THINC: A Remote Display Architecture for Thin-Client Computing

Ricardo A. Baratto, Jason Nieh and Leo Kim
{ricardo, nieh, lnk2101}@cs.columbia.edu
Department of Computer Science
Columbia University
Technical Report CUCS-027-04
July 2004

Abstract

Rapid improvements in network bandwidth, cost, and ubiquity combined with the security hazards and high total cost of ownership of personal computers have created a growing market for thin-client computing. We introduce THINC, a remote display system architecture for high-performance thin-client computing in both LAN and WAN environments. THINC transparently maps high-level application display calls to a few simple low-level commands which can be implemented easily and efficiently. THINC introduces a number of novel latency-sensitive optimization techniques, including offscreen drawing awareness, command buffering and scheduling, non-blocking display operation, native video support, and server-side screen scaling. We have implemented THINC in an XFree86/Linux environment and compared its performance with other popular approaches, including Citrix MetaFrame, Microsoft Terminal Services, SunRay, VNC, and X. Our experimental results on web and video applications demonstrate that THINC can be as much as five times faster than traditional thin-client systems in high latency network environments and is capable of playing full-screen video at full frame rate.

1 Introduction

In the last two decades, the centralized computing model of mainframe computing has shifted toward the more distributed model of desktop computing. But as these personal desktop computers become prevalent in today's large corporate and government organizations, the total cost of owning and maintaining them has become unmanageable. This problem is exacerbated by the growing use of mobile laptop computers and handheld devices to store and process information, which poses additional administration and security issues. These mobile devices often contain sensitive data that must be carefully secured, yet the devices themselves must travel in insecure environments where they can be easily damaged, lost, or stolen. This management and security problem is particularly important for the medical community, given the increasing use of computing in medicine, the urgent need to comply with HIPAA regulations[18], and the huge privacy consequences for compromised patient data. As a result of the rising management complexity and security hazards inherent in the current computing model, there is a growing movement to leverage continued improvements in network bandwidth, cost, and ubiquity to return to a more centralized, secure, and easier-to-manage computing strategy. Thin-client computing is an embodiment of that movement.

A thin-client computing system consists of a server and a client that communicate over a network using a remote display protocol. The protocol allows graphical displays to be virtualized and served across a network to a client device, while application logic is executed on the server. Using the remote display protocol, the client transmits user input to the server, and the server returns screen updates of the user interface of the applications to the client. Since data and applications are accessed from a single remote location, several significant advantages over desktop computing are achieved. The client is essentially a stateless appliance that does not need to be backed up or restored, requires almost no maintenance or upgrades, and does not store any sensitive data that can be lost or stolen. Server resources can be physically secured in protected data centers and centrally administered, with all the attendant benefits of easier maintenance and cheaper upgrades. Moreover, computing resources can be shared across many users, resulting in more effective utilization of computing hardware.

Given these enormous potential advantages, it is not surprising that the market for thin-client systems is expected to grow substantially over the next five years [27, 29]. However, thin-clients 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 the computing desktop. While previous thin-client approaches have focused on supporting office productivity tools in LAN environments and reducing data transfer for low bandwidth links such as ISDN and modem lines, they do not effectively support more display-intensive applications such as multimedia video, and they are not designed to operate effectively in higher latency

WAN environments.

In this context, we introduce THINC (*THin-client InterNet Computing*), a remote display architecture for thin-client computing that can provide high fidelity display and interactive performance in both LAN and WAN environments. THINC provides a virtual display driver that takes drawing commands, packetizes them, and sends them over the network to a client device to display. In doing so, THINC leverages the video display driver interface to work seamlessly with existing unmodified applications, window systems, and operating systems.

With THINC, higher-level graphics calls used by applications are transparently mapped to a small set of low-level commands that form the basis for the THINC remote display protocol. Application-level display commands are handled by novel semantic-preserving transformation optimizations, including offscreen drawing awareness and native video support. THINC's low-level commands mirror the video display driver interface and are easy to implement and accelerate using widely-available commodity video hardware on clients.

THINC also incorporates several latency-sensitive display mechanisms which give high performance even in high latency WAN environments. These include local cursor drawing support based on commodity video hardware, a push display update model that minimizes synchronization costs between client and server, shortest-job-first display command scheduling to improve response time for interactive applications, and a non-blocking drawing pipeline that integrates well with and maximizes the performance of today's single-threaded window servers. THINC also provides server-side screen scaling, which minimizes display bandwidth and processing requirements for small display handheld devices.

We have implemented THINC as a virtual display driver in the XFree86 window system and measured its performance on real applications. We have compared our THINC prototype system against several popular thin-client systems, including Citrix MetaFrame, Microsoft Terminal Services, SunRay, X, and VNC. Our experimental results on web and multimedia applications in various network environments demonstrate the importance of not just the choice of display commands used, but also how the mapping of application-level drawing commands to protocol primitives can affect performance. In this regard, THINC's approach allows it to achieve overall superior performance both in terms of application performance and network bandwidth usage. Most notably, it is capable of displaying full-screen video at full frame rate with modest resource requirements.

This paper presents the design and implementation of THINC. Section 2 presents the overall THINC system architecture. Section 3 presents THINC's mechanisms to improve system interactivity, while Section 4 describes THINC's screen scaling support for heterogeneous display devices. The implementation of THINC as a virtual display driver in the XFree86 window system is discussed in Section 5. Section 6 presents experimental results measuring THINC performance and comparing it against other popular commercial thin-client systems on a variety of web and multimedia application workloads. Section 7 discusses related work. Finally, we present some concluding remarks.

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 a pixmap rectangle in a region
BITMAP	Fill a region using a bitmap image

Table 1: THINC Protocol Display Commands

2 THINC Architecture

The THINC architecture is based on a thin-client model in which all persistent state is maintained by the server. Display updates are sent to the client only when the display content changes. These updates are stored as soft state in a local framebuffer at the client, which is used for screen refreshes and can be overwritten at any time.

2.1 THINC Protocol Commands

Within this basic architecture, one important design consideration is the choice of commands used to encode display information for transmission from the server to the client. The choices range from encoding high-level graphics calls to sending raw pixel data.

Higher-level display encodings such as those used at the application level allow the server to simply forward commands to the client. However, this forces a maintenance issue since the client must keep its own set of libraries for decoding application-level commands, thus requiring software updates for both the client and the server. Also, while high-level display encodings are thought to be more bandwidth efficient, previous studies show that this is often not the case in practice [23, 42]. Furthermore, they may be more platform-dependent and can result in additional synchronization overhead between client and server, substantially degrading display performance in WAN environments.

On the other hand, raw pixel encodings are very portable and easy to implement. 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.

The design of THINC is based on the idea that a small set of low-level display encoding commands provides a simple yet powerful low-latency mechanism that translates to superior remote display performance. THINC uses a set of commands that mimic operations commonly found in client display hardware. The five display commands used in THINC's display protocol are listed in Table 1.

These commands were selected because they are ubiquitously supported, simple to implement, and easily portable to a range of environments. They also represent a subset of operations accelerated by most graphics subsystems. Graphics acceleration interfaces such as the XFree86 XAA architecture [41] and Microsoft Windows' GDI Video Driver interface [28] use a set of operations which can be synthesized using THINC's commands. In this manner, clients need to do little more than translate protocol commands into hardware calls, and servers avoid the need to do full translation to actual pixel data, reducing the latency of display processing. Furthermore, THINC commands can capture important semantic information regarding the content of display updates so that they can be encoded in a bandwidth-efficient manner. The THINC protocol display commands allow clients to be simple and stateless and operate in a wide-range of network environments.

The five core THINC display commands are as follows. RAW is used to present 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. RAW commands are the only commands that are compressed to mitigate their impact on the network. COPY instructs the client to copy a region of the screen from its local framebuffer to another location. This command improves the user experience by accelerating scrolling and opaque window movement without having to resend screen data from the server. SFILL, PFILL, and BITMAP are commands that paint a fixed-size region on the screen. They are useful for accelerating the display of solid window backgrounds, desktop patterns, backgrounds of web pages, text drawing, and certain operations in graphics manipulation programs. SFILL fills a sizable region on the screen with a single color. PFILL replicates a tile over a screen region. BITMAP performs a fill using a bitmap as a stipple to apply a foreground and background color.

To provide higher fidelity display, all THINC commands are designed to support full 24-bit color as well as an alpha channel. The alpha channel enables THINC to use alpha blending to work with more advanced window system features that incorporate transparency, such as Mac OS X. However, most current window systems such as XFree86 do not yet provide support for advanced transparency features. Due to space constraints, and since we use the XFree86 window system for the THINC prototype implementation and experimental results, we skip the details of transparency support for this paper.

2.2 Application Command Interception

Another important design issue is the method for obtaining display information from application display commands so that they can be translated into THINC protocol commands. To be a viable replacement for the traditional desktop computing model, THINC needs to be able to obtain display updates from application display commands without modifying existing applications. Also, since good performance even in WAN environments is essential, THINC 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.

Given that traditional display systems are structured in multiple abstraction layers, there are a number of ways in which THINC can interact with existing display systems. One approach is to intercept commands at the application layer using functions provided by a display library. Intercepting at this layer provides a high-level view of the overall characteristics of the display system including the operation and management of windows, input mechanisms, and display capabilities of the system. Though it gives the ability to fully optimize the encoding of display updates to the client, the translation of application layer requests down to THINC display commands entails a significant amount of application logic, software complexity, and computational power on the client.

Another possibility is to intercept commands at the middleware layer, a hardware-independent abstraction of the display hardware created to meet the requirements of the display system and its applications. To maintain consistency across hardware with differing capabilities, this layer is provisioned with fallback mechanisms and software routines that can implement missing hardware features. However, the complexity of the middleware layer can make implementating display command interception difficult. Moreover, non-standard middleware implementations quickly become outdated and need constant revision to keep up with advances in commodity middleware systems.

Another approach is to operate at the lowest level possible by simply reading the actual pixel data in the framebuffer, implicitly intercepting commands after they have been processed completely. In this manner, it is possible to develop a very portable system since raw pixels are ubiquitously supported. Nonetheless, this approach is unsatisfactory because semantic information that may have been associated with application display commands is no longer available. This makes it computationally expensive to translate pixel data into bandwidth-efficient display protocol commands, despite advances in compression algorithms developed for this purpose [9, 8].

THINC takes an approach based on the video device abstraction layer which sits below the middleware layer and above the framebuffer. This layer is a well-defined, low-level, device-dependent layer that exposes the video hardware to the display system. Instead of relying on a real hardware-specific driver, THINC is designed as a virtual video device driver that can intercept graphics commands at the device layer, encode them as THINC protocol commands, and send the commands over the network to the client. The virtual video device approach enables THINC to maximize the use of available client resources without requiring a significant amount of application logic and computational power. The advantage to this approach is that it works with existing unmodified middleware layer implementations, thus allowing THINC to leverage continuing advances in existing window systems and avoid re-implementing substantial functionality available in those systems. THINC can also support new video hardware features with at most the same amount of development work required for supporting them in traditional desktop display drivers.

THINC can utilize this semantic information at the video device driver layer to translate application commands and transmit them from the server to the client in a way that is computationally and bandwidth efficient. Still, this semantic information is not guaranteed to be maintained throughout the translation process. Consider the case where display commands are intercepted and fully rendered into actual pixel data. At this point, no semantic information for the intercepted display command is available, and THINC is left to analyze only pixel data to determine the best commands with which to encode. While the choice of abstraction layer for intercepting display commands and the underlying protocol commands are important in thin-client design, the mechanism used for translating from one to another is also critical for performance.

2.3 THINC Translation Architecture

Having described the basic THINC protocol and the mechanism used to transparently intercept application drawing requests, we discuss how THINC translates application commands to THINC commands. As we will show, this translation architecture is a key component of several THINC optimizations.

THINC introduces an abstraction layer that allows it to perform the translation from application draw requests to protocol commands, and to efficiently manage protocol commands as they traverse the system, The abstraction layer builds upon 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. All command objects are based on a generic interface that allows the THINC server to operate on all of the THINC commands in the system. Objects specific to each protocol command are concrete implementations of this generic interface. Several attributes and methods are shared by all command objects, though particular objects implement their own set of attributes and methods specific to the command they represent. Attributes can indicate specific characteristics of a command such as its scheduling priority or whether the command is opaque or transparent.

The generic command interface consists of the following methods:

- *Create* a new instance of a command object.
- *Destroy* an object instance.
- *Copy* an instance of an object. Commands are free to implement copying efficiently, for example by using copy-on-write mechanisms to share private data across object instances.
- *Modify* the characteristics of an object instance. Current supported modifications are clipping and translation of the draw region of the object, and merging of two objects into one.
- *Query* information about an object instance, for example, the region where it draws, or its size.
- *Flush* an object instance by creating the appropriate protocol representation of the command encapsulated by the object, and passing it to the network layer for delivery to the client.

Functions within the application interception layer pass appropriate information to the translation layer to create a particular type of object as an opaque handle. This handle is passed to the *Create* method which returns a generic object instance that the translation layer can manipulate. However, as discussed later, 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 facilitate this process, THINC introduces the notion of a *command queue*. A command queue is a typical queue structure where commands are ordered according to their arrival time. However, the queue also has a single invariant: no overlap exists among opaque commands inside the queue. To guarantee correct drawing, the queue distinguishes between self-contained opaque commands and commands with transparent regions that depend on commands previously executed. The former are allowed to overwrite previous commands whereas the latter cannot. The invariant is maintained by modifying queue insertion. When newer commands are added to the queue, existing commands are automatically modified to maintain no overlap. If a command in the queue is partially overwritten by the new object, the intersection of its draw region and the region of the new object is modified. For example, if a RAW command is partially overwritten by a new FILL object, the queue will clip and discard the RAW pixel data associated with the overwritten region. Furthermore, if the new command completely overwrites a queued object, the queued object is purged from the queue.

Command queues provide a powerful mechanism for THINC to manage groups of commands. For example, queues can be merged and the resulting queue will maintain the invariant automatically. Command queues also play a key role in THINC's non-blocking operation. In the following section, we discuss an important translation optimization based on command objects and queues.

2.4 Offscreen Drawing

Over the past few years, there has been a trend in graphic applications to move towards 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.

Though this practice provides the user with a more pleasant experience on a regular local desktop client, it can pose a serious performance problem for thin-client systems. Thin-client 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 thin-client server send a corresponding display update to the client. However, at this point, all semantic information regarding the offscreen data has been lost 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 using image compression algorithms, these algorithms 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, the appropriate THINC protocol command object is generated and added to the command queue associated with the destination offscreen region. Since the command queue guarantees that no overlap exists among commands in the queue, only relevant commands are stored for each offscreen region. In addition, transparent manipulation of offscreen commands is made possible by the generic command object interface. The command queue also allows 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 area being copied in the source offscreen region to the destination region's queue. 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 the commands are copied, they are clipped so that they do not draw outside of the source area. In addition, they are translated to the proper position in the destination offscreen region. Finally, the destination command queue guarantees that commands are properly modified so that no overlaps exist with the newly-added commands.

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 2: THINC Video Commands

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. This process is encapsulated by the command's *Flush* method. 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 6.

2.5 Video Support

From video conferencing and presentations to movie and music entertainment, multimedia applications play an everyday role in desktop computing. However, existing thin-client platforms have little or no support for multimedia applications, and in particular for video delivery to the client. Video delivery imposes rather high requirements on the underlying remote display architecture. If the video is completely decoded by applications on the server, there is little the thin-client server can do to provide a scalable solution. Real-time re-encoding of the video data is computationally expensive, even with today's high end server CPUs. At the same time, delivering 24fps of raw RGB data can rapidly overwhelm the capacity of a typical network. Further hampering the ability of thin-client systems to support video playback are the lack of well-defined application interfaces for video display. Most video players use ad-hoc, unique methods and architectures for video decoding and playback, and providing support in this environment would require prohibitive per-application modifications. This section describes THINC's video support. We first present THINC's generic video architecture, designed to provide scalable remote video delivery. Then we describe how THINC's video architecture transparently supports today's multimedia applications.

Video support in THINC is implemented as a separate set of protocol commands. We decided against reusing or extending the basic display update protocol commands since they could not cleanly provide the appropriate framework that video delivery requires. While typical display updates are stateless and self-contained, video display updates are deeply interconnected and require considerable amounts of state. The video architecture is built around the notion of video stream objects. Each stream object represents a video being displayed. Available formats for a session are negotiated at client connection time to allow the system to adapt to varying client capabilities. All streams share a common set of characteristics that allow THINC to manipulate them such as their format, position on the screen, and the geometry of the video. In addition, each stream encapsulates information and state for its respective format.

The commands used to manipulate video streams are described in Table 2. When an application attempts to display a video, the THINC server sends an `INIT` message to the client that sets up the video stream. Playback does not start until the client acknowledges successful stream initialization. This synchronization step is needed because the client will normally need to make use of hardware resources that may not always be available. (However, after a video stream is initialized, no additional synchronization is needed.) The `INIT` message also assigns a unique ID to the stream. Any other video command will use this ID to identify the stream that is being modified. Video playback is accomplished using the `NEXT` command. `NEXT` encapsulates the data needed to display the next frame in the video stream, and is sent in response to requests from the application. Because applications have complete control over video playback, THINC does not need to implement playback control commands, for example to 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 which is useful for displaying videos at resolutions different from the actual encoded stream. `SRCSIZE` informs the client that the dimensions of the encoded stream have changed. The command may not be supported with all video formats. For example, to change the geometry of an MPEG stream, the server would have to re-encode the stream on the fly.

To provide transparent video playback functionality, THINC supports alternative YUV pixel formats commonly used in applications that manipulate video content. 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. In addition to the obvious compression gains, the use of YUV data has the benefit of being natively supported by virtually every off-the-shelf video card available today. This allows THINC to take full advantage of the capabilities of client video hardware while incurring minimal overhead for video processing. The video data need only be transferred to the client video hardware, which automatically and efficiently performs the required colorspace conversion and scaling to the stream's destination size. Moreover, in the absence of suitable video hardware, the colorspace conversion can be optimized using high-speed operations such as Intel's MMX or PowerPC's Altivec extensions, both of which are found in almost all CPUs in common use today. Widely used application interfaces already exist today to allow video players to transfer YUV data directly to the video card. THINC is able to leverage these interfaces to provide its support for YUV pixel formats without requiring any modifications to existing applications.

Application interfaces that support the YUV pixel model enable applications to initially query the video device to find out what pixel formats are supported. From the list of formats, the application chooses one to use and then forwards subsequent images in this format to the video device. THINC's virtual video device operates in the same manner, such that at the query stage, THINC can steer an application towards a particular pixel format optimal for its environment. The relative simplicity of the YUV formats allows the THINC server to do on-the-fly resampling to support video playback in small screen devices. As demonstrated by our performance results, the resampling operation incurs very low overhead while producing excellent gains in resource constrained environments.

3 Improving Interactivity

Thin clients must provide a high-quality interactive experience in order to become a viable replacement to traditional desktop computers. The interactive performance of a thin-client system is directly dependent on its response time and, more importantly, on its ability to effectively support network latency variations. Unfortunately, today's thin-client systems are either optimized for LAN or low bandwidth environments and, consequently, use continuous synchronization or have client-driven display update mechanisms which can only give subpar interactive performance. Furthermore, they may become completely unusable as network latency increases.

We have designed THINC with responsiveness and latency tolerance as a top priority. Previous sections have described THINC's low overhead architecture. We now describe the mechanisms built on top of this architecture, employed by THINC to maximize the interactive feel of the system and adapt to variable network latency. As we demonstrate with our results, these mechanisms allow THINC to provide an interactive experience superior to any other existing system, particularly in high-latency network environments.

3.1 Server-Push Model

At the heart of THINC's interactive architecture lies its design around 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 [3] and GoToMyPC [17], 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.

3.2 Non-Blocking Operation

Although a push mechanism can outperform client-pull systems, a server that blindly pushes 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 core server 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 keeping a per-client command buffer based on the command queue structure described in Section 2.3. The command queue within the buffer ensures no command overlap, thus any outdated commands in the buffer are automatically evicted. Periodically, THINC attempts to flush the buffer 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. If the server detects that committing a command may cause it block, the operation is postponed 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 bluffer. Commands are not broken up in advance to guarantee minimum overhead and allow the system to adapt to changing conditions.

Command	Description
CHANGE	Changes the shape of the cursor, described as two bitmaps: source and mask
SHOWHIDE	Show or hide the cursor
MOVE	Move the cursor (only in response to application request)
COLOR	Change the color of the cursor

Table 3: THINC Cursor Commands

3.3 Scheduling Updates

Alongside the client buffer is 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 [5], a primary goal in improving the interactivity of a system. THINC uses remaining size instead of the update's original size to shorten the delay between delivery of segments of an update, and 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. 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.

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, she expects immediate feedback from the system in the form of a pressed button image. By marking this update as real-time and delivering it sooner as opposed to later, THINC improves the perceived responsiveness of the system.

3.4 Managing the Cursor

Guaranteeing quick cursor response has a direct effect on the perceived feel of the system. Owing to the fact that today's commodity video hardware has the ability to manage a hardware cursor, THINC optimizes cursor management by transferring the responsibility of drawing the cursor to the client. Since hardware support exists for cursor drawing, a local client cursor does not impose additional overhead on the client. In contrast, approaches where the cursor is drawn on the server and delivered as normal display update cannot guarantee the response time required by cursor movement. This is particularly true in high latency WAN environments where the cursor updates have a continuous lag of at least the round-trip time. In THINC, the server continues to maintain cursor state and transmits changes to the client using the commands shown in Table 3. Each command modifies a component of the cursor state while the client is responsible for using this state to continuously draw the cursor in response to local mouse movements. Cursor commands are treated as high priority commands by THINC's scheduler, thus minimizing the perceived user delay between local cursor activity and any corresponding display changes.

4 Supporting Heterogeneous Displays

The promise of ubiquitous computing access has been a major driving force in the growing popularity of thin-client systems. To deliver on this promise, THINC enables access from a variety of devices by supporting variable display sizes and dynamic resizing. For instance, to view a desktop session through a small-screen mobile device such as a PDA, THINC initially presents a zoomed-out version of the user's desktop, from where the user can zoom in on particular sections of the display. In sharp contrast to similar client-only approaches in existing thin-client systems, THINC's small screen clients are fully supported by the server. After a client reports its screen size to the server, subsequent updates are automatically resized by the server to fit in the client's smaller viewport. When the user zooms in on the desktop, the client presents a temporary magnified view of the desktop while it requests updated content from the server. The server updates are necessary when the display size increases, because the client has only a small-size version of the display, with not enough content to provide an accurate view of the desktop.

Server resize support is designed to minimize processing and network overhead while maintaining display quality. For this reason, resizing is supported differently for each protocol command. `RAW` updates can be easily resized because they consist of pure pixel data which can be reliably resampled, and more importantly, the bandwidth savings are significant. Similarly for `PFILL` updates the tile image is resized to save client computation, since the region to be filled can be large. On the other hand, `BITMAP` updates cannot be resized without incurring significant loss of display information and generating display artifacts. Traditionally, antialiasing techniques are used to minimize the loss of information from the downsize operation. However, antialiasing requires the use of intermediate pixel values which bitmap data cannot represent. In this case, `BITMAP` updates are sent unmodified to the client, which takes care of resizing and merging them into the display. Also, resizing `SFILL` updates represents no savings with respect to bandwidth or computation, and therefore they are sent unmodified. As we show in our results, our approach provides substantial performance benefits by taking advantage of server resources and reducing bandwidth consumption, vastly outperforming the client-only support present in other thin-client systems. Furthermore, since THINC leverages the powerful server CPU to do most of the resize work, it can use high quality resampling algorithms to provide superior display content to the user.

As a final note, we wish to draw attention to the interesting differences in providing this kind of support in thin-client systems versus the prevalent local computer model. In particular, the topic of web page display in small screen devices has received lots of attention over the last couple of years. Mechanisms like WAP, specialized web browsers, and even different website versions tailored to different screen sizes, have all attempted to provide desktop-like web experience in mobile devices with varying degrees of success. On the other hand, we have shown that THINC easily provides this kind of support without requiring any changes to existing protocols, infrastructure, or applications.

5 THINC Implementation

We have implemented a prototype THINC server as a video device driver for XFree86 4.3.0 in Linux, and a prototype THINC client as a simple X application. We also have a Java client implementation, both as a standalone application and a web browser applet, demonstrating THINC's client portability and simplicity. XFree86 4.0 introduced a modular device driver infrastructure that allows THINC to be confined to a single, dynamically loadable module. The module encapsulates all the THINC server functionality, along with simple, network-aware, cursor, mouse, and keyboard drivers. THINC's module seamlessly hooks into XFree86's existing driver infrastructure to capture display commands and translate them to THINC protocol commands. Since THINC uses well-defined and standard interfaces, no changes are required to applications or the window system. XFree86 is designed around a single-user workstation model where a server has exclusive access to the computer's display hardware, and multiple server instances are not allowed to be active simultaneously. Because the THINC server does not access local hardware, THINC modifies XFree86's behavior from within the video driver interface and without any changes to XFree86, thus allowing for multiple THINC servers to be active at the same time. To implement THINC's drawing infrastructure, we have made use of XFree86's *Drawables* to track and record all display operations in the system. In particular for offscreen updates, THINC attaches to all *Pixmap* objects a command queue where all draw operations on the Pixmap are recorded.

As previously discussed, the RAW command is the only command where we apply additional compression to mitigate its impact on the network. The current prototype uses zlib's implementation of *deflate*[13] for this purpose. We have experimented with other compression algorithms and have found zlib's implementation to have the best size/speed ratio. To support resizing, we use a simplified version of Fant's resampling algorithm [15], which produces high quality, antialiased results with very low overhead. To provide video support, THINC implements XFree86's standard XVideo driver interface. THINC primarily exports the YV12 format to applications, which we chose not only for its intrinsic compression characteristics, but more importantly, for the wide range of applications supporting it, and its use as one of the preferred formats in MPEG codecs.

Even though the thin-client model presents a leap forward in overall computer security, its reliance on insecure networks makes it vulnerable to sniff attacks that can potentially compromise sensitive data such as passwords typed on the client's keyboard. To further improve the thin-client security model, THINC encrypts all traffic using RC4, a streaming cipher particularly suited for the kind of traffic prevalent in thin-client environments. Although block ciphers can have a significant effect in the performance of the system, we have found the cost of RC4 to be rather minimal, and the benefits far outweigh any minor overhead in overall system performance. Our prototype also implements standard UNIX authentication through the use of PAM (Pluggable Authentication Modules). Our authentication model requires the user to have a valid account on the server system and to be the owner of the session she is connecting to. To support multiple users collaborating in a screen-sharing session, the authentication model is extended to allow host users to specify a session password, that is then used by peers connecting to the shared session.

6 Experimental Results

We measured the performance of THINC on common web and multimedia applications in a range of different network environments and compared our unoptimized THINC prototype with a number of state-of-the-art popular thin-client platforms in use today, including Citrix MetaFrame, Microsoft Terminal Services, SunRay, X, and VNC. Citrix MetaFrame and Terminal Services are often referred to by their respective remote display protocols, ICA (Independent Computing Architecture) and RDP (Remote Desktop Protocol), which we also do here. We also used a local PC as a baseline representing today's prevalent desktop computer model. Section 6.1 describes our experimental testbed. Section 6.2 describes the application benchmarks used for our studies. Section 6.3 presents our measurement results.

6.1 Experimental Testbed

We used an isolated network testbed to measure the performance of THINC and other thin-client systems under different network conditions. As shown in Figure 1, our experimental testbed consisted of seven computers connected on a switched FastEthernet network: two thin-client client/server pairs, a network emulator machine, a packet monitor, and a web server used for testing web applications. Only one client/server pair was active at a time. Table 4 summarizes the characteristics of the machines. The Web Server used was Apache 1.3.27, the network simulator was NISTNet 2.0.12, and the packet monitor was Ethereal 0.9.13.

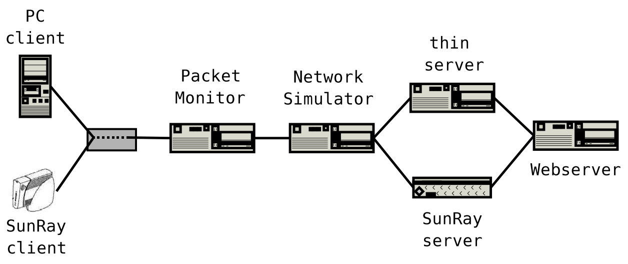

Figure 1: Experimental Testbed

Role	Hardware	Software
Thin-Client Server Packet Monitor Network Simulator Web Server	IBM Netfinity 4500R 2x933MHz PIII, 512MB RAM	Debian Linux Unstable (2.4.20 kernel), Windows 2000/2003 Server
Thin-Client client	450MHz PII, 128MB RAM, nVidia Riva TNT	Debian Linux Unstable (2.4.20 kernel), Windows XP Pro
SunRay server	SunFire V210 2x1GHz UltraSPARC IIIi, 2GB RAM	Solaris 9 4/03, OpenWindows 6.6.1
SunRay client	SunRay I Terminal, 100MHz uSPARC IIep, 8MB RAM	SunRay OS

Table 4: Testbed Machine Configurations

All of the thin-client systems except SunRay used the Pentium II PC as the client, and a Netfinity server as the thin-client server. We used a SunRay I hardware thin client with a Solaris 9 SunFire v210 server for SunRay measurements, since it does not run with the common hardware/software configuration used by the other systems. To minimize application environment differences, we used common thin-client configuration options and common applications across all platforms whenever possible. All of the tests were done with the client display set to 24-bit color and, if supported by the system, 128-bit encryption enabled. Any remaining thin-client configuration settings were set to their defaults. Some thin-client 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. Finally, because VNC's adaptive compression mechanisms compromise display quality by using variable color depths, we disabled this mechanism and set VNC to use the best compression algorithm available for 24-bit color to guarantee a fair comparison with the other systems.

For each thin-client system we used the server operating system which delivered the best performance for the given system. Terminal Services only runs on Windows, MetaFrame ran best on Windows, THINC, VNC, and X ran best on UNIX/Linux, and SunRay only runs on Solaris. We used the most recent system versions available on each platform at the time of our experiments, namely Citrix MetaFrame XPe, Microsoft Terminal Services built into Windows XP and Windows 2003, VNC 3.3.7, XFree86 4.3.0, and SunRay 2.0.

We considered two different display resolutions for our experiments, one with the client display set to 1024x768 for a desktop-like viewing experience, and the other with the client display set to 320x240 for a PDA-like viewing experience. We used the network emulator to adjust the network characteristics to match those of various LAN and WAN network conditions. For desktop screen resolution, we measured the performance on a 100 Mbps LAN network, and on a 100 Mbps Internet2 WAN network, where the round-trip network latency was set to 66 ms to represent US cross-country network latency [23]. These environments are identified as *LAN Desktop* and *WAN Desktop*, respectively. For PDA screen resolution, we measured the performance on an idealized 802.11g network by limiting bandwidth to 24 Mbps [2]. We chose 802.11g over 802.11b to reflect 802.11g's emergence as the next standard for wireless networks. In addition, the added bandwidth capacity guarantees a more legitimate comparison for bandwidth intensive applications, such as video playback. Since the purpose of the test was to measure performance on small-screen displays, we did not add the latency and packet loss characteristics typical of wireless networks. This environment is identified as *802.11g PDA*, and results are only reported for those architectures with support for small screens, as we discuss later on.

We conducted our WAN experiments using the kind of high-bandwidth network environment that are becoming increasingly available in public settings [1]. For example, South Korea is building a nationwide Internet access infrastructure to make speeds up to 100 Mbps available to the home by 2010 [26]. Because most of the thin-client 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, on the WAN environment we used a 1 MB TCP window size to take full advantage of the 100 Mbps Internet2 network bandwidth capacity available. Network packet loss was set to zero for our experiments.

6.2 Application Benchmarks

We evaluated display performance using three popular desktop application scenarios, web browsing, video playback and interactive graphics editing. Web browsing performance was measured using the Mozilla 1.6 browser to run a benchmark based on the Web Text Page Load test from the Ziff-Davis i-Bench benchmark suite [43]. 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 a video player to play a 34.75 s video clip of original size 352x240 pixels displayed at full-screen resolution. The video player used was MPlayer 1.0pre3 for the Unix-based platforms, and Windows Media Player 9 for the Windows-based platforms. PC performance was measured by running the web browser and video player on the thin-client client computer. Graphics editing performance was measured by recording a one minute long graphics editing session, and then replaying the session on each of the systems. Recording was done using Xnee and the graphics editor used was The Gimp 1.2. Because Xnee is only available for X/Unix-based platforms, the Windows-based systems were not measured with this benchmark.

We used the packet monitor in our testbed to measure performance on the thin-client systems using slow-motion benchmarking [31, 24]. This allowed us to quantify system performance in a non-invasive manner by capturing network traffic. The primary measure of web browsing performance was the average page download latency. The primary measure of video playback performance was video quality [31], 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. The primary measure of interactive graphics editing performance was completion time.

6.3 Measurements

We compared THINC to other popular thin-client platforms by measuring performance in three representative application scenarios, web browsing, video playback, and interactive graphics editing, and three network/display environments. Figures 2 and 3 show web browsing performance results in terms of the perceived latency and average per page data transfer, respectively. Figures 4 and 5 show the video playback performance results in terms of video quality and total data transferred, respectively. Due to space constraints, figures are not shown for the interactive graphics editing performance but are discussed below.

For LAN and WAN environments, Figure 2 shows that the local PC is the most bandwidth efficient platform for web browsing. However, Figure 3 shows that THINC is 2.5 times faster than the local PC, and provides the best web page download latencies across all thin-client systems. THINC is 1.25 to 1.6 times faster in the LAN environment, with a more marked difference in the WAN environment, where THINC is 1.8 to 5 times faster than all measured systems. THINC outperforms the PC because it leverages the faster server machine to process web pages more quickly than the web browser running

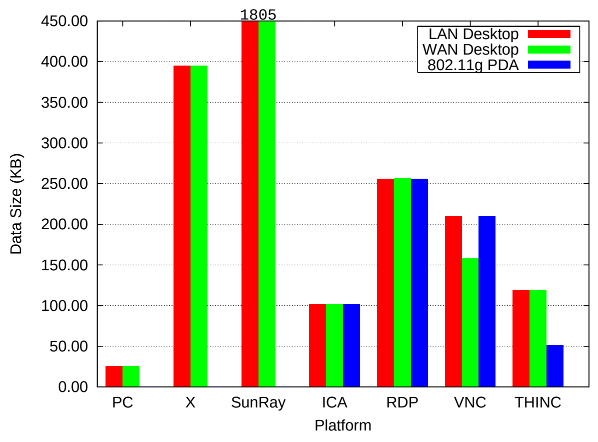

Figure 2: Web Benchmark: Average Page Data Transfer

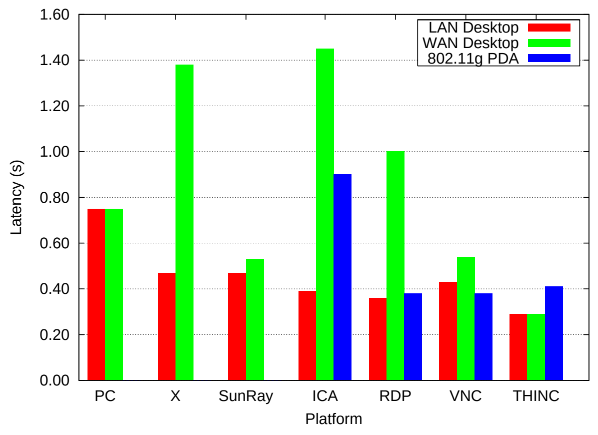

Figure 3: Web Benchmark: Average Page Latency

on the slower client PC. It is worth mentioning that X and VNC are the only platforms with no encryption support. We have measured the latency penalty of tunneling X and VNC over ssh - the preferred method to secure these platforms - to be approximately 10% and 40%, respectively.

The results show that due to its latency-sensitive design, THINC is the only system that does not suffer any performance penalties in the WAN environment. Platforms employing a high-level display approach such as X, ICA, and RDP, have the worst WAN performance - up to five times slower than THINC in ICA's case - because of the tight coupling required between the application running on the server and the viewer running on the client. VNC's WAN performance degradation is partially due to its reliance on a client pull display update model. THINC's server push model avoids round trip latencies for each update and provides a better interactive response time.

The results also demonstrate the drawbacks of VNC's single compression strategy for all types of display data. Although many graphics compression algorithms exist, none of them can effectively and efficiently compress every type of graphics data - a fact best illustrated by VNC's subpar performance. In contrast, THINC's multiple command approach results in much better bandwidth utilization and higher performance. Since most of the compression work is done by the command separation and translation layer, very few of the elements in the web pages actually need to be sent using `RAW` and thus need compression using a generic algorithm.

Finally, the importance of offscreen drawing awareness is illustrated by the large difference between THINC and SunRay's bandwidth usage in both LAN and WAN, where SunRay transfers 15 times more data per page than THINC. While SunRay and THINC use a similar multi-command protocol, SunRay is unable to leverage its own protocol due to Mozilla's heavy use of offscreen drawing. As previously discussed, by the time Mozilla finally renders the web page onscreen, SunRay has lost all semantic information and must resort to its equivalent `RAW` command to draw updates. We have measured the impact of disabling offscreen awareness in THINC to cause a 70% slowdown in latency. The slowdown is caused by having to fallback to `RAW`'s expensive deflate compression for all data transferred. We note that this demonstrates both the importance of THINC's offscreen drawing awareness, and the drawbacks of relying on a single encoding mechanism for all display data.

Small screen results are also shown in Figures 2 and 3. We only report results for VNC, RDP and ICA since only these architectures have support for a client display geometry different than the server's. The results show that THINC has the best performance overall, particularly with respect to bandwidth usage where THINC transfers between 2 to 5 times less data than the other systems.

Support for small screen devices can be divided in two models: systems which clip the client's display and systems which actually resize the contents of the display. RDP and VNC fall within the first category, which requires users to scroll around the display to see the full screen and offers a more cumbersome usage model. Citrix and THINC fall within the second category though THINC differs from ICA in that the THINC server does most of the resizing work. As shown by our results, this approach achieves the best performance across all the architectures. Since most of the display updates are resized before being sent on the network, THINC's bandwidth utilization is reduced by more than a factor of two while only marginally affecting the latency of the system. In contrast, ICA's client-only resize approach increases latency to more than twice its LAN latency, with no improvement in bandwidth consumption. In the CPU- and bandwidth-limited environment of mobile devices this approach adversely affects the overall user experience. As a final note, we also conducted tests on different PDA devices using the respective PDA specific clients for each platform, where available. All the systems behaved the same except RDP's client. Specifically, RDP's desktop client delivers all display updates, but RDP's PDA client only sends those display updates that draw within the client's viewport. As such, the results reported here do not show the effects of RDP's clipping support on performance.

It is also worth mentioning the large difference in quality of THINC's resized display compared with ICA's. THINC's

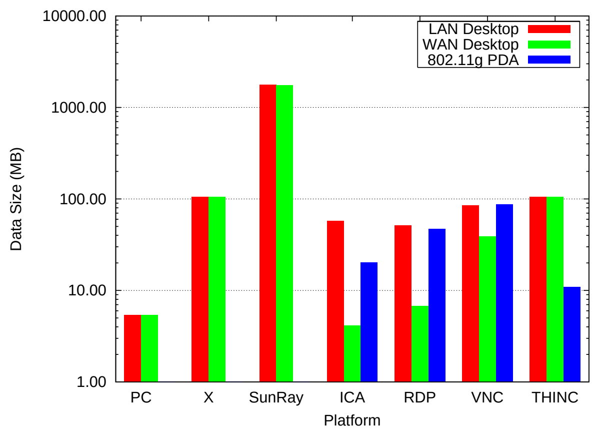

Figure 4: Video Benchmark: Total Data Transferred

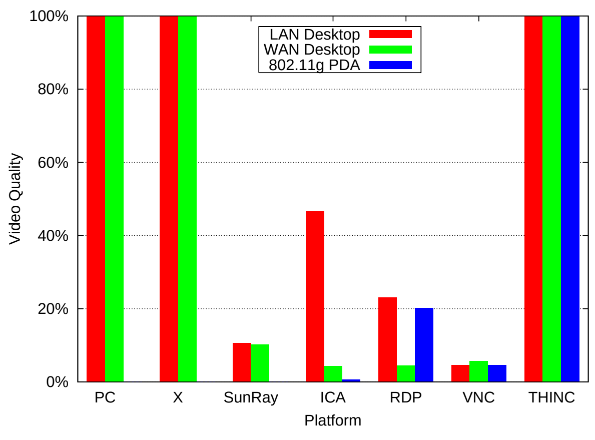

Figure 5: Video Benchmark: Video Quality

resize algorithm appropriately interpolates pixel data and uses antialiasing to provide high quality results such that the web page is still readable even when displaying the 320x240 client window on a computer with a resolution of 1280x1024. On the other hand, ICA's resized display version is barely readable and appears to be useful only for locating portions of the screen to zoom in to. Clearly, ICA's choice of resizing algorithm is restricted by the client's computational power, while THINC can take advantage of the server's powerful CPU and make use of better algorithms that produce higher quality results.

Video playback performance results are shown in Figures 4 and 5. Figure 4 shows that the local PC is also the most bandwidth efficient platform for video playback, using about 1.2Mbps of bandwidth. However, Figure 5 shows that THINC provides perfect video quality in the same manner as the local PC and X. Figure 5 also shows that all of the other platforms deliver very poor video quality, specifically 46%, 23% and 4% for ICA, RDP, and VNC, respectively. They suffer from their inability to distinguish video data from normal display updates and apply ineffective and expensive compression algorithms on the video data. These algorithms are unable to keep up with the stream of updates being generated, resulting in dropped frames or extremely long playback times. In contrast, THINC's ability to leverage client hardware to deliver video provides substantial performance benefits over the other systems. VNC has the worst overall performance primarily because of its use of a client pull 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. Figure 4 also shows that THINC's 100% video quality does not translate into high resource utilization. The total data transferred corresponds to bandwidth usage of roughly 24Mbps. While VNC, RDP and ICA consume less bandwidth - 13, 11 and 19Mbps respectively - their video quality is too low to provide useful video delivery. X and THINC have the same video quality and bandwidth consumption as both are using a similar mechanism to provide remote video display. However, X is not able to provide the scaling benefits shown by THINC for small screen devices, as discussed next.

The small display results again demonstrate the benefits of THINC's server resize mechanism. THINC still performs at 100% video quality, demonstrating the minimum overhead incurred on the server by resampling the video data while significantly reducing bandwidth consumption to 2.5Mbps, well below any of the other systems. In fact, we have conducted tests that demonstrate that THINC's resized bandwidth requirements are more than enough to provide perfect video playback over an 802.11b wireless network, which cannot be done by any of the other thin-client systems, including X. ICA's client-side resize mechanism aggravates its low video quality, reducing it to less than 1% while consuming the same amount of bandwidth. RDP and VNC's clipping mechanisms are not particularly useful for video playback since the user only sees the section of the video that intersects with the client's viewport. The user could potentially watch the video at a smaller size and make the video window fit within the client's display. However, we believe that adding such awkward constraints to the user interface is detrimental to the overall usability of the system.

Finally, we measured graphics editing performance as a benchmark of a highly interactive application with reasonable bandwidth demands. Although the application runs on X/Unix-based systems, neither VNC nor SunRay were able to run the benchmark. VNC lacks support for the appropiate X mechanisms needed to record the session, demonstrating the drawbacks of implementing alternative middleware systems. VNC is implemented as a variation of an old version of XFree86, and has grown outdated and lacking support for otherwise ubiquitous features. SunRay could not run the benchmark because of incompatibilities between its X server and Xnee's implementation. The only systems that ran the benchmark were X and THINC.

THINC and X both completed the benchmark in one minute in the LAN environment, but THINC vastly outperformed X in terms of the amount of data transferred, sending about 5 times less data than X during the benchmark. Furthermore, only THINC completed the benchmark successfully in the WAN environment. Xnee's replay operation relies heavily on timing and synchronization information between mouse and keyboard events, and the graphic responses to those events. If timing at replay time is not equivalent to the recorded information, Xnee will be unable to know if responses were missed and it should continue, or if responses have yet to arrive and it should keep waiting. Although Xnee has considerable resilience to timing differences, if synchronization is lost for an extended period of time it will give up on the replay operation. X's large synchronization overhead made it significantly slower, causing Xnee to lose synchronization and give up after only 20 seconds.

7 Related Work

A number of remote display systems have been developed, and previous studies have compared many of them and identified those with superior performance [23, 31]. We used those systems as a basis for comparison with THINC. These systems can be loosely classified by their choice of protocol primitives. X [36], Citrix Metaframe [10], and Microsoft Terminal Services [12] use high-level commands which are widely thought to allow for more efficient encodings. However, this approach suffers from substantial performance degradation in high-latency WAN environments. VNC [33] takes a low-level approach and uses a single encoding mechanism providing a simple and portable solution. Though a number of encoding algorithms have been developed for thin-client systems, such as VNC's ZRLE, FABD [16], PWC [4], and TCC [9, 8], none of them can effectively and efficiently compress all types of display data, resulting in subpar performance as illustrated by VNC's results. SunRay [37] is designed around a set of simple commands similar to those used in THINC. However, its inability to recognize new application display approaches adversely affects its overall performance. Furthermore, it lacks many of THINC's other design features, including screen scaling for heterogeneous display devices, transparent video support, and latency-sensitive optimizations.

Many other systems for remote display exist including Tarantella [35], Laplink [25] and PC Anywhere [38], along with extensions to other systems such as Kaplinsk's VNC tight encoding[21], low-bandwidth X (LBX)[6], and more recently, NoMachine's NX system[32]. Previous studies have shown the limitations in several of them [19, 22, 30] and demonstrated that they perform worse than the thin-client systems we compared against THINC. In particular, these systems have primarily been designed for LAN or low-bandwidth networks, without regard for latency and responsiveness. Still, a growing number of ASPs such as services from FutureLink [7], Runaware [34], and Expertcity [14] are employing thin-client technology to host desktop computing sessions over WAN environments.

Specialized architectures that provide remote access to specific applications have also been proposed over the years. The topic of remote access to multimedia content has been extensively explored, and in particular the Infopad project [39] created a terminal device optimized for wireless access to multimedia content. Many commercial systems provide remote access to 3D content, for example SGI's VizServer [40]. Similarly, WireGL and Chromium [20] enable cluster rendering systems that distribute the load of processing 3D content, but require high bandwidth environments to operate efficiently.

While technology has changed, the vision of customers renting their computing services from a public computer utility harkens back to the days of Multics [11]. Unlike Multics, ASPs are faced with supporting not just simple text programs but also graphics and multimedia-oriented applications. THINC provides a key componet to support these kinds of applications, thereby modernizing the vision (and reality) of utility computing.

8 Conclusions

We introduced THINC, a remote display architecture for high-performance thin-client computing for LAN and WAN environments. THINC uses a simple, low-level protocol that mimics operations commonly found in commodity display hardware, and introduces a low overhead, semantic-preserving translation architecture to convert high-level application drawing calls to THINC protocol commands. On top of this architecture, THINC implements a number of latency-sensitive optimizations to provide a high fidelity visual and interactive experience. These include client-side cursor management, a server-push update model, shortest-job-first command scheduling, and a non-blocking drawing pipeline. Furthermore, THINC provides native support for video display by leveraging client display hardware, and is amenable to use in small screen devices with server-side scaling of display updates.

We implemented a THINC prototype system as a virtual display driver for XFree86 4.3.0 and an Xlib client application. Our implementation illustrates the simplicity of THINC's protocol and the effectiveness of its translation architecture. We measured THINC's performance in web and video applications in a number of network environments and compared it to other thin-client systems. Our experimental results in web applications have shown that THINC delivers superior performance and is as much as five times faster than traditional systems in high latency environments. Our results also demonstrate the effectiveness of THINC's server-side scaling mechanism, reducing THINC's bandwidth consumption by more than a factor of two. Finally, THINC's video support outperforms other existing systems with reasonable network usage, and coupled with server-side video scaling, THINC is the only system capable of delivering full-screen video on 802.11b wireless networks.

References

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

[2] 802.11 Wireless LAN Performance. `http://www.atheros.com/pt/atheros_range_whitepaper.pdf`.

[3] Virtual Network Computing. `http://www.uk.research.att.com/vnc`.

[4] P. Ausbeck. A Streaming Piecewise-constant Model. In *Data Compression Conference (DCC), Snowbird, UT*, Mar. 1999.

[5] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *SIGMETRICS/Performance*, pages 279–290, 2001.

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

[7] Charon Systems. `http://www.charon.com`.

[8] B. O. Christiansen and K. E. Schauser. Fast Motion Detection for Thin Client Compression. In *Data Compression Conference (DCC), Snowbird, UT*, Apr. 2002.

[9] B. O. Christiansen, K. E. Schauser, and M. Munke. A Novel Codec for Thin Client Computing. In *Data Compression Conference (DCC), Snowbird, UT*, Mar. 2000.

[10] Citrix MetaFrame 1.8 Backgrounder. Citrix White Paper, Citrix Systems, June 1998.

[11] F. J. Corbato and V. A. Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the Fall Joint Computer Conference*, volume 27, pages 185–196, June 1965.

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

[13] An Explanation of the DEFLATE Algorithm. `http://www.gzip.org/deflate.html`.

[14] DesktopStreaming Technology and Security. Expertcity White Paper, 2000.

[15] K. M. Fant. A Nonaliasing, Real-Time Spatial Transform Technique. *IEEE Computer Graphics and Applications*, 6(1):71–80, Jan. 1986.

[16] J. M. Gilbert and R. W. Brodersen. A Lossless 2-D Image Compression Technique for Synthetic Discrete-Tone Images. In *Data Compression Conference (DCC)*, Snowbird, UT, Mar.30 - Apr.1 1998.

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

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

[19] B. Howard. Thin Is Back. *PC Magazine*, 19(7), July 2000.

[20] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. T. Klosowski. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. In *SIGGRAPH 2002*, 2002.

[21] C. Kaplinsk. Tight Encoding. `http://www.tightvnc.com/compare.html`.

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

[23] A. Lai and J. Nieh. Limits of Wide-Area Thin-Client Computing. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 228–239, Marina del Rey, CA, USA, June15-19 2002. ACM Press.

[24] A. Lai, J. Nieh, B. Bohra, V. Nandikonda, A. P. Surana, and S. Varshneya. Improving Web Browsing on Wireless PDAs Using Thin-Client Computing. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, New York, NY, May17-22 2004.

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

[26] D. Legard. Korea to build 100Mbps Internet system. *InforWorld*, Nov.18 2003. `http://www.infoworld.com/article/03/11/18/HNkorea_1.html`.

[27] Worldwide Enterprise Thin Client Forecast and Analysis, 2002-2007: The Rise of Thin Machines. `http://www.idcresearch.com/getdoc.jhtml?containerId=30016`.

[28] MSDN Library: Display and Print Devices: Windows DDK. `http://msdn.microsoft.com/library/en-us/graphics/hh/graphics/dispprnt_1%odj.asp`.

[29] Thin-Client market to fatten up, IDC says. `http://news.com.com/2100-1003-5077884.html`.

[30] J. Nieh, S. J. Yang, and N. Novik. A Comparison of Thin-Client Computing Architectures. Technical Report CUCS-022-00, Department of Computer Science, Columbia University, Nov. 2000.

[31] J. Nieh, S. J. Yang, and N. Novik. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. *ACM Trans. Computer Systems*, 21(1):87–115, Feb. 2003.

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

[33] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), Jan./Feb. 1998.

[34] Runaware.com. `http://www.runaware.com`.

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

[36] R. W. Scheifler and J. Gettys. The X Window System. *ACM Trans. Gr.*, 5(2):79–106, Apr. 1986.

[37] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, volume 34, pages 32–47, Kiawah Island Resort, SC, Dec. 1999.

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

[39] T. E. Truman, T. Pering, R. Doering, , and R. W. Brodersen. The InfoPad Multimedia Terminal: A Portable Device for Wireless Information Access. *IEEE Transactions on Computers*, 47(10):1073–1087, Oct. 1998.

[40] SGI OpenGL Vizserver. `http://www.sgi.com/software/vizserver/`.

[41] The XFree86 Project. `http://www.xfree86.org`.

[42] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari. The Performance of Remote Display Mechanisms for Thin-Client Computing. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002.

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