OPTIMIZATION METHODS FOR THE PERFORMANCE ENHANCEMENT OF THIN CLIENT COMPUTING OVER BANDWIDTH-LIMITED AND SLOW LINKS

SUN YANG

NATIONAL UNIVERSITY OF SINGAPORE

2007

OPTIMIZATION METHODS FOR THE PERFORMANCE ENHANCEMENT OF THIN CLIENT COMPUTING OVER BANDWIDTH-LIMITED AND SLOW LINKS

SUN YANG

(*B.Eng., HUAZHONG UNIV. OF SCI. & TECH.*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2007

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my research supervisor, Dr. Tay Teng Tiow, for his guidance, insight and invaluable support throughout my doctoral course and for his careful reading of this thesis and the constructive suggestions given on it.

I would like to acknowledge all the anonymous reviewers who have reviewed and commented parts of this work published in journals and conference proceedings. I really appreciate their invaluable comments and suggestions which have helped me to improve the quality of this work.

I was fortunate enough to have wonderful lab-mates during my time at NUS. I owe thanks to Mr. Xia Xiaoxin, Mr. Pan Yan, Mr. Zhu Xiaoping, Mr. Bao Chunyu and Miss Mo Wenting for their valuable suggestions and inspiring talks with me. I would also like to thank the other lab-mates and friends for their help and friendship.

Finally I would like to take this opportunity to thank my family for their steadfast love, encouragement, inspiration and unwavering support throughout my time at NUS.

TABLE OF CONTENTS

SUMMARY

After a decade of moderate development, thin client computing has entered into an era of rapid growing as it offers a secure, cost-effective and easily managed computing model and facilitates mobility. Nevertheless, to deliver a satisfactory user experience, especially over the bandwidth-limited and slow network links, performance enhancement of existing thin client systems is still essential. Analysis shows that there is still a large amount redundancy in the screen updates generated by existing thin client systems, and improving the interactive performance of thin client computing by reducing the redundancy is still practicable. Towards this direction, we proposed a few optimization techniques for thin-client systems. These techniques include a static object cache technique, a long-distance redundancy reduction scheme, and a data spike reduction scheme. The static object cache technique is motivated by the observation that many graphical user interface objects cause similar screen updates when they are shown for multiple times. By caching the static parts of their presentation data on the clients, we can significantly reduce the data transferred cross network. We implemented this method on VNC, an open-sourced thin client system. Our experimental results show that this technique can reduce the network traffic and user operation latency of VNC by up to 60%. The long-distance redundancy reduction method provides a flexible and scalable way to increase the history size in LZ compression algorithm, which is widely used in thin client systems. This scheme is intended to reduce the redundant data that are located far from each other in the screen update stream. Our experiments show that compared to the baseline compressor used in Microsoft Terminal Server, this scheme significantly reduces network traffic and user operation latency. The data spike reduction scheme provides a hybrid cache-compression scheme to reduce the screen update data that are generated in a short time (data spikes). Experimental results show that reducing data spikes can considerably reduce the long operation latencies over bandwidth-limited and slow links with reasonable cache memory and slightly more computing time.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Background

In the early 1990s, thin clients were thought to be an alternative to personal computers (PCs). Thin clients are scaled-down PCs without permanent data storage. They are designed to be used in a special client/server architecture where the majority of processing occurs on the server. Thin clients were supposed to save money for companies by providing a cheap thin client device rather than an expensive PC for each employee. But the idea was soon eclipsed by the fact that PCs got much cheaper and much faster than anyone had expected. As a result, throughout the 1990s, thin clients remained a niche market, and PCs dominated the desktops in most enterprises.

While PC started out only as a simple computing tool, it has evolved into a powerful information processing system that can supports diverse, complex software applications. On the other hand, PCs have become so cheap that in many companies almost each employee has his own PC. Hence for a modern large- or medium-sized company, there are a substantial amount of hardware and software to administrate. The maintenance and management cost for these facilities has become a significant portion of the total expense of today's IT infrastructures. In response to this, thin client computing regained attentions of academia and industry by the end of 1990s [67-70, 102-104]. But different from the initial concept, thin client is now used more often to refer to a computing model (thin client computing) where users can interact with various applications running on remote servers with a single client even if the applications were not built in client/server architecture. The solutions making use of this model were no longer limited to thin client hardware. Some thin client solutions provide a simple software client that allows users to operate their applications on remote servers. Software based thin client solutions can be used by a company to configure their low-end PCs as thin clients while using their high-end PCs or servers to host applications for multiple users.

Compared to personal computing where most applications are installed and run on user PCs, thin client computing can reduce administration and maintenance cost in many enterprises [92, 93, 105, 106]. The reduction is mainly achieved by reducing the complexity of application deployment and security management. Firstly, with thin client solutions, application deployment becomes easier. A new application only needs to be installed and configured on a small number of servers that are usually located at one or a few places. Accordingly, compatibility only needs to be ensured between the new application and the software/hardware systems on servers. But for personal computing, installation, configuration, and compatibility tests need to be performed for thousands of or even more client machines which may have distinct machine architecture, hardware configurations, operating systems and installed applications, and may be dispersed all over the world. Secondly, with thin client solutions an IT department can focus its resources on the servers, which reduces the chance of viruses and other malware breaking into the network. As security management becomes more and more complex and also more and more important in enterprises, this merit of thin client computing considerably improves IT staffs' productivity and cuts off administration and maintenance costs. Of course, thin client computing will require more management and maintenance job for the server side. However, the sum of resources is greatly reduced because these resources when deployed on servers can be more efficiently allocated and shared. Hence the increased maintenance and management complexity on the servers will be much lower than the reduction of that on the client side. And since the client management becomes trivial, thin client computing removes or at least reduces the need of IT staffs' travel to different business locations in client supporting.

Besides the cost consideration, the growing demand of user mobility is another important reason for the revival of thin client computing. The user mobility trend is strong: More than 10% of users are working outside the LAN for more than 50% of their working time [87]. Thin client computing in nature supports user mobility as it allows users to access their applications from anywhere with a resource limited hardware or a simple software. If the enterprise adopts thin client infrastructure, when traveling a user can authenticate and login at any available network terminals, PCs or even mobile devices [88-90] and has immediate access to his unique computing environment. This improves his productivity as well as reduces risks of important data missing which is caused by the portable computers damaged or stolen when traveling from one place to another. If the enterprise does not adopt thin client computing infrastructure, mobile users can still benefit from this technology. For example, to avoid data missing a mobile user can choose to leave the important data with his desktop PCs in his company providing a network connection is available in the guest location. When he needs the data, he can use thin client technology to access them. Not only can he access the data, but also the applications installed on his PCs or other PCs which he has right to access, even though the PCs may have different architecture and operating systems. This is very useful when he wants to use an application or software tool which is not available on his laptop or the provided PC in the guest location. Or in some cases, some specific computation needs very long time to complete. So a mobile user can start the computation on PCs or servers. When traveling, the user uses mobile devices to monitor the status, control the process and view the final results.

As a secure, cost-effective and easily managed computing model, thin client computing offers IT users a valuable choice, but it may not be the ideal choice for all situations. Like any network-based computing model, it requires a robust and reliable network connection. Its performance also relies on the network performance to a great extent. If this computing model is deployed in a low-bandwidth network, some applications with intensive GUI display requirements like AutoCAD and Photoshop may get unsatisfactory performance over it. Fortunately, the network bandwidth is improving rapidly and the reliability of networks is increasing at the same time. Thanks to the efforts in both industry and academic areas, thin client computing itself also made remarkable enhancement and is continuing to enhance the performance. With the improvements of networks and thin client computing, we can anticipate thin client computing to be more widely used in the future.

1.2. Motivations and Goals

For most client/server systems, end-users generally have little concern about where processing occurs and where data are stored, as long as the interaction is fast, consistent, and seamless. Hence, interactive response time is the key performance of a thin client computing system. As shown in Figure 1.1, if the response time of an interactive application is less than a threshold value (tolerable threshold), then from a performance point of view, the application is fast enough. Above that point, the delays will be perceptible and lead to decreased performance. But until a second threshold (unusable threshold) the application is still tolerable for the users. Above the second threshold, the user feels the application unusable. The actual threshold values vary from person to person and are affected by the stimulus intensity [1, 75].

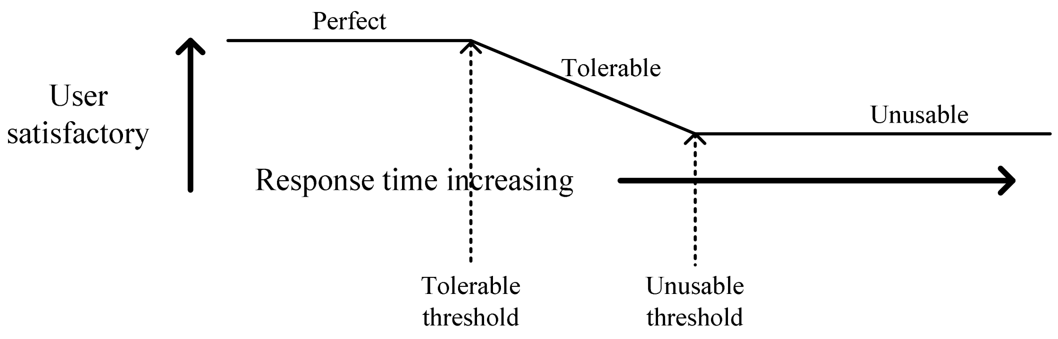

Figure 1.1: Impact of interactive response time on user satisfaction (adapted from [12])

After 40-year researching on the impact of interactive response times on user satisfaction and task productivity [2-7], the human-computer interaction (HCI) community reaches the consensus on acceptable response times for trivial interactions: when the latency is below 150 milliseconds user productivity is not impacted by the latency; when the latency is above 150 milliseconds users become increasingly aware of the delay; above 1 second, users become unhappy [8].

While various optimizations have been applied, the interactive performance of existent thin client systems is still unsatisfactory in many practical situations. The results in [8-10, 52, 53, 94] show that when network latency increases and network bandwidth decreases the interactive performance of thin client system degrades; for some tested WAN configurations, the response time may increase to the tolerable threshold or even unusable threshold.

To illustrate the cause of unsatisfactory interactive performance of thin client systems in these situations, we present a simple analysis here. Thin client computing splits an application into the user interface on clients and the application logic on servers, and uses network as the communication tie between them. In this architecture, the response time of an operation can be divided into two parts, computation time and network access time. On the client side, computation time is mainly the time to process thin client protocol, which is usually insignificant with today's PCs and thin client terminals. On the server side, computation time mainly includes application processing time and thin client processing time. Application processing time depends on the server's software/hardware environment and the computation nature of the requested operation. Thin client processing time consists of the time to process user input and the time to process presentation data. It is determined by the server's hardware/software environment and the thin client protocol adopted. Processing user inputs mainly involves parsing thin client protocol packets to get user input information and generating system events as if the inputs had happened locally. This normally takes only a small amount of time as the input data is limited. Processing presentation data mainly involves capturing and representing the presentation data, encapsulating them into protocol packets and applying various optimization methods. This processing may take a long time when some operations produce lots of presentation data. Network access time mainly consists of the time of transferring user input data and presentation data. It is dependent on the inherent network latency, the available bandwidth and the amount of data to be transferred. Network latency is determined by the network path in use. The available bandwidth is associated with whether and how much data are being transferred across the links in the network path. User input data usually only incurs small network traffic while presentation data can be very large and is a main contributor to the network access time and the total latency.

From the analysis we can find that using different thin client computing techniques affects two main parts of the total latency: thin client processing time to capture and process the presentation data, and network transmission time to transfer the presentation data. Though desirable, it is very hard to optimize one of them without affecting the other. In many situations it would be more favorable to reduce the network transmission time at the cost of increasing some processing time. This is because compared with the processing time, the presentation data transmission time usually contributes much more to the total latency especially in some non-ideal networks (high-latency and low bandwidth).

Although network bandwidth has been greatly increased in recent years, available bandwidth in many networks may still not be able to meet the needs of transferring the presentation data of thin client computing adequately fast. This is either because there are other data on the network or because the network capacity is limited. When available bandwidth is not enough, some of the screen update data will be queued on the sending machine or some network nodes which will increase interactive latency. When too much data are queued up on the sending machine the machine's network output buffer may become full, and some packets may be discarded (packet loss). This will induce packet retransmission, further increasing the interactive latency.

Different user operations result in screen update data of different sizes. The peak bandwidth requirement can be much higher than the average bandwidth requirement. For example, some GUI-intensive user operations such as opening a complex dialog or menu produce a large volume of presentation data in a short time. Usually delivering such an update to the clients within a user-perceptible latency needs a bandwidth that is much higher than the average bandwidth requirement. These GUI-intensive operations will induce long latencies in many non-ideal network environments [99-101].

In summary, the interactive performance of existing thin client systems could be potentially improved by reducing the network traffic, especially when the available network bandwidth is low and when the user operations generates a lot of screen updates. The goal of this dissertation work is to investigate the optimization techniques that can achieve this. Nevertheless, Figure 1.1 also implies that there is a performance threshold above which users will not be able to tell more improvements. As the thin clients perform well in high-bandwidth low-latency networks, the optimization techniques presented in this thesis do not target such networks.

1.3. Major Contributions

The major contributions of this research work are as follows:

- We propose a static object caching scheme [99]. This scheme reduces the redundant presentation data sent across networks which is caused by recurrent display of GUI objects. We implemented this scheme on VNC [25], a popular thin client computing system. Our experiment results show that for bandwidth-limited networks this scheme reduces the network traffic and interactive latency by up to 60% with only a little more CPU usage.
- We present a flexible and scalable method [101] to extend the history buffer of LZ algorithm [13] that is often used to compress screen update data. We empirically studied the effectiveness of our scheme on three different types of screen update traces of Microsoft Terminal Server [72]. The numerical results show that this scheme can reduce 15.0% - 33.5% network traffic for the tested traces with a history

buffer of 2M bytes. Moreover, it also can reduce 20.6% - 27.8% noticeable long latencies for different types of applications with 2M bytes history buffer. This scheme costs only a little additional computation time and the cache size can be negotiated between the client and server.

- We present a hybrid cache-compression scheme, DSRS, to reduce the data spikes [100]. We empirically studied the effectiveness of DSRS using a number of screen update traces of Microsoft Terminal Server. The experimental results show that DSRS can reduce 26.7% - 42.2% data spike count and 9.9% - 21.2% network traffic for the tested data with a cache of 2M bytes. DSRS can reduce 25.8% - 38.5% noticeable long latencies for different types of applications with the same cache configuration. This scheme costs only a little additional computation time and the cache size can be negotiated between the client and server.

Publications arising from this work

- Y. Sun and T.T. Tay, "Analysis and Reduction of Data Spikes in Thin Client Computing", *Journal of Parallel and Distributed Computing*, in press, DOI: 10.1016/j.jpdc.2008.05.007.
- Y. Sun and T.T. Tay, "Improving Interactive Experience of Thin Client Computing by Reducing Data Spikes", *ICIS2007*, Melbourne, Jul. 2007.
- Y. Sun and T.T. Tay, "Reducing Long Distance Redundancy of Thin Client Systems", *IEEE-IWEA2007*, Melbourne, Jul. 2007.
- T.T. Tay and Y. Sun, "A Novel Performance Optimization Approach for Thin Client Computing", *PDCS'05*, Las Vegas, Sep. 2005.

1.4. Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides necessary background knowledge on thin client systems. It gives an insight into one important thin client architecture, distributed presentation architecture. A brief survey of the thin client systems taking this architecture will provide a good background for the performance optimization techniques discussed in the following sections. Chapter 3 presents a brief survey of the past and ongoing researches in thin client performance optimization followed by a summary that reveals the gap between the reality and the expectation in this area. Chapter 4 describes an application-specific optimization approach, static object cache scheme. The performance evaluation of this scheme is presented. Chapter 5 presents an experimental framework to evaluate the performance of two optimization techniques proposed in Chapter 6 and Chapter 7. It describes the test traces, an interactive latency measurement methodology, the testbed configuration and a baseline MPPC encoder. Chapter 6 analyzes the long-distance redundancy in the screen updates of thin client systems and proposes a long-distance redundancy reduction scheme. Chapter 7 introduces the concept of data spikes, followed by an analysis of their influence on interactive latency. An efficient data spike reduction scheme is then proposed in this chapter. Chapter 8 concludes this thesis and discusses possible research directions.

CHAPTER 2

THIN CLIENT COMPUTING SYSTEMS

2.1. Background

A thin client system is a special form of client/server (C/S) system. In the early 1990s, Gartner Research proposed a famous application partitioning scheme to represent different designs of C/S systems [74]. According to this scheme, an application is divided into three logic layers: presentation layer, application layer and data access layer. A layer can be wholly or partially assigned to the client or the server. Different assignments of the layers result in five distinct models: *distributed presentation*, *remote presentation*, *distributed logic*, *remote data management* and *distributed data management*, as shown in Figure 2.1. Among these models distributed presentation and remote presentation model are used by existing thin client systems. While a few thin client systems such as Canoo's Ultra Light Client (ULC) [32, 33], Classic Blend [63], Remote JFC [66] and IBM's Thin Client Framework (TCF) [34] adopt a remote presentation model, the main stream of existing thin client systems adopt the distributed presentation model. A key advantage of distributed presentation model over remote presentation model is that the latter requires applications to be specially developed for it and thus can not support legacy applications while the former has no such requirements.

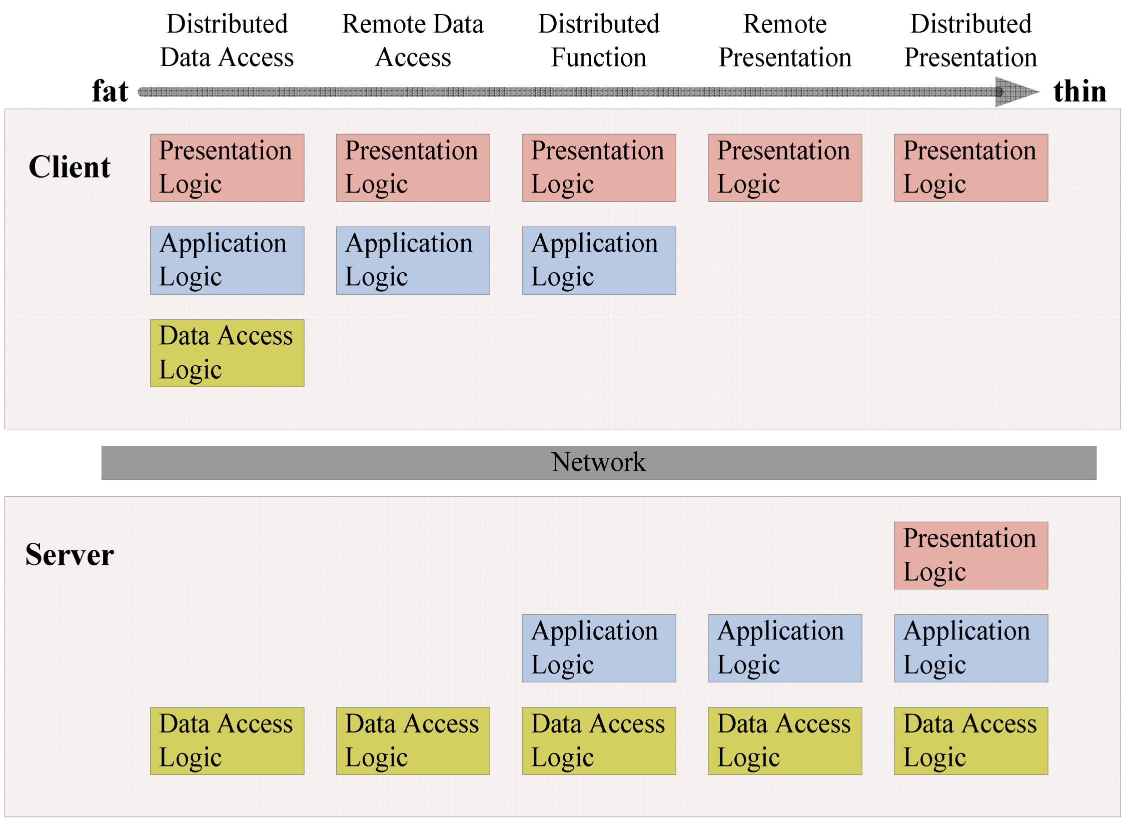

Figure 2.1: Gartner Group C/S application partition models

In contrast to the remote presentation model which assigns the entire presentation layer to the client, the distributed presentation model assigns only the application-independent part of the presentation layer to the client. The remaining presentation layer parts as well as the application and data layers all run on the server. The client merely displays the graphical representation of the application's GUI on the client and forwards user inputs such as keystrokes and mouse clicks to the server. As a result, distributed presentation architecture only requires limited processing power from the client and is able to manipulate existing legacy applications remotely. In the remaining of this dissertation, we only discuss the thin client systems with distributed presentation architecture.

2.2. Display Updates Encoding

A critical part of a thin client system is a thin client protocol. Thin client protocol defines the display primitives to be used to represent the screen updates sent from the server to the client. Different types of display primitives require intercepting the screen updates at the different layers of a display system, and require different encoding/decoding methods. Generally, the display primitives used in existing thin client systems can be classified into three types: *high-level vector primitives*, *low-level vector primitives, and bitmap-based primitives*. Accordingly, we classify the existing thin client systems into three categories: *high-level vector thin client systems (HVTC)*, *low-level vector thin client systems (LVTC)* and *bitmap-based thin client systems (BTC)*.

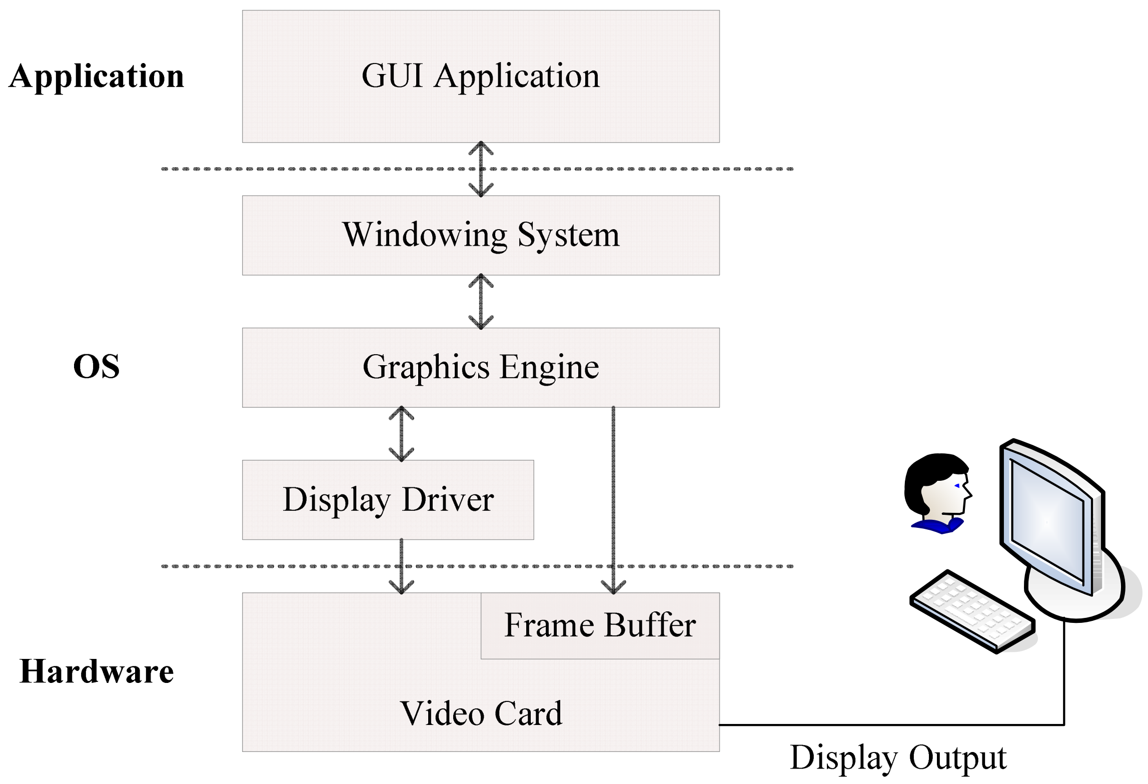

Figure 2.2: A desktop display subsystem

In order to illustrate how different types of thin client systems work, here we briefly describe the key components of a display subsystem on a modern computer. As shown in Figure 2.2, display subsystems are usually organized in a layered structure, with three software layers over the displaying hardware: *windowing system*, *graphics engine* and *display driver*. *Windowing system* is a graphics user interface (GUI) based on windows. A window is a virtualized area of the display, typically rectangular in shape, containing some graphics elements that displays the output of a program and may allow keyboard and mouse input from the user. *Windowing system* provides a high-level abstraction of the graphics hardware. It manages windows from different programs, which enables the computer user to work with multiple programs at the same time. *Graphics engine* interprets the sophisticated device-independent graphics operation requests from the *windowing system*. It translates these requests into some low-level drawing primitives which the underlying display driver can understand. *Display driver* then translates the primitives into commands for the video hardware to draw graphics on the screen. The video hardware inside a PC usually consists of a set of graphics chips and a dedicated memory to hold the contents of a single screen image. Such a video memory is called frame buffer. In some cases frame buffer shares the PC's main memory. The raster image held in frame buffer is sent to the user's display from the video hardware via a VGA or DVI connector.

All three types of thin client systems are able to intercept the screen updates without modifying the legacy applications. BTC operates at the lowest layer by simply reading the actual pixel values in the frame buffer. LVTC intercepts the elementary display primitives that are sent from the graphics engine to the display driver. HVTC intercepts screen updates at the application layer, capturing the display commands issued by the application. In the following sections, we will separately introduce these three types of thin client systems in terms of architecture, features and representative examples.

2.3. Bitmap-based Thin Client Systems (BTC)

Generally, bitmap-based primitives are operations for drawing bitmaps. BTC treats screen updates as a sequence of bitmaps without differentiating between synthetic images and real images. The pixel values of the bitmaps are obtained from the lowest abstraction layer of the display subsystem, frame buffer. Figure 2.3 shows the typical architecture of BTC. As shown in the figure, the thin client server simply reads the screen updates from the frame buffer and forwards them to a protocol encoder. The encoded data are sent to the client and are decoded, and then updated to the client's frame buffer. The encoded data can be optionally compressed on the server side to further reduce the data size.

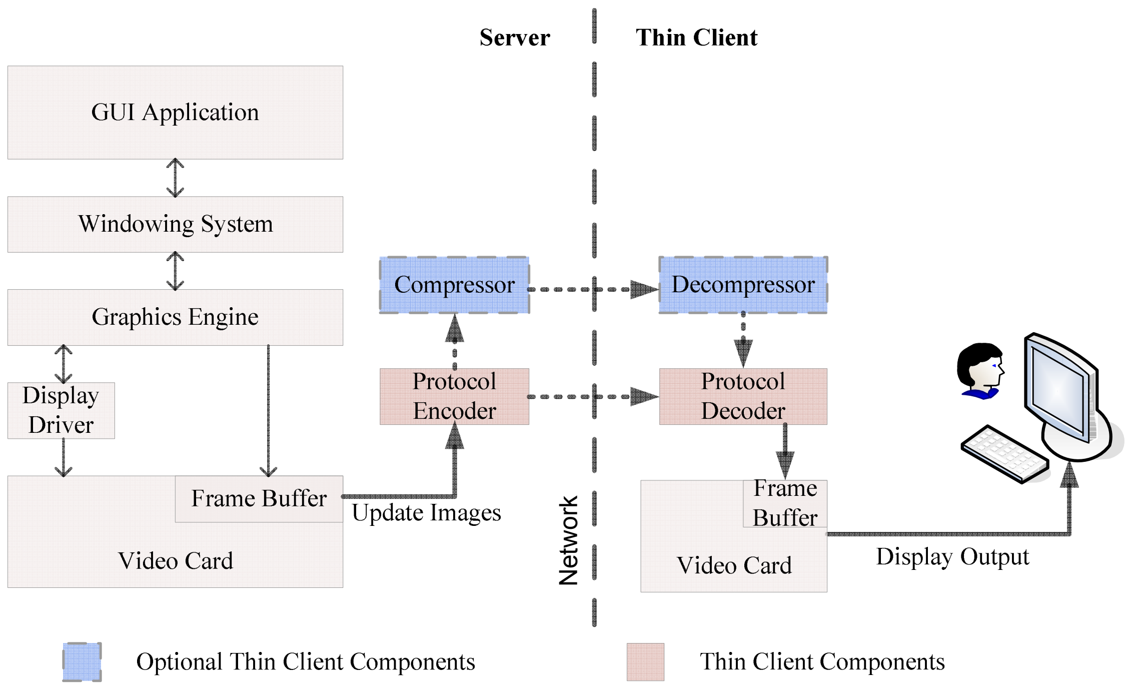

Figure 2.3: BTC architecture

Synthetic images can be a mixture of images, vector graphics, texts, etc. This is often the case for an application's GUI. For vector graphics and texts, using the image-oriented encoding and compression methods usually is not very efficient. As a result, BTC is generally considered to be less bandwidth efficient than other types of thin client systems. Nevertheless, it requires a less complex client and is more platform-independent. For example, AT&T VNC [25] employs a single graphics primitive for filling a screen region. Such a simple encoding results in its very thin client and its applicability cross various operating platforms.

2.3.1. Virtual Network Computing (VNC)

VNC [25] is a freeware developed at AT&T Laboratories in Cambridge, England aimed for remote operating of X windows and Microsoft Windows. The technology underlying VNC is a simple remote display protocol, named RFB for Remote Frame Buffer [26]. As the name implies, RFB protocol works at the frame buffer level. This protocol employs a single graphics primitive: "put a rectangle of pixel data at a given x, y position" [26]. RFB protocol can be implemented on various operating systems and hardware devices with some basic display capabilities and a communications link. This protocol can work over any reliable transport protocol such as TCP/IP.

VNC supports various encoding schemes for screen updates: RAW, RRE, CoRRE, Hextile and COPYRECT. RAW encoding is the lowest common denominator supported by any client. With this encoding the update data for a rectangular region is simply sent in left-to-right scanline order. Other three encodings, RRE, CoRRE and Hextile, are all based on run length encoding. In run length encoding a sequence of data of the same value (*runs*) is represented with a single data value and a count. COPYRECT encoding can be used when the client already has the same update data elsewhere in its frame buffer. This encoding simply replaces the repeated rectangular region with an (x, y) coordinate and the region's width and height. The coordinate gives a position in the frame buffer where the client can copy the update data of that rectangular region.

The above encoding techniques can greatly reduce the original data size. Some derived variants of VNC provide additional optimization options. For example, UltraVNC [27] has an on-screen cache and a cache encoding mechanism which saves two latest updates for each pixel of the screen. The latest update of the pixel is saved in a remote frame buffer, and the second latest update is saved in a cache. The new updates will be check against the saved updates to pass only the different screen regions.

2.3.2. Sun Ray

Sun Microsystems in 1999 came out with Sun Ray [76, 77], which was introduced as a stateless thin client solution aimed at enterprise environment. Among all existing thin client systems Sun Ray is the only product tightly coupled to the hardware since the client software is stored in the firmware and loaded on start-up. Sun Ray server software runs on Sun servers with the Solaris or Linux operating system. Sun Ray clients are dumb terminals connected to the Sun Ray server software. Sun Ray client comprises a smartcard reader and often comes with a flat panel display. It allows users to log in or to reconnect just by plugging in their identification cards.

The underlying display protocol for Sun Ray is a bitmap-based network protocol, Appliance Link Protocol (ALP), which is similar to the RFB protocol used in VNC.

Supporting audio output redirection makes ALP different from RFB. And, it also roams the audio streams from one Sun Ray terminal to the other when a session is disconnected and reconnected from somewhere else. With no outstanding compressions, the protocol is only usable on 10/100Mbps Ethernet networks. The protocol operates over a proprietary network protocol based on UDP/IP. ALP is a proprietary protocol and Sun Microsystems keeps away any in-detail information about the exact protocol.

2.4. Low-level Vector Thin Client Systems (LVTC)

Low-level vector primitives are simple display commands such as drawing lines, drawing rectangles, blitting bitmaps, etc. These low-level vector primitives are captured on the display driver layer. The typical architecture of a LVTC system is shown in Figure 2.4. On the server side, LVTC inserts a virtual display driver into the display system to intercept the low-level drawing primitives passed from the graphics engine. The captured drawing primitives are encoded by a protocol encoder and then usually processed by a compressor. On the client side, the received data are decompressed and decoded. The decoded drawing primitives are either passed directly to the client's display driver or indirectly to a graphics engine if the display driver cannot handle some of the drawing primitives defined by the protocol. In that case, the graphics engine will translate the primitives into display commands that the display driver supports.

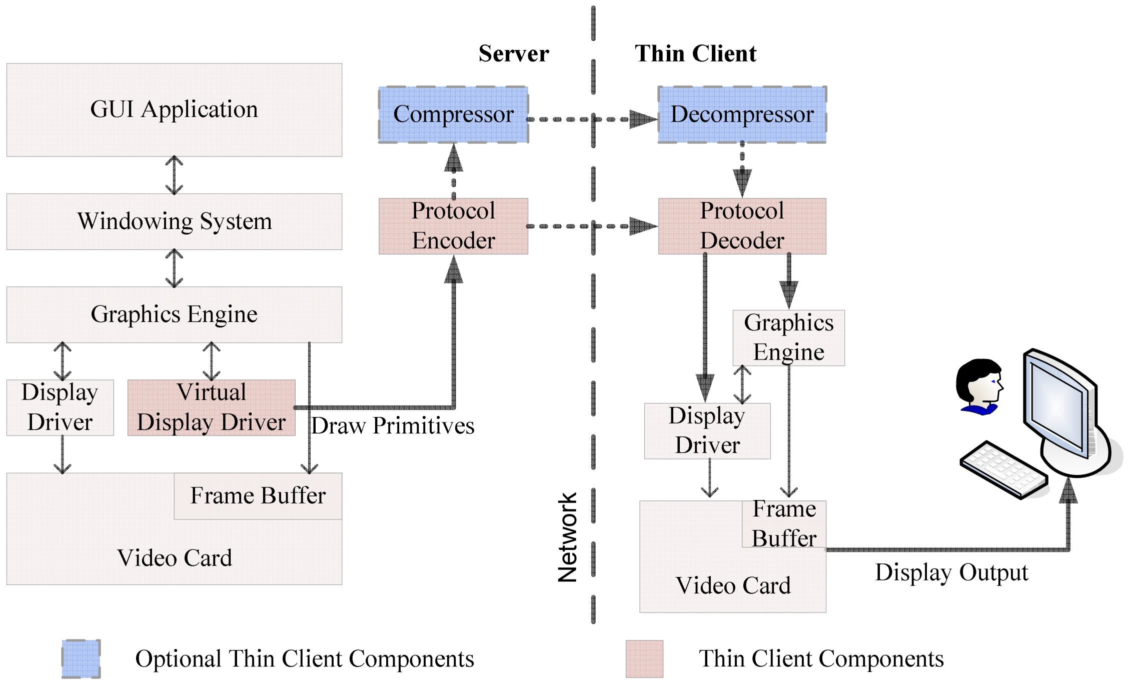

Figure 2.4: LVTC architecture

Compared to bitmap-based primitives, low-level vector primitives may lead to a more bandwidth efficient thin client system as using vector primitives can more efficiently represent most synthetic images than using bitmaps. However, as shown in the figure, such primitives require additional supports on both servers and clients and thus increase the complexity of the whole thin client system. In addition, the protocol encoder and decoder of a LVTC will also be more complex than that of a BTC as the vector drawing primitives are much more complex than merely bitmaps. A more complex client may prevent LVTC being used on some resource-limited computers. Below, we discuss three representatives of LVTC: Microsoft Terminal Server [72, 73, 78], Independent Computing Architecture (ICA) [71, 79] and Tarantella [30, 80, 81].

2.4.1. Microsoft Terminal Server (RDP)

In the early 1990s, Windows NT was designed for single-user workstations with no multi-user operation option. In 1995, Citrix promoted an independent version of NT 3.51 called WinFrame to provide multi-user support. Later, Microsoft authorized Citrix to introduce the thin-client support into the NT code base. In 1998, Microsoft released Microsoft Windows NT server 4.0, Terminal Server Edition [72, 73, 78], the commercial version of its thin client solution.

The default communication protocol between a Terminal Server and its clients is called Remote Desktop Protocol (RDP). RDP is based on T.128 protocol [28], a member of the International Telecommunications Union's (ITU's) T.120 [29] protocol family. Although RDP can be extended to run on various transport protocols including NetBEUI and IPX/SPX, the latest RDP version (RDP 5.0) only supports TCP/IP.

MS Terminal Server employs various mechanisms to reduce the amount of data transferred over the network. Examples are glyph cache, string cache, bitmap cache, and compression. Glyphs and strings are respectively cached in memory buffers on the client. Images, icons and cursors etc. are cached in another 1.5M bytes buffer called bitmap cache. In RDP 5.0, Terminal Server is augmented with a 10M bytes persistent disk cache for bitmaps. If necessary, the bitmaps cached in memory can be saved to the persistent disk cache, which can be reused cross user sessions. Before being passed to the network interface, the update data are sent to a Microsoft-Point-to-Point-Compression (MPPC)

[43] encoder which implements the general Lempel-Ziv (LZ77) [13] compression algorithm to further reduce the data size.

These optimization mechanisms are helpful to improve the overall performance, especially when running applications across low-bandwidth and high-latency network connections. They make the Terminal Server bandwidth efficient with an average data volume of about 30kbit/s [50].

2.4.2. Independent Computing Architecture (ICA)

In 1995, Citrix, one of the pioneers in the server-based computing, licensed the source code of Windows NT 3.51 and developed their proprietary Independent Computing Architecture (ICA) protocol [35]. The ICA reproduces the user interface of an application on separate terminals to support multiple remote users. The product was called WinFrame (later superseded by MetaFrame). MetaFrame offers clients for various platforms like Macintosh, OS/2, Linux and all kinds of UNIX, and even Java clients, Netscape plug-ins, and Active X for Microsoft Internet Explorer.

MetaFrame is equivalent to Terminal Server in architecture except for several features which are not supported in Terminal Server. These features include application publishing, multiple servers load balancing, session shadowing, anonymous users, etc. Like RDP, ICA utilizes various compression and caching mechanisms such as object-specific compression and bitmap cache. These mechanisms are shown to be efficient and on average reduce 50% original network traffic [98]. MetaFrame is one of the most bandwidth-efficient thin client solutions and is usable over low-bandwidth connections like dial-ups for some applications. The average bandwidth usage of MetaFrame was said to be below 20kbit/s [98] but the test conducted by Tolly Research [31] argued that 30kbit/s seems a more realistic figure. The dynamic configuration feature of ICA protocol stack makes it independent on the underlying transport protocol. So it can run on many common transport protocols, such as TCP/IP, NetBEUI, IPX/SPX, and PPP and on many popular network connections, such as dial-up, ISDN, Frame Relay and ATM.

2.4.3. Tarantella (AIP)

In late 1997, SCO released the Tarantella [30, 80, 81] (later superseded by Tarantella Enterprise) which "delivers any application, to any user, anywhere". Tarantella is a middleware that allows thin clients to connect to server-based applications via normal Java-enabled browsers. This makes Tarantella applicable to various platforms, including Windows, Web, UNIX, Mainframe and AS/400.

Tarantella Enterprise is typically installed on one or more dedicated servers. It maintains the information about the users and their applications. To access their applications, users are authenticated by a Tarantella server through their web browsers. This is done by a downloaded Java applet. The Tarantella server determines if a user has the right to access an application. This set of applications accessible to a user is then presented in the form of a "webtop".

By introducing a Tarantella server between application servers and client devices, the servers and clients are completely shielded from each other. The Tarantella server handles the communications between application servers and client devices through the proprietary Adaptive Internet Protocol (AIP). AIP is a wrapper to a number of protocols such as RDP, ICA and X11. Its job is to deliver the best user experience that makes the user feel as if the application is running on the client device. This is achieved by using heuristic mechanisms to evaluate the current network conditions, and then accordingly adapting the way data is transferred between the client device and Tarantella server. AIP also provides a set of optimization means such as command-specific compression, interlaced images, graphical optimization and delayed updates for administrator to fine-tune the AIP settings.

2.5. High-level Vector Thin Client Systems(HVTC)

High-level vector primitives provide drawing commands associated with a graphical context. One example is: "draw a line from (x_1, y_1) to (x_2, y_2) using the graphical context GC with dashed line mode set". Like LVTC systems, HVTC systems are usually more bandwidth-efficient than BTC systems as they often can represent some screen updates with simple parameterized drawing commands.

However, HVTC systems are significantly different from LVTC systems. The former is based on high-level vector primitives whereas the latter is based on low-level vector primitives which have no semantic knowledge of the application's GUI objects such as "the rectangle (x_1, y_1, x_2, y_2) is the boundary of the window *w*". The difference is induced by the different ways of representing a display command. LVTC represent a display command in a way that can be easily understood by a display driver or hardware, while HVTC express a display command in a way that is more convenient to the programmer. In the following we describe X Window, the most widely used HVTC system.

2.5.1. X Window

X Window system [14, 15, 16] was developed in the mid 1980s at MIT to provide distributed graphical computing in UNIX environments. The current protocol version, X11, appeared in September 1987. X client and X server communicate over a reliable duplex (8-bit) byte stream. Because the communication requires nothing more than a reliable duplex byte stream, X is usable over various transport protocols such as TCP [17], DECnet [18], and Chaos [19].

X protocol [20] splits an application's presentation and application logic by introducing a layer of graphics directives called X Library (Xlib) [21]. Since the separation is within an application, applications have to be developed specifically for X protocol with the help of Xlib. On the top of Xlib is a set of toolkits which defines the actual look-and-feel widgets. The toolkits are helpful to accelerate the speed and efficiency of developing X applications.

In X Window system, X server runs on a machine physically connected to a display monitor and manages the monitor, keyboard and mouse. X clients are applications that send drawing requests to the X server. Client applications can run either remotely via network or on the local machine. An X server allows multiple simultaneous connections from different clients. To achieve this, an X server multiplexes requests from the clients to the display, and demultiplexes keyboard and mouse inputs back to the appropriate clients. On the other hand an X client can have connections with multiple servers at the same time, sending every server the same display requests. Figure 2.5 gives a schematic overview of the architecture of X Window.

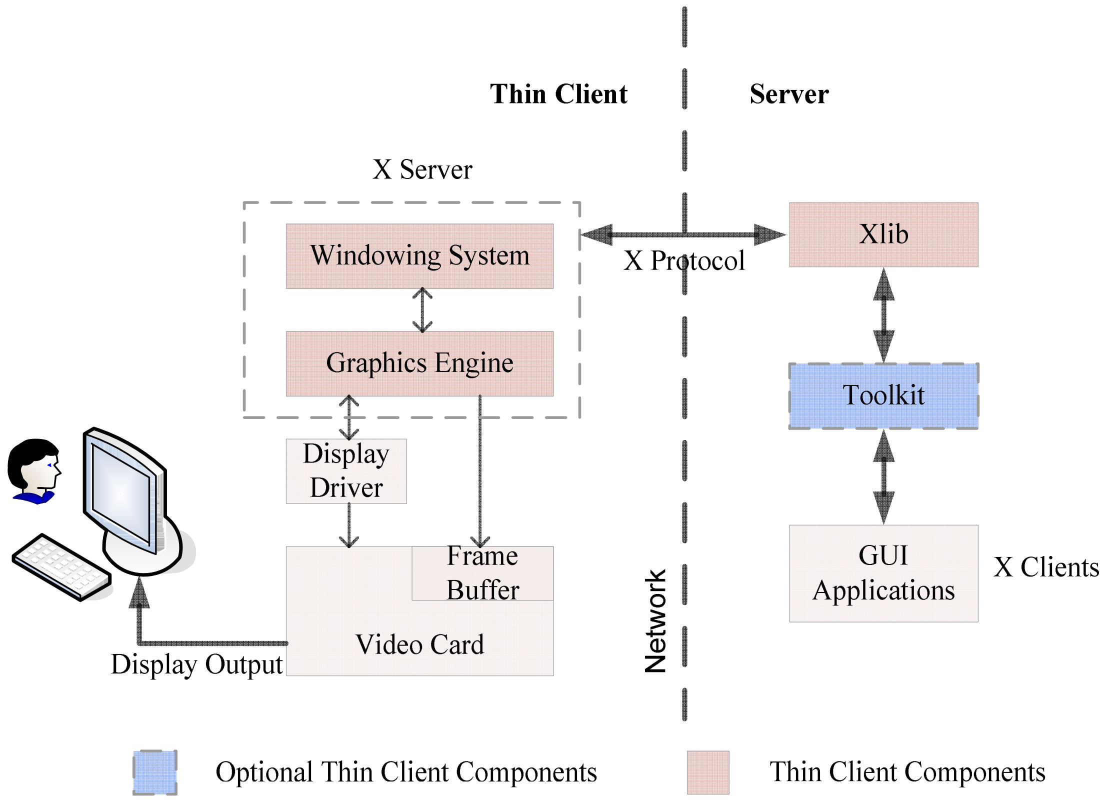

Figure 2.5: HVTC architecture: X Window

Apparently X window should have a higher performance than RDP as it employs high-level vector primitives. But the original X window contains almost no sophisticated data compression or caching optimizations. This makes its performance evidently lower than that of RDP. Nevertheless, some extensions of X (e.g. Low-Bandwidth X [22, 23, 24], NoMachine NX [82]) adds optimization methods that can greatly boost X window's performance. A limitation of X Window system is that it only supports the applications aware of X protocol.

2.6. Summary

In this chapter we presented a background for subsequent chapters by looking into various thin client systems. We classified the existing thin client systems into three types according to the different display encoding mechanisms they used. In particular, we reviewed six popular thin client systems: AT&T VNC, Sun Ray, Microsoft Terminal Server, Citrix Metaframe, Tarantella Enterprise and X Window in this chapter. A summary of the characteristics of the six systems is given in Table 2.1.

Bitmap-based primitives are the simplest display encoding method to implement a thin client system. Although it is considered to be less bandwidth efficient compared to the ones with higher-level primitives, its simplicity, platform-independence and small client footprint make it naturally suitable for mobile computing applications.

Among the six systems, Microsoft Terminal Server, Citrix Metaframe and Tarantella Enterprise which adopt low-level vector primitives as display encoding are the most bandwidth efficient. As commercial products, all of them deploy various optimization mechanisms like data compression, client caching to reduce as much network traffic as possible. However, this is not always the case. Some studies [9, 10] show that VNC and Sun Ray may surpass LVTC systems in some situations (e.g. for some wide-area network environments).

X Window employs high-level vector primitives. X server is quite platform-dependent for it relies on the special Xlib to handle the display requests. Without using the extension [22-24, 82], X behaves the worst among the six in many network environments especially in wide-area network environments [9, 10].

	VNC [25]	**Sun Ray** [76]	**RDP** [28]	**ICA** [35]	**Tarantella** [30]	**X Window** [14]
Encoding	bitmap-based primitives	bitmap-based primitives	low-level vector primitives	low-level vector primitives	low-level vector primitives	high-level vector primitives
Server platform	Windows, UNIX, Mac	Solaris, Linux	Windows	Windows, UNIX	Windows, UNIX	UNIX
Client platform	Windows, UNIX, Mac, Java	No OS (only a firmware)	Windows, UNIX, Mac, Java	Windows, UNIX, Linux, OS/2, Mac, Java	Any platform supports Java-based browser	Windows, UNIX, Mac, Java
Transport protocol	TCP/IP	UDP/IP	TCP/IP	TCP/IP, NetBEUI, PPP, etc.	TCP/IP	TCP/IP, DECnet, Chaos, etc.
Compression	included in some encoding methods	none	MPPC	LZ	adaptively enabled (RLE and LZW at low bandwidths)	supported by some extensions
Client-side caching	on-screen cache, local frame buffer	local frame buffer	glyph, bitmap, string cache (1.5 MB RAM, 10 MB disk)	glyph, bitmap, string cache (3 MB RAM, 1% of disk)	depends on protocol tunneled	application-/toolkit-specific, usually none
Bandwidth efficiency	low	low	Very high	Very high	Very high	High (with extensions)

Table 2.1: Thin client computing systems summary

CHAPTER 3

THIN CLIENT OPTIMIZATION TECHNIQUES

For most client/server systems, end-users generally have little concern about where processing takes place and where data are stored, as long as the interaction is fast, consistent, and seamless. The most critical performance criterion is the crispness of the user interaction [8, 9, 10]. Ideally, the response time of an operation should be short enough to let end-users feel that the applications are running on their local machines. It's hard to give users a pleasant experience if the dialog appears with perceptible delay after users press a button, or if the onscreen rubber band cannot track users' mouse movements in resizing windows.

To satisfy the end-users' expectation on interactive performance, almost all existent thin client systems make use of some kinds of generic or proprietary optimization techniques. These techniques can be largely classified into three categories: data compression, client caching and client localization. In the following sections we will review these optimization techniques to detail.

3.1. Data Compression

As mentioned in Chapter 1, reducing network traffic is an effective way to reduce user operation latency. To reduce network traffic, data compression is possibly the most direct choice. Various data compression methods have been used or proposed for thin client systems. These include generic compression algorithm such as LZ (Lempel-Ziw) [13] and specially designed compression algorithm such as Thin Client Compression (TCC) [36] and Differential X Protocol Compression (DXPC) [83], etc. We will discuss them in the following.

3.1.1. LZ Compression

Abraham Lempel and Jacob Ziv developed the lossless data compression algorithm LZ77 [13] which is broadly used in various scenarios where data compression is needed. Both the encoder and the decoder of LZ77 keep track of an amount of recently processed data (history) in a structure called a sliding window. The encoder searches for matches between the data being compressed and the history data in the sliding window. It achieves compression by replacing matched strings with length and distance pairs. The decoder restores the matched strings from the length and distance pairs by referencing the history.

LZ77 is a generic compression algorithm that can be applied to any data if only the data can be represented as a byte stream. LZ77 is therefore independent of the thin client systems and the underlying operating systems. It is also applicable to the various application workloads of thin client systems.

Many empirical results show that LZ77 is an efficient compression algorithm. For example, after using MPPC (a compression protocol based on LZ77), Microsoft Terminal Server on the average reduced data size to 40-50% of the original; after upsizing the sliding window of LZ77 from 8K bytes to 64K bytes on the average, a further 30% is reduced [95].

To maximize the compression performance, LZ77 searches for the longest matches between history data and the data being compressed. Searching for the longest matches is a classic computer science problem. The fastest exact searching algorithm proposed by Masek and Paterson takes $O(n^2 log\ log\ n/\ log\ n)$ time (n is the sliding window size) for unbounded alphabet size [44]. Such a speed is not acceptable for compressing the data on the fly. However, the searching speed can be greatly improved by using an approximate searching method, though at the cost of possibly missing the longest matches. For example, with the help of a hash table the searching time can be reduced to $O(n)$ [45].

3.1.2. Thin Client Compression (TCC)

In 2000, B. Christiansen et al. proposed a new compression algorithm called Thin Client Compression (TCC) [36], aiming to exploit both local and global redundancy in synthetic images. It consists of three phases: pattern matching, substitution of the matched patterns with references, and coding.

To compress an image, TCC makes two passes. In the first pass, TCC scans the image from left to right, top to bottom and segments the image into marks and the residue image [49]. A mark is defined as "a set of connected pixels that is surrounded by a single-color 4-connected [96] boundary, and no pixel in the set that is adjacent to this boundary is of the boundary's color" [36]. Once a mark is identified, TCC searches for an exact match of the mark in a codebook. If no match is found the new mark is added to the codebook. If a match is found, TCC replaces its bounding box with a reference into the codebook and fill the box with the surrounding color. In the second pass, the unique marks in the codebook and the residue image after extracting all marks are coded by three piecewise-constant models [47, 48]. The pointers into the code book and the positions of all marks are coded by a multi-symbol QM coder [84].

TCC is designed for compression of synthetic images. In compression, it needs to scan the images to extract marks. Thus it can only be applied in bitmap-based thin client systems where screen updates are represented by bitmaps. In addition, TCC can hardly extract marks from smooth-toned images that are popular in multimedia applications. This makes TCC unsuitable for multimedia applications.

Experiments showed that TCC has better compression performance than other four image compression algorithms, PWC [47, 48], GIF, FABD [85] and JBIG2 [86], especially for the textual images [36]. On average, for individual image TCC outperforms PWC by a factor of 1.5 and GIF by 4.8; for a sequence of images, it outperforms PWC by a factor of 2.6 and GIF by 8.2.

Experiments also showed that when compressing individual screendump, TCC is about one order of magnitude slower than GIF (using LZW algorithm [97]), a bit slower than PWC, but faster than FABD and JBIG2. When decompressing the statement keeps true with the only exception that TCC outperforms PWC in decompressing textual images.

3.1.3. Streaming Thin Client Compression (STCC)

In 2001, B. Christiansen et al. proposed a streaming version of TCC (STCC) [37], aiming to reduce the end-to-end latency by pipelining compression, transmission and decompression. The pipeline shields the long delays caused by large updates and thus improves the user experience. Compared to TCC, STCC uses a less strict definition of marks and, based on this definition, a streaming boundary trace. In addition, STCC introduces a tree-structured codebook that allows for incremental coding. As a result of these changes, STCC only need to make one pass over images while TCC makes two passes.

As a streaming extension of TCC, STCC is inherently only suitable for thin client systems with bitmap-based encoding and applications involving no or not much smooth-toned images.

STCC were compared with TCC and PWC in terms of compression performance [37]. The results are summarized as the following. In an individual-screendump test, STCC outperforms PWC for the textual images by a factor of about 1.7 but only a slightly better for graphical images. Despite of the row-by-row encoding, STCC performs very closely to TCC and is outperformed by 4.4% on average. In a sequence-of-screendumps test, STCC outperforms PWC by a factor of 2.6. Compared to TCC, STCC compresses some images more efficiently since it extracts more marks from the image than TCC and replaces them with only pointers. However, in most cases TCC averagely outperforms STCC by 4.7%. The authors also showed that STCC achieves shortest end-to-end latency among the three methods when testing on low bandwidth connections (64-512Kbit/s) like DSL and cable modem connections.

Experiments showed that STCC has shorter end-to-end latency than TCC for the bandwidth above 64Kbit/s, and has shorter end-to-end latency than PWC for the bandwidth below 512Kbit/s [37]. This implies that like TCC, STCC is slower than PWC.

3.1.4. 2-D Lossless Linear Interpolation Coding (2DLI)

In 2002, Fei Li and Jason Nieh developed a new coding algorithm, called 2-D lossless linear interpolation algorithm (2DLI) [38], to improve the support for multimedia applications in thin client computing environments.

2DLI treats screen updates as linear pixel arrays by defining the x-coordinate of a pixel as the cardinal number of the pixel in the update rectangle and defining the y-coordinate as the R/G/B value of the pixel. The server selects and transmits only a small subset of pixels from the pixel array representing a screen update. The client recover the screen update from the received pixels using piecewise linear interpolation to achieve the best visual quality. The goal of 2DLI is to select an optimal set of pixels from the array to minimize the data size transferred from server to client while maintaining the same visual quality.

Selecting an optimal set of pixels is an exponential computational problem. 2DLI is a greedy algorithm that achieves the linear computational complexity by searching a minimal isolated pixel set. An isolated pixel is a pixel that can not be horizontally or vertically interpolated by gradients with a constant δ through its neighboring pixels in the update region. The values of isolated pixels are delivered to clients to reshow the update region through 2-D linear interpolation.

Since 2DLI only requires the positions and the R/G/B values of pixels to interpolate and compress, it can be applied on both synthetic images and smooth-toned images. Nevertheless, the values of all pixels are only available for bitmap-based encode. Hence 2DLI can only be used for bitmap-based thin client systems such as VNC.

The compression performance of 2DLI was compared to that of JPEG, LZ and Hextile (used by VNC) on four different workloads: smooth-toned images, web pages, screendumps, and instructional videos [38]. 2DLI outperforms the other three for the last three workloads while it performs second to JPEG for the smooth-toned images.

2DLI has a linear computational complexity O(*n*) for both encoding and decoding, where *n* is the number of pixels. On average the encoding is 2.41 times faster and decoding is 1.75 times faster than JPEG. But 2DLI is slower than LZ and Hextile for both encoding and decoding.

3.1.5. Differential X Protocol Encoding (DXPC)

Differential X protocol compression (DXPC) [83] is an X protocol compressor designed by Brian Pane to improve the speed of X11 applications over low-bandwidth links like dialup or satellite connections. It compresses X protocol in six steps. In the first step, it strips unnecessary data fields like some padding data in an X message. In the second step, it shrinks fields with a limited value range. For example, DXPC transmits a one-byte long Boolean field in an X message as a single bit value. In the third step, it shrinks fields with typically small values but still handles the cases where the values are large. In the fourth step, DXPC caches different X message types on both sides of the link. With the help of the cache, instead of transmitting a full X message DXPC transmits a much shorter one with differential field values based on the previously sent message. In some cases, field values keep changing in some pattern. For example, the sequence numbers contained in messages generated by X server keep increasing until they reach 65536. In the fifth step DXPC caches the last six deltas so that it can do further encoding based on the pattern of deltas. Lastly, DXPC caches large blocks of data that need to be repeatedly transmitted (e.g., X resources).

DXPC requires semantic knowledge of the X message fields for its differential compression algorithm. Thus it can not be deployed in thin client systems without such semantic knowledge. DXPC is capable of compressing X messages by as much as 10 times. However, for most X applications, DXPC achieves compression ratio ranging from 3:1 to 6:1, while on the average a compression ratio of 4:1 is achieved [83].

3.2. Client Caching

The idea of client caching is very simple. It caches some screen updates that may be used more than once on the client. Two representative examples of client caching are: graphics object cache used in Microsoft Terminal Server and Citrix MetaFrame, and Message Store used in NoMachine NX.

3.2.1. Basic Graphics Object Cache

Microsoft Terminal Server and Citrix MetaFrame cache basic graphics objects including bitmaps, glyphs, and strings to improve the encoding performance of their proprietary thin client protocols, RDP and ICA (a variant of RDP).

A glyph is made up of a character (such as a letter or number) and the font information that is required to calculate display of the character. In other word, a glyph is a bitmap representation of the character. The same character in different fonts is thus represented by different glyphs. A string is a sequence of characters. RDP saves bitmaps, glyphs and strings sent previously in memory caches on both client and server computer. A bitmap could be an icon, image, etc. A large bitmap is usually split into small bitmap blocks such as 64 pixels by 64 pixels. The server improves the protocol efficiency by replace the real presentation data of the cached items with cache references in the subsequent screen updates. The client recovers the cached presentation data and draws them onto the screen.

In addition to memory caches, MS Terminal Server and Citrix MetaFrame added a persistent disk cache for bitmaps. The persistent cache size is 10M bytes in Terminal Server and 1% of the total disk in MetaFrame. The RAM cache is always the destination for the initial bitmap cache. Whether to use the persistent disk cache or not is negotiated during the session establishing stage. If it is turned on, the server may choose to permanently store some bitmaps to the client's disk. The server instructs the client to retrieve these bitmaps from the persistent cache when they are needed but have been evicted from the memory cache.

The basic graphics object cache mechanism requires simple semantic knowledge of screen updates in order to distinguish these objects. It therefore cannot be applied to thin client technologies with bitmap-based encoding.

From the information published by Microsoft, the basic graphics object cache is very effective which together with compression make the average bandwidth requirements as low as 10kbit/s [95]. However, the Tolly research [50] showed that the 30kbit/s bandwidth requirement seems to be more reasonable. The information on the computational complexity of the basic graphics object caches used by RDP and ICA is not publicly available. However for RDP we can observe that LRU is used for the cache management. Furthermore, an algorithm is needed to determine whether a bitmap has been cached or not, possibly by comparing bitmap pixels.

3.2.2. NX Message Store

NoMachine NX [82] stores the most recent X messages in a cache called Message Store. NX divides an X message into two parts: a fixed-size identity part and a variable-size data part. NX calculates a MD5 checksum of each X message covering its data part and some fields of identity part. The chosen fields of identity part are those that will not change across different instances of the same X message. NX uses this checksum as a key to search a message in Message Store. If a message is found, NX sends only the status information, together with the reference to the cached message and a differential encoding of all those identity fields that are not used in calculating the checksum.

NX Message Store requires the semantic knowledge of the X message fields to partition the message into data part and identity part and to choose the non-changeable fields as identity. Thus it can not be deployed in thin client systems without such semantic knowledge.

The average cache hit ratio of Message Store is 60%-80%, but can be close to 100% for some messages such as fonts, images, etc. [82]. Overall, Message Store and other compression algorithms together can make X protocol usable over low-bandwidth links such as PPP (28.8 kbps), which is not possible with the original X protocol. As Message Store works closely with other compression algorithms in NoMachine, we can not know its computation complexity.

3.3. Client Localization

Client localization techniques localize parts of the application logic on the client. They directly reduce roundtrip times or reduce the total transfer time of a sequence of screen updates. Active Component Localization [39], Keyboard Activity Localization [40, 41] are examples of client localization.

3.3.1. Active Component Localization (ACL)

Researchers showed high interests in introducing thin client computing architecture into the mobile computing environment. However, the inherent characteristic of high latency in mobile communication has great ill effect on the end-to-end latency of thin clients. In year 2000, Cumbur Aksoy and Sumi Helal proposed an active component localization (ACL) mechanism [39] to reduce this ill effect on active-media applications.

Active components are parts of an application, presenting a recurring sequence of images, sound or data. Animated GIF is an example and is used by the authors to show how to localize active components. The basic idea of the localization is quite simple. The first step is to detect whether the application display keeps looping over several images. Once an *n*-state loop is detected, the server instructs the client to set aside a buffer for the *n* images in the loop. Then the client waits for *n* images, buffering them and also recording their arrival times. When the buffer is filled, the client goes into a zero communication mode. In this mode, the client displays the images in its buffer one after another according to the timing information recorded. At the same time, the server goes into a watch mode, where it watches the display behavior of the application to make sure that it obeys the loop. Once an image out of the loop occurs, the server passes it to the client and puts the client back into original thin client mode.

While the active component extraction is platform-independent and application-independent, it only optimizes the active components which do not appear quite often in normal applications except for GIF in browser and circularly played music.

The authors did a series of case studies which include one animated GIF with different number of states, and a mixture of multiple GIFs. In all tested cases, ACL detected active components with low error rate and considerably reduced the roundtrips and traffic between the server and the client.

ACL uses an image comparison method to detect the states of the application's displays. The computation complexity of the detection increases with the number of states in an active component. Besides that, active component extraction relies on horizontal and vertical line scanning over images, which is computation intensive. In sum, the average processing time of the localization is around 350ms, 750ms and longer than 1.5 second when the application's display size is 351×286, 475×419 and 631×597 pixels respectively. Compared to a user-perceptible latency which is around 100ms [1], the processing time is too long.

3.3.2. Keyboard Activity Localization (KB Pro)

In 2001, Sivasundar Ramamurthy and Sumi Helal proposed a keyboard activity localization technique [40] to reduce the roundtrips between client and server when users are typing. The keyboard localization is triggered only for certain kinds of keyboard activities, such as a "KB Blitz". A KB Blitz is defined as the situation"when the thin client user types in such a speed that display of the characters is delayed for reasons such as a slow network or a slow server".

A localization system, called KB Pro system, was developed for the Win32 ICA clients. It has three core components: a virtual channel driver *VdHook*, a client side localization process *KBWin* and a server side process *KBServer*. VdHook registers a keyboard hook when the latency is above a threshold (i.e. 200ms) and watches out for a KB Blitz on a typing application. Once such a keyboard blitz is detected, the keyboard events that can be localized are handled and displayed by KBWin in the localized area. Once a refreshing event occurs (e.g. a caret moves beyond the localized area), KBWin performs a refreshing and exits. KBServer provides a few services for VdHook. These services include opening virtual channel, providing typing applications information for VdHook, refreshing the text when refreshing requests are received from VdHook, and inspecting the network latency together with VdHook.

KB Pro is particularly useful for typing applications in which the majority of key presses actually get displayed and only a few of them involve other events like refreshing. In addition, keyboard localization requires the client to be able to hook the keyboard activities, to detect a KB Blitz, to monitor the network traffic and detect a high latency situation, and to display characters locally.

It was shown by the authors that KB Pro works best on benchmarks without refreshing event. It can clearly decrease the latency, the packets exchanged, and the bytes exchanged. The performance decreases when refreshing events increase. It was also shown that KB Pro caused only a slightly increase to CPU usage on the client.

3.4. Summary

In this chapter we classified existent thin client optimizations into three categories: data compression, client caching and client localization. We examined several representative examples of them in terms of effectiveness, applicability and computation complexity.

Data compression methods are very effective in reducing the size of the screen update data. On the average we can expect the screen update data to be reduced to 50% or less of its original size with a general compression algorithm such as LZ. Specially designed compression algorithms TCC, STCC, 2DLI, and DXPC can achieve even higher compression ratio. Data compression methods make use of the data redundancy in different types and different scopes of screen updates, and use various encoding methods from information theory to minimize the data size. LZ compression applies to any types of screen updates, but only exploits the recurrent byte strings within its sliding window (usually smaller than 64KBytes). TCC, STCC and 2DLI are designed for bitmap-based thin client systems. Among them, TCC and STCC exploit the data redundancy (identical marks, the same colors in background image) in a synthetic image or a number of synthetic images (by sharing its codebook). 2DLI exploits the data redundancy in a synthetic image or a smooth-toned image by not sending the pixels whose color values can be inferred from neighboring pixels following a linear interpolation algorithm. DXPC is designed for X Protocol. It efficiently encodes X messages by specializing on each field's value range, frequency and applying delta encoding. While complex compression algorithms can get additional reduction of the data size than a simple one, it requires more computation time and may not improve the end-to-end latency. Thus we can see that the compression algorithms used or proposed for thin client systems usually have a low computation complexity at the cost of optimal compression performance. The technique presented in Chapter 6 of this thesis aims to improve the compression performance of LZ compression algorithm for thin client computing, while at the same time maintaining a low computation complexity.

Client caching techniques are also very effective and important in reducing the size of screen update data. Some caches such as bitmap caches can on average reach a hit ratio above 80%. Like data compression, client caching techniques also make use of the data redundancy in different types and different scopes of screen updates. But client caching techniques are usually used for "objects", which can be distinguished from other types of screen updates. The objects are saved as cache items. When a cached item appears later in the screen updates, the reference to the item rather than the item itself is sent by the server. Unlike data compression methods, no complex encoding is needed for caching techniques. In addition, caching techniques usually search for redundant data in a larger range as a cache only stores the same type of objects and is usually larger than a sliding window or codebook. With simple object matching and encoding, and a simple cache management policy such as LRU, cache techniques usually cost less processing time than data compression. But generally cache techniques require more memory than data compression algorithms and can only benefit certain kinds of screen updates. While caching technique usually deal with simple object, in Chapter 4 of this thesis, we present a cache technique attempting to cache some complex objects in bitmap-based thin client computing systems to minimize network traffic. In Chapter 7 of this thesis, we present a hybrid compression-cache scheme which can benefit different kinds of screen updates.

Different from data compression and client caching, client localization techniques emulate parts of the presentation logic of an application on the client side. This reduces the number of round-trips needed for some user operations, which gives a better user experience for the network connections with long latencies. But as we mentioned for each example, the presentation logic that can be emulated on the client side with reasonable resources (memory, CPU usage) is very limited. So these techniques are only suited to a narrow range of scenarios.

After reviewing previous optimization techniques we can identify a number of requirements on the optimization techniques of thin client computing. We summarized them as below.

- As mentioned in Chapter 1, the main performance issue with existing thin client systems is that they cannot deliver satisfactory user response time in low-bandwidth and/or high-latency network environment. Hence a performance optimization technique must be able to effectively reduce user-perceptible latencies in such networks. In the following chapters, we present and discuss the performance of the techniques we proposed in such networks.
- Reducing the data size of screen updates may not necessarily lead to a reduction of the end-to-end latency. The computation time of an optimization technique must be taken into account. As the optimization mechanisms must be able to process the screen updates online, it is important for an optimization mechanism to have low enough computation complexity to operate in real time. In addition, as the thin client may only have limited memory and CPU power, the optimization technique must not consume too many resources. For each technique we proposed later, we will discuss the resources they consumed.
- A thin client system may execute a wide range of applications. Therefore, it is desirable for an optimization mechanism to be applicable to a wide range of

applications with different types of screen updates. In Chapter 5, we will describe a few quite different application workloads which we used in evaluating the techniques presented in Chapter 6 and 7.

CHAPTER 4

STATIC OBJECT CACHING SCHEME

In Chapter 3, we reviewed various performance optimization methods used in existing thin client systems. While these methods proved to be effective, they only aim to reduce the data redundancy at relatively low level such as the redundancy between pixels, primitives or bitmaps. High-level data redundancy in screen updates was not adequately considered. High-level data redundancy is commonly seen in an application's screen updates. For example, a menu or a dialog is often displayed for many times during a remote user session without any change in its presentation. We call this kind of high-level redundancy object-level redundancy, since the redundancy is between the screen updates of two graphical user interface (GUI) objects. Without any optimization, object-level redundancy may produce a large amount of network traffic and cause long latencies in user operations. The previous optimization methods such as bitmap cache, string cache, etc., which remove low-level redundancy in screen updates, can only indirectly reduce object-level redundancy while leaving a large margin for improvement. In this chapter we present a static object cache technique aimed at directly reducing object-level redundancy.

4.1. Static Object Caching Technique

We proposed a static object cache scheme to enhance a thin client server [99]. The modified architecture of the server is shown in the Figure 4.1. The core module of the architecture is called Object Filter (OF). It monitors the screen updates to be sent to the client, filtering out the presentation data of the static window objects based on the object information collected by offline profiling and instructing the client to use the previously cached presentation data of these objects. For example, in Figure 4.1, the presentation of the dialog 'logon' is static except for the areas occupied by the two edit boxes. In the original architecture, the whole presentation either encoded in bitmap-oriented drawing primitives, or vector-based draw primitives, is sent to the client. However with our static object cache technique, only the variable parts of the presentation are sent, as shown in the right cloud of Figure 4.1. Following, we will describe the terms used in this chapter and give the details of object filter.

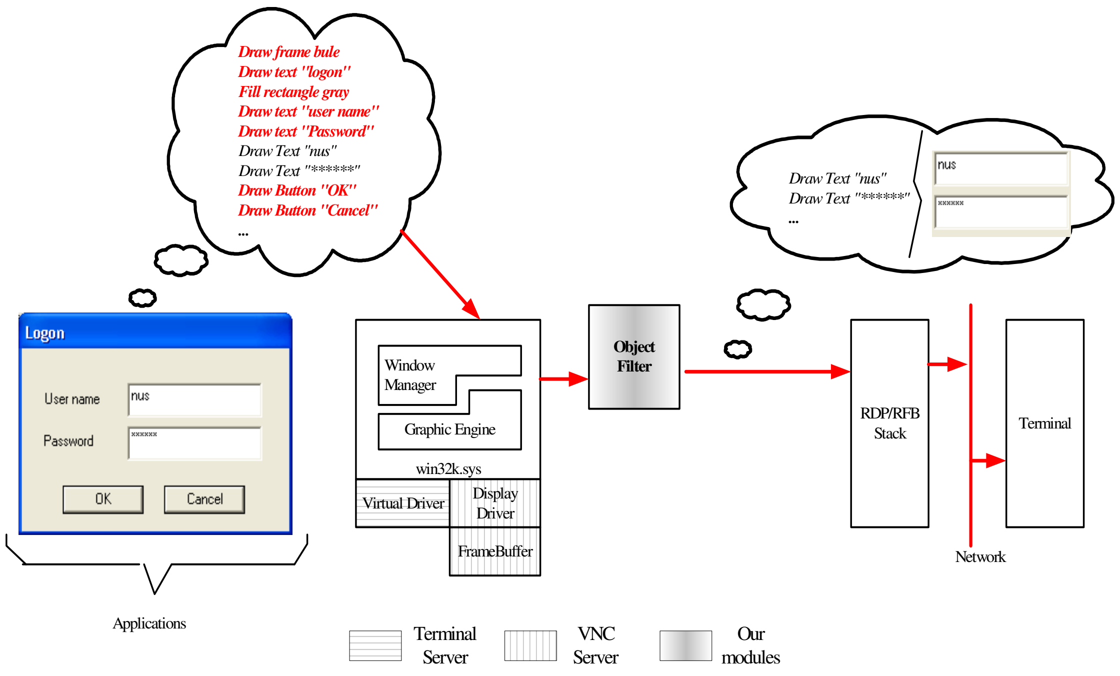

Figure 4.1: High-level overview of the proposed static object caching technique

4.1.1. Definitions

A *GUI component* is any graphical element that composes one application's GUI. It can be as basic as a picture, a button or a text label. It can also be a set of basic components clustering together to provide one specific function. The GUI components in an application are not isolated. They are linked to each other by some kinds of relationship and together form the GUI structure [51]. During run time, the GUI structure is incarnated as some instances of the GUI components and the relationship between them. We define an instance of a GUI component as a *GUI object*. The relationship between a GUI component and a GUI object is similar to the relationship between an application and a process. After a GUI object is created, it may be painted many times. One painting of an object is termed an *update*. Generally an update will only cause the visible region of the object to be painted. By 'visible region' we mean the region of the GUI object that is not hidden by other GUI objects.

Among the GUI components of an application, some have static presentation: all of their instances share the same presentation (or can be considered so without affecting the use of them) except some fixed areas that are non-overlapped. We refer to such components as *static components*. We refer to the areas of a static component's presentation that is invariable as *static areas* while referring to the areas of a static component's presentation that is variable as *non-static areas*.

4.1.2. Object Filtering Algorithm

Our OF technique consists of four elements. The first is a set of static GUI components of an application (denoted as S). For Java applications, S can be extracted from the .class files. For Win32 applications, S can be obtained from the applications' source codes or from profiling of the applications. The second is a mapping function, **Get-Component**. Given a GUI object o, this function determines whether there exists an s ($\in$ S), of which o is an instance. The third is a server-side OF algorithm, **Object-Filter**. Given the static component set S and the mapping function Get-Component, the filtering algorithm is as shown in Figure 4.2. The fourth is a client-side static component caching and presenting module. Presently we assume that all presentation data are placed in memory ahead of running the program. With this assumption, the client side handling is straightforward. On receiving a static component update message, the client first retrieves the presentation data locally and then clips them against the visible region and finally renders them at the right position.

Object_filtering

```
PROCEDURE Object-Filter (u: update of object o)
BEGIN
    s = Get-Component (o)
    IF s Exists
        IF Update-Region (u) in Non-Static-Region (o)
            Send-Client-Update (u)
        ELSE
            p = Position (o)
            v = Visible-Region (o)
            Send-Client-Message (s, p, v)
        ENDIF
    ELSE
        Send-Client-Update (u)
    ENDIF
END
```

Figure 4.2: Object filtering algorithm

4.2. An Implementation on VNC

As the implementation of our technique requires modification to both the server and the client, we choose VNC, a popular open sourced thin client system as our test bed.

4.2.1. Acquiring The Static Component Set

We developed a tool called GUI Extractor to discover static components. This is done in two stages. In the first stage, a background process monitors the target application and extracts GUI information by hooking windows message. To determine if a window is an instance of a static component, its properties such as styles, size, caption, and the relationship with the known static components are inspected. Currently we only consider top-level windows that are created without WS_CHILD style. After identifying a static component, we link it to other static components according to the owner-owned window relationship among them. If the new static component has no owner, it will be linked to the root static component (i.e. the application's main window, which is considered as a special static component). If the owner of the static component is not static, it will be linked up to its nearest ancestor owner which is static. At the second stage, the GUI data obtained are inspected and edited visually to ensure their correctness. Some components may be incorrectly considered as static components in the first stage. And it may also be wrong to consider the areas occupied by some controls as static areas based on the controls' properties. At this stage users can select appropriate static components and adjust the static regions of static components to ensure they meet the run-time requirements. In the current implementation we require that at any time there is only one instance for each static component.

Application	Extracted SCs	Static Ratio
WordPad	18	44.17%
Visual C++	66	48.06%
Internet Explorer	26	46.34%
Outlook Express	36	48.45%

Table 4.1: Static components in some Windows applications

We run our tool on some MS Windows applications including Microsoft WordPad, Visual C++, Internet Explorer and Outlook Express. Table 4.1 summarized the tool's output. The presentation data is calculated in pixels. Static ratio is the fraction of the static area on a GUI component. The results show that all identified components on average have more than 40% static areas.

4.2.2. Mapping between The Static Components and The Objects

To establish and maintain the link between static components and static objects at run time, we need to monitor the life cycle of a GUI object. This is done by the routine **LifeCycle** (shown in Figure 4.3). This routine is invoked at three important points within an object's life cycle: create time, paint time and destroy time. In MS Windows, these three points respectively correspond to the time when a window receives a WM_CREATE, WM_PAINT and WM_DESTROY message. **Add-To-Candidate-List** is executed at create time. It checks the object's relationship with other static objects to see if it is a static object candidate. We inserted the candidates into a candidate list. **Bind-Static-Component** is called when a window is to be painted and it is in the candidate list. The function will further examine the window to see if it is a static object. **Unbind-Static-Component** breaks a link between the static object and the static component when the object is to be destroyed. In an object's lifetime, these functions will only be called once. In function Bind-Static-Component and Unbind-Static-Component we need to compare the attributes of an object with those of static components. Nevertheless, as we only target top-level windows, the number of the static components is not very large (see Table 4.1). As a result, the extra CPU usage is insignificant.

Link_comp_obj

```
PROCEDURE LifeCycle (w: window, t: time)
BEGIN
    SWITCH (t)
        CASE Create_Time:
            Add-To-Candidate-List (w)
        CASE Paint_Time:
            Bind-Static-Component (w)
        CASE Destroy_Time:
            Unbind-Static-Component (w)
    END SWITCH
END
```

Figure 4.3: Maintaining the links between static components and objects

4.2.3. Object Filtering Algorithm Implementation

VNC handles an update in three steps. At the first step, the possible updates are collected using various mechanisms. At the second step, the possible updates are checked by comparing the local frame buffer with a mirror of the remote frame buffer. At the third step, the true updates that indicate differences between local frame buffer and remote frame buffer are sent to the client.

We made the following modifications to this update handling process. When a static object is to be painted, a message will be sent to notify the VNC server of an update of this static object. On receiving this message, VNC server adds the static object into a waiting list. Before checking the updates, for each static object in the waiting list, VNC server obtains its position and computes its visible static region. The information collected will be sent to the client to get these static object painted. The visible static regions will be excluded from the possible updates collected by VNC server. This prevents VNC server from checking the updates of static areas, which is CPU-intensive. At the same time, we copy the contents of the visible static regions from local frame buffer into the mirror frame buffer to ensure that the mirror frame buffer is synchronized with the remote frame buffer. The overhead caused by these modifications is very small. The LifeCycle routine described in Section 4.2.2 has established the mapping between static objects and static components. So we only need to search the mapping table to identify if an object is static or not. The position and visible region are the attributes of GUI objects which are always updated by MS Windows.

4.3. Performance Evaluation & Results Analysis

4.3.1. Evaluation Methodology

In order to evaluate the bandwidth efficiency and interactive performance of our technique, we modified the VNC client to measure the network traffic and latency associated with GUI operations. We time-stamped a user input and the last screen update it induces. All the presentation data received between the two time stamps are attributed to the network traffic resulting from this operation. The difference between the two time stamps is used as a measure of the operation latency. To guarantee the accuracy of the measurement, we altered the input acceptance process of the client by introducing delays between any two continuous inputs. This method is similar to the slow-motion versions of application benchmarks [52] employed by other thin-client performance analysis experiments [9, 53]. However, the measured latency in our experiments also includes the time between the client input is made and the input being sent, and the time between the client receives a screen update from the network and the actual image being drawn to the screen.

Each dialog was opened and closed (using an automated way) for many times and the average latency and network traffic of opening each dialog was used as the evaluation criteria. To eliminate the inconsistency due to human operations, we altered the client to record the mouse messages that lead to the events and play them back in our experiments. This is similar to the mechanism used in other capture/playback tools [54]. To model different network bandwidth, we employed a high-resolution timer to limit the data size that the client can receive during each timer interval.

4.3.2. Numerical Results

VNC provides various pixel encoding methods, allowing a large degree of flexibility in trading off various factors such as network traffic, client processing time and server processing time. The typical encoding method for LAN network environments is Hextile, which is supported by all VNC versions. The encoding method for low-bandwidth network environments varies from version to version. We chose ZRLE, an encoding method used by RealVNC [55], in our experiments. We compared the performances of our modified version to the original version for these two encoding methods. We chose from MS WordPad 6 dialogs (shown in Table 4.3), which have different static ratio (calculated in pixels) for our experiments. The configurations of the testbed machines in our experiments are shown in Table 4.2.

Role	Hardware	OS
Client	1500MHz Intel PIV 512M RAM 10/100BaseT NIC	MS Windows XP Professional
Server	1800MHz Intel PIV 26M RAM 10/100BaseT NIC	MS Windows XP Professional

Table 4.2: Testbed machine configuration

Dialogs	Static Ratio
Open	39.0%
Save As	41.2%
Date & Time	60.6%
Font	63.8%
Page Setup	80.8%
Paragraph	86.7%

Table 4.3: Dialogs used in experiments

Figure 4.4 through Figure 4.7 show the relative network traffic and interactive latency of VNC with OF technique compared to regular VNC. As shown in Figure 4.4 and 4.6, VNC with OF generates 38%~85% and 30% ~88% of network traffic compared to VNC when using Hextile and ZRLE encoding respectively. This reduction in network traffic is because the presentation data of some static objects are not transferred when OF technique is used. The reduced traffic ratio is consistent with the static ratio when using Hextile encoding but not when using ZRLE encoding. The reason being we use pixel numbers to calculate the dialog's static ratio. This may not reflect the true static ratio when a compression algorithm is used on the presentation data, which is ZRLE's case. As shown in Figure 4.5 and 4.7, VNC with OF technique and VNC have very similar latency at 100 Mbps when using both Hextile and ZRLE encoding. However, as the bandwidth drops down below 1.5Mbs, the VNC with OF begins to outperform VNC. The performance difference between them becomes obvious when the network bandwidth is 128Kbps. The reason is that computation time and network access time are two principal contributors to operation latency. At 100Mbps, the network access time is minimal compared to the computation time. So with OF technique the operation latency is even a bit larger because OF takes some computation time. When network bandwidth is low, the network access time becomes dominant with respect to operation latency. Consequently, VNC with OF technique has shorter operation latency than VNC.

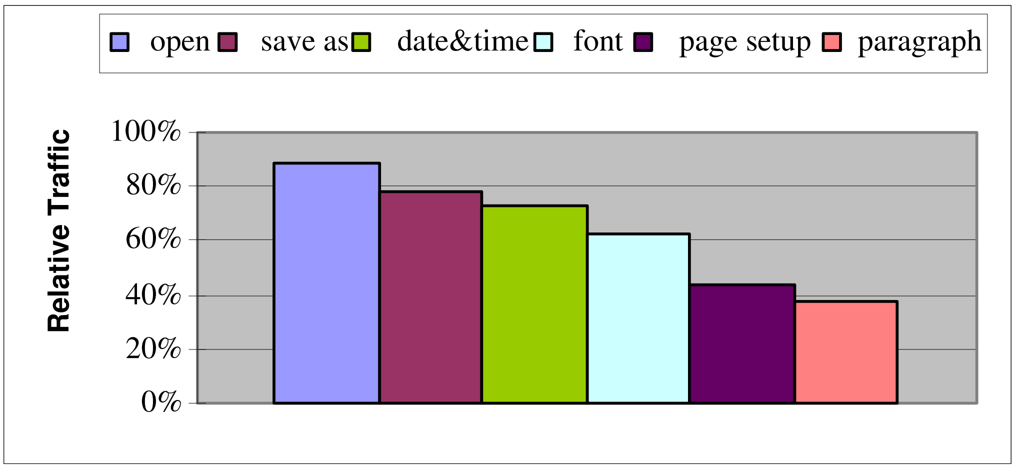

Figure 4.4: Relative network traffic of VNC with OF technique compared to original VNC. Network bandwidth is 100Mbps. Hextile encoding is used.

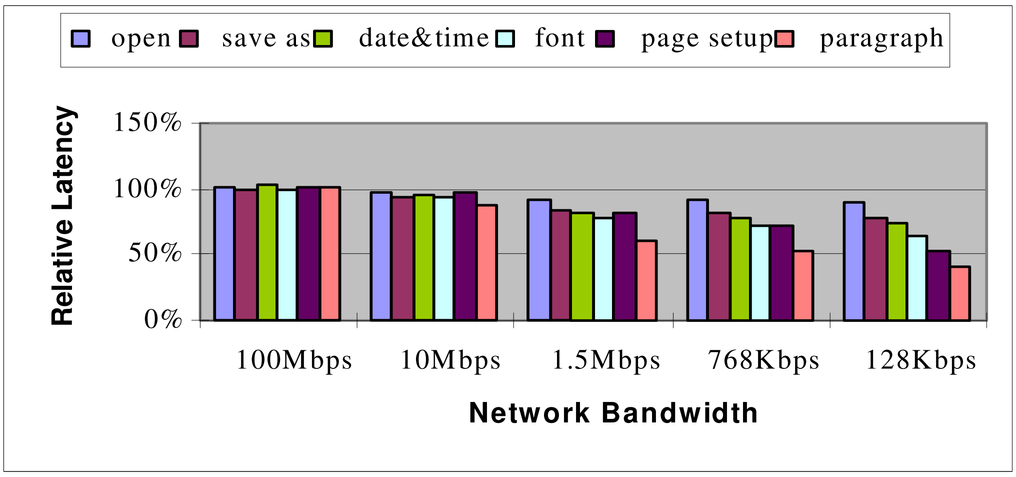

Figure 4.5: Relative operation latency of VNC with OF technique compared to original VNC. Different network bandwidths and Hextile encoding are used.

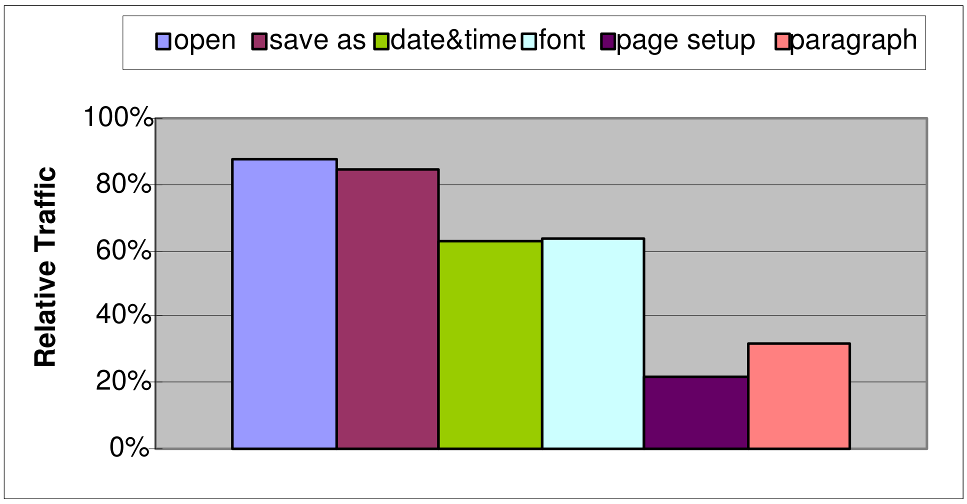

Figure 4.6: Relative network traffic of VNC with OF technique compared to original VNC. Network bandwidth is 100Mbps. ZRLE encoding is used.

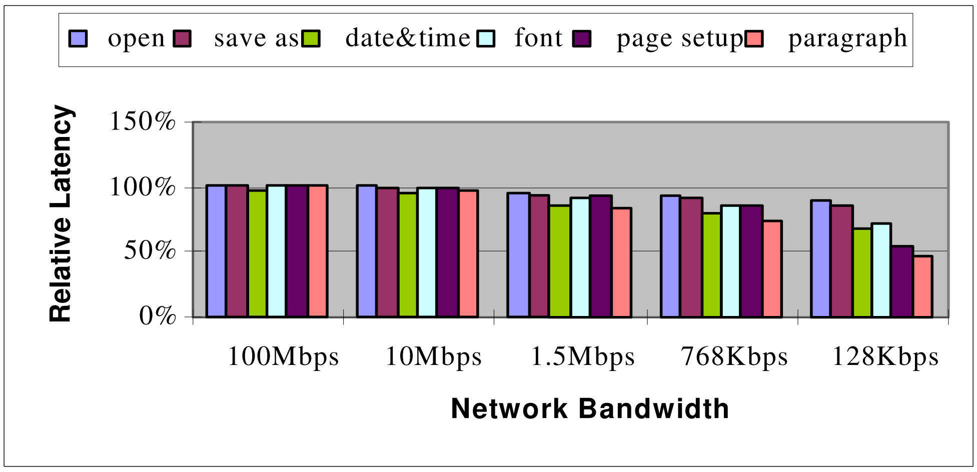

Figure 4.7: Relative operation latency of VNC with OF technique compared to original VNC. Different network bandwidths and ZRLE encoding are used.

Figure 4.8 and 4.9 show the client and server CPU utilization of VNC and VNC with OF technique. These results were obtained by monitoring the CPU utilization when operating different dialogs continuously. In comparison, we can see that with ZRLE encoding, the CPU utilizations on both server and client are higher for ZRLE is more complex than Hextile. It can also be seen that the OF technique will only slightly increase VNC's CPU utilization on the server when using either Hextile or ZRLE encoding. This is mainly due to the extra computation time spent by OF technique in object binding and object filtering. But on the contrary, on the client OF reduces a little on the CPU usage. Due to the static cache in OF technique, part of presentation data comes from the cache instead of the server. Processing the cached data is easier as it just retrieves data from memory and then displays the data. The storage required by the OF technique is also not large. WordPad only need 200 kilobytes. With a cache of 1 megabyte, several applications can cache their static presentation data on the client side at the same time. For a modern desktop computer having hundreds of megabytes memory, this storage is trivial. For mobile devices, running several applications simultaneously is rare because of the display device's limitations. Using 200 or 300 kilobytes to achieve a fast interaction experience is acceptable, especially for some frequently used applications.

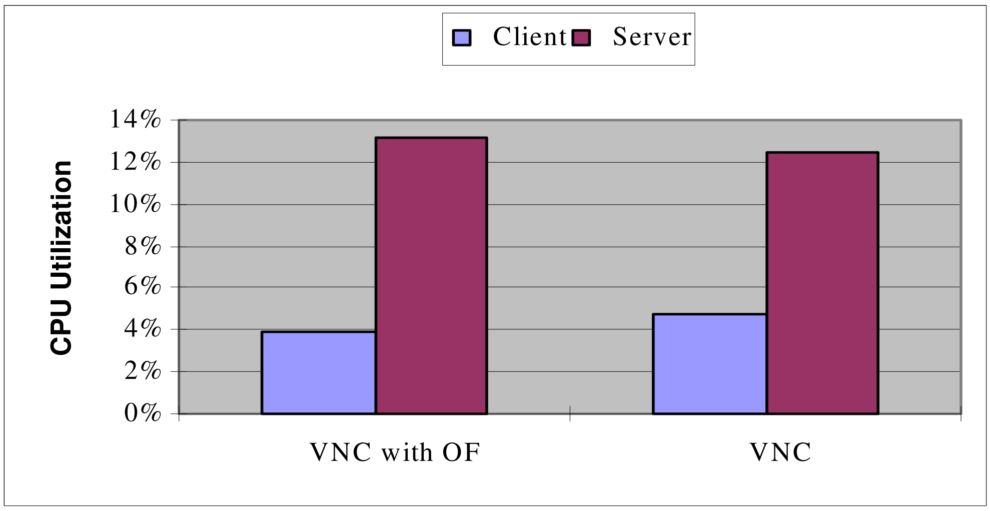

Figure 4.8: Average CPU utilization when using Hextile encoding

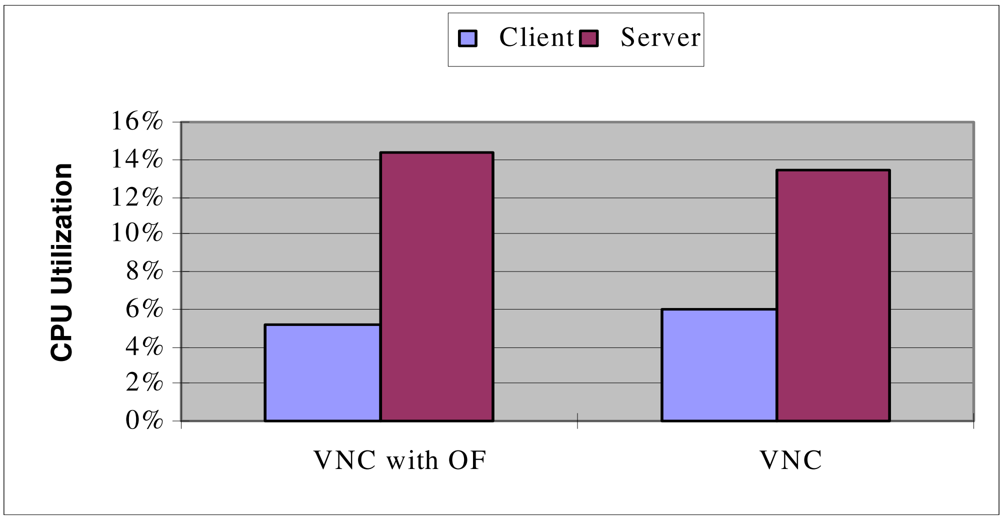

Figure 4.9: Average CPU utilization when using ZRLE encoding

4.4. Summary

Object-level redundancy may cause a large amount of network traffic and cause long latencies of user operations. To reduce object-level redundancy, we proposed a new optimization approach, static object cache. This application-specific approach works in two steps. In the first step it automatically analyzes and extracts static objects from the application's GUI. In the second step, the kernel module of the approach, Object Filtering, reduces the redundant presentation data sent across networks according to the extracted static object information. We empirically studied the efficacy of this static object cache technique on the open-sourced VNC. The experiment results show that for bandwidth-limited networks this approach reduces the network traffic and interactive latency by up to 60% with only a little more CPU usage.

CHAPTER 5

PERFORMANCE EVALUATION FRAMEWORK

In this chapter, we describe an experimental framework that will be used later to evaluate the performances of two optimization techniques respectively presented in Chapter 6 and 7. Network traffic and user operation latency (the delay between a user operation and the visible presentation changes incurred by it) are two key performance metrics of thin client systems. Measuring network traffic can be achieved by monitoring the network activities using software or hardware, while measuring the latency of user operations usually takes more efforts as the timing information of the user inputs and the screen updates they caused must be recorded in an accurate and timely manner.

In Chapter 4, we modified the source code of VNC server to integrate our optimization techniques and added some instrumentation code to VNC client to measure user operation latency. This method cannot be used for Microsoft Terminal Server or other commercial products where the server's source codes are not publicly available. In this chapter, we present another latency measurement method based on trace collection and simulation. This method is in light of the measurement method proposed in [9, 10, 52].

In the rest of this chapter, we will respectively describe how traces are collected, a latency measurement methodology, the testbed we used, and a baseline Microsoft Point-to-Point Compression (MPPC) encoder. In chapter 6 and 7 we will compare the performance of our optimization techniques to this baseline.

5.1. Trace Collection

We choose three representative interactive applications, Adobe Photoshop, Microsoft Visio, and Microsoft Word for our performance evaluation. The three applications produce different types of screen update data. Adobe Photoshop is a popular image processing application and induces a high percentage of bitmap data. Microsoft Visio is a vector drawing application that is widely used in computer aided design. Its main screen updates are vector graphics. Microsoft word is a commonly used word processing application which represents a wide range of office automation software. Its main screen updates are texts but also include graphics and bitmaps to some extent. For convenience, we call the three traces captured from them respectively *image*, *vector* and *text*.

The *image* trace captured the screen updates when a user eliminated the noises of a photograph, separated a portrait from the photo, and fused it into another image using Adobe Photoshop. The *vector* trace captured the screen updates when a user edited a large graph with Microsoft Visio. The user operations included opening documents, creating various block shapes, adjust the positions of block shapes, adding texts, and importing external images. The *text* trace captured the screen updates when a Microsoft Word user edited an existing document, inputting some new paragraphs, importing some external images, editing a few tables and adjusting the formats of the document. All of the three traces were about 10 minutes long.

These traces are obtained when users carried out the above workloads on MSTS through an open-sourced RDP client, rdesktop [60] in a LAN environment with submillisecond round-trip time (RTT). These traces recorded information about user operations and screen updates. We obtained both compressed and decompressed screen update data. Compressed data are used only for comparison purpose. As MPPC is a lossless compression protocol, the decompressed data should be the same as the uncompressed data on the server side. Hence the decompressed data are used as the input of our study. We made the same assumption as in [8]: an operation begins at the time the client sends a user input and ends with the last screen update received from the server before a succeeding user operation. To separate update packet sequences for different user operations in the traces, we appended an end mark to the last packet of each operation. To model the application processing time, we also recorded the time interval between a user input and the first update packet of it as well as the intervals between two consecutive update packets of an operation. In this modeling we neglected the network transferring time which is very small in LAN environment.

5.2. Latency Measurement Methodology

Figure 5.1 shows how a user operation is simulated and how its latency is measured. A small packet is sent from the client to the server to mimic a user input. On receiving this 'user input', the server begins processing the update packet sequence for this 'user input'. The processing includes inserting the recorded time intervals between two consecutive packets in the sequence, compressing the packets using either MPPC or our optimization schemes, and sending the compressed data to the client. An end mark attached with a packet indicates the end of a user operation. During the simulation we recorded the sending time of a 'user input' and the receiving time of the last update packet triggered by this 'user input'. The difference of them is used to measure operation latency.

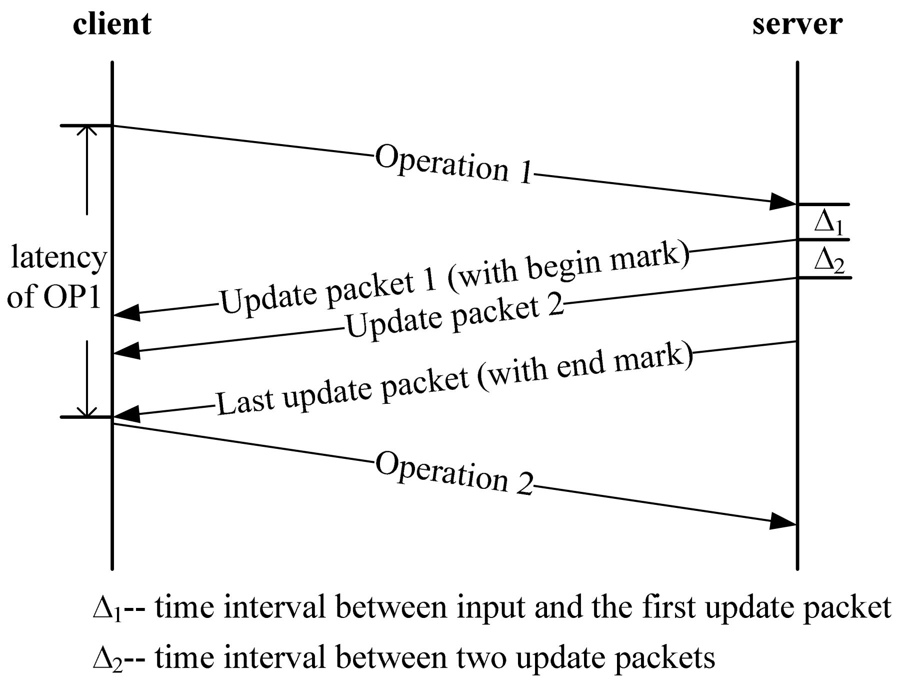
client
server
Operation 1
latency of OP1
Δ_1
Δ_2
Update packet 1 (with begin mark)
Update packet 2
Last update packet (with end mark)
Operation 2
Δ_1-- time interval between input and the first update packet
Δ_2-- time interval between two update packets

Figure 5.1: Latency measurement

5.3. Testbed Configuration

Figure 5.2 shows our experimental testbed. In this tested, two workstations respectively simulate the server and the client of a thin client system. WAN is emulated by a WAN emulator, NistNet [62] which runs on a machine with two network interfaces. This machine is configured as the router of two subnets where the client and server sit respectively. The emulator is able to emulate a variety of wide area network environments by adjusting the available bandwidth, latency, packet loss rate and other network parameters. Table 5.1 and Table 5.2 respectively show the machine configurations and the network settings used in our experiments. We considered a number of typical bandwidths in contemporary network environments as well as the impacts of different network latencies.

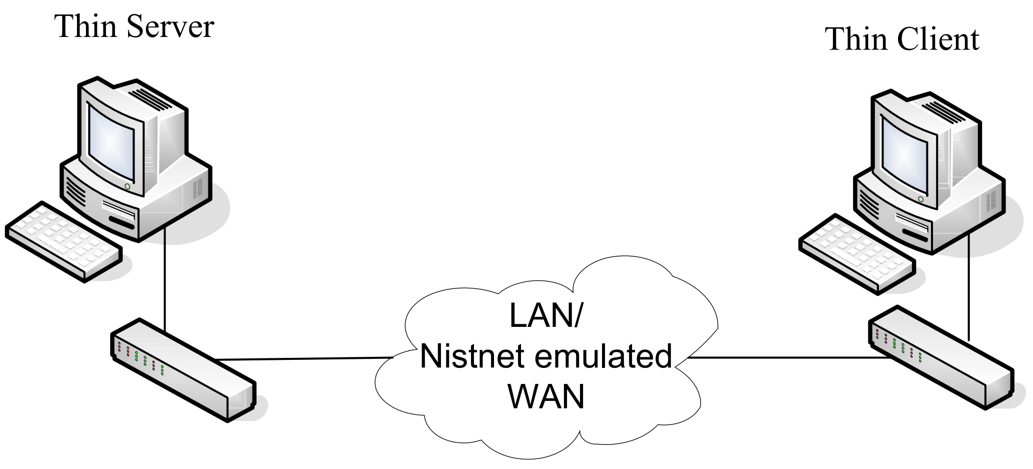

Figure 5.2: Experimental testbed

Role	**Hardware**	**OS**
Server	500MHz Intel Pentium III 256M RAM 10/100 BaseT NIC	MS Windows XP professional
Client	500MHz Intel Pentium III 256M RAM 10/100 BaseT NIC	Redhat Linux 9.0
WAN emulator	1.5GHz Intel Pentium IV 512M RAM 10/100 BaseT NIC (2)	Redhat Linux 9.0

Table 5.1: Machine configurations

Network configuration	**Bandwidth**	**RTT**	**Packet loss rate**
LAN	100Mbps	1ms	≈ 0
WAN1	1.5Mbps (T1)	20ms	0.5%
WAN2	768Kbps(DSL)	20ms	0.5%
WAN3	256Kbps(ISDN)	20ms	0.5%
WAN4	1.5Mbps	60ms	0.5%
WAN5	768Kbps	60ms	0.5%
WAN6	256Kbps	60ms	0.5%

Table 5.2: LAN & WAN scenario descriptions

5.4. Baseline

Our baseline is a MPPC encoder modified from the open-sourced MPPE/MPPC kernel module for Linux [46]. By studying the compressed and the decompressed stream obtained from rdesktop, we made our encoder as similar to the MSTS's MPPC encoder as possible. The correctness of our encoder was verified using a decoder taken from rdesktop [60]. We tested the performance of our MPPC implementation on a Pentium III 500 MHz machine using the decompressed streams obtained. From the results shown in Table 5.3, we can see that our implementation has a comparable compression ratio as the MPPC implementation in MSTS. The difference in compression ratios may be caused by the different hash functions used. Table 5.3 also gives the compression speed of our implementation. Though we cannot compare it with MSTS as MSTS is not open-sourced, our experimental results showed that the compression time has only insignificant impact on interactive latency.

Display update traces	**Compression ratio (%)** ($\frac{compressed\ size}{original\ size}$)		**Compression speed of our impl.** (μs/KB)
	MPPC (MSTS impl.)	**MPPC (our impl.)**	
image	33.72	33.77	61.53
vector	26.28	26.35	55.38
word	30.13	30.13	48.72

Table 5.3: Baseline performance

CHAPTER 6

REDUCING LONG-DISTANCE REDUNDANCY

In Chapter 4, we presented a static object cache technique to reduce the object-level redundancy in the screen updates of thin client systems. In this chapter we focus on another kind of redundancy, which is caused by the same screen updates repeated after a long trace of other screen updates. We call this kind of redundancy *long-distance redundancy*. In a thin client session, long-distance redundancy is generally induced by repeating some operations after a series of other operations. This scenario is very common in graphics-based user interaction. For example, a user may use the 'Open' dialog in Acrobat 6.0 to open a file. After some editing operations, the user may use the same dialog to open another file. When the 'Open' dialog is shown for the second time, the screen updates of it have only a trivial change. The same situations may occur when users repeat menu operations, switch between some frame windows, and so on. In this chapter, we present an analysis of long-distance redundancy and a flexible and scalable extension of LZ algorithm to it.

6.1. Introduction

Some cache techniques deployed in the existing thin client computing systems can reduce long distance redundancies to some extent. Bitmap cache in MSTS caches bitmap data on client and reuses them when the same bitmap data need to be redisplayed. As MSTS bitmap cache uses up to 1.5M bytes for memory cache and 10M bytes for persistent cache, some bitmap data redundancies with distance much larger than 64K can be reduced. String cache and glyph cache in MSTS can mitigate long distance redundancies of non-bitmap data. Cache-based optimization techniques require the reoccurring items to be the same as saved items. Any change even only one byte in the reoccurring item will be considered as a new item.

In comparison, dictionary-based compression techniques such as LZ algorithm can tolerate such minimal changes by matching reoccurring data in a finer granularity. LZ algorithm is very effective to reduce the redundant data in screen updates while at the same time is fast enough for online processing [9, 95, 91]. So it's widely used in thin client systems. Microsoft Point-to-Point Compression (MPPC) scheme [43] used by MS Terminal Sever, is based on LZ algorithm. AT&T VNC uses Zlib [45], which is also based on a variant of LZ algorithm. However, a small history buffer (or sliding window) that is popular in the implementations of LZ algorithm and its variants can not reduce the redundancies that occur at a long distance. A straightforward solution for the long-distance redundancies is a flat extension of the LZ algorithm. In this extension, the history buffer of LZ algorithm, which is organized as a bounded continuous memory area, is simply enlarged. However, this solution is not flexible. History buffer size is limited by the available memory resources that vary from system to system. To adapt to different situations, the extension should be able to resize the history buffer easily. But in the flat extension, the encoding/decoding logic changes as history buffer size (offset range) changes. Moreover, a flat extension has scalability issue. Most existing implementations use a history buffer below 64K bytes. When the history buffer size is above 64K bytes, the offset needs a 32-bit representation rather than a 16-bit representation. This will cause the auxiliary data structures (hash tables, suffix trees, etc.) to expand significantly. Expanding history buffer flatly may also excessively increase computational time as the LZ algorithm has to search for matches in an auxiliary data structure with more items. Additionally, a larger offset range requests more bits to encode an offset, and thus will counter some gains from increasing history buffer.

We proposed another way to extend the history buffer of LZ algorithm, which is a vertical extension where history buffer is organized as a number of separate memory blocks [101]. This extension is more flexible and has no such scalability issue. We named this optimization scheme long-distance redundancy reduction scheme (LDRS) for its objective is to reduce the long-distance redundancy.

6.2. Analysis of Discrete History

If all the data packets arriving at the compressor are saved in a single history buffer in the order of arriving, we call the history contained in the history buffer a continuous history. If only a subset of the data packets is saved, we called the history a discrete history.

Formal definitions of continuous history and discrete history will be given later (Section 6.2.2). In this section we present an empirical analysis of the possibility of replacing a continuous history with a short, discrete history when compressing each packet. Before presenting the analysis, we define some terms used in it.

6.2.1. Edit Distance between Byte Strings

In a thin client system, screen update data are usually encapsulated into packets proprietary to a thin client protocol such as T.128 [28] and Remote Frame Buffer Protocol (RFB) [26]. A packet usually contains the screen update data to paint some graphical objects and some additional information that describe the screen update data contained. A history usually contains a sequence of packets. Either an individual packet or a sequence of packets can be treated as a byte string (a sequence of bytes). If a byte string B in a byte string sequence is similar to a previous byte string A in the same sequence and A has been stored on the client, instead of transferring B to the client, we can pass to the client a small edit script to build B based on A.

Two kinds of edit scripts can be used for this purpose. The first kind consists of predefined reversible operations to modify one string into another, which has been used in text comparison [56]. The second kind consists of two predefined operations to build a new string B based on an existing string A: insert a substring of A into B or insert a symbol not in A into B. The first kind of edit script allows bidirectional conversion. The second kind of edit script only allows unidirectional conversion but can be more easily constructed and integrated with a dictionary-based compressor (a class of lossless data compressors which search matches between the data to be compressed and a dictionary of strings and replace the matched data with the index of the string in the dictionary). As we only need to build a byte string based on a previous byte string, we adopt the second kind of edit script. Next we will define a partition that is equivalent to it.

DEFINITION 6.1. A **match partition** of string *B* about string *A*, denoted as $MP(B \mid A)$, is a sequence of *B*'s substrings each of which is either a substring of *A* (termed a **match**) or a symbol that cannot be found in *A* (termed a **literal**), and concatenation of which equals to *B*.

From each match partition of string *B* about string *A* we can construct an equivalent edit script to build string *B* based on string *A*, using the following two operations: insert a substring of string *A* into string *B* and insert a literal into string *B*. So we will not distinguish a match partition and its equivalent edit script in the rest of this text. Different match partitions of string *B* about string *A* must have the same number of literals but may have different number of matches and different sizes. For example, suppose *A* = *abcicdejdef* and *B* = *abcdefgh*, one match partition (*ab*, *cd*, *ef*, *g*, *h*) has 3 matches and a size of 5 while another (*abc*, *def*, *g*, *h*) has 2 matches and a size of 4.

DEFINITION 6.2. A **minimal match partition** of string *B* about string *A***,** denoted as $MMP(B \mid A)$, is a $MP(B \mid A)$ of minimal size.

Given string *A* and *B*, $MMP(B \mid A)$ is not necessarily unique. For example, suppose *A* = *abcbde* and *B* = *abde*, both {*ab*, *de*} and {*a*, *bde*} are minimal match partitions of *B* about *A*. Inspired by the LZ algorithm [13], we define a special minimal match partition.

DEFINITION 6.3. A **LZ partition** of string *B* about string *A*, denoted as $LZ(B \mid A)$, is a $MP(B \mid A)$ that is built from the following procedure. We use *B*(*i*, *j*) to denote a substring of *B* which starts at position *i* and ends at position *j*.

1. Find a longest substring *B*(*0*, *j*) that is also a substring of *A*. If not found, add the symbol at position *0* of *B* to $MP(B \mid A)$. Otherwise add *B*(*0*, *j*) to $MP(B \mid A)$.
2. If $(j != \mid B \mid -1)$ then $B = B(j, \mid B \mid -1)$ and go to 1.

The above procedure is very similar to the parsing phase of LZ compression algorithm [13]. A key problem in the above procedure is to search the longest match substring *B*(*0*, *j*). Various solutions [57, 58, 59] are available for it.

LEMMA 6.1. $LZ(B \mid A)$ is a $MMP(B \mid A)$.

Proof. Assume that $LZ(B \mid A)$ is not a $MMP(B \mid A)$. Hence in $LZ(B \mid A)$ some consecutive matches can be repartitioned into fewer matches. Obviously 2 matches cannot be repartitioned into 1 match; else it should be found in the above procedure. Assuming 3 matches can be repartitioned into 2 matches, the first match after repartitioning must contain itself and part of the second match; else the remaining 2 matches should be combined into 1 match. This contradicts with step1 in Definition 6.3, which searches for a longest match each time. Iteratively, we can infer that any N matches (N>3) in $LZ(B \mid A)$ cannot be repartitioned into N-1 matches. ■

DEFINITION 6.4. The **match edit distance from string *A* to *B*,** denoted as $D(B \mid A)$, is defined as the size of $MMP(B \mid A)$.

$D(B \mid A)$ reflects the relevance of string B to string A . It equals to the total cost of operations in the equivalent edit script of a minimal match partition with each operation given a unit cost. Hence, $D(B \mid A)$ indicates the potential compression performance that can be obtained from a LZ algorithm or its variants.

6.2.2. Discrete History vs. Continuous History

Suppose P_0 …P_i … P_n is a packet sequence. For each packet P_i, the subsequence $H_i = P_k ... P_{i-1}$, $(0 \le k < i)$ is called a **continuous history** of packet P_i. The subsequence $H_i^{'} = P_{j_1} P_{j_2} ... P_{j_k}$ $(0 \le j_1 < j_2 < ... < j_k < i)$ is called a **discrete history** of packet P_i. If the packets in H_i' have the k smallest match edit distances to P_i, H_i' is called a ***k*-best discrete history** of packet P_i.

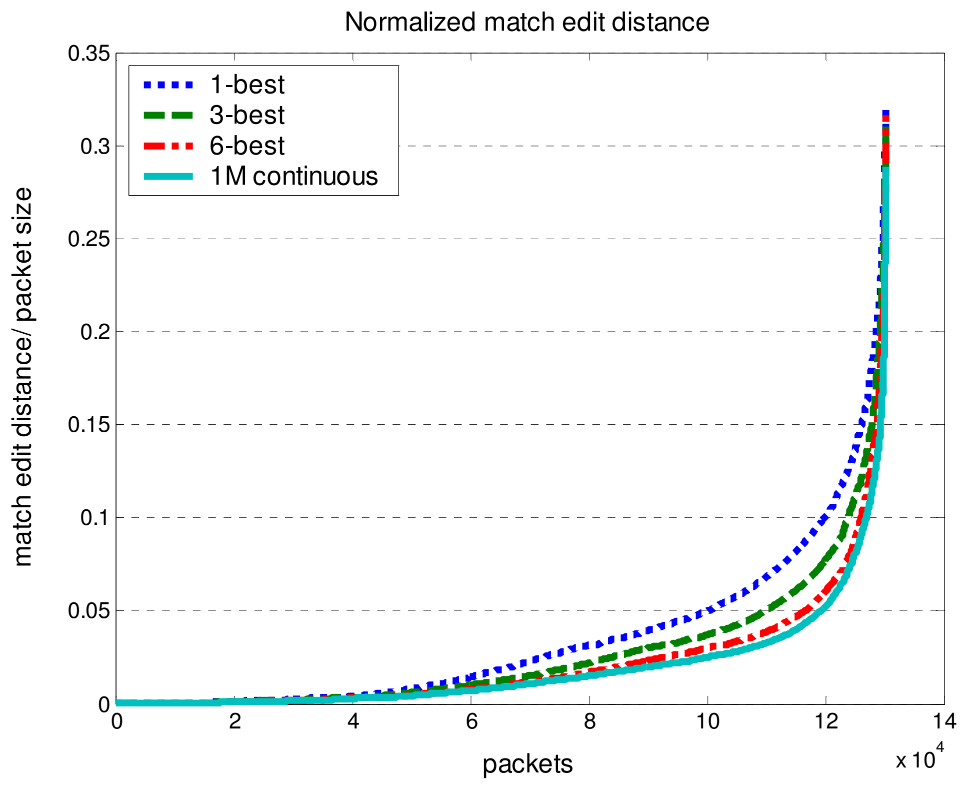

Figure 6.1: Continuous history vs. *k*-best discrete history

We empirically analyzed the effect of using continuous history and discrete history in compression for some long packet sequences obtained from an open source MSTS client (rdesktop[60]). In our experiment, we calculate two match edit distances for each packet P_i. One is the match edit distance from H_i to P_i, where H_i is a continuous history of P_i and the size of H_i is no more than 1M bytes. The other is the match edit distance from a *k*-best discrete history H_i' to P_i. H_i' is a subset of H_i. The analysis program calculates continuous and discrete edit distance for each packet P_i $(i=0,\ldots,n)$ in the sequences. Figure 6.2 shows the comparison between them. From the result we can see that the difference between the *k*-best edit distance and the continuous edit distance reduces as *k* increases. And when *k* is increased to 6, the *k*-best edit distance is very close to the continuous edit distance. This shows that when compressing a packet, using a short discrete history is possible to achieve similar compression performance as using a long continuous history. As the discrete history is short, it is not necessary to use a large offset range for the matches. For example, in MSTS, the average packet size is only 1.5K bytes. So 6 packets in the discrete history only need a 9K offset range on average.

6.3. Long-distance Redundancy Reduction Scheme

Based on the above analysis, we propose a vertical history extension scheme in this section. We start with the history buffer organization in this scheme, followed by a description of the history selection algorithm used in this scheme.

6.3.1. Organization of History Buffer

Figure 6.3 (a) and (b) respectively show the history buffer organization in a flat extension scheme and the proposed vertical extension scheme. In contrast to the flat extension which contains a single history buffer, the proposed extension scheme contains a number of history blocks of the same size. The number of history blocks can be negotiated between the server and the client at the beginning of a thin client session. Each history block is restricted to 64K bytes or less. In compression, the packet which is being compressed (current packet) will use one block as history for its compression. The current packet, if to be saved, should be saved in that block. The index of that block and whether this packet is saved or not need to known by the client to ensure correct decompression and synchronization of the history state. In this extension scheme, each history block has its own auxiliary data structure. As the history block's size is fixed and restricted to 64K bytes, the offset range will not change when the total history buffer size increases. Thus it's always sufficient to represent an offset with 16 bits in the auxiliary data structure. The number of items in a block's auxiliary data structure will keep unchanged and will not increase match search time. The number of bits needed to encode an offset will not scale either. Hence this extension scheme has more flexibility and scalability than the flat extension. The flexibility and the scalability of the history buffer provide the possibility of adaptively allocating history buffer according to the available memory resources.

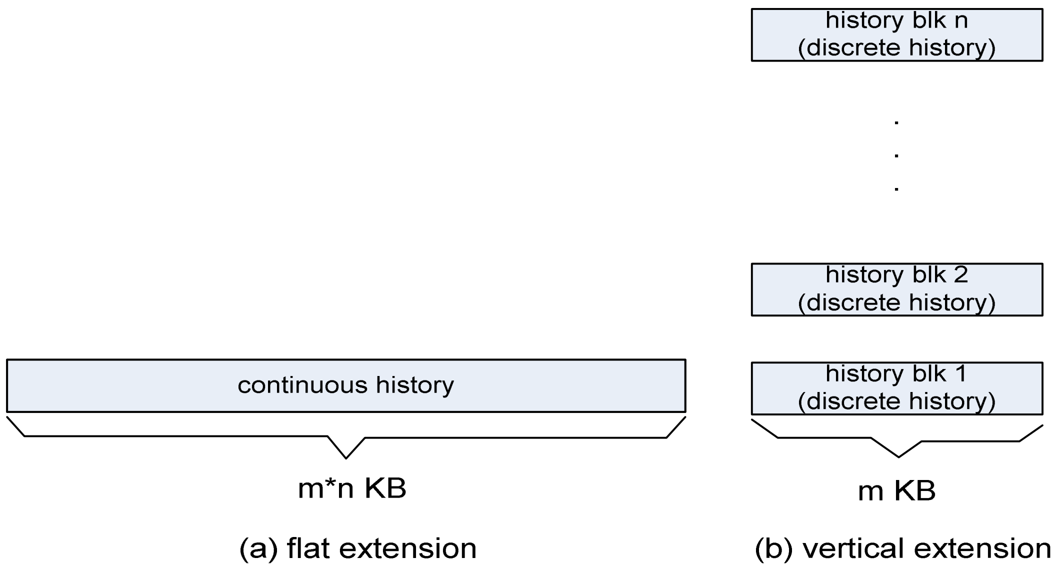

Figure 6.2: Flat history extension vs. vertical history extension

6.3.2. History Selection

Finding *k* best history packets with minimal match edit distances to the current packet will cost a substantial amount of time if these packets are dispersed in different blocks. Fortunately we found that in thin client sessions, these best history packets are often clustered. In other words, these best history packets will appear in the same history block if we always save them in the block that has the minimal match edit distance to each of them. We also observed that even sometimes a few best packets are missing, the rest still form a good history. As such, we only need to select a best history block for the current packet. We use the algorithm GREEDY shown in Figure 6.4 to achieve this.

In order to quickly detect which block is the best history block for the current packet *P*, we find out all long matches (at least 16 bytes) between *P* and all cache blocks by a fast rolling signature technique. For each history block a hash table is used to index every 16 bytes history in the block. During compression GREEDY calculates a 16-bit fast rolling signature of a 16 bytes window from the current point in packet *P*. This signature is used to index the global hash table. When an offset is retrieved, we find its corresponding position in the history blocks and compare it with the data in *P* from the current point. If a long match is found, GREEDY adds the long match's length to the block's total match length. The total match length is used to select the best history block.

Greedy

```
Input: packet $P$ and its size $sz$, number of blocks $N$.
Output : Selected history block $H$.

$ptr$ = 0; /*$ptr$ is a pointer that indicates the current point of $P$*/
for ($i$=0;$i$<$N$;$i$++)
  $matches[i] = 0$;
while ($ptr$ <= $sz$-16)
{
  $s$ = rolling_signature($P$, $ptr$);
  $offset$ = hash_lookup($s$);
  $longest\_match$ = find_longest_match($offset$, $ptr$);
  if($longest\_match$ >= 16 and $offset$ in block $i$)
  {
     $matches[i]$ +=$longest\_match$;
     $ptr$ += $longest\_match$;
  }
  else
     $ptr$ ++;
}

sort($matches$);
$k$=$N$;
for ($i$ = 1; $i$ < $N$; $i$ ++)
{
   $diff$ = $matches[i-1]$ - $matches[i]$;
   if ($diff$ > $threshold$)
   {
     $k$=$i$; break;
   }
}

$H$= least_used_block($matches$, $k$);
return $H$;
```

Figure 6.3: History selection algorithm: GREEDY

However, always selecting the block with longest total match length may lead to an unbalanced memory usage. To avoid this, we choose one block from a number of candidates using another criterion. Candidates are those blocks which have similar total match lengths, and have larger total match length than the rest blocks by a threshold. In details, GREEDY saves total match length for each block into an array. After searching the long matches, GREEDY sorts the array in a descending order and calculates the differences between the neighboring numbers in the array. If the difference between two neighboring blocks *i* and *j* is larger than the predefined threshold, all blocks before and including block *i* in the array are candidates. GREEDY selects the least used block in the candidates. If no difference larger than the threshold is found, GREEDY chooses the least used block from all of the history blocks. When a block is selected, GREEDY saves the data of *P* into that block after compressing and updates the hash table at the same time. To prevent one block saving the same data for many times, we do not save the packet if its total match length is almost equivalent to the packet size.

6.4. Experiment Results and Analysis

In this section we compare the performance of LDRS with MPPC in terms of network traffic and interactive latency. We measured the performance of the scheme with 2, 4, 8 and 32 history blocks. Unlike network traffic, interactive latency is related to network configuration. We measured interactive latency in the seven different emulated network configurations described in Chapter 5. We also commented the computation overhead and memory usage in Section 6.4.2.

6.4.1. Performance

Figure 6.5 shows the network traffic reduction of LDRS compared to MPPC for the 3 traces described in Chapter 5: *text*, *vector*, and *image*. We can observe that more history blocks lead to better improvements. Using more history blocks permits more data to coexist in the history, improving the chance to find matches between current packet and the history. Compared to original MPPC, LDRS can eliminate around 4.3% - 8.9% network traffic with 2 history blocks, around 15.0 - 33.5% with 32 blocks. This indicates LDRS can efficiently reduce long-distance redundancies in different type of screen update traces. Compared to *image* and *vector* traces, a lower percentage (15.0%) of network traffic in *text* trace is reduced by our vertical extension scheme. Further analysis shows that this is because there are more long-distance redundancies in *image* and *vector* than in *text* trace. With more history blocks redundancies at longer distances can be reduced.

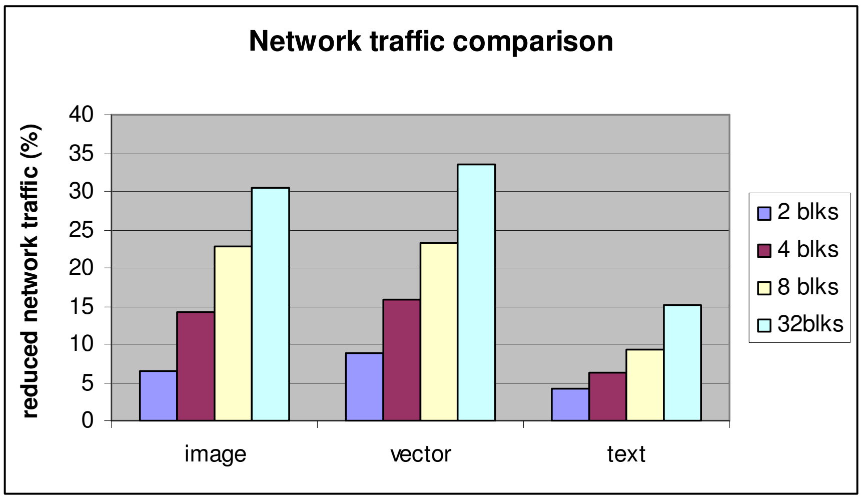

Figure 6.4: Network traffic comparison between LDRS and MPPC

To evaluate the effect of LDRS on interactive latency, we partition the interactive response times into three time bins based on the five time bins proposed by Tolia el. dl [8]: *crisp* – shorter than 150ms; *noticeable* – longer than 150ms but shorter than one second; *annoying* – longer than one second. Table 6.1 gives the number of operations that fall into noticeable or annoying time bin for different traces, history buffer configurations and network environments. According to the results in Table 6.1, LDRS achieves the same interactive performance as MPPC in LAN. In LAN environment the time to transfer a data spike is usually small. The long latencies are mainly induced by long application processing time rather than network transferring time. For all chosen WAN network configurations, LDRS reduces the number of noticeable and annoying response time compared to MPPC. Like the network traffic reduction, more history blocks produce better results. With 2 blocks LDRS reduces 1.4% - 3.1%, 2.2% - 5.8%, and 3.0% - 6.2% noticeable or even annoying latencies for *image*, *vector* and *text* respectively. With 32 blocks LDRS respectively reduces 22.9% - 26.7%, 24.6% - 27.8%, and 20.6% - 21.3% noticeable or even annoying latencies for the three traces. These results show that LDRS can reduce quite a few long operation latencies in WAN environments. This is achieved by prohibiting redundant large screen updates to send across the network. With the limited bandwidth in WAN environments, a large screen update will need a long time to be transferred to the client.

Trace	Network	Noticeable 150 ms - 1 sec					Annoying >1 sec				
		M*	D2	D4	D8	D32	M	D2	D4	D8	D32
Image (13344 operations)	LAN	24	24	24	24	24	6	6	6	6	6
	WAN1	189	183	177	161	143	24	24	24	24	20
	WAN2	343	335	321	300	256	37	37	36	34	29
	WAN3	521	510	484	453	386	48	46	43	38	31
	WAN4	262	253	244	231	202	31	31	30	28	20
	WAN5	395	390	376	346	303	44	43	41	37	31
	WAN6	694	689	669	625	542	70	64	60	53	47
Vector (17912 operations)	LAN	17	17	17	17	17	3	3	3	3	3
	WAN1	228	220	209	197	165	29	29	29	28	21
	WAN2	397	380	362	338	300	34	33	32	31	25
	WAN3	639	611	583	557	466	47	46	45	41	33
	WAN4	306	290	281	270	229	33	31	30	28	23
	WAN5	460	450	431	399	345	48	47	42	38	31
	WAN6	881	832	805	765	643	58	53	47	40	35
Text (13834 operations)	LAN	10	10	10	10	10	2	2	2	2	2
	WAN1	63	61	58	54	50	4	4	4	3	3
	WAN2	126	122	118	109	100	4	4	4	3	3
	WAN3	234	220	214	201	186	7	7	6	6	5
	WAN4	93	87	84	79	74	4	4	4	3	3
	WAN5	145	136	132	127	115	5	5	5	4	3
	WAN6	301	287	275	253	239	7	7	6	6	5

* M – MPPC; D2, D4, D8, D32 – DSRS with 2, 4, 8 and 32 cache blocks.

Table 6.1: Interactive latency comparison between LDRS and MPPC

6.4.2. Computation Overhead & Memory Usage

Except for the history selection process, all the other parts of LDRS are the same as MPPC. So the additional computation is mainly induced by selecting a block as history which includes auxiliary data structures updating time. Since LDRS uses a global hash table to index the history, the overhead caused by the selection process does not increase with the number of history blocks. Column 2 of Table 6.2 shows the average history selection overhead to process 1K bytes data for different traces. Column 3 shows the average compression speed of our extension. Compared to the baseline compression speed given in Table 5.3, we can find that LDRS increased about 25.5%, 23.6% and 16.3% compression time for *image*, *vector* and *text* respectively. Generally, such an overhead contributes little to the total latency since the compression itself is very fast and has no bad effect on the server's scalability. In fact, in the case that the scalability of a thin client server is mainly bounded by network bandwidth, using LDRS can improve its scalability. Of course, LDRS is essentially a tradeoff between computation time and network transferring time. Hence it is not suited to be used where the scalability of the server is tightly bounded by the computation resources.

Screen update trace	**History Selection** (μs/KB)	**Compression Speed** (μs/KB)
Image	18.32	77.21
Vector	15.20	68.43
Word	14.57	56.64

Table 6.2: Computation overhead of LDRS

In our implementation, we use hash tables as auxiliary data structures on the server side. For every 16 bytes written to a history block, we create a hash table entry for it. Hence the number of hash entries needed is one sixteenth of the block size. The total additional memory required on the server side can be calculated by

$(blk\ size + \frac{blk\ size}{16} \times bytes\ per\ hash\ entry) \times blk\ number$. If the block size is 64K bytes, each hash entry needs two bytes to store the offset. The additional memory required on the client side is equal to the cache size.

6.5. Summary

In thin client computing systems, long-distance redundancies may cause a large amount of network traffic but have not received adequate attention. This chapter presented a way to extend the history buffer of LZ compression algorithm that is often used for screen update data, aiming at reducing long-distance redundancies. We empirically studied the effectiveness of our scheme on three different types of screen update traces of Microsoft Terminal Server. The numerical results show that this scheme can reduce 15.0% - 33.5% network traffic for the tested traces with a history buffer of 2M bytes. Moreover, it also can reduce 20.6% - 27.8% noticeable long latencies for different types of applications with 2M bytes history buffer. This scheme costs only a little additional computation time and the cache size can be negotiated between the client and server.

CHAPTER 7

A DATA SPIKE REDUCTION SCHEME FOR THIN CLIENT SYSTEMS

In Chapter 6 we proposed an optimization scheme to reduce the long-distance redundancy in screen update data by extending the dictionary-based Lempel-Ziv (LZ) algorithm. While this scheme provides a flexible and scalable extension scheme to the history buffer of LZ algorithm, the history buffer size is limited by the available memory resource of the server and client in practice. So how to improve the user experience as far as possible with limited resources is a problem we should address. In this chapter we present a data spike reduction scheme, aimed to reduce the long, noticeable latencies that are caused by sending a large amount of screen update data in a very short time.

7.1. Introduction

Although various optimization techniques have been used, we observed that many user operations can still produce a large amount of screen update data when using existing thin client computing systems. These screen update data are generated in a very short period, resembling a 'data spike' in an update sequence to be transferred over network. Usually delivering such a data spike to clients within a user-perceivable latency needs a bandwidth that is much higher than the average bandwidth requirement. Although network bandwidth has been greatly increased in recent years, available bandwidth in many networks may still not be able to meet the needs of transferring data spikes adequately fast. In that case, the user will feel the interaction with remote application "unsmooth". An unsmooth user experience may make users unsatisfied with the whole thin client systems. After all, "it is the worst case not the average case determines the usability of the thin client" [8]. Therefore, reducing the number and peak values of data spikes can improve the interactive performance of thin client systems, giving users a more satisfactory experience.

As data spikes are essentially a large amount of screen update data produced in a short time, existing techniques that reduce data size can reduce data spikes to some extent. UltraVNC [27] employs an on-screen cache and a cache encoding mechanism which save the latest two updates for each pixel of the screen. The new updates will be compared with the saved updates to pass only the unsaved screen regions. The two cache schemes can only reduce data spikes caused by recurrences of the latest two updates of the same screen region.

Citrix Metaframe [71], Microsoft Terminal Server [72], Tarantella [30], and NoMachine NX [83] cache basic graphic objects or X messages in memory. Caching basic graphic objects and the data parts of X messages can efficiently reduce the transferred data size and thus can reduce some data spikes caused by them. However, only exact matches of graphic objects or data parts can benefit from these caches. Similar graphic objects or data parts, even having only a little change compared to the saved ones, have to be transferred as they are new. Furthermore, these cache techniques cannot reduce the number of drawing orders or X messages, and thus can not reduce the data spikes caused by some complicated screen update consisting of many drawing orders or X messages.

Thin client compression [36] and streaming thin client compression [37] are designed to compress synthetic images. Both of them can only be used with bitmap-based thin client computing systems such as VNC and cannot be used to compress smooth-toned images that are popular in multimedia applications.

2-D lossless linear interpolation coding (2DLI) [38] is designed for both synthetic images and smooth-toned images. Although 2DLI performs very fast and efficiently on compressing update images, it only leverages the redundancy in a single update image. The redundancy across frames cannot be reduced. Like thin client compression, 2DLI can only be used with pixel-based thin client computing systems such as VNC.

Differential X protocol compression (DXPC) [83] efficiently encodes X messages by specializing on each field's value range, frequency and applying delta encoding. It also employs different compression methods for different image types. Similarly, T.128 protocol [28] specifies two compression schemes that are separately applied to orders (order encoding) and bitmap data (bitmap compression). Order encoding avoids transferring the same field values in consecutive orders for multiple times. All these compression schemes can reduce the length of orders/messages, but cannot reduce the number of them. Thus data spikes consisting of lots of orders/messages cannot be reduced.

The Generic compression algorithm LZ searches a history for matches and encodes them with length and offset pairs. In order to restrict the number of bits to encode the offset, the history is usually small (not greater than 64KB). Since all screen updates (before compression) will be saved into such a small history, the screen updates that cause data spikes are easily overwritten before similar screen updates are produced.

In this chapter we propose a data spike reduction scheme (DSRS) to reduce data spikes [100]. The proposed scheme is a hybrid cache-compression one which reduces data spikes by caching their main contributors and uses the cached data as history to better compress the recurrent screen updates in possible data spikes.

7.2. Analysis Model for Screen Update Data

In a thin client system, screen update data are usually encapsulated into packets proprietary to a thin client protocol such as T.128 [28] and Remote Frame Buffer protocol (RFB) [26]. A packet contains the screen update data to paint some graphical objects. Later these graphical objects may be repainted when the user performs the same operation. This will generate a packet that may be similar to a previous one. Our objective is to exploit the similarities between similar packets to reduce network traffic as well as user-perceptible latency in a thin client system.

7.2.1. Packet Reference

In Definition 6.4, we defined the **match edit distance from string *A* to *B*** (denoted as $D(B|A)$) to reflect the relevance of string *B* to string *A*. It can also be used as an indication of the potential compression performance that can be obtained from a LZ algorithm or its variants. In the following, we examine the relationship between $D(B|A)$ and the communication cost in a further step.

DEFINITION 7.1. The communication cost of a match partition is defined as the number of bytes needed to encode the literals and matches in it.

Using different coding schemes may result in different communication costs. Suppose *l* and *m* are respectively the maximal number of bytes to represent a literal and a match in a coding scheme. Normally, $l \le m$ because for a match its position and length need to be encoded while for a literal only the symbol needs to be encoded.

LEMMA 7.1. If $D(B|A) \le |B| \times D_{th}$ ($|B|$ is the length of string *B*, D_{th} is a constant named difference threshold), the communication cost of a $MMP(B|A)$ using a coding scheme for which $l \le m$ is less than $|B| \times D_{th} \times m$.

Proof. Assuming a $MMP(B \mid A)$ has M matches and L literals, we can use $M \times m + L \times l$ bytes to encode a $MMP(B \mid A)$. $M \times m + L \times l = (M + L) \times m - L \times (m - l) \leq |B| \times D_{th} \times m$. ■

Lemma 7.1 shows that if the match edit distance from string *A* to string *B* is very small, by caching string *A* on the client and using an appropriate coding scheme, we can considerably reduce the network traffic needed to communicate *B* to the client. For example, if D_{th}=0.1 and *m*=3, the communication cost will be less than 30% of the length of *B*. Further, if the difference between the sizes of packet *A* and *B* is small, we can quickly locate where *A* is saved in the cache, as shown later. We can also use a replace algorithm to efficiently manage the cache space. This assumption is reasonable because many windows, when repainted, only change a little. In other words, only a very small number of bytes are inserted or deleted compared to a previous packet. So the new packet's size usually differs little from the previous packet. We characterize the relationship between such two packets in the following definition.

DEFINITION 7.2. Suppose P_i and P_j (*j*>*i*) are two packets in a packet stream, if packet P_i and packet P_j satisfy the following two conditions, we say P_i is a **reference** of P_j.

1) $D(P_j \mid P_i) < |P_j| \times D_{th}$

2) $||P_j| - |P_i|| < |P_j| \times D_{th}$

Obviously, difference threshold (D_{th}) plays a very important role in this definition. We will discuss later in Section 7.5.1 how we choose an appropriate value for it by experiments.

7.2.2. Analysis Model

Figure 7.1 shows an analysis model for screen update data, which is very similar to the stack reference model used in traditional cache performance analysis [61]. In the model, a sequence of packets in a screen update stream collected from a thin client system is fed to a FIFO-managed buffer whose capacity is *C*. If a packet in the buffer is a reference of the current incoming packet as per the definition in last subsection, we say the current packet **references** (or **hits)** the packet in the buffer. The total data contained in the packets between the current packet and its nearest reference in the buffer is defined as **reference distance** Δ **.** If the current packet has no reference in the buffer, $\Delta = \infty$. **Distance distribution** $d(n)$ is the probability that the current packet has a reference at distance *n* from it and has not another reference with a distance smaller than *n*. Note that *d*(*n*) is a 'defective' distribution in that it may not sum to unity because of the possible infinite reference distance. The corresponding **cumulative distance distribution** is defined as $D(C) = \sum_{n=1}^{C} d(n) \times P(n)$, where *P*(*n*) is the proportion of data in the packets with reference distance *n* against the total data in the screen update trace. The cumulative distance distribution is actually the trace's hit rate for cache size *C*, i.e. the proportion of data in the packets which have a reference within a distance of *C*. For a given screen update trace, *D*(*C*) indicates the amount of data redundancy that can be exploited by a FIFO-managed cache with capacity *C*. Therefore, *D*(*C*) can be used to investigate the amount of data redundancy among a subset of screen update data and to guide the selection of cache capacity.

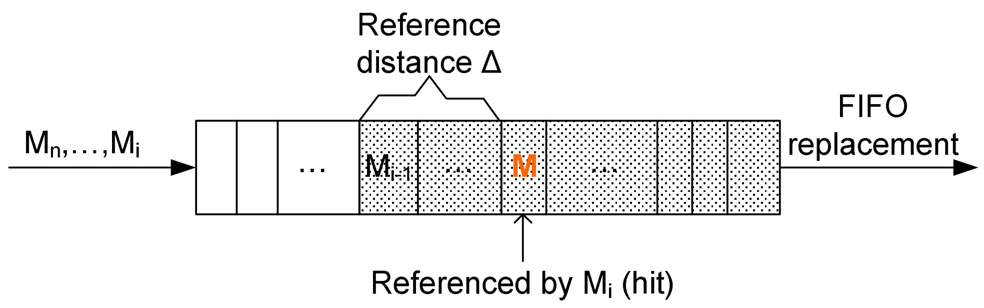

Figure 7.1: Data redundancy analysis model

7.3. Data Spike Analysis

7.3.1. Data Spike Definition

The bursty nature of screen updates is the source of data spikes in thin client systems. Once user input is processed, some screen update data are generated in a very short time. Some user operations induce a large amount of screen update data, raising data spikes. Examples of such user operations include launching a new application, switching to another application, opening a dialog with complex graphic elements and displaying some large menus such as the 'start' menu in Microsoft Windows. In a thin client system, screen update data are normally packetized, compressed and then sent over network. In this process, the network interface of the server can be viewed as a queue system. We assume that the data in the same packet arrive in the queue at the same time. Given a short period τ and an integer S_{th} , if the total data arriving at the queue in time interval (t- τ, t) amounts to S_{th} bytes or above, we say there is a (τ, S_{th})-**data spike** at time t. τ is called **data spike width** and S_{th} is called **data spike threshold**. Any packet arriving in period (t- τ, t) is said to be a **contributor** of the data spike at time t. If a contributor's individual size is above half of data spike threshold, we call it a **main contributor**. A data spike can have one or more main contributors. For example, in Figure 7.2, data spike *A* has 2 main contributors. Two data spikes can share contributors and main contributors. In the figure, data spike *B* and *C* share a main contributor.

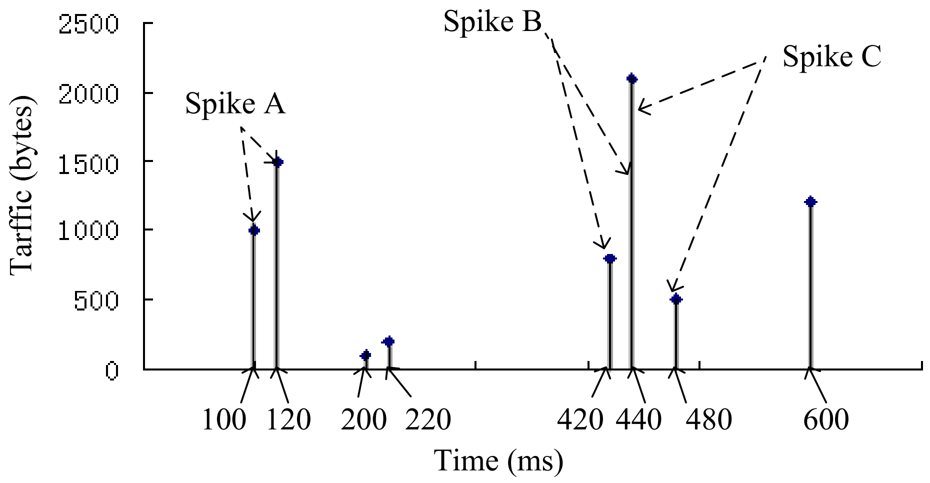

Figure 7.2: Data spikes (τ = 50ms, S_{th} = 2K bytes)

7.3.2. Distance Distribution of Data Spike

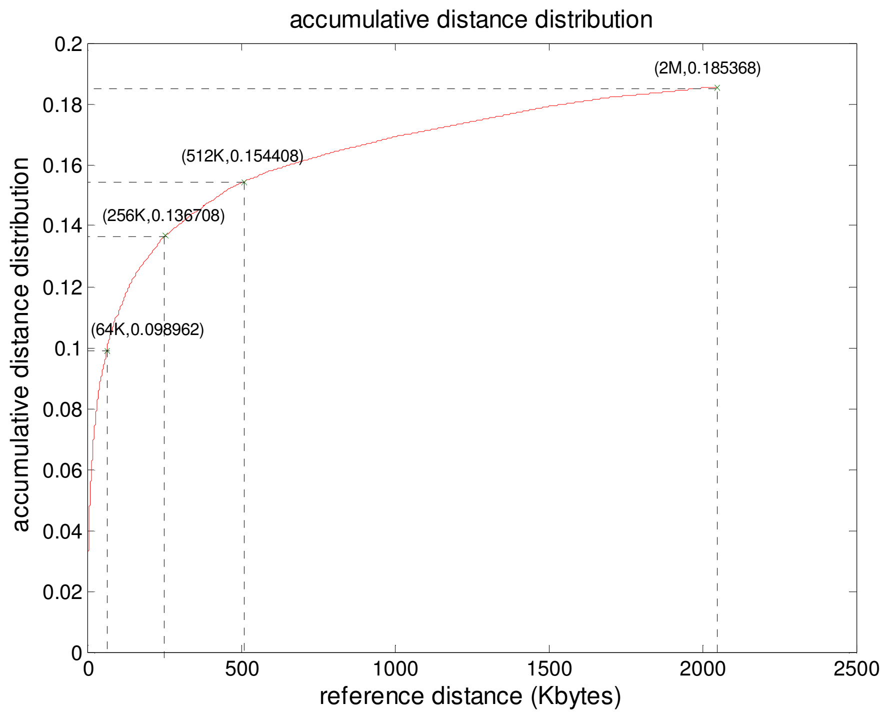

Figure 7.3: Average cumulative distance distribution of data spike main contributors

Using the analysis model proposed in Section 7.2 we analyzed the packet-level redundancy of data spikes in 100 screen update traces. In this analysis, the cache capacity C is specified as 2M bytes. Such a cache size is acceptable in many practical situations for thin client computing. We chose 2K bytes and 0.15 respectively for the parameter S_{th} and D_{th}. The data spike width (τ) used is 20ms. The reason to choose these values will be given later in Section 7.5.1. We only analyzed the redundancy between main contributors. The experimental results show that in all traces redundant main contributors account for 6.3 - 34.4% of the total data, and in 79 traces this number is larger than 10%. Figure 7.3 shows the cumulative reference distance distribution of these screen update traces when they are concatenated together. It can be seen that the cumulative reference distance distribution increases with the reference distance but the increase gradually drops. When the reference distance is 512K bytes the distribution is 15.4% while the value is 18.5% at reference distance 2M bytes. This indicates that there is a considerable amount of packet-level redundancy in data spikes of screen updates that can be exploited by a reasonably small cache.

7.4. Data Spike Reduction Scheme

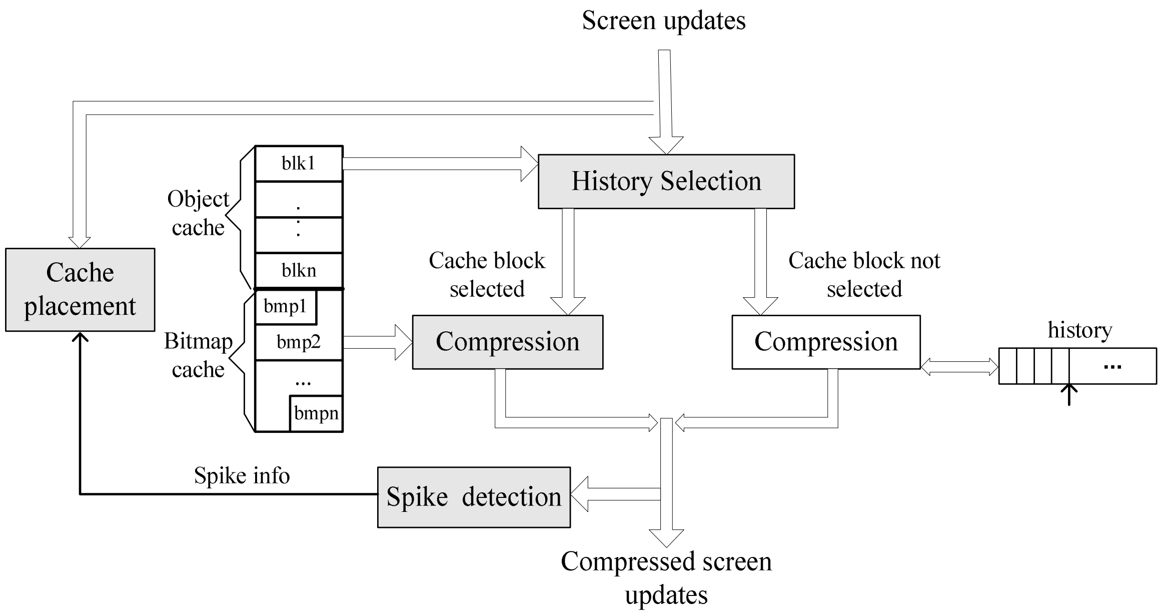

Figure 7.4: Overview of DSRS (server side)

Based on the analysis in Section 7.3, we propose in this section a hybrid cache-compression scheme, called Data Spike Reduction Scheme (DSRS), to reduce the data spikes in screen update streams. Figure 7.4 shows an overview of DSRS. DSRS is built around a general online compression scheme. It enhances the compression scheme with two caches separately on the server and client side. The spike detection component in DSRS identifies data spikes in compressed screen updates and saves the main contributors of them into the cache. The history selection component in DSRS selects a good history to compress a large packet. If such a history does not exist, the normal compression method is used for that packet. The client-side cache is always synchronized with the server-side cache so that the encoded data can be decoded using the client-side cache as history. Compared to the general compression scheme, DSRS is able to exploit some long-distance references and has the flexibility to reduce some specific screen updates. By selecting history, some large packets can be better compressed, reducing the possibility of causing a data spike.

7.4.1. Cache Organization

The cache in DSRS consists of two parts, respectively called object cache and bitmap cache. Object cache is used to save packets that contain various drawing orders. Bitmap cache is used to save large bitmaps each of which can result in a data spike. Object cache and bitmap cache are organized differently. Object cache is partitioned into equal-size history blocks. All the packets saved in one block can be used as a history to compress a new packet. To limit the number of bits to encode a match in a history block, we restrict the history block size to be 64K bytes or smaller. But the number of history blocks can be negotiated between the client and the server, allowing a tradeoff between performance and resource requirement. Packets are saved in a history block continuously from the beginning to the end. As in MPPC, if the remaining space in a history block is not enough to hold a new packet, the new packet will be saved from the beginning of the block, overwriting the old data. Bitmap cache is a continuous memory area which is organized as a ring buffer. Bitmaps are saved into the buffer one after another. Each saved bitmap can be used as a history to compress a new bitmap. An additional data structure records the width and height of a bitmap as well as its location in bitmap cache. When an old bitmap in bitmap cache is overwritten, its information is removed from the additional data structure. Like object cache, the size of bitmap cache is also negotiable.

7.4.2. Data Spike Detection and Main Contributor Saving

A large bitmap itself will cause a data spike. Hence all of them will be saved into the bitmap cache. To detect data spikes caused by other screen update data, DSRS monitors the compressed packets. According to the definitions in Section 7.3, if the total data received within a data spike width exceed the data spike threshold a data spike is assumed and its main contributors will be saved. As the main contributors of a data spike may be received before the data spike is identified, we need to trace the recently received large packets that are possible main contributors. Since two contiguous data spikes can share main contributors, a large packet may be saved already. So we trace only the recent large packets that are not saved. According to the definition of main contributor, only one possible main contributor within a data spike width may not be saved before a data spike is found. After a data spike is identified, its main contributors will be saved into the object cache. When the content of the server-side cache changes the same change should be made to the client-side cache to keep them synchronized. The client also records the possible main contributors as the server does. Later when a spike is detected, the server will inform the client of the main contributors to be saved and which history block they will be saved in. For bitmaps, we need to pass the bitmap size information to the client. We utilize a few bits in packet header to pass such information.

7.4.3. History Selection

How to select a history for a new packet or bitmap from the saved history blocks or bitmaps based on proximity is a key problem in DSRS. To reduce searching time, we first shortlist the candidates by comparing packet sizes or bitmap sizes. As per Lemma 7.1, using a packet or bitmap's reference as history to compress it, we can get a small communication cost. According to the definition of reference, a packet or bitmap has similar size as its reference. For a new bitmap we compare its width and height with each saved bitmap's width and height to determine if the saved one is a possible reference. For other screen updates, we assign a packet size range (l_i, u_i) to each history block B_i and only store the packet whose size is larger than l_i and not larger than u_i into block B_i. This way, we shrink the search space to the cache blocks whose assigned sizes are close to the current packet's size.

At the second stage, we calculate the match edit distance from the index of each shortlisted candidate to the index of the new packet or bitmap. If the match edit distance associated with a candidate is smaller than a threshold, that candidate is selected and the selection process stops. Using a byte sequence's index rather than the byte sequence itself makes this process very fast for the index is much shorter. Computation of the indexes is enlightened by the content-based partition technique proposed in [42]. This technique calculates a 64-bit Rabin Fingerprints [11] for every 48-byte region (overlapping) in a byte sequence. When a fixed number of lower bits of a region's fingerprint equal to a chosen value, the region forms a breakpoint. These breakpoints divide a byte sequence into chunks. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. Rabin Fingerprints can be computed in linear time with simple logic and shifting operations and can be computed in a rolling style [64]. This partition method avoids sensitivity to data shifting in a byte sequence by setting breakpoints based on packet contents rather than the positions within a byte sequence. Insertions and deletions therefore only affect the local breakpoints [42].

In our context, a breakpoint is formed when the low-order 7 bits (for order data) or 12 bits (for bitmap data) of a region's fingerprint equal to a chosen value. The distances between every two breakpoints are recorded in a sequence and this sequence is defined as the byte sequence's index. To prevent breakpoints from being too dense or too sparse, we restrict the distance between two breakpoints to be at least 64 bytes and no more than 256 bytes (8K bytes for bitmap) by eliminating/inserting breakpoints if necessary. We use more low-order bits for bitmap data because a bitmap is usually much larger than a packet. When a packet or bitmap is saved in the cache, the index of it is also saved. Since each distance in an index needs two bytes and the minimal distance is 64 bytes, the indexes for all packets in a 64K bytes history need at most 2K bytes.

7.4.4. Encoding

For screen updates using object cache as history, we use LZ algorithm to calculate an edit script. If a packet is encoded using a history block on the server side, it must be decoded using the same block on the client side. We utilize a few bits in the packet header to inform the client which history block should be used to decode the current packet. The number of bits needed is subject to the number of cache blocks used. For the bitmap data we use a two-dimensional LZ algorithm (2D-LZ) [65] to calculate an edit script. This algorithm takes advantage of the inherent two-dimensional nature of bitmap data. The match between the target bitmap and the reference bitmap is a rectangular region rather than a linear data sequence. This gives better compression efficiency compared to LZ algorithm. As illustrated in Figure 7.5, in 2D-LZ algorithm, a match region can be specified with 4 fields: the match region's x offset, y offset, width and height. In contrast, LZ algorithm needs to encode each line of the match region.

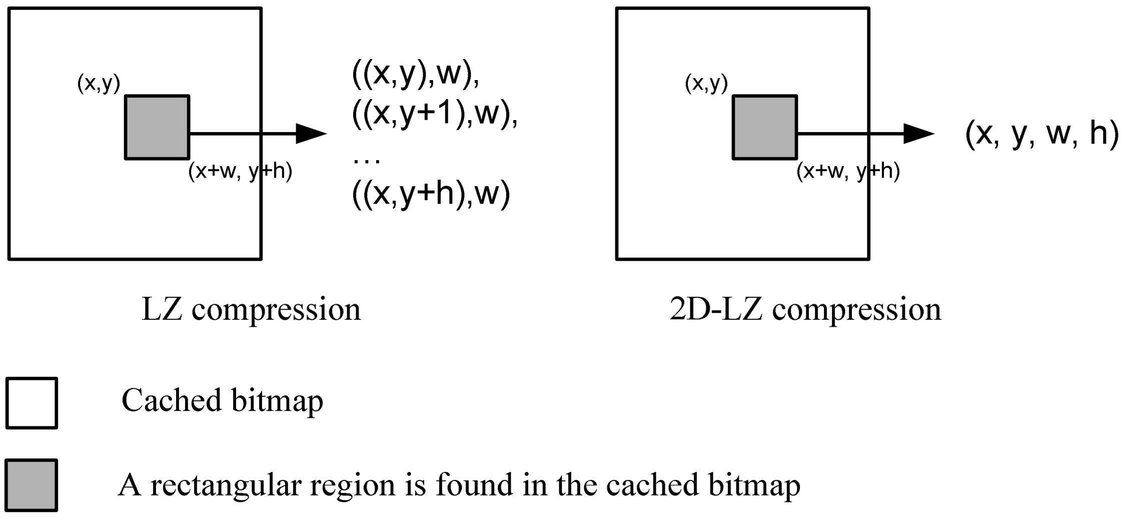

Figure 7.5: Encoding a matched region of bitmaps: 2D-LZ versus LZ

7.5. Experiment Results and Analysis

In this section we compare the performance of DSRS with MPPC in terms of data spikes, network traffic and interactive latency. We described how we reach the values of the three important parameters data spike threshold (S_{th}), data spike width (τ), and difference threshold (D_{th}) in Section 7.5.1. We measured the performance of DSRS with 2, 4, 8, and 32 cache blocks and presented the results in Section 7.5.2. For the configuration of 2, 4, and 8 cache blocks, bitmap cache is not used. For the configuration of 32 cache blocks, we set aside 16 blocks to the object cache and the rest to the bitmap cache. Unlike data spikes and network traffic, interactive latency is related to network environment. We measured interactive latency in 7 emulated network environments described in Chapter 5 Section 5.3. The computation overhead and memory usage of DSRS are discussed in Section 7.5.3.

7.5.1. Parameter Selection

Data spike threshold (S_{th}), data spike width (τ), and difference threshold (D_{th}) are important parameters in DSRS. We experimented with various values for them. According to our experiments, if data spike threshold is too large (e.g. 4K bytes), some large packets cannot constitute data spikes. These large packets could cause a long latency in some practical low-bandwidth situations (e.g. DSL). But if data spike threshold is too small (e.g. 1K bytes), many small packets will form data spikes and will be cached. This causes some large packets to be overwritten before they reoccur. When data spike width is too short (e.g. 10ms), some continuous packets cannot together constitute a data spike. But when it is too long (e.g. 50ms) a lot of meaningless data spikes (having the same main contributors) will appear. If the difference threshold is too large (e.g. 0.4), the selection process may not select a good enough history when it stops. But if the difference threshold is too small (e.g. 0.1), the selection process frequently does not find a good history. For the experiments presented in Section 7.5.2, we used the same empirical values for data spike threshold (2000 bytes), data spike width (20ms), and difference threshold (0.15).

7.5.2. Performance

Figure 7.6 shows the data spike reduction of DSRS compared to baseline MPPC (for details about the baseline MPPC please refer to Section 5.4). Overall, DSRS eliminates around 8.5% - 12.1% data spikes with 2 cache blocks, and around 26.7% - 42.2% with 32 blocks. This indicates DSRS can efficiently reduce the data spikes in different types of user traces. We can see that for all three traces the reduction ratio increases with the number of cache blocks used. Using more cache blocks permits more data to coexist in the cache, improving the hit chance. Nevertheless, not every data spike can be removed by DSRS. Some data spikes are caused by the screen updates that do not appear before and thus cannot benefit from a cache scheme. Compared to *image* and *vector* traces, a lower percentage (20.69%) of data spikes in *text* trace are reduced by DSRS. Further analysis shows that this is because there are more long-distance redundancies in *image* and *vector* than in *text* trace. From Figure 7.6 we can also notice that with 8 or less cache blocks *vector* trace improves faster than *image* trace as the number of cache blocks increases. But from 8 to 32 blocks, *image* trace improves more than *vector* trace. In *image* trace most of the data spikes are formed by bitmap data while in *vector* trace many data spikes are formed by a large number of drawing orders. The bitmap data can be better compressed with the separate bitmap cache in 32-block configuration.

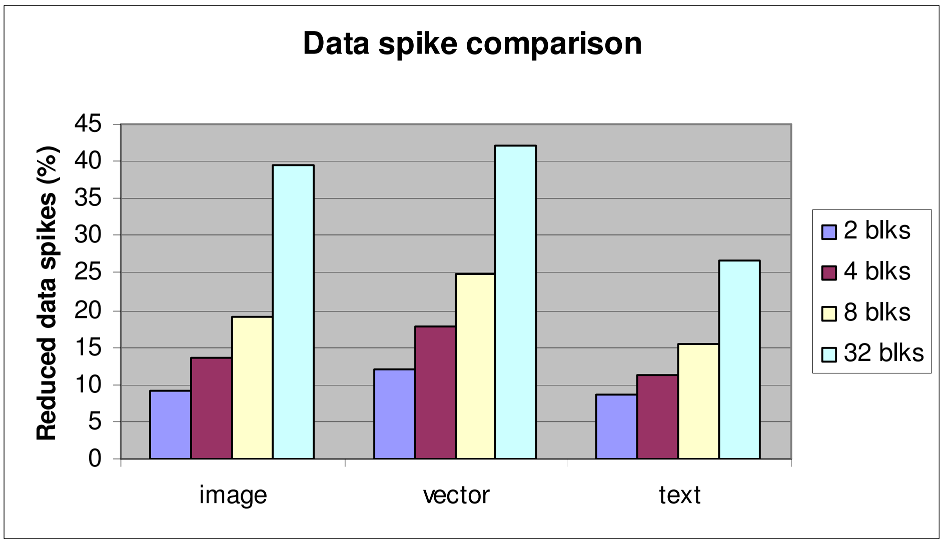

Figure 7.6: Data spike comparison between DSRS and MPPC

Figure 7.7 shows the network traffic reduction of DSRS compared to MPPC. Again we can observe that more cache blocks lead to better results and overall DSRS is able to significantly reduce network traffic for all three traces. With 32 blocks, DSRS reduces 21.2%, 19.2% and 9.9% network traffic for *image*, *vector* and *text* respectively. Data spikes are quite data intensive. Reducing them can directly decrease network traffic. In contrast to data spike reduction, *image* trace achieves more network traffic reduction than *vector* trace in 32-block configuration. This is because the data spikes in *image* trace account for a higher proportion of the total network traffic than in *vector*.

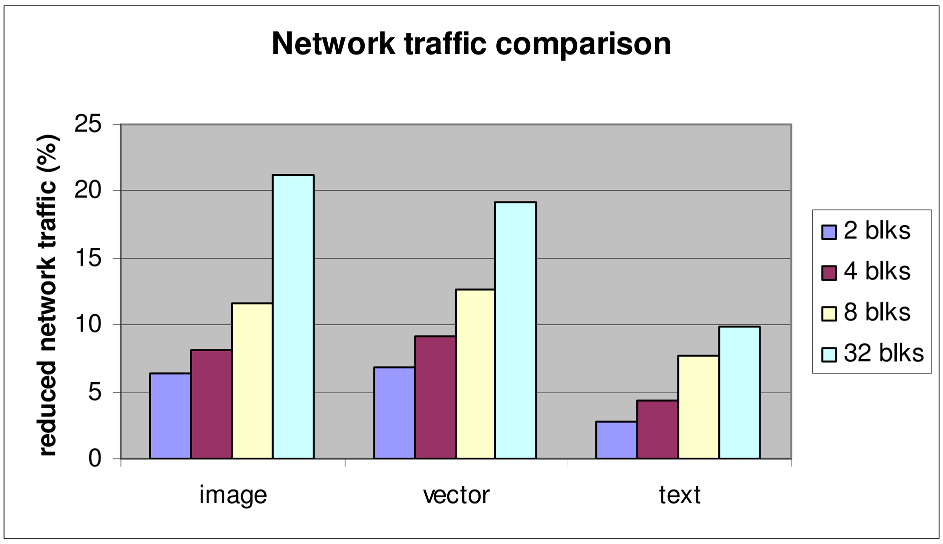

Figure 7.7: Network traffic comparison between DSRS and MPPC

To evaluate the effect of DSRS on interactive latency, we also partition the interactive response time into three time bins as used in Chapter 6: *crisp* – shorter than 150ms; *noticeable* – longer than 150ms but shorter than one second; *annoying* – longer than one second. Table 7.1 gives the number of operations that fall into noticeable or annoying time bin for different traces, cache configurations and network environments. According to the results in Table 7.1, DSRS achieves the same interactive performance as MPPC in LAN. In LAN environment the time to transfer a data spike is usually small. The long latencies are mainly induced by long application processing time rather than network transfer time. For all chosen WAN network configurations, DSRS reduces the number of noticeable and annoying response time compared to MPPC. Like data spike reduction, more cache blocks produce better results. With 2 cache blocks DSRS reduces 3.4% - 4.4%, 6.2% - 7.0%, and 6.9% - 8.0% noticeable or even annoying latencies for *image*, *vector* and *text* respectively. With 32 blocks DSRS respectively reduces 28.5% - 33.0%, 33.4% - 38.5%, and 25.8% - 28.6% noticeable or even annoying latencies for the three traces. These results show that DSRS can efficiently reduce the number of long operation latencies in WAN environments. With the limited bandwidth in WAN environment, a data spike will need a long time to be transferred to the client. The large network latency will exacerbate this situation as more time is needed before all the data in a data spike are delivered to the client. As a result, we can see in Table 7.1 that there are a larger number of noticeable latencies when the network bandwidth decreases and the network latency increases.

Trace	Network	Noticeable 150 ms - 1 sec					Annoying >1 sec				
		M*	**D2**	**D4**	**D8**	**D32**	**M**	**D2**	**D4**	**D8**	**D32**
Image (13344 operations)	LAN	24	24	24	24	24	6	6	6	6	6
	WAN1	189	181	171	157	128	24	24	24	22	17
	WAN2	343	328	313	292	242	37	35	34	31	26
	WAN3	521	500	476	431	349	48	44	41	36	32
	WAN4	262	251	239	222	184	31	30	27	24	17
	WAN5	395	382	367	340	289	44	40	38	35	25
	WAN6	694	673	646	588	482	70	65	59	51	42
Vector (17912 operations)	LAN	17	17	17	17	17	3	3	3	3	3
	WAN1	228	213	201	185	145	29	28	28	26	19
	WAN2	397	369	353	324	245	34	32	31	29	20
	WAN3	639	598	576	541	422	47	44	40	35	28
	WAN4	306	285	272	253	196	33	31	29	25	20
	WAN5	460	431	419	387	299	48	43	38	34	27
	WAN6	881	829	802	746	592	58	51	46	40	33
Text (13834 operations)	LAN	10	10	10	10	10	2	2	2	2	2
	WAN1	63	58	55	51	46	4	4	4	2	2
	WAN2	126	117	110	104	92	4	4	4	2	2
	WAN3	234	216	202	191	168	7	6	6	4	4
	WAN4	93	86	81	76	70	4	4	4	2	2
	WAN5	145	133	128	119	106	5	5	5	3	3
	WAN6	301	280	261	247	220	7	6	6	4	4

* M – MPPC; D2, D4, D8, D32 – DSRS with 2, 4, 8 and 32 cache blocks.

Table 7.1: Interactive latency comparison between DSRS and MPPC

7.5.3. Computation Overhead & Memory Usage

On the server side, the computation overhead of DSRS has two components: selecting a cache block as history, and saving main contributors of data spikes, including updating auxiliary data structures. Both components are approximately proportional to the amount of data processed. Column 2 and 3 of Table 7.2 show the average cost to process 1K bytes of data for the three traces. As mentioned in Section 7.4, not all screen updates need to be processed by them. In *image*, *vector* and *text*, the cached main contributors respectively contain 29.3%, 28.2%, and 17.8% of the total screen update data. The packets involving history selection respectively contain 35.4%, 30.7%, and 15.5% of the total screen update data. Column 4 of Table 7.2 shows the compression speed of DSRS. Compared to the baseline compression speed given in Table 5.3, we can find that DSRS respectively increased 9.4%, 13.6%, and 7.0% computation time for the three traces. On the client side, the only difference between DSRS and MPPC is that DSRS uses the history block specified by the server to decompress a packet while MPPC uses the only history block. The decompression process is the same. So DSRS introduces little computation overhead on the client side. Like other compression schemes, DSRS is a tradeoff between computation time and network transferring time. Hence DSRS is not suited to be used where the scalability of the thin client server is tightly bounded by the computation resources. In the case that the scalability of a thin client server is mainly bounded by network bandwidth, using DSRS can improve its scalability.

Screen update trace	**History Selection** (μs/KB)	**Contributor saving** (μs/KB)	**Compression Speed** (μs/KB)
Image	12.39	20.93	67.29
Vector	14.12	20.14	62.91
Word	10.08	17.60	52.13

Table 7.2: Computation overhead of DSRS

In our implementation, we use hash tables as auxiliary data structures on the server side. For every 4 bytes written to a cache block, we create a hash table entry for it. Hence the number of hash entries needed is one fourth of the cache block size. Besides, each cache block needs 2K bytes to save the index of the data in it. In sum, the total additional memory required on the server side can be calculated by $(cache\ blk\ size + \frac{cache\ blk\ size}{4} \times bytes\ per\ hash\ entry + index\ size) \times cache\ blk\ number$. If the cache block size is 64K bytes or less, each hash entry costs 2 bytes to save the offset. The additional memory required on the client side is equal to the cache size.

7.6. Summary

In thin client computing, data spikes are caused by the need of transferring a large amount of screen update data in a very short time. These data spikes cause long interactive latencies when the available bandwidth is not enough, giving users unsmooth usage experiences. This chapter analyzed the object-level redundancy in data spikes. Based on the analysis result, we proposed a hybrid cache-compression scheme, DSRS, to reduce the data spikes. We empirically studied the effectiveness of DSRS using a number of screen update traces of Microsoft Terminal Server. The experimental results show that DSRS can reduce 26.7% - 42.2% data spike count and 9.9% - 21.2% network traffic for the tested data with a cache of 2M bytes. Moreover, DSRS can reduce 25.8% - 38.5% noticeable long latencies for different types of applications with the same cache configuration. This scheme costs only a little additional computation time and the cache size can be negotiated between the client and server.

CHAPTER 8

CONCLUSIONS

Thin client computing has made rapid development during recent years. To make thin client computing more widely used, especially in low-bandwidth and high-latency networks, the performance of existing thin client computing systems still need to be improved. The purpose of this dissertation is to investigate techniques to optimize the performance of thin client computing. In the first part of this thesis, we revealed the existence of static objects in an application's GUI and proposed a static object cache technique to reduce the network traffic caused by redundant transferring of static objects' presentation data. Our experiment results show that this approach reduces both the baseline VNC's network traffic and interactive latency up to 60% with only a little more CPU usage. While this application-specific approach is efficient, it requires offline profiling to collect static object information of an application. Hence it's only suited for the enterprise environment where applications need to be shared by many users.

In the second part of this thesis we proposed a long-distance redundancy reduction scheme (LDRS), to reduce the data redundancy occurring at long distance. LDRS is based on the dictionary-based Lempel-Ziv algorithm (LZ) [13], which is widely used in thin client systems. LDRS extends the history buffer of LZ algorithm in a vertical and discrete way. This gives LDRS more flexibility and scalabilities over a flat extension. Our analysis shows that LDRS can achieve a compression performance similar to using a flat long continuous history. Experimental results further show that compared to the baseline LZ-based MPPC compression scheme, LDRS can reduce 15.0% - 33.5% network traffic and 20.6% - 27.8% noticeable long latencies for different types of applications with a history buffer of 2M bytes. LDRS can be used to compress any kinds of screen update data. It is especially efficient when the long-distance redundant data account for a large portion of the total screen updates.

The memory and computing resource may be limited on a thin client. When the resources are limited, how to use them wisely will become important. Not painting every graphical object will cause user-perceptible latency. Painting a simple and small graphical object may only induce a small amount of screen update data and short user latency. But painting complex and large objects can generate a data spike, which is a large amount of screen update data produced in a short time. Delivering data spikes to thin clients may induce long user latencies and unsmooth user experiences in many contemporary networks. Motivated by this, we proposed the data spike reduction scheme (DSRS) which utilizes the limited memory resources to improve the user experience as far as possible.

DSRS is a hybrid cache-compression scheme which reduces data spikes by caching their main contributors and uses the cached data as history to better compress the recurrent screen updates in possible data spikes. We empirically studied the effectiveness of DSRS using a number of screen update traces of Microsoft Terminal Server. The experimental results show that DSRS can reduce 26.7% - 42.2% data spike count and 9.9% - 21.2% network traffic for the tested data with a cache of 2M bytes. Moreover, DSRS can reduce 25.8% - 38.5% noticeable long latencies for different types of applications with the same cache configuration. These results suggest that removing data spikes in thin client screen update streams can reduce user interactive latency and improve user-perceptible performance. Like LDRS, DSRS costs only a little additional computation time. The cache size can be negotiated between the client and server. DSRS is intended to reduce long latencies. In high-bandwidth, low-latency network environment, long latencies are rarely seen for thin client computing. In that situation DSRS has a little influence on interactive latency. Nevertheless, it will still reduce network traffic in that situation if data spikes are produced.

During the course of carrying out this research, we also found some possible directions for future research. We highlight them as below.

- The optimization techniques presented in this thesis are designed for GUI-intensive applications. We aim to reduce the interactive latency of GUI operations. Multimedia applications such as MediaPlayer are not the target applications of these optimization techniques as such applications need only infrequent interactions and are not sensitive to latency. However, such applications produce a lot of screen updates and still can cause unsmooth screen updates when the network bandwidth is not enough. How to reduce the network traffic produced by these applications in a thin client computing system needs to be further investigated, especially for the thin clients constrained by memory and CPU power. For still or motion pictures, a little loss of accuracy is often tolerable. Lossy compression methods such as those presented in JPEG and MPEG standards are very efficient. But before such methods can be applied, two problems should be addressed. First, this requires the thin client to have more resources for decoding. Hence, simplified versions of such compression methods may be more practical. Another problem is that there should be a method for the server to detect that the current display updates are of such specific types.

- The optimization techniques presented in this thesis are designed for existing thin client computing techniques which perform well in PC environments. However, the thin client computing techniques are more and more widely deployed in mobile devices such as mobile phone and PDA. For these devices with limited computation and memory resources, merely deploying these optimization techniques may not be enough. Some new display encoding methods should be proposed to fit in these new networking applications. A new display encoding method should take into consideration both the pros and cons of existing display encoding methods. How to achieve as low bandwidth consumption as the vector-based thin client encodings and at the same time achieve as low client complexity as the bitmap-based thin client encodings need to be further researched.

BIBLIOGRAPHY

[1] S.K. Card, T.P. Moran and A. Newell. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, U.S.: L. Erlbaum Associates, 1983.

[2] H. Kalmus, D.B. Fry and P. Denes. "Effects of Delayed Visual Control on Writing, Drawing and Tracing". *Language Speech*, 1960, v3, pp. 96-108.

[3] J.L. Guynes. "Impact of System Response Time on State Anxiety". *Comm. ACM*, 1988, vol. 31, no. 3, pp. 342-347.

[4] G.L. Martin and K.G. Corl. "System Response Time Effects on User Productivity". *Behavior and Information Technology*, 1986, vol. 5, no. 1, pp. 3-13.

[5] R.B. Miller. "Response Time in Man-Computer Conversational Transactions". *Proc. AFIPS Fall Joint Computer Conf.*, 1968, AFIPS Press, pp. 267-277.

[6] A. Rushinek and S.F. Rushinek. "What Makes Users Happy?". *Comm. ACM*, 1986, vol. 29, no. 7, pp. 594-598.

[7] B. Kahneman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 3rd ed. Reading, MA: Addison-Wesley, 1997.

[8] N. Tolia, D.G. Andersen, and M. Satyanarayanan. "Quantifying Interactive User Experience on Thin Clients". *IEEE Computer*, March 2006, vol. 39, no. 3, pp. 46-52.

[9] A. Lai and J. Nieh. "Limits of Wide-Area Thin-Client Computing". *Proceedings of the ACM SIGMETRICS 2002*, Marina del Rey, CA, June 15-19, 2002.

[10] A. Lai and J. Nieh. "On the Performance of Wide-Area Thin-Client Computing". *ACM Transactions on Computer Systems (TOCS),* May 2006, vol. 24, no. 2, pp. 175-209.

[11] M.O. Rabin. "Fingerprinting by Random Polynomials". Center for Research in Computing Technology, Harvard University, Technical Report TR-15-81, 1981.

[12] J.M. Danskin. "Compressing the X Graphics Protocol". Ph.D Dissertation, Department of Computer Science, Princeton University, NJ, U.S., Jan. 1995.

[13] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". *IEEE Trans. Inform. Theory*, May 1977, vol. IT-23, pp. 337-343.

[14] R.W. Scheifler. “The X Window System Protocol”. M.I.T. Laboratory for Computer Science, 1988.

[15] R.W. Scheifler and J. Gettys. *X Window System*. Burlington, MA: Digital Press, 1992.

[16] R.W. Scheifler and J. Gettys. “The X Window System”. *Transactions on Graphics*, April 1986, vol. 5, no. 2, pp. 79-109, and *Software Practice and Experience,* October 1991, vol. 20(S2), S2/5-S2/34.

[17] J. Postel. "Transmission Control Protocol". USC/Information Sciences Institute, Marina del Rey, CA, Report RFC 793, Sept. 1981.

[18] S. Wecker. "DNA: The Digital Network Architecture". *IEEE Trans. Commun*, Apr. 1980, vol. 28, no. 4, pp. 510-526.

[19] D. MOON. "Chaosnet". Artificial Intelligence Laboratory, MIT, Cambridge, MA, AI Memo 628, June 1981.

[20] "Introduction to the X Window System Protocol". Available: http://www.x.org/about_x.htm.

[21] J. Gettys and R.W. Scheifler. "Xlib – C Language X Interface Reference", *X Consortium Standard*, X Version 11, Release 6.4.

[22] D. Converse et al. “LBX - Low Bandwidth X Extension”. X Consortium, 1996.

[23] J. Fulton and C.K. Kaatarjiev. “An update on low bandwidth X (LBX)”. *Proceedings of the 7th Annual X Technical Conference*, January 1993, O’Reilly and Associates.

[24] *LBX X Consortium Algorithms.* X Consortium, 1996.

[25] T. Richardson, Q. Stafford-Fraser, K.R. Wood and A. Hopper. "Virtual Network Computing". *IEEE Internet Computing*, Jan./Feb. 1998, vol. 2, no.1.

[26] T. Richardson and K. Wood. “The RFB Protocol”. ORL Cambridge, January 1998.

[27] *UltraVNC*. Avaiable: http://ultravnc.sourceforge.net/.

[28] *ITU T.128 Protocol*. Avaiable: http://www.itu.int/rec/T-REC-T.128/en.

[29] *ITU T.120 Protocol*. Avaiable: http://www.itu.int/rec/T-REC-T.120/en.

[30] "Tarantella Web-Enabling Software: One World, One Network, One answer". SCO Inc., Tarantella White Paper, May 2001.

[31] "Bandwidth Usage using Citrix ICA". Tolly Research, Technical Report, Sep. 2001.

[32] "UltraLight Client ". Canoo Engineering AG, Technology White Paper, Apr. 2004. Available: http://www.canoo.com/ulc.

[33] "ULC Rich Thin Clients for J2EE". Canoo Engineering AG, Technology White Paper, June 2004. Available: http://www.canoo.com/ulc.

[34] P. Bahrs and B. Feiqenbaum. "Introduction to Thin Client Framework, Part 1: The basic elements". IBM technical paper, Jan 2003.

[35] "Citrix ICA Technology Brief". Boca Research, Boca Raton, FL, Technical White Paper, 1999.

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

[37] B.O. Christiansen, K.E. Schauser, and M. Munke. "Streaming Thin Client Compression". *Proceedings of the Data Compression Conference (DCC)*, Mar. 2001, Snowbird, UT.

[38] F. Li and J. Nieh. "Optimal Linear Interpolation Coding for Server-Based Computing". *Proceedings of the IEEE International Conference on Communications (ICC) 2002*, New York, NY, April 28-May 2, 2002, pp. 2542-2546.

[39] C. Aksoy and S. Helal. "Optimizing Thin Clients for Wireless Active-media Applications". *WMCSA*, 2000, pp. 151-160.

[40] A. Helal and S. Ramamaurthy. "Optimizing Thin Clients for Wireless Computing via Localization of Keyboard Activities". *Proceedings of the 2001 International Performance, Computing, and Communication Conference*, Phoenix, Arizona, April 2001, pp. 249-252.

[41] S. Ramamurthy. "Localization of Keyboard Activity during High Network Latency". Master Thesis, University of Florida, 2000.

[42] A. Muthitacharoen, B. Chen and D. Mazieres. "A Low-bandwidth Network File System". *ACM SIGOPS Operating Systems Review*, 2001, vol. 35, pp. 174-187.

[43] "Microsoft Point to Point Compression (MPPC) protocol". RFC2118. Available: http://rfc.net/rfc2118.html.

[44] W.J. Masek and M.S. Paterson. "A Faster Algorithm Computing String Editing Distances". *Comput. System Sci.*, 1980, vol. 20, pp. 18-31.

[45] *Zlib*. Available: http://www.zlib.net/.

[46] *MPPE/MPPC Kernel Module for Linux*. Available: http://mppe-mppc.alphacron.de/.

[47] P.J. Ausbeck. "Context Models for Palette Images". *Proceedings of the IEEE Data Compression Conference*, Apr. 1998, Snowbird, UT.

[48] P.J. Ausbeck. "A Streaming PWC Model". *Proceeding of the IEEE Data Compression Conference*, Apr. 1999, Snowbird, UT.

[49] I.H. Witten, A. Moffat, and T.C. Bell. "Managing Gigabytes – Compressing and Indexing Documents and Images". 2nd edition. Morgan Kaufmann Publishers, Inc., 1999.

[50] "Thin-client Networking". Tolly Research, Report TCN0102RT01E.

[51] K. Marsh. "Win32 Window Hierarchy and Styles". Microsoft msdn. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwui/html/msdn_ styles32.asp.

[52] S.J. Yang, J. Nieh, and N. Novik. "Measuring Thin-Client Performance Using Slow-Motion Benchmarking". *Proceedings of the USENIX 2001 Annual Technical Conference*, June 2001, Boston, MA, pp. 35-49.

[53] S.J. Yang, J. Nieh, M. Selsky and N. Novik, “The Prformance of Remote Display Mechanisms for Thin-Client Computing”, in *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, USA, June 10-15, 2002.

[54] J.H. Hicinbothom and W.W. Zachary. “A Tool for Automatically Generating Transcripts of Human-computer Interaction”. *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, 1993, vol. 2 of SPECIAL SESSIONS: Demonstrations, pp. 1042.

[55] *Virtual Network Computing*. Available: http://www.realvnc.com.

[56] J.W. Hunt and M.D. McIlroy. "An Algorithm for Differential File Comparison". Bell Laboratories, Technical Memorandum 75-1271-11, October 1975.

[57] K. Sadakane and H. Imai. "Improving the Speed of LZ77 Compression by Hashing and Suffix Sorting". *IEICE Transactions on Fundamentals*, 2000, E83-A(12), pp. 2689-2698,.

[58] T. Bell and D. Kulp. "Longest-match String Searching for Ziv–Lempel Compression". *Software—Practice and Experience*, July 1993, 23(7), pp. 757–771.

[59] N.J. Larsson. "Extended Application of Suffix Trees to Data Compression". *Proceedings of the Conference on Data Compression*, March 31-April 03, 1996, pp.190.

[60] *Rdesktop*. Available: http://www.rdesktop.org/.

[61] E.G. Coffman and P.J. Denning. *Operating System Theory*. Englewood Clis, NJ: Prentice-Hall, 1973.

[62] M. Carson and D. Santay. "NIST Net – A Linux-based Network Emulation Tool". *Computer Communication Review*, 2003, vol. 33, no 3, pp. 111-126.

[63] "Classic Blend Whitepaper". Applied Reasoning, Whitepaper, Oct. 2001. Available: http://www.appliedreasoning.com.

[64] A.Z. Broder. "Some Applications of Rabin's Fingerprinting Method". *Sequences II: Methods in Communications, Security, and Computer Science*, Springer-Verlag, 1993, pp. 143-152.

[65] V. Dai, A. Zakhor. "Lossless Layout Compression for Maskless Lithography Systems". *Proc. of the SPIE*, 2000, vol. 3997, pp.467-477.

[66] S. Lok, S.K. Feiner, W.M. Chiong and Y.J. Hirsch. "A Graphical User Interface Toolkit Approach to Thin-Client Computing". *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 7-11, 2002.

[67] J. Krikke. "Thin Clients Get Second Chance in Emerging Markets". *IEEE Pervasive Computing,* Oct./Dec. 2004, vol. 3, issue 4, pp. 6-10.

[68] N. Tolia, D.G. Andersen and M. Satyanarayanan. "The Seductive Appeal of Thin Clients". Computer Science Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, Technical Report CMU-CS-05-151, February, 2005.

[69] J. Golick. “Network Computing in THE NEW Thin-Client Age”. *Association for Computing Machinery*, 1999, pp.30-40.

[70] S. Kissler and O. Hoyt. "Using Thin Client Technology to Reduce Complexity and Cost". *SIGUCCS 2005*, pp. 138-140.

[71] T.W. Mathers and S.P. Genoway. "Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame". Macmillan Technical, Indianapolis, Technical Paper, 1998.

[72] "Microsoft Windows NT Server 4:0, Terminal Server Edition: An Architectural Overview". Microsoft Corporation, Redmond, Wash., Technical White Paper, 2000.

[73] "Windows 2000 Terminal Services Capacity and Scaling". Microsoft Corporation, Redmond, Wash., Technical White Paper, 2000.

[74] "Strategic Planning Assumptions for Information Technology Management". Gartner Group, Summary of Cause92 Conference Presentation, Dec. 1992.

[75] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 2nd ed. Reading, MA: Addison-Wesley, 1992.

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

[77] *Sun Ray 1 Enterprise Appliance*. Available: http://www.sun.com/products/sunray1

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

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

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

[81] A. Shaw, K.R. Burgess, J.M. Pullan and P.C. Cartwright. "Method of Displaying an Application on a Varity of Client Devices in a Client/Server Network". US Patent US6104392, Aug. 2000.

[82] G.F. Pinzari. "NX X Protocol Compression". Nomachine, Technical Report D-309/3-NXP-DOC, 2003.

[83] B. Pane. "Differential X Protocol Compressor". Available: http://www.vigor.nu/ dxpc/.

[84] P.G. Howard. "*Text Image Compression Using Soft Pattern Matching*". *The Computer Journal*, 1997, 40(2/3).

[85] J.M. Gilbert and R.W. Brodersen. "A Lossless 2-D Image Compression Technique for Synthetic Discrete-Tone Images". P*roceedings of the IEEE Data Compression Conference*, Snowbird, UT, Apr. 1998.

[86] P. Howard, F. Kossentini, B. Martins, S. Forchhammer, and W. Rucklidge. "The Emerging JBIG2 Standard". *IEEE Transactions on Circuits and Systems for Video Technology*, 1998, vol. 8, no. 7.

[87] A. Volchkov. "Server-Based Computing Opportunities". *IT Professional*, Mar/Apr, 2002, vol. 04, no. 2, pp. 18-23.

[88] J. Jing, A. Helal and A. Elmagarmid. "Client-Server Computing in Mobile Environments". *ACM Computing Surveys*, 1999.

[89] M. Le and S. Seshan. "Software Architecture of the InfoPad System". *Proceedings of the Mobidata Workshop on Mobile Wireless Information System*, Nov. 1994.

[90] R.A. Baratto, S. Potter, G. Su and J. Nieh. "MobiDesk: Mobile Virtual Desktop Computing". *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, Sep. 2004.

[91] "Microsoft Windows 2000 Server: Remote Desktop Protocol (RDP) Features and Performance". Microsoft Corporation, Redmond, Wash, Whitepaper, June 2000.

[92] M. Wild and S. Herges. "Total Cost of Ownership (TCO)". Ein Uberblick, University Mainz, 2000.

[93] "Total Cost of Application Ownership". Tolly Research, Technical Report, Oct. 2001.

[94] S.J. Yang, J. Nieh, S. Krishnappa, A. Mohla and M. Sajjadpour. "Web browsing performance of wireless thin-client computing" *Proceedings of the twelfth international conference on World Wide Web*, 2003.

[95] A.Y. Wong and M. Seltzer. "Evaluating Windows NT Terminal Server Performance". *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999, pp. 145-154.

[96] I.H. Witten, A. Moffat and T.C. Bell. *Managing Gigabytes — Compressing and Indexing Documents and Images*. 2nd edition. Morgan Kaufmann Publishers, Inc., 1999.

[97] T.A. Welch. "A Technique for High-performance Data Compression". *Computer*, June 1984, vol. 17, pp. 8-19.

[98] "ICA Client Bandwidth Analysis". Citrix Consulting Services, Citrix Systems, Inc., Whitepaper, 2001.

[99] T.T. Tay and Y. Sun. "A Novel Performance Optimization Approach for Thin Client Computing". *IASTED Intl. Conf. on Parallel and Distributed Computing and Systems 2005 (PDCS)*, Las Vegas, Sep. 2005.

[100] Y. Sun and T.T. Tay. "Improving Interactive Experience of Thin Client Computing by Reducing Data Spikes". *6th IEEE International Conference on Computer and Information Science (ICIS*), Melbourne, July 2007.

[101] Y. Sun and T.T. Tay. "Reducing Long Distance Redundancy of Thin Client Systems". *IEEE 1st international workshop on e-activities*, Melbourne, July 2007.

[102] R.W. Brodersen. "The Network Computer and Its Future". *IEEE Solid-State Circuits Conference*, San Francisco, Feb.1997.

[103] B. Howard. "Thin Is Back". *PC Magazine*, Ziff Davis Media, New York, NY, Apr. 2000, 19(7).

[104] S.J. Yang and J. Nieh. "Thin Is In". *PC Magazine*, Ziff-Davis Media, New York, NY, July 2000, 19(13).

[105] "A White Paper on GartnerGroup's Next Generation Total Cost of Ownership Methodology". Gartner Group, Gartner Consulting, Stamford, CT, White Paper, 1997.

[106] B. O'Donnell and R. Perry. "Thin Computing ROI: The Untold Story". IDC, White Paper, Nov. 2005.