single-word queries of text selected randomly from the respective database. For real desktop usage, we report the average query time for ten multi-word queries, with a subset limited to specific applications and time ranges, to mimic the expected behavior of a THINC user. Figure 6.3 shows that on average, THINC is able to return search results in no more than 10 ms for the application benchmarks and in roughly 20 ms for real desktop usage. These results demonstrate that the query times are fast enough to support interactive search. Another important measure of search performance is the relevance of the query results, which we expect to measure based on a user study; this is beyond the scope of this dissertation.

We measured browsing performance by using the display content recorded dur-

ing our application benchmarks and access it at regular intervals. Since the full fidelity recorded stream contains many points where the user was not actively using the system, and the display was not being actively updated, replaying the minimal commands needed to access those points would minimize the average time needed to regenerate the contents of the screen. Since these are points in time a user is unlikely to want to regenerate, and to avoid skewing the results in THINC's favor, we eliminate search points if less than 100 display commands were issued from the previous point. Figure 6.3 shows that on average, THINC can access, generate, and display the contents of the stream at interactive rates, ranging from 40 ms browsing times for video to 130 ms for web. For real desktop usage, browsing times were roughly 200 ms. These results demonstrate that THINC provides fast access to any point in the recorded display stream, allowing users to efficiently browse their content.

To demonstrate how a user quickly a user can visually search the record, we measure playback performance of all the application scenarios and measure how long it would take to play the entire visual record. Figure 6.4 demonstrates that THINC is able to playback an entire record at many times the rate at which it was originally generated. For instance. Figure 6.4 shows that THINC is able to playback regular user desktops at over 200 times the speed it was recorded. While some benchmarks, in particular web browsing, do not show as much of a speedup, we attribute this to the fact that they are constantly changing data at the rate of display updates. Even in the worst case, THINC is able to display the visual record at over 10 times the speed at which it was recorded. These results demonstrate that THINC can browse through display records at interactive rates.

6.7 Summary

This chapter has introduced a novel mechanism for continuously recording visual desktop output, and indexing it based on text displayed, enabling all desktop output to be played back, arbitrarily browsed, and searched through text and contextual queries.

By leveraging THINC's virtual and remote display architecture, this mechanism can work seamlessly with unmodified applications and operating systems, operate with very low-overhead, produce very efficient representations of the recorded content, and enable recorded output to be efficiently accessed. Alongside, we have developed a novel way to harvest on-screen text that leverages accessibility interfaces to gain access to both text and desktop contextual information.

We have implemented a prototype of this system, and evaluated its performance, in terms of runtime overhead, space usage, and how efficiently it can provide access to recorded content. Our results with common desktop application workloads and real desktop usage demonstrate that THINC can provide a high-performance, space-efficient platform for desktop recording and indexing, and that recorded content can be played back, randomly accessed, and searched fast enough for interactive use.

Finally, we have integrated our display recording and indexing functionality into DejaView [66, 67], a personal virtual computer recorder that provides a complete record of a desktop computing experience. DejaView enables users to not only playback, browse, and search visually recorded output, but to also revive their past computing experiences, making it possible to retrieve both information they have seen before, and the state of their desktops in the past. In this manner we have shown that THINC not only provides an efficient remote display architecture, but it can also be used as a building block for new desktop applications.

Chapter 7

Related Work

This dissertation introduced a virtualization architecture for remote desktop access. Its contributions touch upon many fields, and a number of systems have been proposed in the literature and as commercial products which have some commonality to THINC, either in function or methodology. In this chapter we discuss in detail these systems and THINC's relationship to them.

7.1 Remote Display and Thin-Client Computing

Because of the importance of developing effective remote display systems, many alternative designs have been proposed. These approaches can be loosely classified based on a number of design choices, which we discussed in depth in Chapter 2 while describing the architecture of THINC. In this section, we will follow a similar path, but focused on the choices made by existing systems.

As Figure 7.1 shows, a typical display architecture works as a pipeline, with desktop applications on one end, and the framebuffer and input devices at the other. The purpose of this architecture is to allow applications to generate visual output to users,

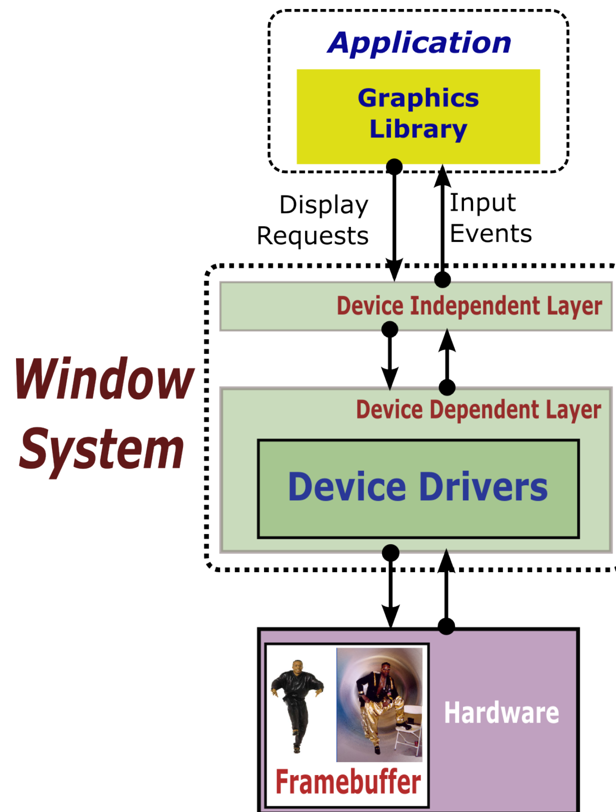


Figure 7.1 – Standard display architecture

and in turn to receive input events generated by these same users as they interact with the applications. Section 2.1 discusses at length how the architecture works by showing the process triggered when the user clicks on a link on a web page, and a new page is rendered on the screen.

Given this pipeline architecture, existing systems that provide remote display for desktops can be loosely classified based on several design choices:[1]

1. *where* the graphical user interface of applications is executed,

2. *how* display commands from applications are intercepted so that display updates

[1] Given that some of these systems are commercial and closed source, the choices which we are able to describe here are only those which we can infer by treating the systems as black boxes. Deeper architectural choices like how translation and delivery of commands is done, which we previously described as being core to the design of THINC, cannot be evaluated this way.

can be sent from server to client, and

3. *what* display primitives are used for sending display updates over the network.

Older remote display systems such as Plan 9 [110] and X [123] provide remote display functionality by pushing all user interface processing to the client computer. Application-level display commands are not processed on the server computer, but simply forwarded to the client. This division of work is more apparent in X, where, somewhat confusingly, the client computer is referred to as the "X server", and applications running on the server computer are called "X clients". X applications perform graphics operations by calling library functions in charge of forwarding application-level display commands over the network to the X server. 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 display 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 [69]. 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 [69, 124]. Finally, storing and managing all display state at the client makes it difficult to

support seamless user mobility across different locations. In most cases, the display state must be duplicated on the server before users are able to move across different clients without losing the state of their desktop.

Proxy extensions such as low-bandwidth X (LBX) [169] and NoMachine's NX [95] have been developed to try to address some of these problems and improve X performance. LBX has been shown to have poor performance [61] compared to other remote display systems [73]. NX is a more recent development that provides 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 application user interface processing on the client. Furthermore, the key compression techniques in NX can also be used for remote display systems which do not have the drawbacks inherent in executing the user interface of applications on the client.

More recent remote display systems such as Citrix MetaFrame [23], Microsoft Remote Desktop [25, 82, 83], Sun Ray [124], and VNC [118] run the graphical user interface of applications at the server, avoiding the need to maintain and run complex window server software at the client. 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 server software 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. One approach is to translate application display commands into a rich set of low-level graphics commands that encode display updates sent over the network. These commands are similar to many of the commands used in X. Citrix MetaFrame, Sun's Secure Global Desktop [142] (previously known as Tarantella [121]), and Microsoft Remote Desktop are three competing commercial product examples of this approach. However, performance studies [69, 172] of these systems indicate that using a richer set of display primitives does not necessarily provide substantial gains in bandwidth efficiency, particularly in the presence of multimedia content. Furthermore, the added overhead of supporting a complex set of display primitives results in slower responsiveness and degraded performance in WAN environments.

A second approach is to use simpler 2D drawing primitives for sending display updates over the network. Sun Ray takes this approach, and is now in its third major product version from Sun Microsystems. While the command set used is simple and easy to implement, Sun Ray is not able to efficiently and transparently translate application display commands into its command set. It instead often relies on reducing application commands to pixel data and sampling the resulting data to determine which drawing primitives to use. Determining the most efficient drawing primitives to use can be difficult and processing overhead can adversely affect overall performance. Some applications which generate display commands that Sun Ray cannot efficiently

translate need to be explicitly modified to deliver adequate performance. For example, Sun Ray lacks transparent support for video playback. Another drawback in Sun Ray's approach is that it intercepts application commands using a customized X server, which is not easy to do effectively given the complexity of window server implementations. Furthermore, a customized server quickly becomes outdated given the difficulty of keeping up with continuing advances in more widely supported window server implementations, such as XFree86 and X.org. Note that Sun Ray was originally designed assuming the use of a private, low-latency LAN environment [124], though more recent product versions attempt to relax this requirement.

A third approach is to reduce everything to raw pixel values for representing display updates, then read the resulting framebuffer pixel data and encode or compress it, a process sometimes called screen scraping. VNC [150] and GoToMyPC [46] are two actively developed and widely-used systems based on this approach. Other similar systems include Laplink [71] and PC Anywhere [106], which have been previously shown to perform poorly [92]. Screen scraping is relatively simple and decouples the processing of application display commands from the generation of display updates sent to the client. Servers must do the full translation from application display commands to actual pixel data, but clients can be very simple and stateless. However, display commands consisting of raw pixels alone are typically too bandwidth-intensive. For example, using raw pixels to encode display updates for a video player displaying at 30 frames per second (fps) full-screen video clip on a typical 1024x768 24-bit resolution screen would require over 0.5 Gbps of network bandwidth. As a result, the raw pixel data must be compressed. Many compression techniques have been developed for this purpose, including FABD [43], PWC [9], and TCC [19, 18]. However, generating display updates in this manner is fundamentally inefficient since the original application display semantics are lost and cannot be used in the process.

Platform	**Display Protocol**	**User Interface**[a]	**Display Encoding**	**Interception Point**
X Window System	X	Client	High-level graphics	Window System[b]
NoMachine NX	Compressed X	Client	High-level graphics	Window System[c]
Citrix Metaframe	ICA	Server	Low-level graphics	Window System [d]
Microsoft Terminal Services	RDP	Server	Low-level graphics	Window System [d]
Sun Secure Global Desktop	AIP	Server	Low-level graphics	Window System [e]
SunRay	SunRay	Server	Simple 2D primitives	Window System [e]
GoToMyPC	GoToMyPC	Server	Simple 2D primitives	Framebuffer
VNC	VNC	Server	Simple 2D primitives	Framebuffer
THINC	THINC	Server	Simple 2D primitives	Device Driver

[a] Specifies where the application's user interface is executed
[b] Provides all window system functionality
[c] Uses a proxy which functions as a fake window system
[d] Intercepts from within the window system
[e] Intercepts using a customized window system

Table 7.1 – Remote Display Systems Comparison

Table 7.1 summarizes the major characteristics of the most popular remote display systems in use today.

7.2 Multimedia Support

Traditional remote display and thin-client systems have been tailored towards supporting remote access over low-bandwidth connections or accessing office applications in LAN environments. Support for multimedia playback and synchronization in existing thin-client systems is generally lacking as has been shown in previous performance studies [69, 172].

Widely-used remote display systems such as VNC [150] and GoToMyPC [46] take the common approach of doing nothing for audio/video playback. They cannot play audio. They play video by simply screen scraping display data from the framebuffer along with any other display content. This results in terrible video quality as the mechanism cannot keep up with video playback rates. Video data typically is overwritten in the framebuffer before it can even be displayed to the client.

Sun Ray [141], a commercial thin-client system in its third major product version from Sun Microsystems, provides native audio playback and provides protocol mechanisms that can be used by application developers to improve video playback. These mechanisms cannot be used with unmodified off-the-shelf applications as they require applications to be rewritten or relinked with special libraries. Sun Ray was originally designed assuming the use of a private, low-latency LAN environment [124], though more recent product versions attempt to relax this requirement.

Recent support was added in Microsoft's Remote Desktop (RDP) [25, 82] and Citrix MetaFrame (ICA) [23] for remote audio/video playback. Both RDP and ICA take advantage of the media playback architecture present in Windows to deliver

media remotely. In particular, they capture the encoded media stream from the application, transmit it over the network, and leave the decoding and playback to the client. Client complexity increases as decoding of media streams relies heavily on local software components, which need to be bundled with and maintained on the client. This creates an additional upgrade and support point in the network, increasing management cost and complexity. Furthermore, these mechanisms do not work with many standard media formats and require that applications be written using the necessary Windows extensions. Not all multimedia applications support the interfaces used by RDP and ICA to stream content. This lack of standard interfaces for media decoding and playback in both the Windows and Unix world is one of the major problems that thin-client systems must overcome in providing multimedia support.

Many modern thin clients evolved from X [123], which is a display-only system that does not natively support audio. Unlike other approaches, X requires a full window server running on the client, increasing client management complexity. X provides extensions, namely Xvideo and XvMC, that support video playback at full frame rate, though only Xvideo works remotely. X-based thin-client systems such as low-bandwidth X (LBX) [169] and NoMachine's NX [95] have no support for these extensions or other video playback mechanisms. A separate audio server can be installed and run on the client to provide remote audio playback for X-based systems. Examples of these servers are ESD (Enlightenment Sound Daemon) [33], aRtsd (Analog Realtime Synthesizer Daemon) [7], NAS (Network Audio System) [91], PulseAudio [114], which is an effort intended to replace ESD, and MAS (Media Application Server) [78], developed by the X consortium to bring media support to the X world. Both ESD and aRtsd are widely used on Linux desktops, the former is bundled with the GNOME desktop suite and the latter is used by the K Desktop Environment.

These require audio applications to be written to use them, and any single server may not support all the audio applications users would like to run.

The Infopad project [146] was one of the earlier systems that provided remote access to multimedia. Infopad proposed the use of hardware-only terminal devices optimized for operation on wireless networks. Perhaps hindered by the restricted environment to which it was tailored (and the limitations of early wireless networks), Infopad was only able to provide reduced quality video, without support for full frame rate, full-screen playback.

Much work has been done on the topic of multimedia synchronization and synchronization methods [55], though most techniques do not apply in every environment or setting where synchronization must be used. For example, the synchronization requirements of a single media type originating from a single source and streamed over a high-bandwidth, low-latency network to a single sink may require much different synchronization scheme than a multimedia stream with media originating from n sources and arriving at m sinks over a lossy and congested network. Multimedia synchronization mechanisms can also apply to local playback or on network environments where applications must be designed significantly differently to support either type of setting. Measuring the quality of synchronization has proven difficult as no widely-used performance measures are available. The sheer number of synchronization techniques underscores the difficulty and importance of coming up with reliable and accurate means of measuring and evaluating multimedia synchronization [55, 124]. Because of the subjective nature of determining what is "good" synchronization versus "bad", research has also been conducted in the human perception of jitter and tolerance of unsynchronized media [138].

Of all mechanisms for providing synchronized multimedia playback over a network, the Real-time Transport Protocol (RTP) is perhaps the most widely-used protocol

today [126]. Though RTP provides a thorough and elaborate protocol of supplying timestamps, rate control, and other information needed to apply synchronization in a variety of settings, RTP itself does not specify a means with which to apply any particular synchronization technique [109].

As a thin-client system, THINC's requirements for a synchronization mechanism are unique in that it must transparently provide synchronization without applications being aware of any mechanism being applied. That is, local synchronization must be possible for multimedia applications running on a THINC server, and network synchronization must be applied for client playback.

A number of remote audio/video conferencing solutions are in wide use today. These include applications such as NetMeeting [89], VIC [77], and RAT [117]. They provide specialized support for bidirectional, synchronized real-time playback of audio and video. These solutions are complementary to THINC. THINC focuses on the problem of providing multimedia application support for the computing infrastructure of large organizations where the primary audio/video input is from outside sources and not directly from end users for specialized conferencing purposes.

7.3 Support for Mobile Devices

The ability for thin clients to improve web browsing performance on wireless PDAs was first quantitatively demonstrated in a previous study [70]. This study demonstrated that thin clients can provide both faster web browsing performance and greater web browsing functionality. The study considered a wide range of web content including content from medical information systems. Our work builds on this previous study and considers important issues such as how usable existing thin clients are in PDA environments, the trade-offs between thin-client usability and performance,

performance across different PDA devices, and the performance of thin clients on common web-related applications such as video.

Many thin clients have been developed and some have PDA clients, including Microsoft's Remote Desktop [25, 82], Citrix MetraFrame XP [23], Virtual Network Computing [118, 88], GoToMyPC [46], and Sun's Secure Global Desktop [142]. These systems were first designed for desktop computing and retrofitted for PDAs. Unlike pTHINC, they do not address key system architecture and usability issues important for PDAs. This limits their display quality, system performance, available screen space, and overall usability on PDAs. pTHINC builds on THINC [13], extending the server architecture and introducing a client interface and usage model to efficiently support PDA devices for mobile web applications.

Other approaches to improve the performance of mobile wireless web browsing have focused on using transcoding and caching proxies in conjunction with the fat client model [38, 58, 59, 76]. They work by pushing functionality to external proxies, and using specialized browsing applications on the PDA device that communicate with the proxy. Our thin-client approach differs fundamentally from these fat-client approaches by pushing all web browser logic to the server, leveraging existing investments in desktop web browsers and helper applications to work seamlessly with production systems without any additional proxy configuration or web browser modifications.

With the emergence of web browsing on small display devices, web sites have been redesigned using mechanisms like WAP and specialized native web browsers have been developed to tailor the needs of these devices. Recently, Opera has developed the Opera Mini [100] web browser, which uses an approach similar to the thin-client model to provide access across a number of mobile devices that would normally be incapable of running a web browser. Instead of requiring the device to process web

pages, it uses a remote server to pre-process the page before sending it to the phone.

7.4 Display Recording and Text Capture

The ability to record display output for later playback has been proposed and implemented in numerous systems in the past. Some of them have focused on providing tools for remote collaboration [42, 119, 133] or as teaching aids [2]. However, most of these approaches rely on using custom-made applications and environments, which can severely affect their wide use. More recently, a VNC-based approach [72] extends the normal VNC architecture [150] to provide a generic environment for asynchronous collaboration and as a teaching aid. In this system, a proxy sits between the VNC server and clients, and records all output. The proxy is able to playback earlier output to support late comers and mobile users (in the case of a collaboration environment), or do complete offline playback (for teaching purposes). While THINC's approach is similar, and both systems rely on a remote display architecture to provide transparent display recording, their goals are different and mostly orthogonal. Furthermore, VNC's approach does not provide a way to index and search the recording, as opposed to THINC's text capture and display indexing system.

Screencasting provides a recording of a desktop's screen that can be played back at a later time [56, 152, 155, 157, 164, 171]. It has become very popular as a tool to create computer tutorials and demonstrations. Screencasting works by screen scraping and taking screenshots of the display many times a second. It requires higher overhead and oftentimes more storage and bandwidth than THINC's display recording, and the common approach of also using lossy video codecs to compensate further increases recording overhead and decreases display quality. THINC's display recording technology overcomes this limitation by recording only updates to the screen

in the form of simple remote display commands, not just raw pixels, to capture the original display fidelity.

Furthermore, THINC goes beyond screencasting by not only recording the display state, but by capturing text and extracting contextual information to enable display search. OpenMemex [98] extends VNC-based screencasting to also provide display search by using offline OCR to extract text from the recorded data. THINC differs from OpenMemex in that it is able to extract information in addition to just text, and its use of accessibility interfaces will result in better text capture, given the current state of OCR technology. Most commercially available OCR systems are designed for document processing, and desktop content (e.g., small point fonts) may prove challenging to process.

Chapter 8

Conclusions and Future Work

This dissertation presented THINC, a virtual and remote display architecture for desktop computing. In building THINC we departed from the common practice of focusing on improved protocols and compression mechanisms, and instead proposed, and demonstrated, that the architecture of the system is just as important to the overall remote display performance.

THINC is built around a virtual device driver model that abstracts a computer's display and input hardware, by introducing simple device drivers that look and behave like traditional, hardware-specific drivers. In this manner, desktop output can be redirected over the network to simple clients, while leveraging continuing advances in window server technology, and working seamlessly with unmodified applications, window systems, and operating systems. THINC's virtual display architecture introduces novel translation optimizations that take advantage of semantic information to efficiently convert high-level application requests to simple low-level protocol commands. On top of these translation mechanisms, a number of delivery optimizations are introduced that prioritize important updates, and automatically discard irrelevant ones.

We implemented this basic architecture as a device driver for the X Window System in Linux, and a simple Xlib-based client application. Our implementation illustrates the simplicity of THINC's protocol and the effectiveness of its translation and delivery architecture. We conducted an experimental evaluation that compared the web browsing performance of THINC to a number of existing commercial remote display products, on both LAN and WAN environments. Our results show that THINC can deliver good interactive performance even when using clients located around the world. THINC provides superior web performance over other systems, with up to 4.5 times faster response time in WAN environments. Our results demonstrate how THINC's unique mapping of application-level drawing commands to protocol primitives and its command delivery mechanisms significantly improve the overall performance of a remote display system.

Going beyond remote display, this dissertation has also shown how THINC provides a fundamental building block for a broad range of applications.

First, we introduced a mechanism to natively support multimedia content in a remote display system. For video playback, THINC extends existing video acceleration interfaces that leverage client hardware capabilities. For audio support, we continue our virtualization approach, and introduce a virtual audio device driver, that looks and behaves like a normal sound card, and provides remote audio playback and capture. Finally, we provide a simple, client-based mechanism that synchronizes audio and video data delivered over separate channels.

Second, recognizing the growing mobility of users, and the increasing popularity of small mobile devices such as PDAs, we introduced the pTHINC architecture for wireless PDAs. pTHINC provides key architectural and usability mechanisms such as server-side screen resizing, video support, optimal use of screen space for display updates, and leverage of existing PDA control buttons for UI elements. In this man-

ner, pTHINC is able to provide mobile users with ubiquitous access to a consistent, personalized, and full-featured web environment across heterogeneous devices.

Third, we integrated THINC's architecture into a full-fledged, desktop utility computing system. This infrastructure provides perhaps a glimpse of the future, where desktop computers are delivered as utility services. We also presented mechanisms that leverage the advantages of overlay networks and THINC's unique protocol characteristics to protect the desktop utility infrastructure from distributed denial of service attacks, with minimal impact on the performance of the system.

Finally, we moved beyond remote display, and used THINC to build a novel display recording system for desktop users. The system takes advantage of THINC's architecture and remote display protocol to do transparent recording with very low storage requirements and performance impact. Using desktop accessibility interfaces, we also provide text capture to index the recording. In this way, users are not only able to browse and playback through their desktop log, but also search its contents.

We have implemented all of these systems, and extensively measured their functionality and performance on a number of real world scenarios. All our results validate the notion that THINC's architecture is sound, and can be effectively extended to provide high performance systems that can transparently interface with existing applications, window and operating systems. They also demonstrate that THINC can provide a valuable foundation for building systems to improve desktop computing.

8.1 Future Work

The work developed on this dissertation opens up the possibility for a number of improvements and new directions for future research work.

Our most immediate goal is to continue actively participating in the develop-

ment and implementation of a standard for remote displays. This standard, named Net2Display [96], and being developed within the Video Electronics Standards Association (VESA) [148]. The standard has taken some of the principles we developed with THINC's architecture [32, 134] and other existing remote display systems to support a wide variety of display devices, from simple, output-only displays, projectors, kiosks, booths, thin-client terminals, to powerful workstations.

One of the most pressing directions for future research is how to effectively support 3D applications inside a remote desktop system. While previous work has focused on remote 3D rendering of specialized applications [129] or taking advantage of rendering clusters [53], effectively supporting 3D applications in the context of traditional desktops has been an elusive goal. This area of work has become even more important given the extensive use of high-end visual effects in the most popular desktop environments in use today.

Supporting high-end desktops and 3D applications remotely on commodity hardware and networks provides some unique challenges. First, 3D applications have high and specialized resource requirements, both in terms of computational power, which have driven the development of complex and powerful video cards, and data transfer requirements, which have helped foster the development of faster internal computer buses, such as AGP and PCI Express [160]. Second, the latency sensitive nature of 3D applications, such as interactive video games, and desktop environments, where 3D effects are normally generated in response to user interaction with the desktop, severely limit the amount of overhead a remote display system is allowed to incur in order to efficiently support this content. Third, a remote 3D display system will need to be able to provide support, and in some cases interoperability, for the two major hardware interfaces in use today, OpenGL [97] and DirectX [81].

In practice, these challenges translate in an uncertainty about the most appropri-

ate division of work between the client and the server. The straightforward approach of pushing all rendering work to the client has the advantage of leveraging the client's video card to do most of the work, effectively extending the internal video card bus over the network. Perhaps the best example of this approach is the X Window System and its AIGLX [3] component. However, this approach has a number of drawbacks. First, blindly pushing all data will overload most commodity networks, which can only provide a fraction of the bandwidth available on a computer's internal bus. Compressing the 3D data may alleviate this problem, but care must be taken to carefully control the compression overhead on the latency of the system. Second, the client is not guaranteed to be able to cope with the requirements of the applications. In environments where clients are meant to be simple devices, this approach may not provide a viable solution. Third, the client becomes a stateful entity in the system, with the attendant drawbacks associated with this situation. In particular, the server will need to do some work to maintain the state of the 3D engine to allow it to be replicated if a client disconnects and reconnects again. Finally, relying on the client to do all rendering work may cause interoperability issues. For example, attempting to access a Windows desktop from a Unix-based client would force the client to convert from DirectX to OpenGL before it can take advantage of its local hardware.

Another approach would be to do all the rendering work on the server and send simple primitives to the client, similarly to the approach taken by THINC for 2D applications. In this scenario, the client can be simple and stateless, and interoperability can be handled transparently on the server by providing a translation layer between the native API and the primitives used by the remote display system. Network overhead could be reduced if the primitives chosen can provide a more compact representation of the changes on the screen than traditional 3D data. Finally, in an environment with powerful servers, the system would able to leverage this computa-

tional power to do the rendering work. The best example of this approach is VirtualGL [149], which uses a GLX forking architecture [136, 137] to perform all rendering on the server and only send the finished images to the client. However, this approach suffers from limited scalability and resource contention, particularly in environments where multiple users share a single computer to host their desktops, and they must compete for access to the server's video card and its computational resources. Recent developments in I/O and GPU virtualization [107] provide a possible solution to alleviate these problems, by allowing multiple users to gain concurrent access to a single video card. Previous shortcomings in video card architectures which provided slow read back speeds, thus limiting the rate at which rendered images could be generated, appear to be mostly addressed by the current generation of video cards [34] and system buses.

A compromising approach that partitions the rendering work at some intermediate stage of the rendering process could provide the most optimal solution, by leveraging the best of the two previously described approaches. However, such an approach may be overly complex, given the high level of abstraction of hardware 3D interfaces, and their large size and complexity. Finding an appropriate partition work that would serve the needs of most applications may prove to be a challenging task in this environment.

Our work on desktop recording provides a new approach for information storage and retrieval that opens up new directions for future research. In the immediate future, we envision conducting user studies to explore usage patterns to better understand how this functionality will be exploited by users over extended periods of time and how the user interface can be enhanced to better fit daily usage needs. In terms of text capture and search, more work is needed on quantifying and improving the relevance and presentation of search results by exploring the use of desktop contextual

information such as time, persistence, or the relationships among desktop objects. It may also be interesting to explore the possibility of recording desktop audio (both played back and captured), and use it as an additional indexing element, and search tool.

Finally, there are a number of improvements that can help THINC's remote desktop access architecture. In terms of multimedia support, reducing bandwidth usage of video playback and providing video capture will become a necessity as video conferencing becomes an integral part of our everyday experience. We envision our virtualization approach to provide a perfectly suitable architecture to accomplish these goals. In terms of mobile device support, user interface improvements in the form of touch screen support and gestures, and leveraging newer image resizing techniques [10] can provide an even better experience for mobile users accessing their desktops remotely.

Bibliography

[1] 100x100 Project. http://100x100network.org/.

[2] Gregory D. Abowd, Christopher G. Atkeson, Jason Brotherton, Tommy Enqvist, Paul Gulley, and Johan LeMon. Investigating the capture, integration and access problem of ubiquitous computing in an educational setting. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 440–447, April 1998.

[3] Accelerated Indirect GLX (AIGLX). http://fedoraproject.org/wiki/RenderingProject/aiglx.

[4] Aditya Akella, Jeffrey Pang, Anees Shaikh, Bruce Maggs, and Srinivasan Seshan. A Comparison of Overlay Routing and Multihoming Route Control. In *Proceedings of ACM SIGCOMM*, August - September 2004.

[5] ALSA asym plugin. http://alsa.opensrc.org/index.php/Asym.

[6] Advanced Linux Sound Architecture. http://www.alsa-project.org/.

[7] Analog Realtime Synthesizer. http://www.arts-project.org/.

[8] David G. Andersen, Alex C. Snoeren, and Hari Balakrishnan. Best-Path vs. Multi-Path Overlay Routing. In *Proceedings of the Internet Measurement Conferencee*, October 2003.

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

[10] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *Proceedings of the* 34th *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, August 2007.

[11] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. Vision, Issues, and Architecture for Nomadic Computing. *IEEE Personal Communications*, 2(6):14–27, December 1995.

[12] Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS/Performance)*, pages 279–290, June 2001.

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

[14] Ricardo Baratto, Shaya Potter, Gong Su, and Jason Nieh. MobiDesk: Mobile Virtual Desktop Computing. In *Proceedings of the* 10th *Annual ACM International Conference on Mobile Computing and Networking (MobiCom)*, September - October 2004.

[15] BBC News. http://news.bbc.co.uk.

[16] Calista Virtual Desktop. http://www.calistatechnologies.net/.

[17] M. Chapman. http://www.rdesktop.org.

[18] Bernd Oliver Christiansen and Klaus Erik Schauser. Fast Motion Detection for Thin Client Compression. In *Proceedings of the Data Compression Conference (DCC)*, April 2002.

[19] Bernd Oliver Christiansen, Klaus Erik Schauser, and Malte Münke. A Novel Codec for Thin Client Computing. In *Proceedings of the Data Compression Conference (DCC)*, March 2000.

[20] Bernd Oliver Christiansen, Klaus Erik Schauser, and Malte Münke. Streaming Thin Client Compression. In *Proceedings of the Data Compression Conference (DCC)*, March 2001.

[21] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[22] Citrix Application Delivery. http://www.citrix.com/English/ps2/products/feature.asp?contentID=683723.

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

[24] CrossLoop. http://crossloop.com/.

[25] Brian Craig Cumberland, Gavin Carius, and Andrew Muir. *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference.* Microsoft Press, Redmond, WA, July 1999.

[26] Diego López de Ipiña. *Visual Sensing and Middleware Support for Sentient Computing*. PhD thesis, Cambridge University, January 2002.

[27] Delegate. http://www.delegate.org.

[28] Desktone. http://www.desktone.com.

[29] Distributed Multihead X Project. http://dmx.sourceforge.net/.

[30] DoS-Resistant Internet Working Group Meetings. http://www.communicationsresearch.net/dos-resistant, February 2005.

[31] 'Dumb Terminals' Can Be a Smart Move. http://online.wsj.com/public/article/SB117011971274291861-oJ6FWrnA8NMPfMXw3vBILth1EiE_20080129.html?mod=blogs.

[32] Efforts pursue separate paths to streamlined PCs. http://www.eetimes.com/showArticle.jhtml?articleID=199904407.

[33] Enlightenment Sound Daemon. http://www.tux.org/~ricdude/EsounD.html.

[34] Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL. http://developer.nvidia.com/object/fast_texture_transfers.html.

[35] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, March 1974.

[36] Fog Creek Copilot. http://www.copilot.com.

[37] For Networks, Thin Is In. http://www.nytimes.com/2007/09/12/technology/techspecial/12thin.html.

[38] Armando Fox, Ian Goldberg, Steven D. Gribble, and David C. Lee. Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the 3Com PalmPilot. In *Proceedings of Middleware '98, Lake District, England, September 1998*, September 1998.

[39] Eric T. Freeman. *The Lifestreams Software Architecture*. PhD thesis, Yale University, May 1997.

[40] GNOME accessibility project. http://developer.gnome.org/projects/gap/.

[41] Jim Gettys. Personal communication, July 2004.

[42] Werner Geyer, Heather Richter, Ludwin Fuchs, Tom Frauenhofer, Shahrokh Daijavad, and Steven Poltrock. A team collaboration space supporting capture and access of virtual meetings. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, September - October 2001.

[43] Jeffrey. M. Gilbert and Robert 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.

[44] Global Crossing's IP Network Performance. http://www.globalcrossing.com/network/network_performance_current.aspx.

[45] Google Browser Sync. http://www.google.com/tools/firefox/browsersync/index.html.

[46] GoToMyPC. http://www.gotomypc.com/.

[47] GraphOn GO-Global. http://www.graphon.com.

[48] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. Improving the Reliability of Internet Paths with One-hop Source Routing. In *Proceedings of the* 6th *Symposium on Operating Systems Design & Implementation (OSDI)*, December 2004.

[49] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation, RFC 2861. ACIRI, June 2000.

[50] Health Insurance Portability and Accountability Act. http://www.hhs.gov/ocr/hipaa/.

[51] http_load. http://www.acme.com/software/http_load/.

[52] George V. Hulme. Extortion online. *Information Week*, September 13 2004.

[53] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In *Proceedings of the* 29th *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, July 2002.

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

[55] Yutaka Ishibashi and Shuji Tasaka. A comparative survey of synchronization algorithms for continuous media in network environments. In *Proceedings of the* 25th *Annual IEEE Conference on Local Computer Networks*, November 2000.

[56] Istanbul. http://live.gnome.org/Istanbul.

[57] Wenyu Jiang, Kazuumi Koguchi, and Henning Schulzrinne. QoS Evaluation of VoIP End-points. In *Proceedings of IEEE International Conference on Communications (ICC)*, May 2003.

[58] Anupam Joshi. On proxy agents, mobility, and web access. *Mobile Networks and Applications*, 5(4):233–241, December 2000.

[59] Jussi Kangasharju, Young Gap Kwon, and Antonio Ortega. Design and Implementation of a Soft Caching Proxy. *Computer Networks and ISDN Systems*, 30(22–23):2113–2121, November 1998.

[60] K Desktop Environment. http://www.kde.org.

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

[62] Joeng Kim, Ricardo Baratto, and Jason Nieh. An Application Streaming Service for Mobile Handheld Devices. In *Proceedings of the IEEE International Conference on Services Computing (SCC)*, September 2006.

[63] Joeng Kim, Ricardo Baratto, and Jason Nieh. pTHINC: A Thin-Client Architecture for Mobile Wireless Web. In *Proceedings of the Fifteenth International World Wide Web Conference (WWW)*, May 2006.

[64] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[65] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux Implementation of a Log-structured File System. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, July 2006.

[66] Oren Laadan. *A Personal Virtual Computer Recorder*. PhD. Thesis, Computer Science Deparment, Columbia University, April 2011.

[67] Oren Laadan, Ricardo Baratto, Dan Phung, Shaya Potter, and Jason Nieh. DejaView: A Personal Virtual Computer Recorder. In *Proceedings of the* 21th *ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[68] Oren Laadan and Jason Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

[69] 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.

[70] 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.

[71] *Laplink*. http://www.laplink.com/.

[72] Sheng Feng Li, Quentin Stafford-Fraser, and Andy Hopper. Integrating Synchronous and Asynchronous Collaboration with Virtual Network Computing. *IEEE Internet Computing*, 4(3):26–33, May - June 2000.

[73] Alexander Ya li Wong and Margo Seltzer. Operating System Support for Multi-User, Remote, Graphical Interaction. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[74] Linphone. http://www.linphone.org.

[75] Linux Terminal Server Project. http://www.ltsp.org/.

[76] Anuj Maheshwari, Aashish Sharma, Krithi Ramamritham, and Prashant Shenoy. TranSquid: Transcoding and caching proxy for heterogenous ecommerce environments. In *Proceedings of the* 12th *IEEE Workshop on Research Issues in Data Engineering (RIDE)*, February 2002.

[77] Steven McCanne and Van Jacobson. vic: A Flexible Framework for Packet Video. In *Proceedings of the* 3rd *ACM International Conference on Multimedia*, November 1995.

[78] Media Application Server. http://www.mediaapplicationserver.net/.

[79] MediaMall Technologies. http://www.themediamall.com/.

[80] Microsoft DirectSound. http://msdn2.microsoft.com/en-us/library/bb219818(VS.85).aspx.

[81] Microsoft DirectX. http://www.microsoft.com/directx.

[82] Microsoft Remote Desktop Protocol: Basic Connectivity and Graphics Remoting Specification. http://msdn2.microsoft.com/en-us/library/cc240445.aspx.

[83] Microsoft Remote Desktop Protocol: Graphics Device Interface (GDI) Acceleration Extensions. http://msdn2.microsoft.com/en-us/library/cc241537.aspx.

[84] Microsoft Remote Desktop Protocol: Graphics Device Interface (GDI) Acceleration Extensions: Alternate Secondary Orders. http://msdn2.microsoft.com/en-us/library/cc241625.aspx.

[85] Microsoft RemoteFX. http://technet.microsoft.com/en-us/library/ff817578(WS.10).aspx.

[86] Microsoft Terminal Services RemoteApp. http://go.microsoft.com/fwlink/?LinkId=84895).

[87] Mozilla.org. https://bugzilla.mozilla.org/show_bug.cgi?id=372201.

[88] .NET VNC Viewer for PocketPC. http://dotnetvnc.sourceforge.net/.

[89] Windows NetMeeting. http://www.microsoft.com/windows/netmeeting.

[90] Netperf. http://www.netperf.org/.

[91] Network Audio System. http://radscan.com/nas.html.

[92] 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.

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

[94] Jakob Nielsen. *Designing Web Usability*. New Riders Publishing, Indianapolis, IN, December 1999.

[95] NoMachine NX. http://www.nomachine.com.

[96] Kenneth Ocheltree, Steven Millman, David Hobbs, Martin McDonnell, Jason Nieh, and Ricardo Baratto. Net2Display: A Proposed VESA Standard for Remoting Displays and I/O Devices over Networks. In *Proceedings of the Americas Display Engineering and Applications Conference (ADEAC)*, October 2006.

[97] OpenGL. http://www.opengl.org.

[98] OpenMemex. http://openmemex.devjavu.com.

[99] OpenSSL Project. http://www.openssl.org.

[100] Opera Mini Browser. http://www.opera.com/products/mobile/operamini/.

[101] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[102] Open Sound System. http://www.opensound.com/.

[103] oSync: Opera Sync and Backup. http://osync.sourceforge.net/.

[104] Pluggable Authentication Modules (PAM) for Linux. http://www.kernel.org/pub/linux/libs/pam/.

[105] David Patterson and Jim Gray. A Conversation with Jim Gray. *ACM Queue*, 1(3), June 2003.

[106] PC Anywhere. http://www.pcanywhere.com.

[107] PCI Express gains I/O virtualization. http://www.virtualization.info/2006/08/pci-express-standard-to-gain-io.html.

[108] PCoIP: Teradici Corporation. http://www.teradici.com/.

[109] Colin Perkins. *RTP: Audio and Video for the Internet*. Addison-Wesley Professional, Boston, MA, June 2003.

[110] 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.

[111] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[112] Shaya Potter and Jason Nieh. WebPod: Persistent Web Browsing Sessions with Pocketable Storage Devices. In *Proceedings of the Fourteenth International World Wide Web Conference (WWW)*, May 2005.

[113] Shaya Potter and Jason Nieh. Highly Reliable Mobile Desktop Computing in Your Pocket. In *Proceedings of the IEEE Computer Society Signature Conference on Software Technology and Applications (COMPSAC)*, September 2006.

[114] PulseAudio. http://www.pulseaudio.org.

[115] Quicken Personal Finance Software. http://quicken.intuit.com.

[116] Qumranet. http://www.qumranet.com/.

[117] Robust Audio Tool. http://www-mice.cs.ucl.ac.uk/multimedia/software/rat.

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

[119] Mark Roseman and Saul Greenberg. Teamrooms: network places for collaboration. In *Proceedings of the ACM conference on Computer Supported Cooperative Work*, November 1996.

[120] Runaware.com. http://www.runaware.com.

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

[122] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

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

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

[125] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, 2nd edition, 1996.

[126] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[127] Select system call. http://en.wikipedia.org/wiki/Select_(Unix).

[128] ServerEngines. http://www.serverengines.com/.

[129] SGI OpenGL Vizserver. http://www.sgi.com/software/vizserver/.

[130] ShowMyPC.com. http://showmypc.com/.

[131] SIMtone Corporation. http://www.xdsinc.net/.

[132] Skype. http://www.skype.com.

[133] Peter J. Spellman, Jane N. Mosier, Lucy M. Deus, and Jay A. Carlson. Collaborative virtual workspace. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, November 1997.

[134] Standard aims to bolster thin client PCs. http://www.eetimes.com/showArticle.jhtml?articleID=199903238.

[135] Angelos Stavrou, Ricardo Baratto, Angelos Keromytis, and Jason Nieh. A2M: Access-Assured Mobile Desktop Computing. In *Proceedings of the* 12th *Information Security Conference (ISC)*, September 2009.

[136] Simon Stegmaier, Joachim Diepstraten, Manfred Weiler, and Thomas Ertl. Widening the remote visualization bottleneck. In *Proceedings of the* 3rd *International Symposium on Image and Signal Processing and Analysis (ISPA)*, September 2003.

[137] Simon Stegmaier, Marcelo Magallón, and Thomas Ertl. A Generic Solution for Hardware-Accelerated Remote Visualization. In *Proceedings of the symposium on Data Visualisation (VISSYM)*, May 2002.

[138] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, July 1995.

[139] Ao-Jan Su, David Choffnes, Aleksandar Kuzmanovic, and Fabian E. Bustamante. Drafting Behind Akamai (Travelocity-Based Detouring). In *Proceedings of ACM SIGCOMM*, September 2006.

[140] Gong Su. *MOVE: Mobility with Persistent Network Connections*. PhD. Thesis, Computer Science Deparment, Columbia University, October 2004.

[141] Sun Ray Clients. http://www.sun.com/sunray.

[142] Sun Secure Global Desktop Software. http://www.sun.com/software/products/sgd/.

[143] Thin-Client market to fatten up, IDC says. http://news.com.com/2100-1003-5077884.html.

[144] TightVNC. http://www.tightvnc.com/.

[145] TPC-W Java Implementation. http://mitglied.lycos.de/jankiefer/tpcw/.

[146] Thomas E. Truman, Trevor Pering, Roger Doering, and Robert W. Brodersen. The InfoPad Multimedia Terminal: A Portable Device for Wireless Information Access. *IEEE Transactions on Computers*, 47(10):1073–1087, October 1998.

[147] International Telecommunication Union. ITU-T Recommendation G.114: One-way transmission time. May 2003.

[148] Video Electronics Standards Association (VESA). http://www.vesa.org.

[149] VirtualGL. http://www.virtualgl.org.

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

[151] VMware, Inc. http://www.vmware.com.

[152] VMware Movie Capture. http://www.vmware.com/support/ws5/doc/ws_running_capture.html.

[153] VMware View. http://www.vmware.com/products/view/.

[154] VMware VMotion overview. http://www.vmware.com/products/vi/vc/vmotion.html.

[155] vnc2swf. http://www.unixuser.org/~euske/vnc2swf/.

[156] VNC Proxy. http://vncproxy.sourceforge.net/.

[157] vncrec. http://www.sodan.org/~penny/vncrec/.

[158] Grant Wallace and Kai Li. Virtually Shared Displays and User Input Devices. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[159] WengoPhone. http://www.wengophone.com/.

[160] Why PCI Express Architecture for Graphics? http://www.intel.com/technology/pciexpress/devnet/docs/PCIExpressGraphics1.pdf.

[161] Wi-Fi Planet. http://www.wi-fiplanet.com/.

[162] Windows XP Remote Assistance. http://www.microsoft.com/windowsxp/using/helpandsupport/learnmore/remoteassist/intro.mspx.

[163] Windows Server 2003 Terminal Services. http://www.microsoft.com/windowsserver2003/technologies/terminalservices/default.mspx.

[164] Wink. http://www.debugmode.com/wink/.

[165] Worldwide Enterprise Thin Client Forecast and Analysis, 2002-2007: The Rise of Thin Machines. http://www.idcresearch.com/getdoc.jhtml?containerId=30016.

[166] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2(1), March 2006.

[167] Dapeng Wu, Yiwei Thomas Hou, Wenwu Zhu, Ya-Qin Zhang, and Jon M. Peha. Streaming Video over the Internet: Approaches and Directions. 11(3):282–300, March 2001.

[168] Wyse Technology. http://www.wyse.com.

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

[170] X.org Foundation. http://www.x.org/.

[171] xvidcap. http://xvidcap.sourceforge.net/.

[172] 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.

[173] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[174] Lei Zhang. *Implementing Remote Display on Commodity Operating Systems*. M.S. Thesis, Computer Science Deparment, Columbia University, January 2006.

[175] Zinkmo: IE Favorite and Firefox Bookmark Synchronization and Sharing. http://www.zinkmo.com/.

Appendix A

THINC Protocol specification

This section provides a formal specification of the THINC remote display protocol. The protocol is designed to provide efficient remote display across LAN and WAN environments with very simple clients. It is built around a server-push model with minimum synchronization. Finally, the protocol assumes a reliable transport channel is used to transport all protocol messages, e.g., TCP.

This specification is organized as follows. Section A.1 provides a description of the basic packet format of all protocol messages. Section A.2 presents the protocol messages used by clients to obtain a connection to a running THINC server. Finally, Section A.3 describes the messages used during normal operation, i.e., after the handshake has succeeded.

A.1 Packet Format

The packet format of all THINC protocol messages is shown in Figure A.1.

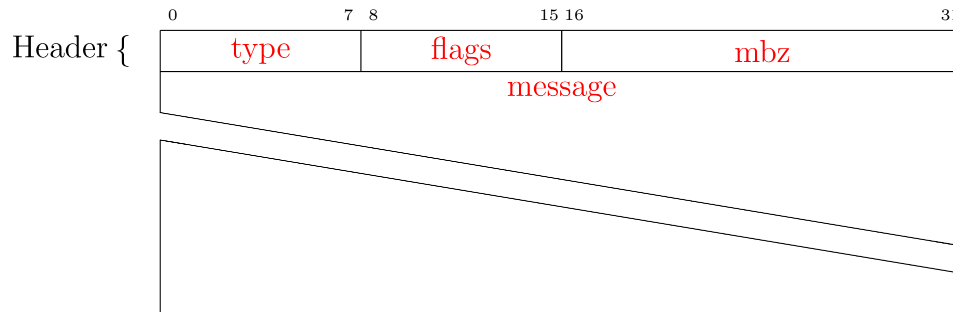


Figure A.1 – THINC packet format

A.1.1 Message Type and Flags

The first byte of the header is the packet type. The second byte is a set of flags which are specific to the message type. The type namespace is separate for server-to-client messages and client-to-server messages. In addition, the handshake protocol has a namespace that is separate from that of normal protocol messages. Tables describing each of the message types and flags can be found in Sections A.2 and A.3.

A.1.2 Unused Field (mbz)

The last 16 bits of the header are currently unused. In a previous version, they were used to store the length of the message. However, this imposed a limitation on the message size of 64KB. The current version of the protocol imposes no limits on the maximum length of a message. Instead, the protocol specification defines the length of each message type, and if additional, type-specific data accompany a message, it is expected that this length will be stored as part of the message. This approach was taken as it allows for easy modification of the protocol, which is crucial for our prototyping purposes. However, it may not be an optimal approach from a performance point of view, since multiple read calls will be needed in order to read any particular message.

A.2 Handshake Protocol

When clients try to connect to a running THINC server, a handshake process takes place before they are allowed to access the desktop, receive display updates, and send input events. This section describes the process. Implementation details and the motivation for this process can be found in Section 2.6.3.

To establish a connection, a client has to go through the following stages.

A.2.1 Version verification

The server sends a version string. The client verifies this string, and if acceptable, sends back its own version string. The server reads the client's version string, and if acceptable moves on to the next stage. Otherwise it closes the connection and the handshake is terminated. A version string is sent as a NULL-terminated ASCII string. Currently, the only version string accepted is:

THINC_0.2	0

A.2.2 Security Handshake

Next, the client and server must set up a secure channel to communicate. By default THINC uses an encrypted and authenticated channel. However, both of these options may be disabled at handshake time. First, the server sends a message containing what security features it supports. This packet has type `T-SERV-SEC-CAPS`[1].

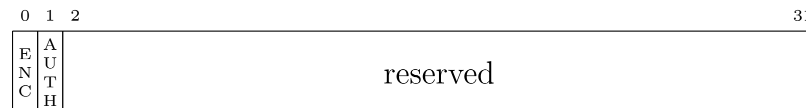
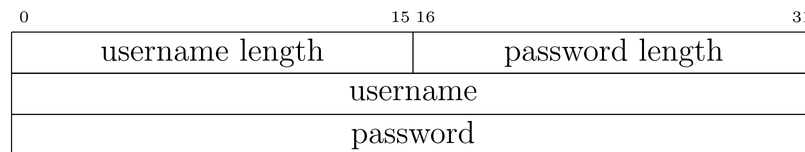


Figure A.2 – Security capabilities packet

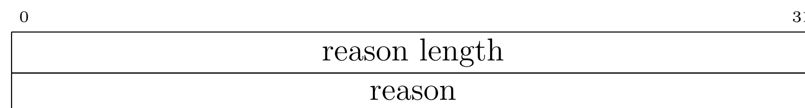
[1]A table summarizing all message types may be found at the end of this section

The ENC bit means the server supports encrypting the communication channel. The AUTH bit means the server requires the client to authenticate before it can proceed any further. The client replies with its own security capabilities, a message with the same structure, but type T-CLIENT-SEC-CAPS. The client capabilities should be a subset of the capabilities sent by the server. Finally, if the server accepts the set of capabilities, it replies with a T-SESS-SEC-CAPS message. This message provides both a confirmation from the server that the client can proceed, and a summary of the security capabilities for the session.

If the ENC bit is set in the session capabilities, the encrypted channel is set up. If the AUTH bit is set in the session capabilities, authentication information is exchanged. While the current protocol specification uses a very simple username/password authentication scheme, any standard authentication protocol can be used. In the current scheme, the client sends a T-CLIENT-AUTH message with the authentication information:

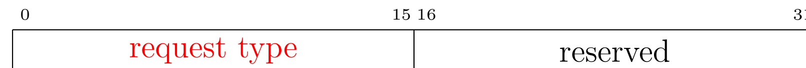


If authentication succeeds, the server replies with a T-SERV-OK message. Otherwise, a T-SERV-NOTOK message is sent back and the connection is closed. The reason is an optional string, specifying the reason for rejecting the client.



A.2.3 Parameter Negotiation

After the secure channel has been established, the client starts sending requests to the server to negotiate aspects of the connection, using the T-CLIENT-REQ message:



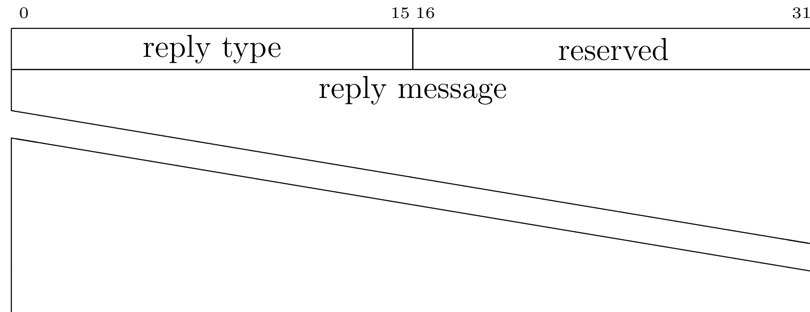
The first 16 bits of the packet represent the identifier of the request being sent. The next 16 bits are unused for now, and reserved for future use. The following table lists the currently available requests:

Type	Name	Description
3	T-REQ-FBINFO	Basic framebuffer information
4	T-REQ-CURSOR	Cursor information and data
5	T-REQ-FBDATA	Contents of the framebuffer
6	T-REQ-ENCODER	How is image data encoded
7	T-REQ-CACHESZ	Size of caches in use
8	T-REQ-VIDEO	Does the server support video playback?
9	T-REQ-NOVIDEO	Client informing the server that it does not support video
10	T-REQ-VIDEO-SERV-FMTS	List of video formats supported by the server
11	T-REQ-VIDEO-CLIENT-FMTS	Client informs server of its video formats
12	T-REQ-KEEPALIVE	Does the client support keepalives? Does the server support them?

Table A.1 – List of client requests

The only request which sends additional data beyond the simple request header is T-REQ-VIDEO-CLIENT-FMTS, which has the same format as the T-REQ-VIDEO-SERV-FMTS reply defined below. The server replies to these requests with a T-SERV-REPLY message:

The reply type field takes the same values as the request type, plus the following extra values:

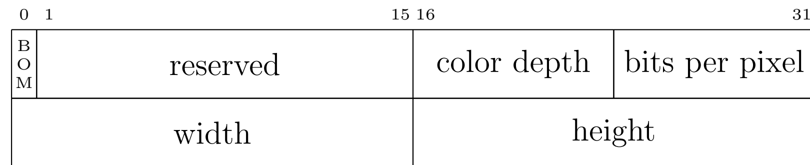


Type	Name	Description
0	T-REPLY-OK	Positive reply
1	T-REPLY-NOTOK	Negative reply (non-fatal)
2	T-REPLY-UNKNOWN	Unknown request sent

Table A.2 – List of server replies

A.2.3.1 Packet Format of Server Replies

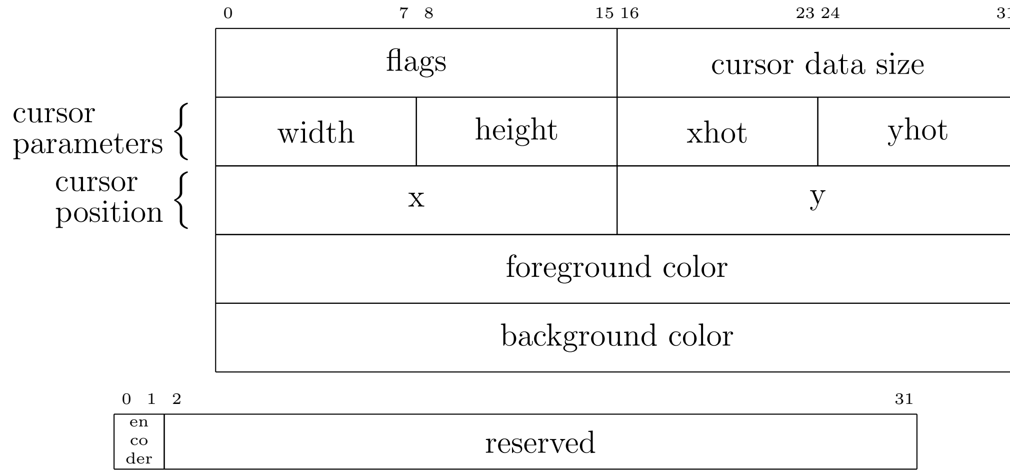
- T-REQ-FBINFO



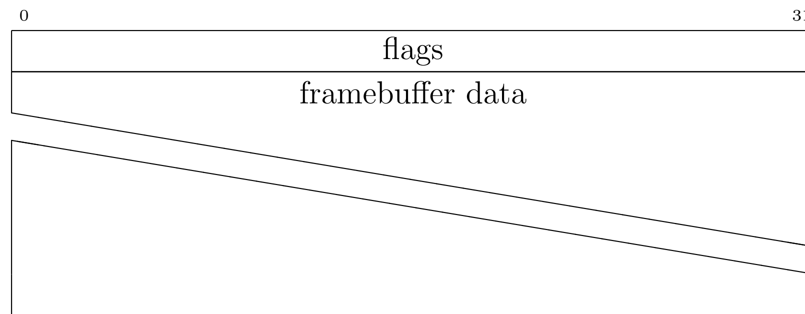
- T-REQ-CURSOR
- T-REQ-ENCODER

  Valid values for the encoder field:

  T-ENCODER-NONE: 0x00
  T-ENCODER-PNG: 0x01
  T-ENCODER-ZLIB: 0x02



- `T-REQ-FBDATA`

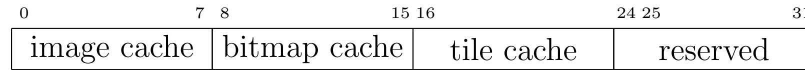


Valid flags:

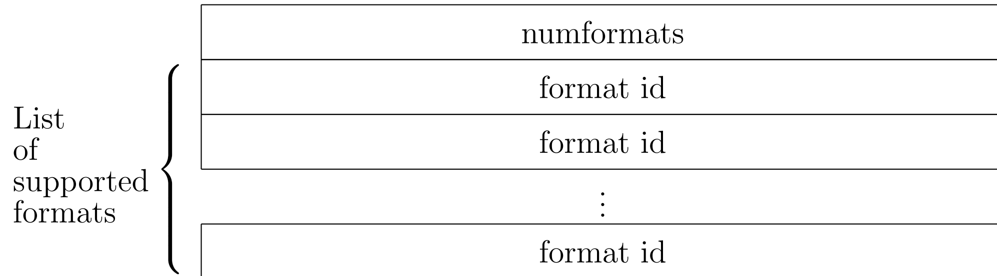
`T-FB-COMPRESSED` `0x01` The framebuffer data is compressed.

- `T-REQ-CACHESZ`

Each field represents the size of a cache, represented as the number of bits that are to be used as an identifier in the cache. Thus, the cache will have 2^{size} entries. `image` is the cache for `RAW` updates. `bitmap` is the cache for `BITMAP` updates. `tile` is the cache for `PFILL` updates.



- `T-REQ-VIDEO-SERV-FMTS` and `T-REQ-VIDEO-CLIENT-FMTS`



Valid formats:

```
T-FOURCC-YV12 0x32315659 ('2'|'1'|'V'|'Y')
T-FOURCC-YUY2 0x32595559 ('2'|'Y'|'U'|'Y')
T-FOURCC-UYVY 0x59565955 ('Y'|'V'|'U'|'Y')
```

Once the client is finished sending requests, it sends the `T-CLIENT-DONE` message and the handshake is finished.

A.2.4 Summary of Handshake Messages

Table A.3 summarizes all the messages used during the handshake.

Table A.3 – List of handshake protocol messages. The first column represents the value of the type field in the message header. The second column is the canonical name of the message. Server to client messages are presented first, followed by client to server messages

Type	Name	Description
Server → client messages		
1	`T-SERV-OK`	Server positive ack
2	`T-SERV-NOTOK`	Server negative ack. After sent, the connection will be closed

Type	Name	Description
3	T-SERV-SEC-CAPS	Server security capabilities
4	T-SESS-SEC-CAPS	Session security capabilities
5	T-SERV-REPLY	Server reply to client request
Client → server messages		
1	T-CLIENT-OK	Client positive ack
2	T-CLIENT-NOTOK	Client negative ack
3	T-CLIENT-SEC-CAPS	Client security capabilities
4	T-CLIENT-AUTH	Client authentication information
5	T-CLIENT-REQ	Client handshake request
6	T-CLIENT-DONE	Client is done with handshake

A.3 Remote Display Protocol

Table A.4 lists all the THINC protocol messages.

Table A.4 – List of protocol messages. The first column represents the value of the type field in the message header. The second column is the canonical name of the message. Server to client messages are presented first, followed by client to server messages

Type	Name	Description
Server → client messages		
9	T-SPING	Keepalive message
12	T-FB-RAW	RAW
13	T-FB-COPY	COPY
14	T-FB-SFILL	SFILL
15	T-FB-PFILL	PFILL
16	T-FB-GLYPH	GLYPH
17	T-FB-BILEVEL	BILEVEL
110	T-VIDEO-START	Start playing video
111	T-VIDEO-NEXT	Next video frame

Type	Name	Description
112	T-VIDEO-END	End video playback
113	T-VIDEO-MOVE	Change the position of a video
114	T-VIDEO-SCALE	Change the destination size of a video
115	T-VIDEO-RESIZE	Change the source size of a video
Client → server messages		
11	T-EV-MOTION	Mouse motion event
12	T-EV-BUTTON	Mouse button event
13	T-EV-KEYB	Keyboard event
110	T-VIDEO-STARTOK	Acknowledge succesful video playback start

A.3.1 Server Messages

- `T-SPING`: A keep alive message. The message does not carry anything beyond the header. The purpose is to try and keep the server to client link from getting idle.

  Size: 0 bytes

- `T-FB-RAW`: A raw rectangular frame buffer update. Carries the description of the location and size of the update.

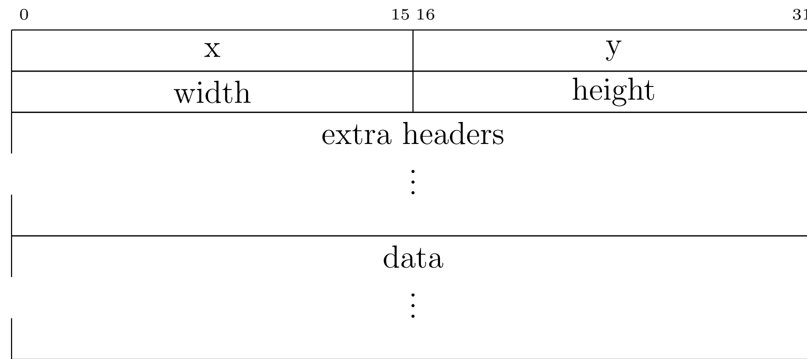
  Size: 8 bytes

  Data: height lines of size width*bpp, representing the new contents of the described frame buffer region

  Header Flags:

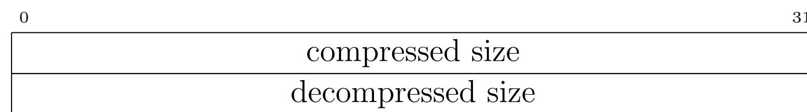
Name	Value	Description
T-FB-RAW-COMPRESSED	0x01	Data is compressed
T-FB-RAW-RESIZED	0x02	Data has been resized
T-FB-RAW-CACHED	0x04	Data is cached
T-FB-RAW-ADDCACHE	0x08	Data should be cached

Packet Format:

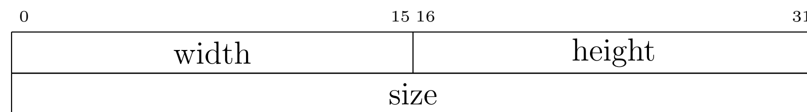


Extra Headers:

- If T-FB-RAW-COMPRESSED is set, the following header contains information about the compressed data:



- If T-FB-RAW-RESIZED is set, the following header contains information about the resized data:



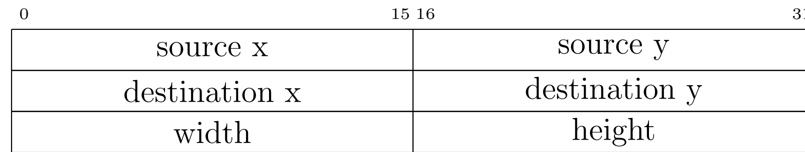
- If T-FB-RAW-CACHED or T-FB-RAW-ADDCACHE are set, the following header has the information to retrieve from or add the data to the cache, respectively:

0–31
cache id

- `T-FB-COPY`: Tells the client to copy the area of size width × height, from the source coordinates to the destination coordinates

  Size: 12 bytes

  Packet Format:



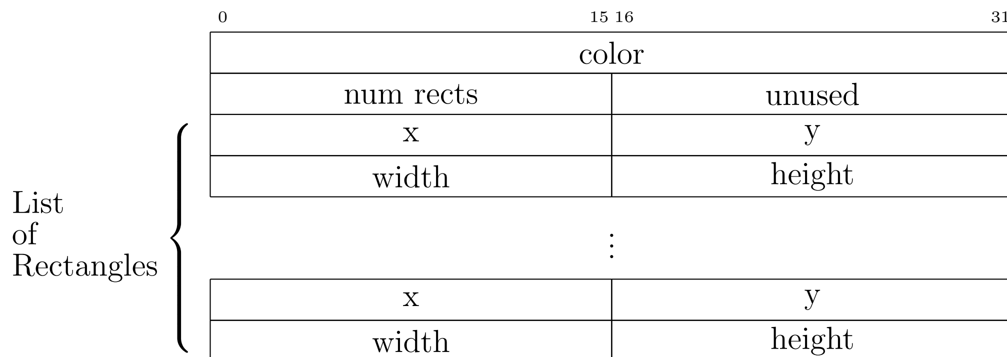
0–15	16–31
source x	source y
destination x	destination y
width	height

- `T-FB-SFILL`: Fill all the passed rectangles with the pixel value color

  Size: 6 bytes

  Data: numrects rectangles

  Packet Format:



	0–15	16–31
	color	
	num rects	unused
List of Rectangles	x	y
	width	height
	⋮	
	x	y
	width	height

- `T-FB-PFILL`: Tile a pixmap along a list rectangles. (x, y) tells the client where's the origin of the tiling region. Note that the rectangles may (and most of the time will) describe a subregion of the tiling region, for example if only the lower left corner of the region needs to be painted. This information is necessary for proper alignment of the pixmap. In other words, the client should not just paint the pixmap starting at the origin of each of the rectangles, clipping and

alignment needs to be done.

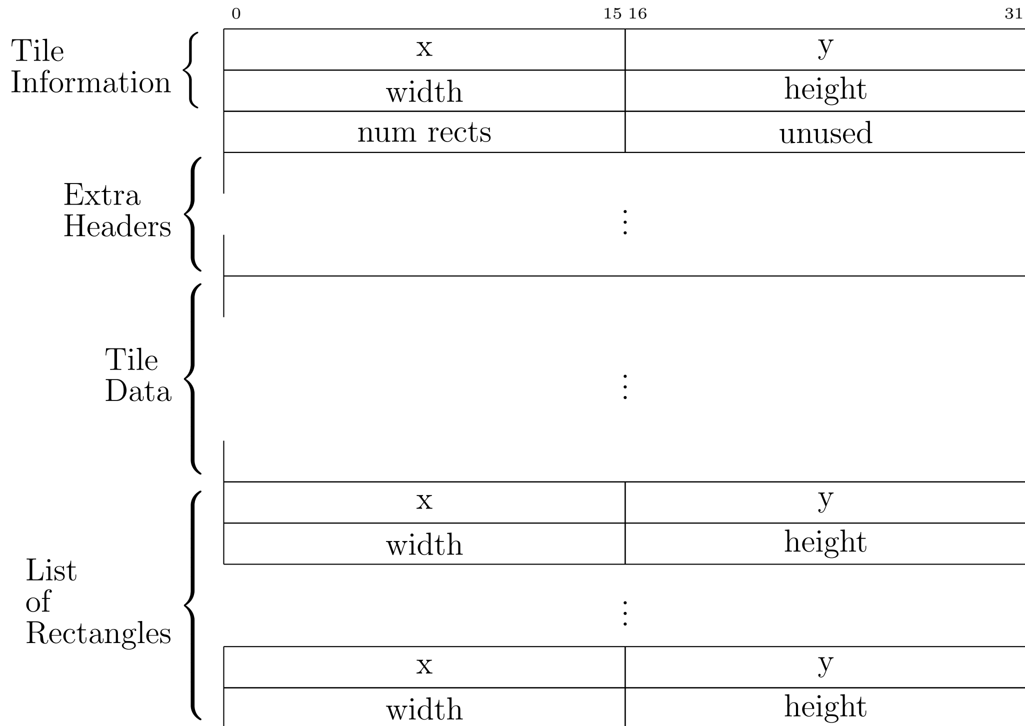
Size: 10 bytes

Data: The tile data followed by numrects rectangles

Header Flags:

Name	**Value**	**Description**
T-FB-PFILL-RESIZED	0x01	Tile data has been resized
T-FB-PFILL-CACHED	0x02	Tile data is cached
T-FB-PFILL-ADDCACHE	0x04	Tile data should be cached

Packet Format:



Extra Headers: The same extra headers defined for T-FB-RAW.

- T-FB-GLYPH: Fill the passed rectangles using the bitmap as a stipple to fill the region: If there is a 1 on the bitmap the color specified in the message should be applied. If there is a 0 no operation should be performed on that pixel.

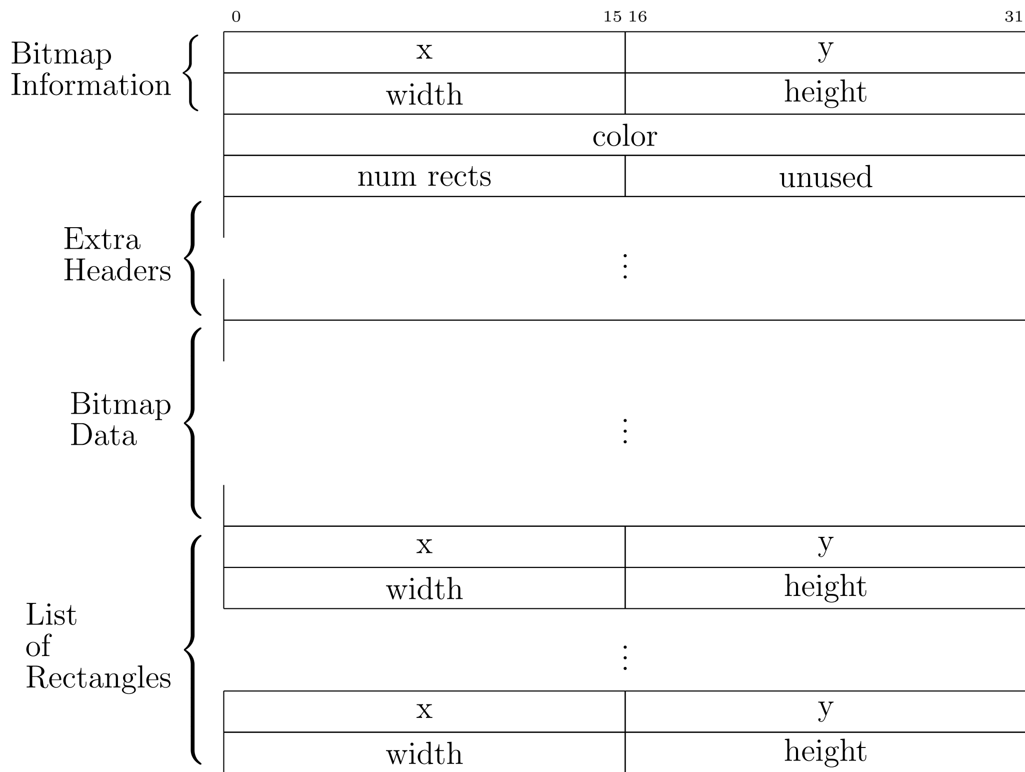
  Size: 12 bytes

Data: The bitmap to use as stipple. Bitmap's size is $\lceil(\text{width}/8)\times\text{height}\rceil$. Followed by list of rectangles to fill.

Header Flags:

Name	Value	Description
`T-FB-BITMAP-ADDCACHE`	0x02	Bitmap data should be cached
`T-FB-BITMAP-CACHED`	0x04	Bitmap data is cached

Packet Format:



Extra Headers: The same extra headers defined for `T-FB-RAW`.

- `T-FB-BILEVEL`: Fill the passed rectangles using the bitmap as a stipple to fill the region: If there is a 1 on the bitmap the foreground color specified in the message should be applied. If there is a 0 the background color should be applied

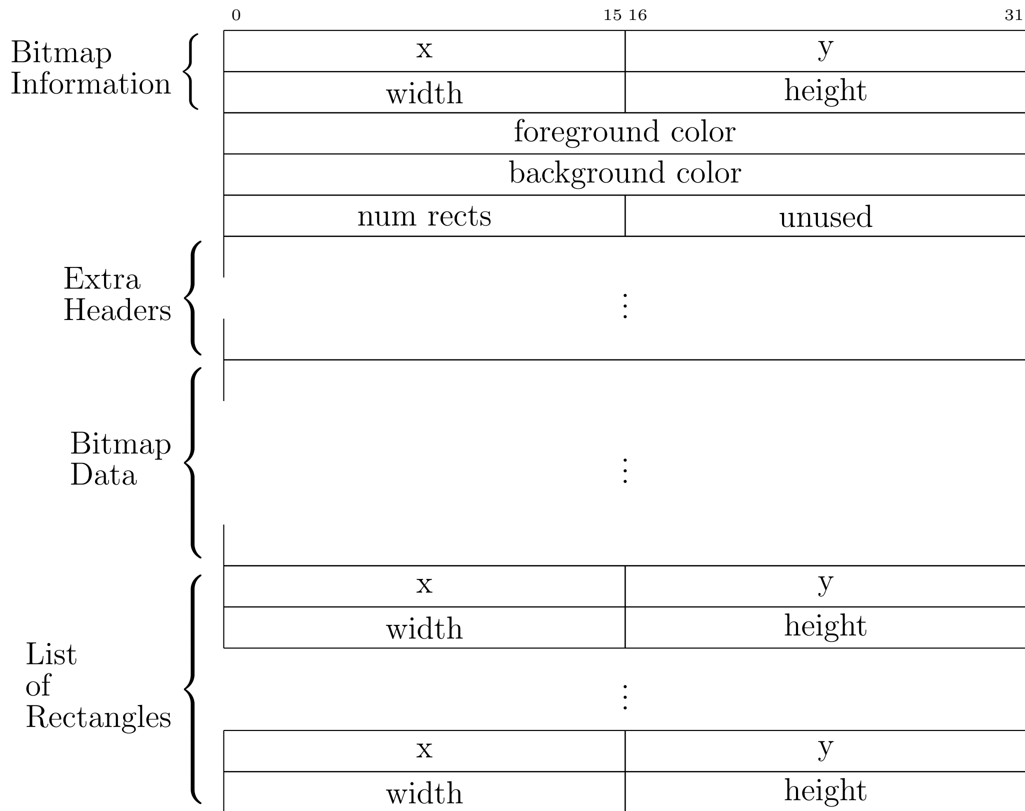
  Size: 16 bytes

Data: The bitmap to use as stipple. Bitmap's size is $\lceil(\text{width}/8)\times\text{height}\rceil$. Followed by list of rectangles to fill.

Header Flags:

Name	Value	Description
`T-FB-BITMAP-ADDCACHE`	0x02	Bitmap data should be cached
`T-FB-BITMAP-CACHED`	0x04	Bitmap data is cached

Packet Format:



Extra Headers: The same extra headers defined for `T-FB-RAW`.

- `T-VIDEO-START`: Asks the client to start playing video which will come in the format specified in the message. It also defines an ID for this video stream which will be used in subsequent messages. Since the client hardware is used to perform scaling, the video has two dimensions associated with it: the size of

the video data, and the size at which it should be displayed

Size: 20 bytes

Packet Format:

0–15	16–31
video id	
format id	
x	y
width	height
destination width	destination height

- `T-VIDEO-NEXT`: Contains the video data for the next frame in the specified video

  Size: 12 bytes

  Packet Format:

0–31
video id
size
timestamp
video data
⋮

- `T-VIDEO-END`: Finish playback of the specified video

  Size: 4 bytes

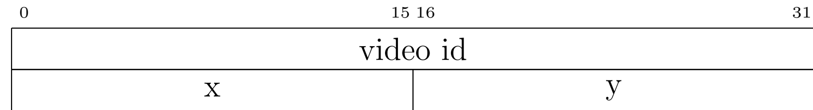
  Packet Format:

0–31
video id

- `T-VIDEO-MOVE`: Change the coordinates of the specified video

  Size: 8 bytes

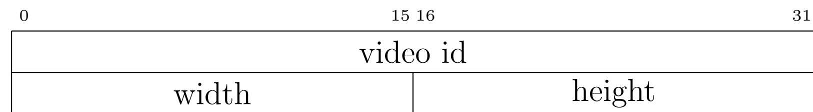
  Packet Format:



- `T-VIDEO-SCALE`: Change the destination dimensions of the specified video

  Size: 8 bytes

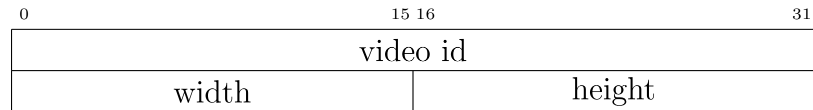
  Packet Format:



- `T-VIDEO-RESIZE`: Change the source dimensions of the specified video

  Size: 8 bytes

  Packet Format:

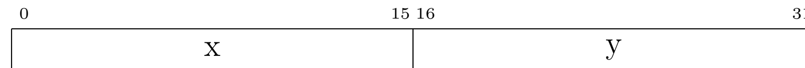


A.3.2 Client Messages

- `T-EV-MOTION`: A mouse motion event being reported by the client. Position is reported as absolute (x,y) coordinates

  Size: 4 bytes

  Packet Format:



- `T-EV-BUTTON`: A mouse button was pressed/released. If the `DOWN` bit is set, the button was pressed. Otherwise, it was released

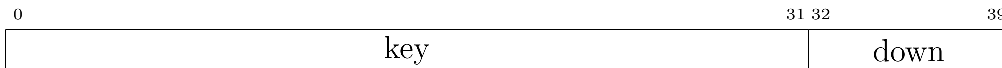
  Size: 1 byte

  Packet Format:

- `T-EV-KEYB`: The specified key was pressed (`down` field is 1) or released

  Size: 5 bytes

  Packet Format:



- `T-VIDEO-STARTOK`: Response to `T-VIDEO-START`. If `ok` is 1, client can play the video. Otherwise, the server should refrain from sending more video data

  Size: 5 bytes

  Packet Format: