WIRELESS THIN CLIENT OPTIMIZATION FOR MULTIMEDIA APPLICATIONS

By

CUMHUR ERCUMENT AKSOY

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Abdelsalam Helal, for giving me an opportunity to work on this challenging topic and for providing continuous guidance and feedback during the course of this work and thesis writing.

I wish to thank Dr. Gerhard Ritter and Dr. Joseph N. Wilson for serving on my supervisory committee and for their careful review of this thesis.

I also would like to thank Citrix Systems Inc., without whom this work would not have been possible. This thesis work was funded by a grant from Citrix Systems Inc. (grant number 4514227-12).

I also wish to take this opportunity to thank my parents for their emotional support and encouragement throughout my academic career.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

WIRELESS THIN CLIENT OPTIMIZATION FOR MULTIMEDIA APPLICATIONS

By

Cumhur Ercument Aksoy

May, 2000

Chairman: Abdelsalam Helal
Major Department: Computer and Information Science and Engineering

The research focuses on improving performance of mobile thin clients in wireless environments where network costs are high due to limited bandwidth and latency. Thin client architecture provides a means for platform independent computing, enabling any kind of device to operate any kind of application through any kind of network connection. Mobile computing and thin clients intersect in the use of resource-poor handheld devices with slow or expensive network connections. The architecture, using powerful servers, converts any kind of mobile device to a capable computer.

The thesis focuses on improvements of multimedia applications run through the thin client architecture. The optimizations reduce the network traffic between the server and the client through localization of multimedia content in the client. The client locally simulates the "active components" that are present in the application's display, without getting any help from the application or restricting the application domain.

CHAPTER 1
INTRODUCTION

Thin client computing takes its origin from the best properties of mainframe computing and the PC era. During the mainframe era, the computing was made centrally, on expensive and powerful mainframes and mini computers. “Green screen” dumb terminals were used to access those computers. These terminals did not perform any processing other than taking user input (usually through keyboard) and displaying output received from the server. The displays were text based allowing only text display and were poor in graphical contexts. Following the mainframe era, PCs entered the market with increasing processing power and decreasing costs. Developments in the microchip technology obeyed the Moore’s Law, that states “Processing power will double every 18 months without an increase in costs.” Several other factors, such as development of graphical User Interface Operating Systems like Windows and MacOS lead to the wide adoption of PCs through out the world. GUIs provided users easy to use, easy to understand applications, whose domain ranges from word processors to spreadsheets to personal planners. PCs also allowed people to customize their computer, operating system and even programs. PC era started the notion of distributed computing, the model of computing where both data and applications are shared and distributed among several computers and they work together to accomplish a certain task.

Having a base of PCs on every desk and old mainframe at the back office in air-conditioned rooms, the client/server architecture was born to extract the most computing power possible from the IT investments so far. In the client/server architecture, the server is the source of those data, and the client accesses the server to make use of that data. However, the client in the client/server architecture is a "fat" client meaning that it needs processing power to work on the data received from the server. The data transferred from the server are usually raw and the client processes them before presenting to the user.

The disadvantage of the client server model was realized as the Internet started to be a common medium of communication, connecting all the systems, whether or not they use compatible software or hardware. "Fat" clients were relying on the local machine for processing and operations, which made it difficult to port applications to new environments [19]. Conversion of a client/server application from Windows to UNIX usually requires rewriting the application. With increasing connectivity among systems, companies started demanding 100% compatibility among their systems, without the cost of redeveloping the whole application.

Thin client architecture was then born as an answer to those demands, and combining the useful properties of mainframe and PC eras. Thin client architecture can be described as storing both applications and data centrally at a server (or a group of servers), running the applications on the server, but deploying the display to a client machine and transferring the inputs from the client to the server.

The clients in this case are very similar to the dummy terminals of mainframes; because they take input and display what they receive form the server. The main difference is that they need not be text based nor be based on any specific platform. Any

device that is capable of communicating to the server and displaying the received displays will suffice in the thin client architecture. In other words, thin client architecture allows users to access to the same services as they receive through a fully functional computer regardless of which client device they use to connect to the network or what kind of connection they have.

Thin clients can be implemented in several ways: NetPCs, network computers and software-based thin clients. Some of these solutions are hardware based, while others are software or a mix. There are many thin client solution-providers in the market, leaders being the Citrix, Network Computing Devices and Microsoft. The major protocols used for the communication between the thin-client and the server are ICA from Citrix and RDP from Microsoft. Each protocol has advantages and disadvantages over another, which will be discussed in the following chapter.

Thin client architecture fits perfectly into mobile and wireless environments. Mobile devices are usually resource-poor devices, in terms of processing power and battery life. These types of devices are designed with mobility in mind and have limited capabilities in every aspect: less processing power, less battery life, less memory, poor display devices and the like. These characteristics do not fit into use of “fat” clients and thus client/server computing model, because, “fat” clients, which have to do processing to present the data or validate user input, make the mobile device run out of resources in a small amount of time. Then the mobile device either turns itself off to prevent data loss, or decreases the clock speed or take any other form of precaution which the user is not happy with.

On the other hand, mobile devices connect to the network through various means. The bandwidth, reliability and latency of those connections form a wide range of network connections. They can use different protocols (e.g. TCP/IP, BlueTooth...) which use different type of network devices (e.g. modems, wireless cards, infrared…). The applications developed for mobile devices assume and rely on three modes of operation:

- Connected
- Weakly connected
- Disconnected

In connected mode, the device is connected to the network with a reliable connection and the application can assume that the network will always be in reach. The weakly connected mode of operation is not as reliable as the first case because of congestion or losses in the network. The disconnected mode assumes there is no network connection. Although there are improvements going on in the area of bandwidth in wireless communications, latency is still the major factor that governs wireless data transfers.

Thin client architecture has the ability to adopt to low bandwidth connections and resource-poor devices, thus, creates a reliable environment for applications on mobile devices that are connected either though cable or wireless. The only part the thin client architecture falls short is the disconnected mode of operation

1.1 Thesis Objective

This thesis' goal is to improve the performance on mobile-wireless thin clients by proposing solutions to the problems imposed by mobility and wireless communications.

The performance of the thin client is measured in terms of the sense of locality that is presented to the user. The user feels that the application is running locally only if the application's active characteristics show the same activeness in the thin client environment and the response time of the application is close to its responsiveness in a local environment.

The active characteristics (called active components or ACs) are studied in this thesis and a solution to the problem of localization is proposed to create the sense of locality. Localization is the simulation of some or all parts of the application in the thin client side, without referring to the server but keeping the same state with the real application running on the server.

Localization, might be useful in two areas:

- Weakly connected or disconnected modes of operation
- Savings in network costs due to decrease in communication between server and client

When the network is down, the thin client can switch to local mode, making use of local resources until the network connection is established again. Or in a situation where costs related to network transmission are high, localization might help decreasing the communication between the thin client and the server.

1.2 Structure of the Thesis

The thesis starts with a literature survey of the thin client architecture, giving an overview of the architecture and looking at the protocols, products, tools and major

players in the thin client computing market. Chapter 3 explains the links between mobile computing and thin clients. Characteristics of mobile devices and wireless communication are presented in detail, and thin client architecture solutions to these problems are explained. Chapter 4 describes details of the sense of locality concept, explaining activity and interactivity of applications and describes the active components and their properties.

The thin client prototype system is introduced in chapter 5. The idea behind it and the algorithms along with implementation details are explained in chapter 5. Chapter 6, building on chapter 5, explains how localization of active components is achieved and what kinds of problems have to be addressed and what algorithms have been used for those.

Chapter 7 presents experimental results that were conducted with the thin client prototype. It presents graphs showing the step by step gains achieved by active component localization.

The thesis ends with the conclusion and a summary of future work to achieve higher levels of optimizations, which were not addressed in the scope of this thesis either because of the focus of the thesis or due to time limitation. The last section lists the references.

1.3 Scope of the Work

The localization proposed in the thesis is focused on the localization of active components of an application, which present a self-repeating structure. An animated GIF

in a web browser application is a good example to this type of self-repeating active component. It displays the same images over and over, which will require a server-client communication every time the display changes. In a wireless environment this communication will be enough to congest the network and reduce the interactivity of the application. The idea is to let the server detect such structures and inform the client to buffer and simulate the active components locally, while saving the network bandwidth for other communication needs.

A thin client prototype has been developed to implement the ideas proposed in this thesis, and to measure the benefits of these improvements.

CHAPTER 2
LITERATURE REVIEW

2.1 The Roots of Thin Client Architecture

2.1.1 Mainframe Era

Thin client computing can be considered as the third phase in computing history. Computing history began with the mainframe era. Mainframes were extremely powerful and expensive machines. The cost of having a mainframe ranged from several hundred thousand dollars to several million, and operating it would sum up to millions of dollars. Later on minicomputers came to market – less powerful however less expensive substitutes to mainframes. This permitted smaller enterprises to own computers. For the purpose of the discussion, from now on the term mainframe will be used to mean both mainframes and minicomputers. Mainframes and minicomputers were at their peak popularity in late 70s and early 80s [21].

The mainframe model consists of centralized computers, usually housed in secure climate controlled computer rooms. The end-user access to mainframes or minicomputers is through the dumb terminals that only have text-based screens and a keyboard connected to that and a communication port to connect to the mainframe. The

function of the dumb terminal is to take user input, transmit it to the server and displaying the output received from the server. These terminals are also called "green screens". They cost below 500$ per terminal. Today, also many PC users connect to mainframes through terminal emulators that give the PC the capability of a standard dumb terminal in addition to traditional PC capabilities.

It is estimated that at the end of 1996 there were still 24 million dumb terminals in use worldwide. In addition, 15 million PCs were deployed functioning primarily as mainframe terminal emulators.

Nowadays, mainframes continue to play a large role in corporate computing for several reasons including:

- Massive computing power is achieved in a single host through symmetric multiprocessing performed by as many as 24 or more RISC based CPUs [21].

- Unprecedented levels of fault tolerance are achieved though clustering - a technology that allows a group (or “cluster”) of computers to act as one in an environment [21].

- They have the ability to run enterprise-level business-critical applications [21].

In a mainframe environment, the dependence is on a single computer or group of computers that can be centrally managed and maintained easily. This configuration has the additional advantage of being more secure not only because the physical security of the computer room, but also because of the end-user's lack of ability (not total inability) to introduce viruses or other types of problems (e.g. file deletion) into the system [20].

The following table (Table 2-1) depicts the advantages and disadvantages of mainframe computing.

Table 2-1: Advantages and disadvantages of mainframe architecture[20]

ADVANTAGES OF MAINFRAMES	DISADVANTAGES OF MAINFRAMES
Scalability	Character based applications
Fault Tolerant (High Availability)	Expensive (High Cost of Entry)
Low Maintenance Desktop Devices: 3 times Mean Time Between Failure (MTBF) rate compared to desktop PCs	Lack of Vendor Operating System Standards and Interoperability in Multi-Vendor Environments
Centralized Backup	Timesharing Systems create a potential bottleneck.
Centralized Management	Potential Single Point of Failure (non-fault tolerant configurations)
Low Cost Desktop Devices (dumb terminals)	
High Level of Security	

2.1.2 PC Era

Due to geometric increase in processing power with decreasing costs, mainframe era gave way to the PC era. Using the leverage of high processing power the personal computer era introduced the Graphical User Interface (GUI) concept. Then came thousands of applications - spreadsheets, word processors, productivity programs, and CAD applications- one after another, creating a savvy PC customer base. Software raised users' expectations from a computer increasing productivity in a user-friendly environment.

PC era also brought the distributed computing, where the computing power is extended out of the computer room to the users' desktop. A user is not dependent on the

availability or load of the mainframe because most of the functions are now performed at the desktop PC.

However, from a business point of view, despite the fact that they increased efficiency and productivity, PCs started to swallow huge budgets dedicated to upgrades and maintenance. Each desktop had to be managed and updated (including both hardware and software), and repaired. Table 2-2 shows the advantages and disadvantages of the PC era:

Table 2-2: Advantages and disadvantages of distributed PC computing[20]

ADVANTAGES OF PC COMPUTING	DISADVANTAGES OF PC COMPUTING
Standardized Hardware	Desktop Device Cost 5 Times Greater Than Dumb Terminals (on average)
Standardized Operating Systems (highly interoperable)	Lack of Centralized Backup
Scalability	Lack of Centralized Management
Graphical User Interface	Security Risks (physical security, data access and virus introduction)
Low Cost Devices (compared to mainframes)	Lack of Cluster Capabilities
Low Cost of Entry	High Management and Maintenance Costs
Distributed Computing	
User Flexibility	
High Productivity Applications	

System configurations, data management, upgrades and user support of PCs are becoming increasingly difficult and expensive with the widespread deployment of PCs. Surveys conducted back in 1992 showed that a large installation of a networked PC client/server system has a cost of $7000 per seat, 64% of this amount allocated to hardware and software maintenance [20].

A software version upgrade in a multi-national company would take years for all PCs to be upgraded. Most PCs would have to be upgraded in hardware to support the

demanding power requirements of the new version and then installation of the software takes place.

As a result of soaring costs, the computer industry went looking for more cost-effective solutions. However, returning to mainframes, going back from the GUI to text-based terminals was not an acceptable solution. What was needed was a method to deliver PC functionality and compatibility with legacy systems while maintaining a centrally managed environment. As a result there were two developments, multi-tier software architectures and thin clients. Multi-tier programming will be discussed in the following paragraphs because it related to thin-clients. And the thin client architecture will be explained in the following sections.

2.1.3 Multi-tier Programming

The issue of easy deployment and central manageability moved software developers into thin clients also giving the additional benefits of portability and platform independence. Innovations like the Internet, Intranets, enterprise-wide networks formed a heterogeneous mix of incompatible systems. "Applications available on a specific platform" evolved into the need to port the application to every available platform, letting the user to work on any device with any kind of network connection accessing any application he needs. So the model of multi-tier programming arrived. For scalability, maintainability and portability the logic of the application is divided and implemented in three distinct layers depicted in Figure 1.

- Data layer,

- Business logic layer and
- Presentation layer.

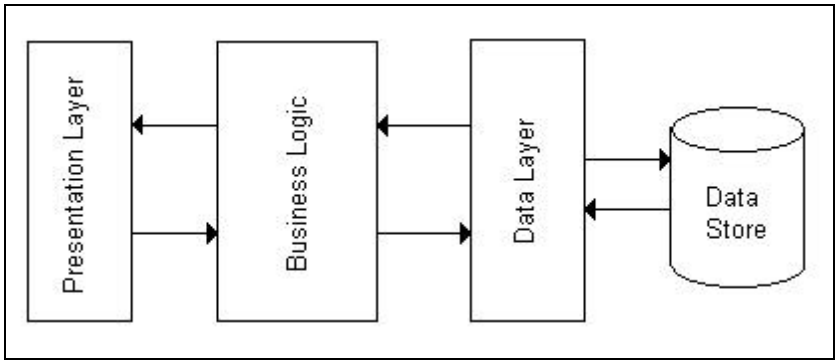


Figure 2-1: Multi-tier application design model

The data layer is where the routines and rules governing data management, storage and querying reside. The business logic layer is the main layer where all the information on how to implement application data and how to react to user requests reside. The metadata rules, governing the application's whole logic are implemented in that middle layer. And the presentation layer embodies the user interface, as thin as possible, only the means of taking input and presenting the output. To make it clearer, in an application designed with respect to the multi-tier model, the user interface will just consist of the means of presenting output and transferring input to the server. For example, consider a customer tracking system where the user has to enter the age of the customer and only customers of age 30 to 40 are eligible to be in the system. The user interface will have an input component to capture age information. The age constraint here is inherent in the applications business logic, so in a multi-tier environment it has to be implemented in the middle-tier. Thus when the user enters 45 as the age, the interface

does not perform any checks. But as the user presses “Save” button, all data goes to the middle tier, and application logic is run against the data. Then middle-tier discovers that age was given 45,which is not valid. Then an error message is sent to the presentation layer, which prompts the user of the mistake. Excluding the age rule creates a thin client, which does not have any logic embedded.

Having split the logic of the application from the interface, portability is achieved just by re-implementing the interface for a particular platform. Moreover, the presentation layer is lightweight. This enables thin clients to be loaded on-demand (no prior installation necessary) and they can work on source limited systems like PDAs or handheld PCs.

2.1.4 Other Disadvantages of Distributed Computing

Most corporate applications today are designed for high-bandwidth networks and powerful desktop computers. This type of application design puts tremendous strain on congested corporate networks and yields poor performance over lower-bandwidth, remote connections. Because of this, many users simply avoid using the vital applications to get their work done. There are still users working on Windows 3.1 because their hardware would not let them upgrade their operating system. The result is excessive work and significant decreases in productivity.

Security is also another issue on today's computing world, because in traditional client/server architectures, business-critical applications and data live on both the server and the client desktops from where it can spread out to the world via the Internet or intranets. It requires extreme care, in such a system to detect any possible information

leaks or security defects, because there is so much to protect. Not only does this increase the risk of unauthorized access, but it also increases the risk of lost or stolen information [20].

2.2 What is Thin Client Architecture?

Combining the advantages of the mainframe and PC eras, increasing portability, performance and security comes a new architecture called thin client architecture. Thin client solutions have the advantages of centralized systems' management and maintenance without leaving the boundaries of graphical user interfaces' productivity gains on multiple platforms.

The thin client approach splits the application processing from the drawing of its graphics, enabling the application to be run on high performance servers while the graphics are drawn on user's display screen. The difference from the multi-tier architecture thin-client is that the separation of the presentation logic is not achieved through design but at the operating system level, thus enabling every application to run as a thin client.

The thin client computing environment uses a server to centrally run and deploy programs to users and to centrally store data. All the applications that are to be run on a client and any data necessary are being held in that server (or a group of servers called “server-farm”). The thin client makes use of an efficient protocol to communicate with the application server.

In the case of an ultra-thin client, the user input – keyboard strokes, mouse clicks, etc. – are transferred to the server without being processed. The server processes them and then deploys back only the display, no data, no query results but just the display, in effect using the thin client as a remote display screen. As in dumb terminals, concurrent thin clients can log on to a single server and run applications in separate, protected and secure virtual sessions, sharing the server's capabilities (Figure 2-2.).

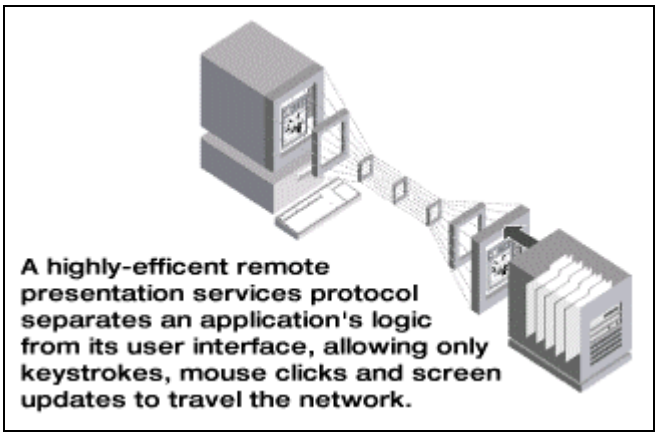


Figure 2-2: Thin client architecture (courtesy of Citrix)[2]

In this new computing architecture a thin layer of software in the client establishes a communications link between the server and the client desktop. The communications protocol only redirects the local screen, keyboard and mouse actions and posts the information received from the server to the client.

With software and data residing on the server or host system, a thin client only needs network access to be operational. Initial configuration (if there is one) can be accomplished in several minutes opposed to the many hours required to load and configure a standard PC. Server based configurations and data storage also make thin clients user independent. Any user can access their applications and data from any thin

client on the network. After logging on, a user will be presented with his/her specific desktop configuration or application options regardless of which thin client in the enterprise he/she is logged on. In this case, users are tied to a profile not their desktop PC [22].

2.3 Thin Client Solutions

There are several ways thin clients can be implemented. The following sections will explain the major methods, namely Windows Based Terminals, network computers and software thin clients.

2.3.1 NetPC Solution (Windows Based Terminals - WBTs)

The NetPC computing model is the closest to traditional client-server model in which there are several NetPCs running off a server. This solution gives multiple users simultaneous access to a single PC/server that can in turn be connected to another server. NetPCs are also called Windows Based Terminals (WBTs). NetPCs can be thought as PCs with some missing features like floppy drive, expansion slots and even hard disk in some systems. WBTs are constructed in a way that enables them to be managed over the network. Users or administrators can perform unattended upgrades from the server without touching the machine directly [15].

NetPCs might or might not offer internal storage. If they do, it is usually a small hard disk to reduce the need for network dependence. The client might download its operating system and applications from a server at startup or use its own copy if there is

one. The application software and data are stored remotely, reducing the cost of service and support of the software. Any software changes on the server or PC are immediately manifested at the user's terminal.

The NetPCs are able to run any standard Windows or UNIX applications, any browser, or any Java-enabled application.

Some implementations of NetPCs require a special connection between the server and the PC. These solutions usually require a card plugged into the server, which manages the high bandwidth connection between WBTs and the server (Figure 2-3). In such a case the machine that the terminals are connected can also be a PC. This approach can be called as desktop cloning. The Windows 95/98 systems can behave as a client to a main server for access to a common database and a server to the very-thin clients simultaneously. This technology eliminates the overhead of network traffic since the communication between the server and terminal is on special dedicated lines. Those connections sometimes can be direct video connections working at 32 Mbits or higher bandwidths. In such a case, the server/PC itself remains fully operable as a local station when access is needed to the diskette or CD-ROM drive.

Because NetPCs are essentially PCs, they have significant hardware costs and the problem of administration still exists.

WBTs can be considered as Windows version of X-Windows UNIX terminals, which have been in the market for a long time. X-Windows is a network based graphical windowing system for UNIX workstations. It was developed by MIT and has been adopted as an industry standard. Clients supporting X-Windows protocol work as thin

clients connected to a remote server that runs an X-Windows server. This type of thin clients will be explained in the software based thin clients.

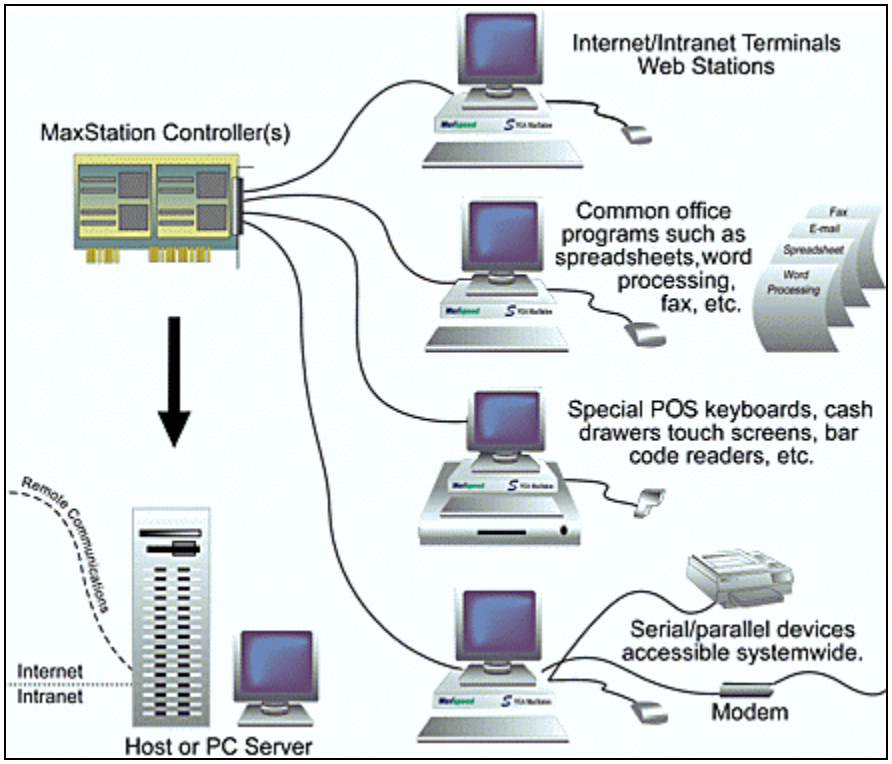


Figure 2-3: NetPC or Windows Based Terminals (WBTs)[11]

2.3.2 Network Computer (NC) Solutions

A consortium of companies (principally Acorn, Apple, IBM, Netscape, Oracle and Sun) have defined a basic set of features for their vision of a network computer through their joint effort Network Computer Inc. (now Liberate Technologies). This is known as the Network Computer Profile 1 (NC-1). NC has created a completely different architecture that uses Java OS and moves all applications to the Web browser. This option eliminates the need for PCs and uses a server that is hooked up to a series of NC terminals. The terminals download the operating system, browser and software from the

server and do Java-based processing locally (the NCs have local Java Virtual Machines). They have local memory and processing and receive data using TCP/IP protocol. There is no persistent local storage [10].

The advantages of this model are lower initial hardware and software costs and lower maintenance costs. Also, because most client system software is downloaded at the time of booting up the terminal, changes to the software are easily accomplished. A major disadvantage to the NC computing option is that existing Windows and server applications are not able to run because the applications must be coded in Java. Moreover, the lack of a local disk also means it is likely that network infrastructure sufficient for a PC LAN will need to be upgraded to cope with the extra traffic that NCs generate by anything more than light use.

Java plays an important role in the thin client architecture in NC solutions because it offers to be an architecture-neutral standard that will break the bonds that tie the applications to particular architectures.

2.3.3 Software Based Thin Client Solutions

X-Windows

X Windows system is the oldest software thin client system in the market. As mentioned above the X Window system, developed at MIT in the late 1980s, rapidly became the industry standard windowing system for graphics workstations. The software is freely available, very versatile, and is suitable for a wide range of hardware platforms, from high-end microcomputers to mainframes [23].

Any X window system consists of 2 distinct parts – the X server and 1 or more X clients. The server controls the display directly, and is responsible for all input/output via the keyboard, mouse or display. The clients, on the other hand, do not access the screen directly - they communicate with the server, which handles all input and output. It is the clients that do the "real" computing work – running applications. The clients communicate with the server, when they need to take input or display something on the screen. This causes the server to open one or more windows to handle input and output for that client.

The X server invariably runs on the machine to which the monitor is connected. The clients may also run on this machine, communicating directly with the server. On most workstations, this is the normal situation. However, X is a *networked* window system, and it is possible for the client to run on a remote machine, communicating with the server via some form of network. In most cases, this would be via TCP-IP over an Ethernet link, but it is possible to utilize alternative protocols and communication media, even a serial line connection [23]. Indeed, it is possible to run an X server on a Personal Digital Assistant (PDA), communicating with a client running on a mainframe at a remote site via a telephone link - even via a mobile phone or wireless connection. Figure 2-4 shows the X-Window system.

Citrix WinFrame and MetaFrame

The Citrix WinFrame solution brings multi-user capability to the Windows NT environment in a similar manner to that which X-Windows provides for Unix. With

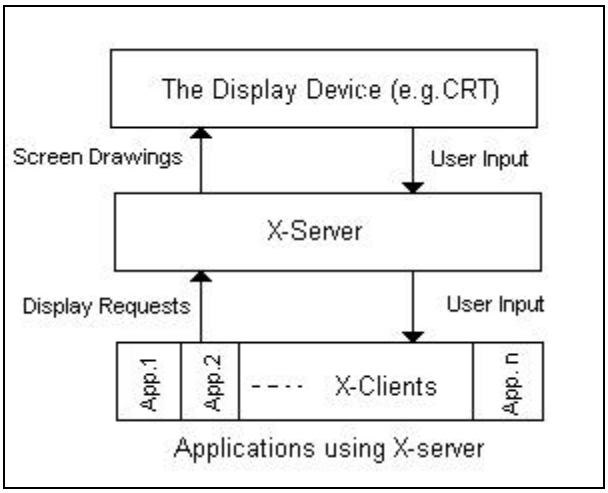


Figure 2-4: X-Windows Architecture

Citrix software, users can access Windows applications on any type of computer and any type of network connection. The applications are hosted at an NT server running Citrix WinFrame or MetaFrame. Citrix has created ICA (Independent Computing Architecture) that allows WinFrame clients to communicate with the server with reasonable performance. When an application is run, the Citrix server intercepts the application's user interface data (display, keyboard and mouse) and transmits this data between the Citrix server and the ICA Client program running on the user's desktop device using the ICA protocol [2].

This solution, being a software based one, and working on existing networks, lowers costs by providing a single software maintenance point like other thin-client solutions, and enables users to run existing Windows applications like a NetPC.

Different types of solutions have different pros and cons. This discussion focuses on Citrix-type software based solutions.

In the thin client architecture – depending on the implementation – all or nearly all of the processing takes place in the server. The thin-client can handle some operations if localization has been done. Localization will be discussed in a chapter of its own, where optimizations on thin clients are investigated. An enterprise-wide thin client solution will look something like Figure 2-5 courtesy of Citrix. All the systems in the enterprise are connected through various networks. Thin client production solutions enable the user to reach out to any type of architecture (UNIX, Windows) regardless of his client device and take the same service as he can through a fully functional client.

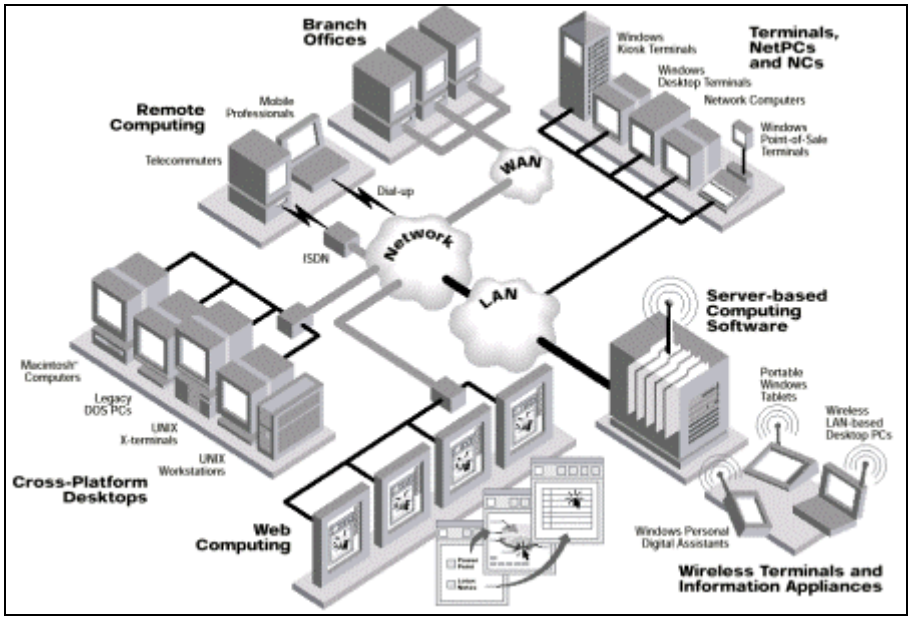


Figure 2-5: Enterprise wide thin client solution (courtesy of Citrix)[2]

The following table (Table 2-3) is a comparison of several computing models on several dimensions like application architectures used, and applications available [3].

Table 2-3 Comparison of computing models(2-3)

	THIN CLIENT COMPUTING	NETWORK COMPUTING	TRADITIONAL CLIENT/SERVER
Processing Model	100% Server Execution	Download and Execute	Local Execution
Hardware Footprint	Thin or Fat	Fat	Fat
Application Architecture	Monolithic, Component or 2- or 3-Tier Client/Server	Component	2- or 3-Tier Client/Server
Native Device	Variable or Fixed Function (PC, NPC, NC, WBT)	Variable Function (NC)	Variable Function (PC)
Native Application Type	UNIX, Windows or Java	Java	Windows

2.4 Thin Client Computing Companies

The thin client market is a large market. According to IDG reports, despite shrinking average selling prices, revenues grew 24%, from $155.2 million in the first half of 1998 to $192.3 million in the first six months of 1999 [8]. There are lots of companies providing thin client solutions. The companies can be grouped into two types: ones that produce the thin client computers and their accessories. These types of companies usually rely on one of the thin client software for connection and management. Most of the hardware companies also provide proprietary management software that helps deploy applications on their thin clients. To name a few, WYSE (www.wyse.com), Network Computing Devices (www.ncd.com) provide such software.

Software companies provide the basis for thin client devices to work. The software from these companies enable multiple concurrent thin client sessions on a single server, and they also have or use a protocol to take care of the communication between

the client and server. Thin client hardware producers usually rely on the protocol to develop their proprietary management software.

Here an overview of three companies Network Computing Devices (NCD), Citrix and Microsoft is provided NCD being the representative of both hardware and software company whereas Citrix and Microsoft are the leading software companies.

2.4.1 Network Computing Devices (NCD)

Founded in 1988 to provide thin client products, NCD has shipped more than 330,000 networked terminals and more than 300,000 copies of its PC-Xware thin client software for PCs throughout the world. The company's thin client desktop hardware and software for PCs enable customers to deliver applications from anywhere in their network to high-performance, easy-to-administer, low-cost desktops without the need for constant upgrading [16].

The company's three product families integrate to form the NCD Thin Client Architecture as in Figure 2-6.

NCD offers thin clients optimized for Windows, Linux and legacy systems access, and for UNIX and multimedia access. The PC-Xware line of thin client software for PCs, for accessing UNIX, Windows NT, and legacy applications over the network from PCs. WinCenter line of multi-user Windows NT server software which enables Windows applications to be used from thin clients, PCs (even ones running Windows 95 or Windows 3.x), UNIX workstations, X terminals, and Macintosh computers throughout corporate networks. It is the focus on connectivity to Windows NT and legacy systems

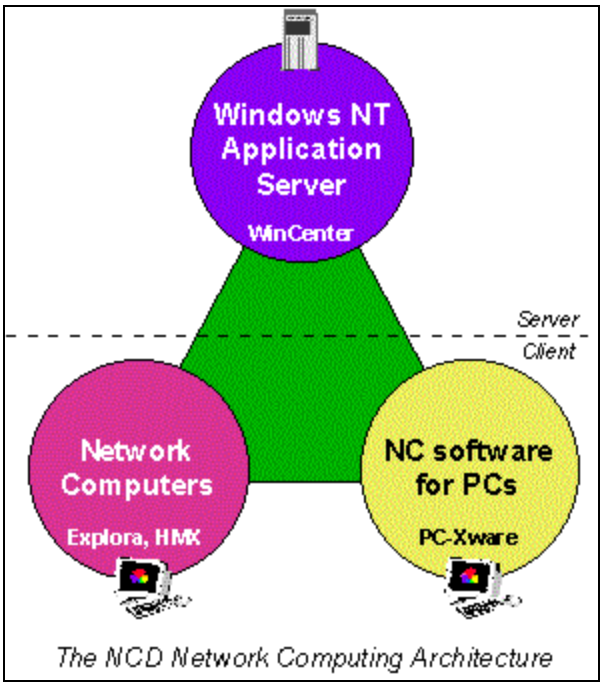


Figure 2-6: NCD thin client architecture[22]

that differentiates the thin client and PCs from older UNIX-oriented X-terminals and X-servers for PCs. Together, NCD's products help organizations provide a common, consistent set of applications on both new and existing desktops anywhere in the network [22].

The product portfolio of NCD provides an easy, cost-effective, and manageable way to deliver the power of the corporate network to the enterprise desktop as depicted in Figure 2-7.

2.4.2 Citrix

Citrix's main product is WinFrame. WinFrame is an integrated server-based computing software that provides access to virtually any Windows application, across

any type of network connection to any type of client based on innovative Independent Computing Architecture (ICA) protocol and MultiWin technology. WinFrame itself contains the operating system for the server and the client.

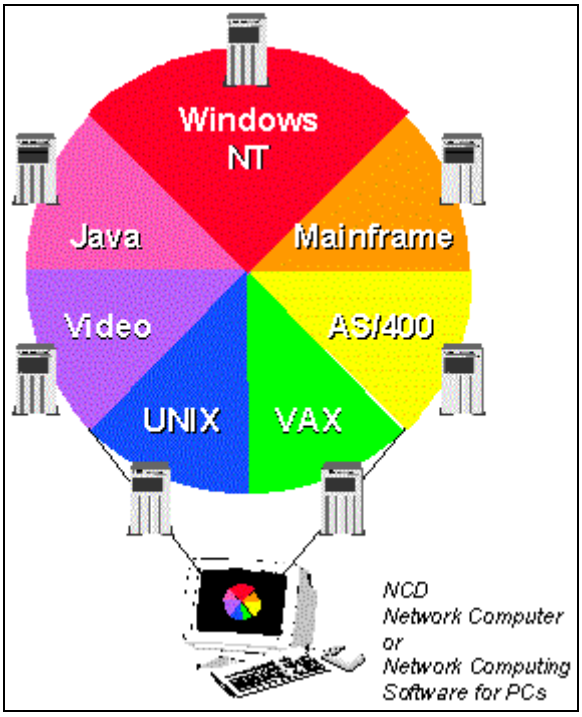


Figure 2-7: NCD architecture [22]

ICA technology provides the foundation for turning any client device – thin or fat– into the ultimate thin client. ICA has become a de facto industry standard for delivering corporate applications across the broadest variety of desktop platforms and networks. On the server, ICA has the unique ability to separate application logic from the user interface thus implementing the 3-tier approach for every application running on the server, even though the application is not developed in that fashion. On the client side, users see and work with the application's interface, but 100% of the application executes on the server. And with ICA, applications consume as little as one-tenth of their normal

network bandwidth. Dial-up connections with speeds of 28.8 or even 14.4 KBPS is sufficient to run a Citrix thin client [2].

MultiWin technology is licensed to Microsoft by Citrix to jointly create Terminal Server. It allows multiple, concurrent users to log on and run applications in separate, protected Windows sessions on the server. As a result, even a single-processor Pentium Pro/II/III server can meet the business-critical application needs of dozens of simultaneous users, with excellent performance.

Citrix's second flagship product is MetaFrame. It is server-based computing software for Microsoft's Windows NT Server 4.0, Terminal Server Edition. MetaFrame delivers a comprehensive server-based solution to the enterprise by extending Windows Terminal Server with additional client and server functionality -- including support for heterogeneous computing environments like UNIX, enterprise-scale management and seamless desktop integration. While Microsoft Terminal Server supports Windows-based devices and IP-based connections, MetaFrame delivers Windows-based application access to virtually all types of client hardware, operating platforms, network connections and LAN protocols [2]. As a result, organizations can keep their existing infrastructure, yet still deploy the most advanced, 32-bit Windows-based applications across the enterprise. The users can sit in front of an SGI IRIX machine and get the desktop of a Windows PC on the screen.

Citrix also offers management software and other performance increasing tools like load balancing software that directs the clients to the least loaded application server to increase speed and efficiency.

2.4.3 Microsoft

After licensing some of Citrix's technology Microsoft went into the thin-client market with NT Server Terminal Server Edition (TSE) (Figure 2-8). TSE (code named Hydra) enables windows-based terminals to access server processing power and other resources. TSE is made up of three components: the multi-user server core, the Remote Desktop Protocol (RDP) and the TSE client software. The multi-user server core provides the basic ability to host multiple, simultaneous client sessions and includes administration tools for management of both the server and the client sessions. RDP is a display protocol allowing communication between the server and the TSE client software. TSE allows connections to Windows 3.11, Windows 95/98 and Windows NT devices using the RDP protocol as well as Windows CE devices or WBTs. RDP and ICA will be discussed in the next section as the major thin client communication protocols.

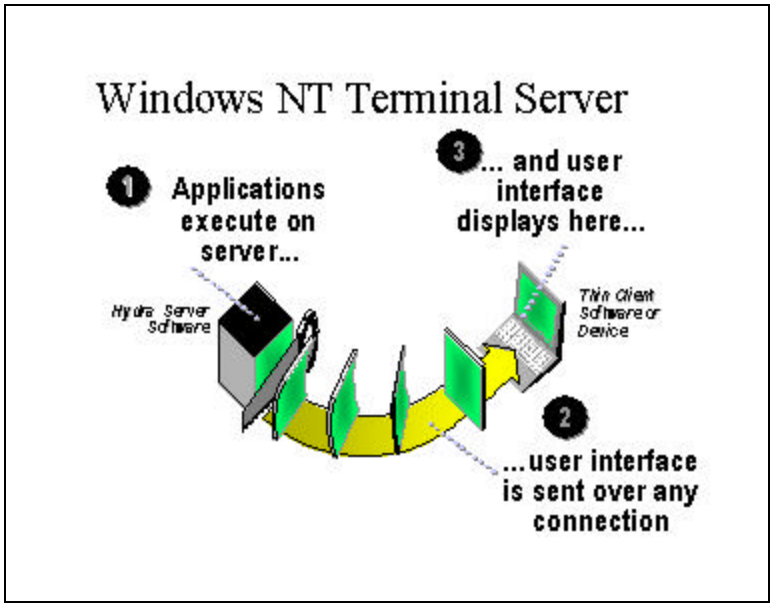


Figure 2-8: Windows NT Terminal Server Edition Architecture[12]

To achieve the multi-user capabilities required in Windows NT Server 4.0, Terminal Server Edition, components, services, and drivers have been added and or modified to the Windows NT 4.0 core operating system. Windows NT 4.0 components such as the Virtual Memory Manager and Object Manager have been modified to perform in a multi-user environment.

TSE only allows Microsoft Operating System based clients. To be able to connect from non-Windows platforms, users need to install Citrix MetaFrame on top of TSE.

With Windows 2000, terminal server is embedded a service. And users can start those services without requiring any additional versions or software installments, just like a Remote Access Server (RAS), or an Internet Information Server (IIS).

2.5 Major Thin Client Communication Protocols

Today there are two major protocols enabling the communication of the thin client to the server and vice versa. One is ICA from Citrix, and the second is RDP from Microsoft.

2.5.1 ICA

The ICA protocol sends only keystrokes, mouse clicks, screen updates and audio across the network. Applications consume just a fraction of the network bandwidth usually required by a regular client/server implementation. This efficiency enables the

latest, most powerful 32-bit applications to be accessed with exceptional performance from existing PCs, Windows-based terminals, network computers and new generation of business and personal information appliances.

On the server, ICA has the unique ability to separate application logic from the user interface. On the client users see and work with the application's interface, but 100% of the application executes on the server. And with ICA, applications consume as little as one-tenth of their normal network bandwidth. The key ICA differentiators are:

Thin resources: ICA requires the equivalent of an Intel 286 processor and access to a minimum of 640k of RAM to operate. This is dramatically thinner than X-terminals and proposed PC alternatives (e.g. Java PCs)[2].

Thin wires: The ICA protocol consumes an average 20kb of bandwidth. This allows it to operate consistently - even over dial-up and ISDN connections - without regard to the robustness of the executing application. However, the execution performance of "download and run" objects will be variable based upon a combination of network bandwidth and object size [2].

Universal application client: ICA works with any Win16 or Win32 application. This allows applications to be developed with off-the-shelf Windows tools and deployed with only one piece of ICA-based client software [2].

Platform Independence: ICA in inherently platform independent and has already been incorporated into UNIX, OS/2, Macintosh, and other non-DOS devices to deliver Windows applications to non-Windows and specialized ICA devices [2][20]**.**

There are several development partners using ICA to develop thin client solutions.

2.5.2 RDP

A key component of Terminal Server, this protocol allows a client to communicate with the Terminal Server over the network. This protocol is based on International Telecommunications Union's (ITU) T.120 protocol and it is a multi-channel protocol that allows for separate virtual channels for carrying serial device communication and presentation data from the server, as well as encrypted client mouse and keyboard data. It was first used in Microsoft Net Meeting conferencing tool [12].

One reason that Microsoft decided to implement the RDP for connectivity purposes within the Windows NT Server 4.0, Terminal Server Edition is that it provides a very extensible base from which to build many more capabilities onto. This is because RDP provides up to 64,000 separate channels for data transmission, as well as providing multi-point transmission (Figure 2-9) [12].

Current transmission activities are only using a single channel (for keyboard, mouse, and video -presentation data). RDP is also designed to support many different types of Network topologies (such as ISDN, POTS, and many LAN protocols such as IPX, NetBEUI, TCP/IP, and so forth.). The current version of RDP only runs over TCP/IP. Microsoft says other protocols will be supported in future versions.

With the new version of the Windows server operating system, Windows 2000, TSE will be an integrated service that can be enabled when the server is configured. Microsoft also announced that there would be new features added to TSE and RDP with the new release.

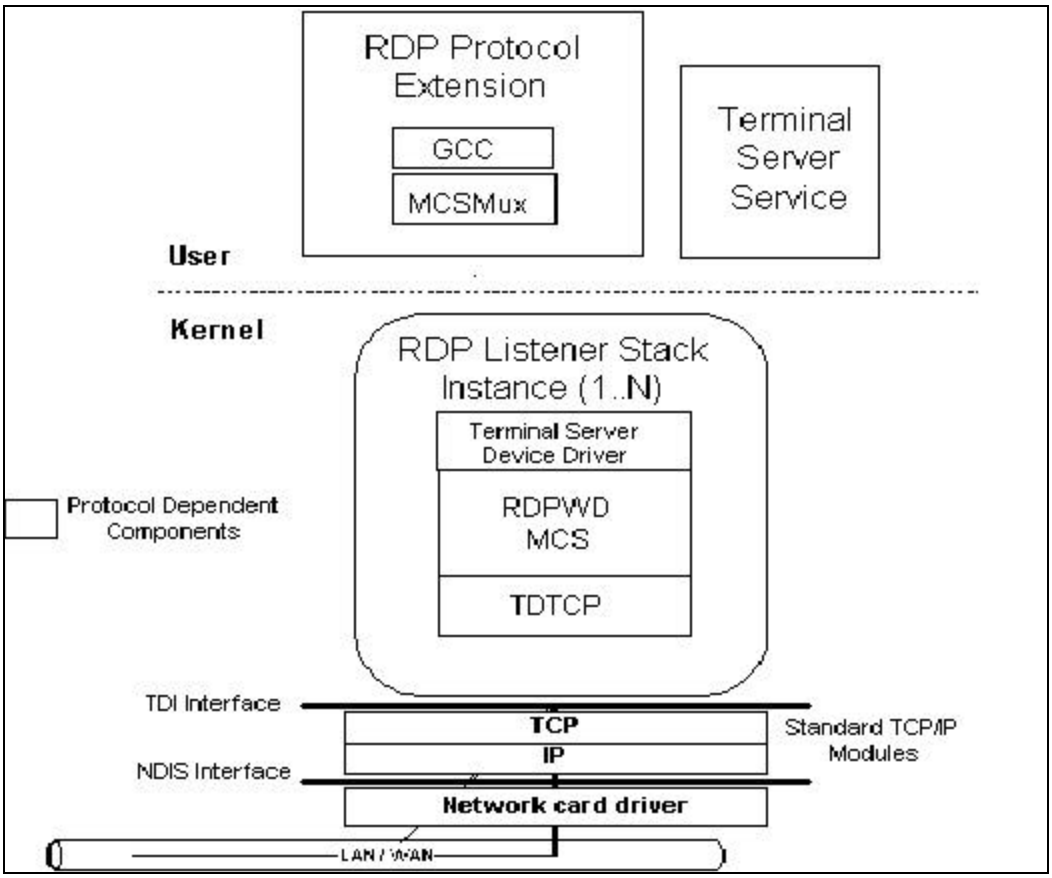


Figure 2-9: RDP Protocol[12]

2.5.3 Comparison of RDP and ICA

TSE is a turnkey product that includes client support for PCs and Windows-based Terminals that are running Microsoft's RDP display protocol. Installing Citrix's MetaFrame product on top of TSE greatly enhances TSE's capabilities. MetaFrame's enhancements include several features that organizations view as mandatory capabilities as opposed to unnecessary bells and whistles. Load balancing, heterogeneous client and protocol support, local device support and enhanced management capabilities are just a few of MetaFrame's capabilities that turn TSE into a true enterprise solution [20].

The main difference lies in the supported architectures. ICA allows UNIX and Macintoshes to connect to the server whereas RDP is only a Windows based protocol. Moreover, the client devices in ICA can be PCs, Palmtops, set-top-boxes and many other, compared to only PC accesses (Windows and Windows CE based devices) in RDP. Although Microsoft announced that more network protocols would be supported in the forthcoming versions, RDP for the time being only works on TCP/IP whereas ICA supports IPX, SPX and NetBEUI. RDP also does not support direct dial-up and serial connections, while ICA does.

Although ICA is more powerful in answering enterprise needs, RDP with TSE is a cheaper solution than Citrix thus affordable for small or middle-sized enterprises. The NT Server Enterprise Edition has all the tools bundled in it, reducing the cost of the system. Enhanced version is now a part of the Windows 2000, which will further decrease the costs.

2.6 Thin Client Benefits

With server-based computing, customers can increase productivity and develop a competitive advantage by gaining universal access to the business-critical applications they need to operate successfully, regardless of the connection, location, or operating systems they are using. Thin clients can be used in branch office computing, to provide manageable, secure application deployment and access over the corporate WAN [2]. Thin clients also enable applications to run on any platform or architecture even ones for which they were not built. This brings the idea of extending vital applications to newer

and lower cost devices. Thin clients can also enable users access to applications through dial-up or wireless connections without the security leaks and performance problems [5].

The following table (Table 2-4) illustrates the advantages and disadvantages of thin-client computing. Depending on the solution type chosen, the advantages and disadvantages might vary slightly (for instance if Network Computers (NCs) were chosen as thin client architecture to be employed, then the applications has to be coded in Java, or at least in a way that provides a Java interface). However, the common major drawback of all the solutions is that all thin clients need a network connection to be alive. The user loses the ability to work offline, although localization, which will be discussed in the following chapter, can carry on off-line processing to some extent.

Table 2-4: Advantages and disadvantages of thin clients[20]

ADVANTAGES OF THIN CLIENTS	DISADVANTAGES
Hefty Performance	Needs network connection. Not functional in disconnected environments. (Unless localization is done)
Easy Installation and Upgrades	
Simpler(Single point)Administration and Management	
Enhanced Security	
Reduced Costs of Ownership	
Flexibility to Ensure the Best and Most Reliable Fit In Any Environment (Any network connection, any client, any application)	
Universal application access	

2.6.1 Total Cost of Ownership (TCO)

The answer to popularity of thin clients in the enterprise world lies in the major concern of the businesses: money. Thin-client technology is designed to reduce the total cost of ownership (TCO) by lowering maintenance and administrative costs, decreasing LAN/WAN bandwidth requirements, and minimizing downtime. According to In-Stat, because full PC functionality is not necessary, thin-client computing can potentially replace from 50% to 70% of PCs now in place in corporations [1]. Thin-client technology eliminates the need to run applications on each employee desktop, which requires high system overhead and intensive IT support. Upgrading and maintaining these applications for each desktop result in additional cost. Because the applications are run and maintained centrally, thin-client drastically cuts the infrastructure expenses [4]. Total cost of ownership is estimated anywhere from $2500 to 3144$ annually for thin clients compared to an estimation of $5731 to $11900 for a PC, based on industry researches conducted by Gartner Group, IDC and Zona Research [8],[7],[24].

2.6.2 Security

On the security side, since the applications and data are centrally stored and controlled, the administrators have increased control over the system. Another point in security is that most thin clients will not have local drives, which disables the introduction of viruses to the system or walk of data through floppy drives. Thin client management tools usually provide methods to create user profiles, set groups of users and

give permissions based on single user or group. All of these permissions are managed from a single point, decreasing the probability of security leaks [20].

2.6.3 Any Client

Thin clients allow access to several platforms, extending the reach of Windows-based applications to virtually any client device including 286, 386, 486, and Pentium PCs, Windows-based terminals, Network Computers, wireless devices, and information appliances as well as X-Windows devices. They support all types of Windows clients, including Windows 3.X, Windows for Workgroups, Windows 95, Windows NT Workstation and Windows CE. Also non-Windows clients including DOS, Unix, OS/2 Warp, Mac OS and Java. Organizations can deliver the same set of applications to virtually any client device without rewriting a single line of code (on many thin client implementations), changing client hardware or adjusting client system configurations [2][20].

2.6.4 Any Network Connection

Users can connect to the system using the network through standard telephone lines, WAN links (T1, T3, 56kb, X.25), broadband connections (ISDN, Frame Relay, ATM) and wireless connections as well as over the Internet or corporate Intranets. Support for all LAN and WAN protocols, including TCP/IP, IPX, SPX, NetBEUI, and direct asynchronous connections are available in several thin client solutions.

This makes thin clients ideal for enterprises that need to extend applications to users everywhere regardless of connection type or available bandwidth. For companies with multiple networks and file servers, it's a convenient and efficient way to get enterprise-wide application deployment [2],[20].

2.6.5 Any Application

Users can access the full range of business and personal productivity applications including the latest Windows-based applications, client/server, mainframe and Java applications, from a client, regardless of available horsepower or operating system.

Organizations can reduce the total cost of application ownership by leveraging their existing technology investments. Users are able to access the most advanced business-critical and productivity applications from their preferred devices, eliminating the expensive training related to introduction of new systems and interfaces [2],[20].

2.7 The Future of Thin Client Computing

Many companies are turning to thin clients to lengthen desktop life and lower costs. Thousands of companies worldwide are using thin clients today to access their local and enterprise-wide networks. Ideal for users, who primarily require access to server-based applications, thin clients harness the power of the network and buffer the effects of change. When necessary, servers, software applications, data and other network resources can be managed and upgraded centrally without the need to disrupt users and

physically handle individual desktops. This allows the IT department to focus on managing the network and improving the effectiveness of the enterprise [20].

Shipments of thin-client devices reached 305,000 units in the first half of 1999, 83% more than in the first half of last year and only 63,450 units less than in all of 1998, according to a study by International Data Corp. (IDC). So the trend is up, and thin clients really produce the results they promise.

The major companies shaping the thin client market are Citrix and Microsoft. The thin client communication protocols provided by these companies are in a battle to be the standard of thin client communications. ICA having advantages in multi platform systems, RDP being a more affordable solution for small enterprises.

CHAPTER 3
MOBILE COMPUTING AND THIN CLIENTS

3.1 Characteristics of Mobile Computing

Mobile computing is characterized by several factors:

- Mobile devices are resource-poor: Although the performance of mobile devices such as laptops, PDA, and handheld computers continue to improve, they will always be resource-poor compared to their static counterparts such as desktops and workstations. Because mobile devices are designed with mobility in mind, they have constraints on weight, power, size and ergonomics. They will lead to less processor power, less memory size, smaller disks, limited battery life and the like.
- Mobile Devices rely on limited battery life: Although the battery life keeps improving, mobile devices will always try to make the most out of their batteries, which will impose constraints in device design. The mobile device has to be sensitive to power consumption and must have means of taking precautions on low battery situations. The software that is designed for mobile devices should also address that problem, making as less use of the battery as possible. Mobile devices' communications also consume a massive part of the whole battery consumption. The applications requiring extensive network

communication will not fit into mobile devices, because network communication is expensive in terms of power consumption [6].

- Network connection is expensive and highly variable in terms of performance: Mobile computers use a wireless communication channel. Although they might have wired connection to the network while they are stationary, mobility requires wireless communication, which is not reliable compared to fixed, static network services. Wireless communication has to deal with more noise, blocking and echoes, which in turn translate into lower bandwidth and high error rates. Wireless communication's second problem is the high latency. Latency, which is already inherent in wireless communications, can increase because of retransmissions, retransmission timeouts, error control protocol processing, and short disconnections. Although the bandwidth of wireless communications are increasing and many improvements are being proposed, the latency problem will be in the scene for a longer amount of time, because its caused by the environment of the communication, where it is most difficult to make improvements [17].

In the light of the constraints listed above, mobile computing applications have to have means to address those problems, or at least be aware and be immunized to those problems. An example can be an application that won't lose any data even in the case of a sudden power failure.

In mobile computing, there are three modes of operation depending on the status of the connection between the mobile device and the network. These are

1. Connected mode of operation: There is reliable and constant network connection, which makes the network services available all the time with guaranteed levels of quality of service. This might be the time when the mobile device is connected to the network through wired Ethernet connection.

2. Weakly connected mode of operation: The connection is not as reliable as it is in the connected mode in terms of bandwidth and availability. An example can be a wireless connection where there are disconnections for short amount of times (where the mobile device moves from one cell to another — deal hand-offs).

3. Disconnected mode of operation: There is no network connection or network service of any kind, the mobile device is in its own in this state.

Among these three types of operation, the last two are more likely to occur in a mobile environment, because wireless communications are susceptible to disconnections and poor connections.

3.2 Mobile Computing and Thin Clients

Mobile computing devices, having limited processing power and capabilities, form a suitable case for thin client architecture. As seen in the section 2, thin client architecture is modest in its demands on processing power and network connections. Even Personal digital assistants with a 14.4 KBPS connection can run thin client applications. That is because all the processing is done on the server and communication only carries inputs from client and displays from server.

Mobile computing characteristics do not fit into old client/server computing model. The client/server architecture assumes that the client is powerful enough to handle some part of the application logic. And mobile devices usually cannot run the operating systems for which these types of applications were written for. In the regular client/server architecture, the client requires services from the underlying operating system to accomplish complicated tasks [9]. However, mobile devices usually have less powerful operating systems which are more specialized in taking care of the mobile device instead of providing fancy tools to the application level.

On the other hand, “fat” clients makes the mobile device run out of resources in shorter time. Then the mobile device either turns itself off to prevent data loss, or decreases the clock speed or takes any other form of precaution which usually suspends the user from using the application any further.

3.3 Thin Client Improvements for Mobile Environments

As mentioned in chapter 2, the only disadvantage of the thin client architecture is, it is not functional in a disconnected environment. This also means that the thin client applications will suffer in handling short disconnections that are experienced in weakly connected mode. The client is purely dependent on the server to do the processing and return the results, however, there may be situations where the thin client’s local resources have the capacity to accomplish the same task.

This brings the idea of improvements in the thin client architecture to better fit into mobile computing domain. Since a completely independent client will solve all the problems related to poor connection status, the way to compensate them is to make the

thin client more autonomous. This way, the client will be object to less network problems with the sacrifice of using local processing power and battery power. This frame of improvements can be combined under the name localization.

3.4 Localization

Localization, in a sense, is a return to the features of old client/server model, "fattening" the client by imposing operations on it. However, the difference between the client/server architecture and a thin client with localization is that, the client is now a smart client, which can decide on the amount of localization that is necessary to handle the current environment. The environment here encapsulates all the participants of the system, the client, the client's processing power, battery life left, the state of the connection, the server, the server's workload.

Degrees of localization can vary from having simple keyboard localization to having the same application stored locally in the client.

Degrees of localization

The very basic localization can be localizing mouse movements in the thin client. If the application is not performing any computation with respect to the position of the mouse, but only responding to mouse clicks (which is generally the case in most legacy applications) then the client can simulate the mouse movements locally. Without any localization, the client has to send the server the mouse position data and then refresh the position of the mouse in its screen according to the response from the server. However, if

the response of the server is just to translate the mouse to its new position, the client can handle that operation without referring to the server.

The ultimate case of localization will be the case where the application that is being run as a thin client is also installed locally in the client. The client then performs thin-client to local copy switches, or vice versa depending on the environment conditions. If we call the thin client version of the application the “Image” and the local copy as the “Replica”, then the client would be able to switch back and forth between these two keeping the two at exactly the same state.

An example would be a text editing application. If the data file that is being edited is too big then the client uses the Image application, so that it does not waste the network by transferring the huge file. However, the editing operation usually includes typing, which in a thin client environment would require extensive communication between the client and the server. As the user presses a key, the key code is sent to the server, the server sends back the resulting display, which is just an echo of the character. This operation would consume a lot of network bandwidth especially if the user types fast, resulting in poor response time. The clever thin client then would use the replica of the application, check pointing with the server copy of the application at various times.

Localization can also be categorized into two with respect to its knowledge about the application that is being run.

Black box localization

Localization can be blind to the application, treating the application as a black box and assuming no help from it. That kind of localization is application independent, thus

works and gives satisfactory results with every type of application, be it a text editor, a legacy application, or a graphics program.

Localization with application support

On the other hand, the application, knowing that it is being run in a thin client environment, can present useful information to the thin client manager, which can in turn be used in localization to improve performance. In this case, the application, in a sense, out-sources some parts of its logic to the client from the server. In the text editing example given above, the text editor can give the thin client manager the file that is being edited, so that the thin client can decide to download that file from the server to the client to allow faster editing in the local copy (the replica application).

The second type of localization requires applications to be designed and developed with thin client support. This will require current applications to be rewritten or modified. Another problem is that, the level of support that is expected from the client can be difficult to estimate, and there should be a common way (a protocol) to describe those thin client interfaces to be able to achieve thin client independence (just like platform independence).

Localization can also be implemented to handle some part of the application whereas other parts of the application still runs in the thin client environment. An animated GIF in a browser application can be an example. Some web pages have several animated GIFs along with regular images and text. While the user views a page with an animated GIF, the image of the application has to be transferred to the client each time a GIF refreshes its display. This increases network traffic, thus creating a bottleneck for the

response time of the application. Localizing the animated GIFs, and handling the rest as a thin client can bring great benefits to the performance. The server (with or without help from the browser) can send these bitmaps to the client along with their position on the screen. The client will then simulate those GIFs and rely on the server for other parts of the display. The server and the client have to make sure that the GIFs in the real application and the locally simulated ones are 100% synchronized.

This thesis will focus on localization without any application support. So the optimizations apply to any kind of application running in a thin client environment. The next section will describe the type of localization this thesis will discuss.

CHAPTER 4
SENSE OF LOCALITY

An application gives the sense of locality through accomplishment of several properties. Once these properties are satisfied, the user will not know whether he/she is running an application locally or as a thin-client. These properties are the necessities to create the sense of locality in the mind of the user.

4.1 Application's Display States

An application can be described as a state machine, each state defined by a screen display of the application. The initial screen that pops up being the starting state and the screen from which the user terminates the application being the final screen. Given the states, the state transition arcs take the user from one screen to the other changing the display. In general there are two kinds of arcs. Self-activated arcs and user activated ones. User activated arcs change the state of the machine as a result of user input. Whenever the user presses a key or moves the mouse, if the application responds with a display change, then it is a new state and the arc connecting these two states is labeled with the user's input. A text-box (an interface component where user can enter text) finite state machine is described in Figure 4-1 through an example where the user types.

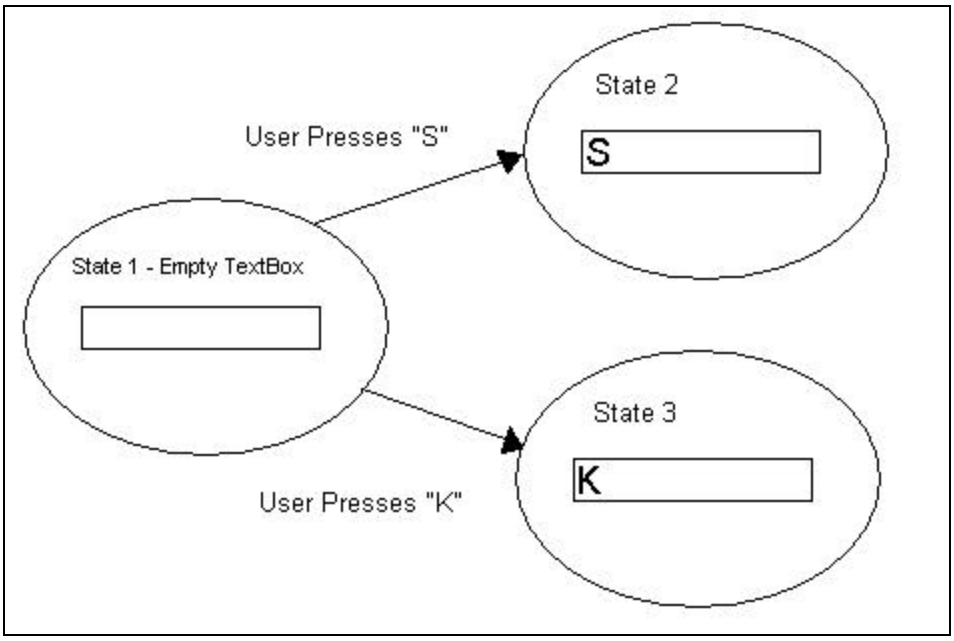


Figure 4-1: Finite state machine of typing action

Self-activated arcs are independent of the user input and will activate on a timely basis or randomly. An example can be a cursor blinking or an animated GIF. In the textbox example above, the cursor blinks while the program waits for user input. In its very basic definition, cursor blinking is achieved by drawing a vertical line in the textbox and deleting it after some time and repeating it all over. With the inclusion of cursor blinking, Figure 4-1 becomes Figure 4-2. The characteristic of the operating system, user preferences or the nature of the active component itself governs the state transitions of that kind. In Windows operating systems, the user can adjust the blinking rate. For animated GIF components, the time information is encoded in the object itself. So the image changes on a timely basis.

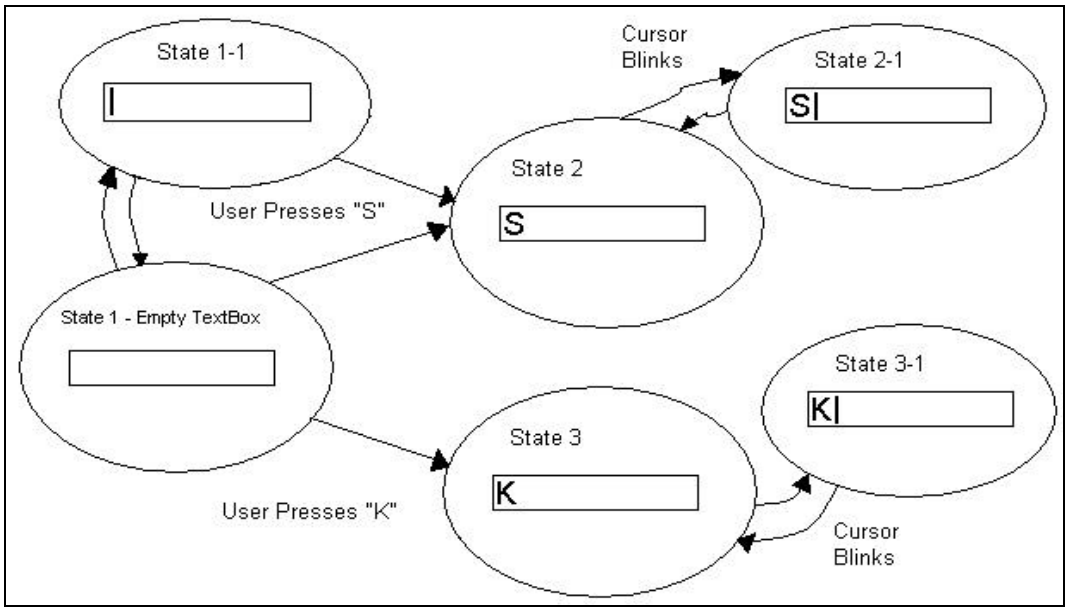


Figure 4-2: Finite state machine of typing and cursor blinking

So we can generalize the items that create the sense of locality under these two types of state transition arcs:

Active components: Active components are parts of the application that change the display of the application without any user interaction. They have self-changing displays and this operation has to occur in a timely manner to give the sense of locality. Examples can be a blinking cursor, an animated GIF, or a program that has a display showing the current time.

Active component display updates have to be done in a timely manner. Any fast display changes or static images will increase doubts about the locality of the application.

Interactivity: This topic includes the display changes produced as a response to a user interaction. Any kind of user interaction that changes the application's display state has to be mimicked the same way so that the user sees the result of his/her action.

The response time – that is the time between the user input and the application response– should be held in reasonable amounts to satisfy the user. Response time is especially lagging in ultra thin clients in mobile environments because of the latency of the wireless network.

4.2 Active Components in Thin Clients

In the thin client architecture, the image on the client has to be updated whenever the display on the server changes. Each update will consume processing power both in the client and the server plus the overhead of network transmission. Active components can be divided into two categories, with respect to their characteristics of displaying repetitive sequences.

4.2.1 Looping Active Components

Active components can be divided into self-repeating and non-repeating types. A display of real-time stock quotas will definitely be a non-repeating display change. An animated GIF on the other hand, is a self-repeating active component. Repeating active components have several states (displays) and they display those states one after another - creating a loop, thus the name. Animated GIFs and a blinking cursor are an example to those kind of active components. The states of a blinking cursor are shown in figure 4-3.

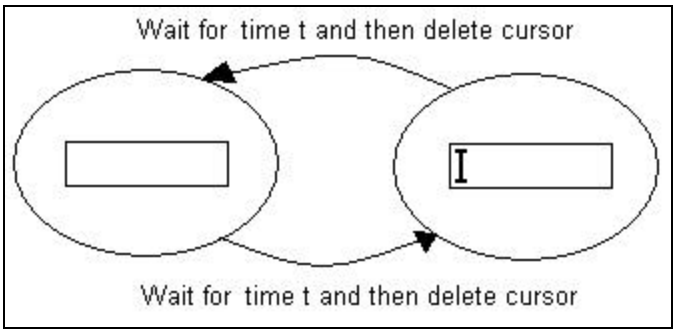


Figure 4-3: Finite state machine of a blinking cursor

The above loop goes on forever creating the display of a blinking cursor on the screen. This active component has two states and is going back and forth on those states. Figure 4-4 shows an animated GIF, with four states. The animation of an envelope bouncing back from the stack and opening is given by displaying the four states one after another in a sequence.

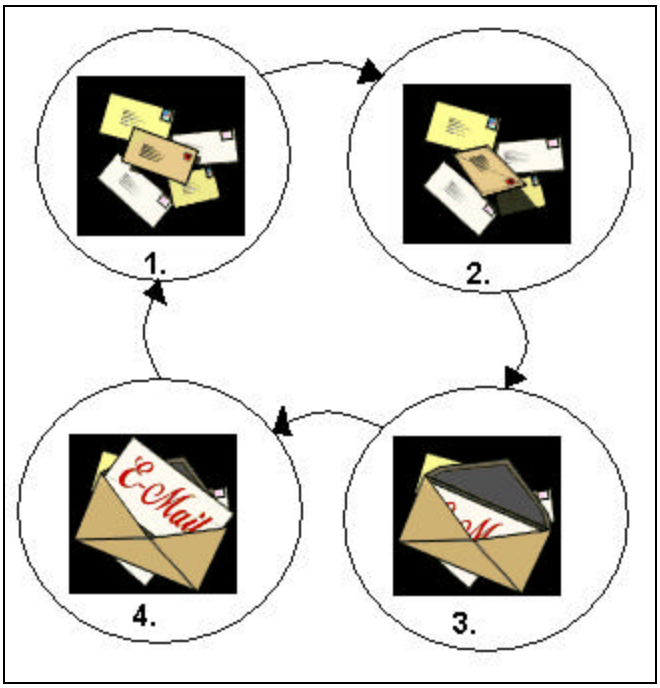


Figure 4-4: Finite state machine of an animated GIF

4.2.2 Non-Deterministically Repeating Components

In some applications, there are components that change their display randomly but have a few display states to choose among. This might be a commercial bar on a web site, which displays a random commercial every time the display changes. Figure 4.5 shows an example of such a bar's display states.

Figure 4-5: States of a commercial bar on a web page

The component in the above diagram has 4 images to choose from. After waiting for a predetermined time of 2 seconds it replaces the displaced ad with a randomly chosen one. Since the process is a random one, it differs from the case explained in the previous section (Looping Active Components). A possible state transition of the above component can be found in Figure 4-6.

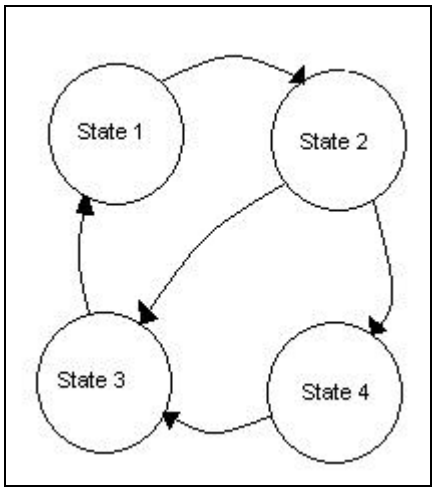


Figure 4-6: Finite state machine for the commercial bar

This thesis focuses on optimizations of active components of the above types. The next section will introduce the Ultra Thin Client Prototype developed to test and measure optimizations.

CHAPTER 5
THE THIN CLIENT PROTOTYPE

5.1 The Idea

A very basic thin client system is implemented from scratch, which gave the capability to test the ideas and quantify their results. The Ultra Thin Client Prototype (UTCP) is composed of two main modules: server and the client. Server is mainly responsible for managing the real copy of the application, sending display updates to the client and tunneling the events received from the client to the application. The client, on the other hand, manages the image of the application; it takes the display from the server and pastes it on the screen. The client manager also keeps track of user input and sends the necessary messages to the server.

Figure 5-1 gives the illustration of a thin client system.

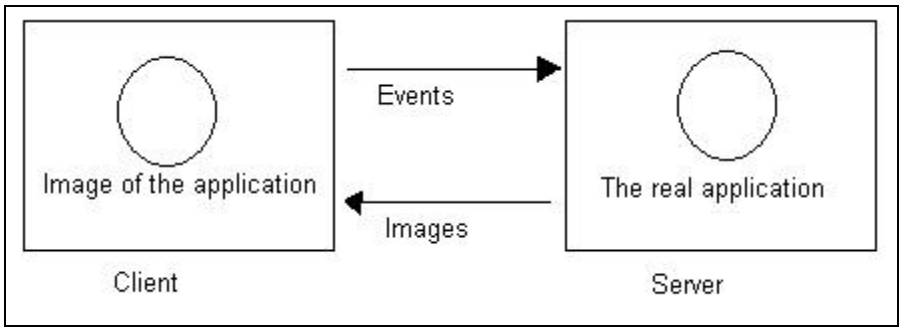


Figure 5-1: Thin client system architecture

The ideas in chapter 4 will be employed to give the user at the client that he/she is actually running the application locally, which is not true.

5.2 The Server Manager

The system is developed under Windows NT 4.0 Service Pack 4, using Visual Basic 6.0 Enterprise Edition, Service Pack 3. Visual Basic provided rapid development as well as a nifty interface and graphical output.

All the operations like the image processing and byte copy operations are implemented using Windows API calls, which provide access to operating system libraries and structures, memory chunks and faster execution. For example, for copying image bits from one place to another, Window's API function *CopyMemory* from *kernel32.dll* is used instead of copying bit by bit.

UTCP server is run on a Pentium II- 266 computer with 256MB of RAM and a screen resolution of 1024x768 Hi-color, and the client is a Pentium II - 233 laptop with 96MB of RAM and 800x600 hi-color screen resolution. The laptop has a wireless network card installed on it that supports 2Mbits/s communication.

The main job of the server manager is to manage the real application, pass the events received from the client to the application, and continuously send the images back to the client.

Once the server manager is run it checks its local IP, initializes port number 1001 and starts to listen for a client on that port. It also gets a list of running applications,

which will later be provided to the client, where client can either choose to start a new application or hookup to one of the already running applications.

UTCP server manager supports only a single client. Once the client connects to the server, it supplies its IP and machine name, which are immediately displaced on the screen of the server and the "Connected" light on the left bottom of the program turns to blue to indicate that a connection has been established successfully. Then the server sends the client a list of available applications. This list includes already running applications and a few other applications. After that the client sends a request either wanting the server to start a new application (e.g. calculator), or hookup to an already running application (e.g. a browser which was already running when the server component was first activated).

If the request is to start a brand new application, the server fires up the application and waits for a few seconds for the application to pop up. Then the server gets the handle to that application's window.

The Windows Operating system window handles

In Windows operating system, every object that is drawn on the screen is a window and has an handle –a 32bit (long) number – uniquely specifying that window. Having that handle, one can access that window and perform operations like getting the bitmap or painting some figures on it. Not only the main windows of applications but every text box and every button is a separate window and (generally) has a window handle associated with it. There is, however, a hierarchy among those windows. The

following screenshot (Figure 5-2) from a sample application shows the handles of each window on a sample application.

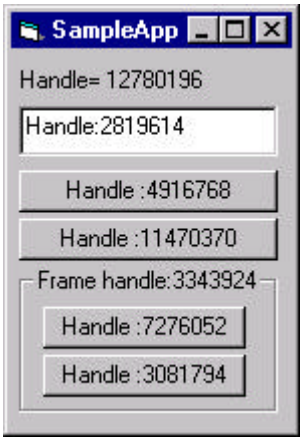


Figure 5-2: A sample application showing window handles

Every item on the application displays its own window handle. And the handle at the top is the handle of the window of the whole application. The hierarchy between the windows is presented below in Figure 5-3:

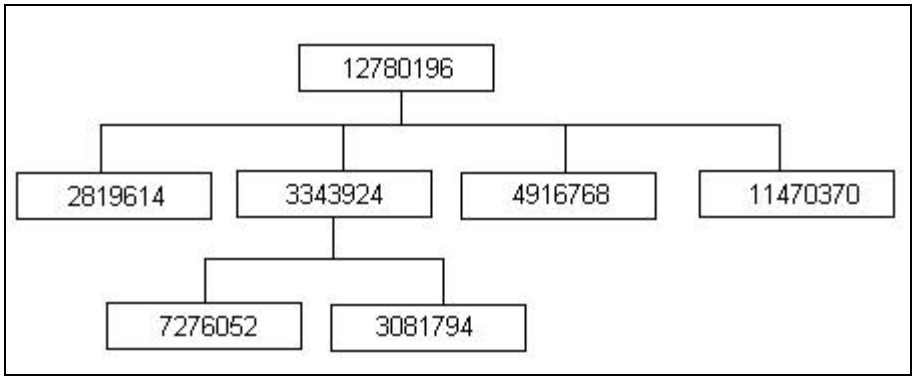


Figure 5-3: The hierarchy of windows in Figure 5-4

Every application's main window is a child under the desktop window. There are several ways one can get handles to application's windows.

One method is to get the handle to the desktop window by calling *GetDesktopWindow* from "*user32.dll*" which returns a long number, the handle to the desktop window. Afterwards, programmer can use *GetWindow* function from the same Dynamic Link Library (DLL) with the handle of the desktop and specify to get the children of that window. This function call, *GetWindow (hwndDesktop, GW_CHILD)*, returns the handle first child that is the application. Then the user can get other applications' handles by repetitively calling *GetWindow (hwndSibling, GW_ HWNDNEXT)*. Every call will return the handle to the next window that is a child of desktop.

Second method is to use *EnumWindows* function from *user32.dll*. This function enumerates and provides handles to all of the currently open top-level windows. It takes a single parameter that is the address of a callback function. Each time a window is located, the function passes that handle to that application-defined callback function. The function continues doing so until all windows have been enumerated, or until the process has been aborted. UTCP uses this second approach since it is recommended in Microsoft's home page and is more reliable and fast.

If the client wants to start a brand new application, then the server uses the *CreateProcess* function from *kernel32.dll*, which starts an application from its command line string, and returns the application's process ID, which is again a unique ID describing that application. *GetWindowThreadProcessId* function has to be called to get the handle to that new application's window.

Getting the display of a window in a bitmap

Once the server has the application's handle, now it can access the display and get the display of that window in a bitmap array. This process is somewhat lengthy and not much detail about the procedure will be covered here.

All the windows in Windows Operating systems are painted onto a device context, which is specially created for the current display adapter. So to store the bitmap of a window, the bitmap needs to be taken out of device context and saved into a device independent bitmap that can later be mapped onto another –but compatible – device context to get the same image.

The procedure, roughly, is to first create a new device context that is compatible with the one the system uses (by *CreateCompatibleDC*). Then create a bitmap handle that is compatible with the new device context (by *CreateCompatibleBitmap*), and associate the bitmap with the newly created device context (by *SelectObject*). Afterwards copy the display from the original device context to the new device context, which in turn fills in the bitmap (by *BilBlt*). Finally, one has to supply an array and use *GetDIBits* –DI stands for Device Independent – to get and save the device independent bitmap.

The structure used to hold the bitmap is a byte array in Visual Basic. Byte is a data type in Visual Basic that has a value space of integers from 0 to 255. The array is a single dimensional array although it can be viewed as a serialization of a 3D array. Since the tests are held in a hi-color (24-bit) resolution, a pixel is formed by 24-bits composed of 3 bytes each representing the red, green and blue intensities of that pixel (RGB values). The way the 1-D array is arranged is pixel by pixel, starting from the left top

corner of the image and traversing the image from left to right and top to bottom. Figure 5-4 gives a visualization of the byte array of the image.

The size of the byte array to hold the application display becomes

Width of the Application (in pixels) x Height of the Application (in pixels) x 24 bits.

So if the application is a full screen application on a 1024x768 screen then the size of the byte array will be 1024x768x24 = 18,874,368 bits, thus 2,359,296 bytes, making an image of about size 2.4 MB image. If the application occupies $1/4^{th}$ or $1/6^{th}$ of the screen then the image array's size will be 400 to 600 KB, which is extremely huge to transfer between the server and client especially in a wireless low-bandwidth and high latency environment. The solution is to compress the bitmap array so that it takes less space, but with the burden of compressing and decompressing. However processing power is more available and cheap than bandwidth.

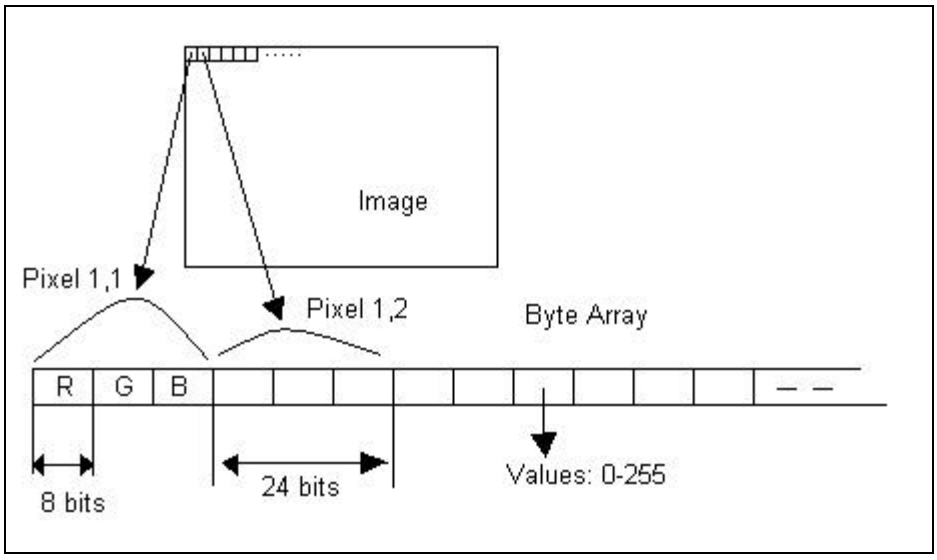


Figure 5-4: Format of the bitmap array to hold application's display

5.3 Compression of Images

Several compression algorithms were tested to compress the images to decide which one performs the best. The nature of the applications' display made the effect on choosing one of them. First have a look at the image characteristics for a few sample applications.

5.3.1 Image Characteristics

Figure 5-5 shows the distribution of values of the image array, the histogram of the bitmap from a sample Windows application. As seen in the chart, more than 90% of the values are accumulated among some values and the count of other values are negligible when compared with those highly popular values. (The figure shows the values for a sample application with image size of 430KB.)

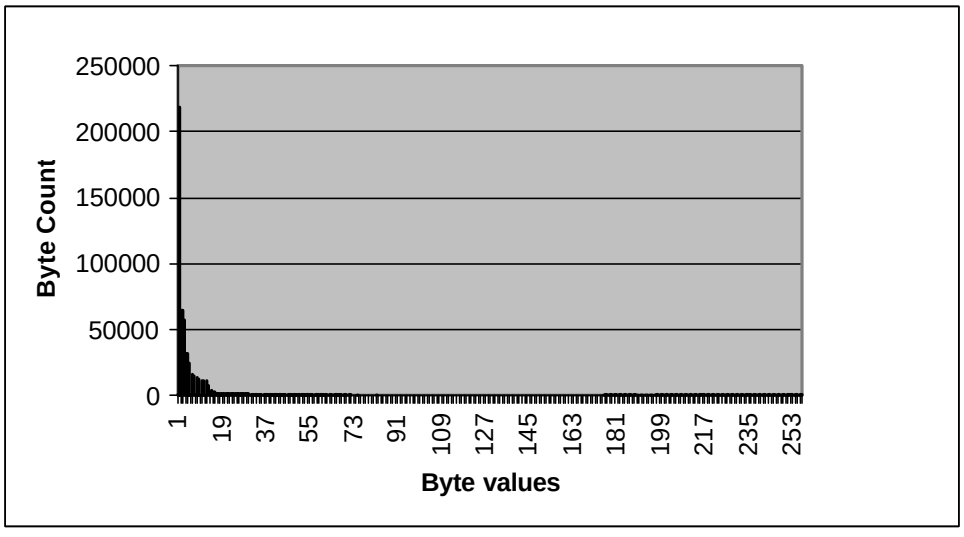


Figure 5-5: Byte value histogram of a sample application

5.3.2 Huffman Coding

This property of repeated values calls for Huffman Coding, which represents highly repeated values by values smaller than that is required if it was transmitted without compression. In this case, the most repeated byte would be replaced by a single bit, generating a saving of 7 bits per byte. However, after that 8th mostly used byte, the savings stop and every byte compressed takes more bits then its original.

Knowing that the array is formed by RGB values, Figure 5-6 shows the histogram of the mostly used 24-bit patterns (RGB values or pixel intensities) in a sample application.

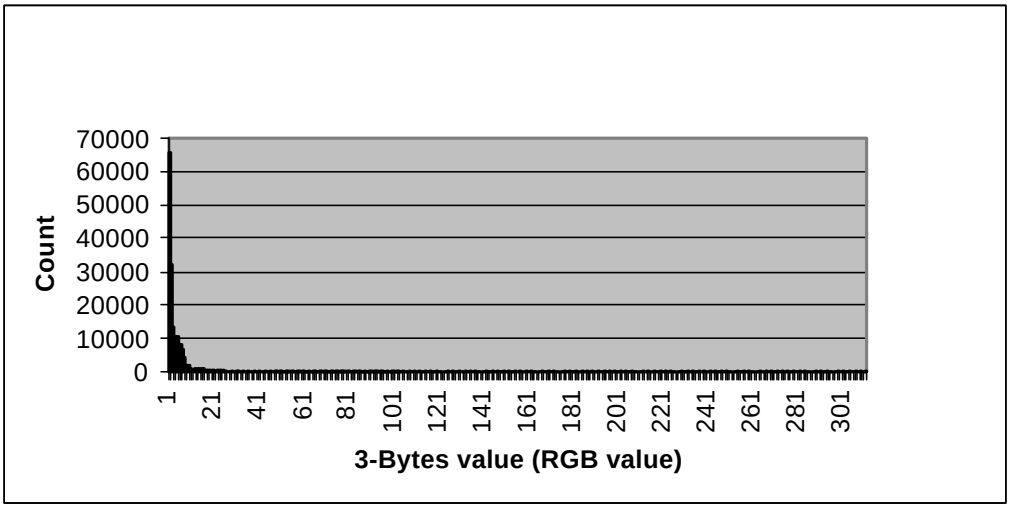


Figure 5-6: 24-bit value histogram

The Y-axis displays the rank of the mostly used 24-bit values. E.g. 1 represents 16777215 which passes 65527 times in the sample image. Calculating the savings using Huffman coding, the sample images were compressed to around 20% to 30% of their

initial size, which for a regular application which takes $1/5^{th}$ of the screen (~500KB) would take 100KBs. The results seemed satisfactory.

5.3.3 Run-Length Encoding

Another characteristic of an application's display is that it usually is not formed of quickly changing pixel values in close neighborhoods. Take a sample application – a word processor- that is pictured in Figure 5-7. As one can easily notice most of the display is formed of gray and white, and they are in a continuous basis. This property calls for the very most basic compression method: run length encoding. Experiments showed that in a full image, run length encoding performs about 10% worse than Huffman coding. That makes savings of 70% to 60% on the total image size.

5.3.4 Lempel – Zif – Welch (LZW) Compression

Finally, before making the decision the Lempel – Zif – Welch (LZW) compression algorithm was also run through several sample images. This algorithm was relatively harder to implement but produced results as good as Huffman coding, savings of 70% to 80%.

Before making a decision, one needs to determine the most common case of the images that will be transferred to the client. Once a screen capture is being transferred, there is no need to transfer the whole image once something in the display changes. Sending only the parts that have changed will suffice and produce great savings. That is the same idea under MPEG compression, which records the first frame as it is and then

records only the differences in the forthcoming screens. Since a change in the display of the whole screen is not expected often, most parts of the applications' display remain the same as the user interacts with it.

The choice of the compression scheme should depend on the assumption that most of the images being transferred will be difference images, which are produced by subtracting the new screen-capture from the previous one. This actually gives rise to different characteristics. A difference image will be white (or black) everywhere but the point of change. Figure 5-8 shows the difference image of the same application displayed in Figure 5-7, that is produced when the user presses Y. It is actually captured when the cursor was also on the screen.

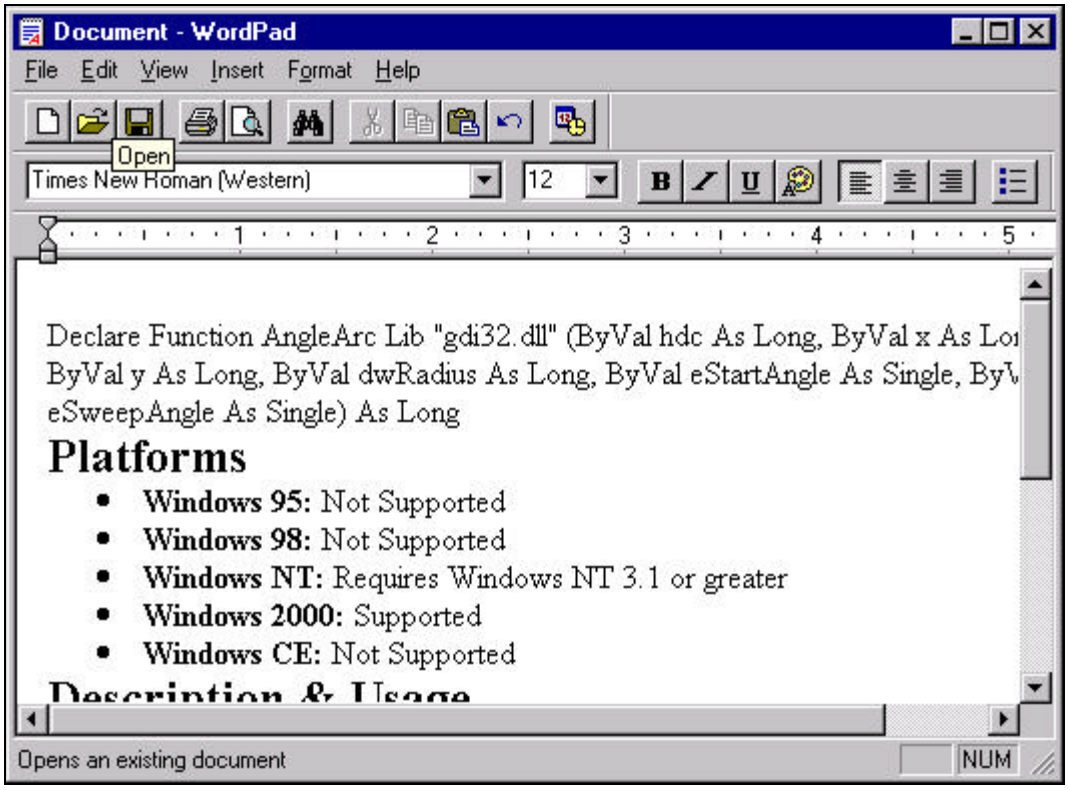


Figure 5-7: A sample text editor

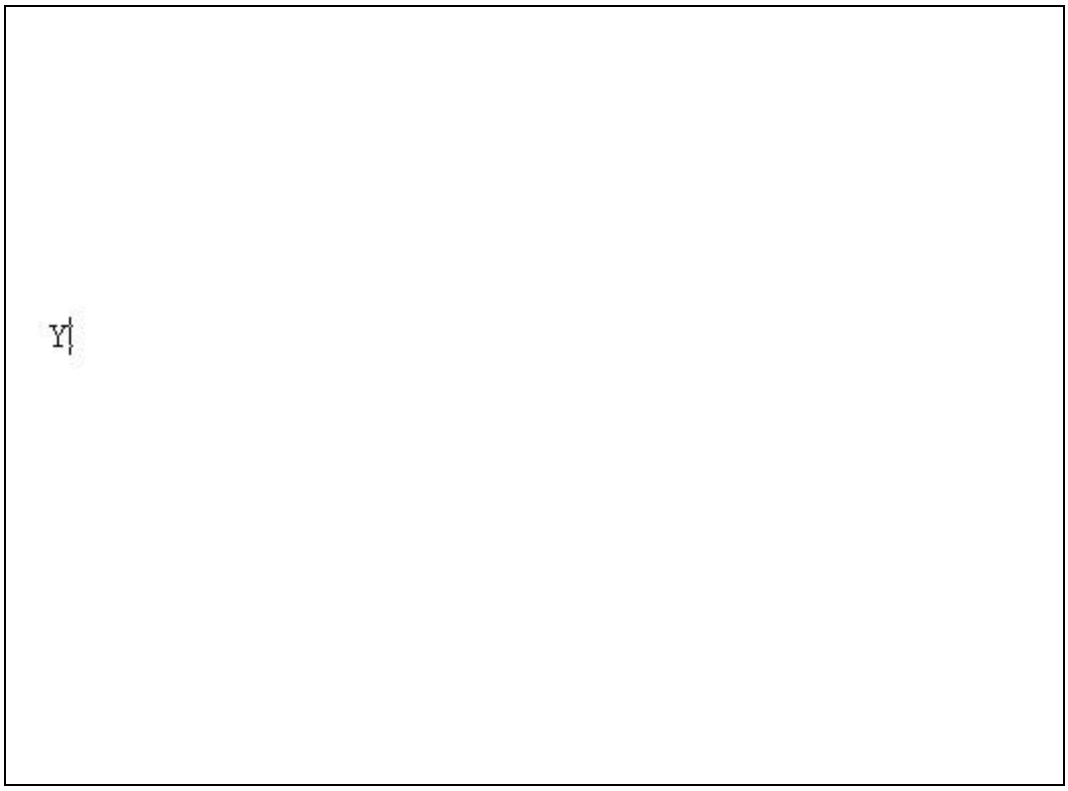


Figure 5-8: The difference image for the text editor

Considering this case, run length encoding produces the best result, compressing the white bits at the very beginning and end to the shortest possible length. Huffman coding could also perform well but server also needs to pass the Huffman Code table to the client for the client to decode the image. Moreover, decoding of Huffman Code compressed data is more computing power intense.

The main loop of the server is composed of checking if the application's display has changed, compressing the difference image and transferring to the client. The applications main algorithm’s outline is given below:

```
Previous Display = (Array of zeros)
Every N milliseconds {
        CurrentDisplay = Get Current Display
        DeltaImage = CurrentDisplay - PrevApplciationDisplay
                If DeltaImage<> 0  {
                        Compress DeltaImage using Run-length encoding
                        Transfer the compressed bitmap to the client
                }
}
```

5.4 Screen Sampling Rate

The algorithm captures the screen shots of the application every N milliseconds. The value of N depends on how much it takes process one screen shot. The timer control is used to create a clock tick every N millisecond. The timer control, however, is not very accurate. The reason is that Windows operating systems (versions 95,98,NT and 2000) are multithreaded and preemptive environments. The timer servicing threads are also subject to CPU scheduling, thus the period between timer thread executions may vary dramatically due to the number of threads with same or higher priority.

If the system executes each thread for 25 milliseconds (the quantum), and if the timer thread is the 5^{th} in the line of execution, then 100 milliseconds would have been passed before the timer even gets a chance to be executed, which would eventually lead to inaccuracies of undetermined time [13].

Moreover, the Windows Operating System Visual Basic environment generates 18 clock ticks per second — so even though the interval property is set in terms of milliseconds, the true precision of an interval is no more than one-eighteenth of a second [14].

On the other hand, the processing of an image takes time, which forms the minimum interval between two screen shots. Taking the difference, checking if the difference is empty and compressing it takes time. In the future sections, more operations will be done on the captured image to get performance improvements on behalf of increasing that interval processing time.

In an application whose screen-capture bitmap size is around 256 K bytes, the mean processing time is around 160 milliseconds. This means that the image capture interval can be at least 167 milliseconds, which means that the server component can miss some states of the display. Further improvements will just increase the processing time, as will be seen in the following chapters.

The following factors are the sources of inaccuracy of the timer in UTCP:

- The accuracy of the timer being used
- The minimum screen capture interval possible

The first problem is inherent in the operating system; thus it is impossible to get rid of that item. The second item can be addressed through making improvements in the processing algorithm. The program has been written with fast execution in mind, so most operations are accomplished by Windows API calls. However, Visual Basic language is a poor language when speed is considered. The code is not compiled but interpreted which increases the execution time. However, with the introduction of Visual Basic 6.0 Enterprise Edition, visual basic code can be compiled into native code, which enables faster execution. This option enables compiler to use loop optimizations and produce a faster executing code.

When the UTCP was compiled into pseudo-code (default type for interpreted Visual Basic), the average processing time was measured as 210 milliseconds. So an improvement of 25% is being achieved by compiling Visual Basic code into native code.

The problems related to the accuracy of timing will be discussed further in Chapter 6.

5.5 The Client Manager

The client manager program is responsible for decompressing the received image and displaying on the screen and collecting user input to send to the server.

When the program is run, the user can choose from a variety of thin client hosts to connect to. (The server manager has to be up and running on the remote host.) Once the server is selected user presses "connect" button to establish the connection. The "Connected" light at the left bottom of the application turns to blue if the connection is successful.

Just after the connection is established, the server supplies the possible applications and they show up in the "Available Applications" list. User then can double click on those applications. If the application is an already running one, the server grabs the display of the application at that time. If not the server fires the application and then gets the screenshot.

Before sending the image, the server sends the size of the window to the client, which in turn resizes its own thin client window. The client uses a border-less window to paste the display received from the server. This window has two special buttons on it,

which enable the server to move the local window or close the display. The empty window is shown in figure 5-9.

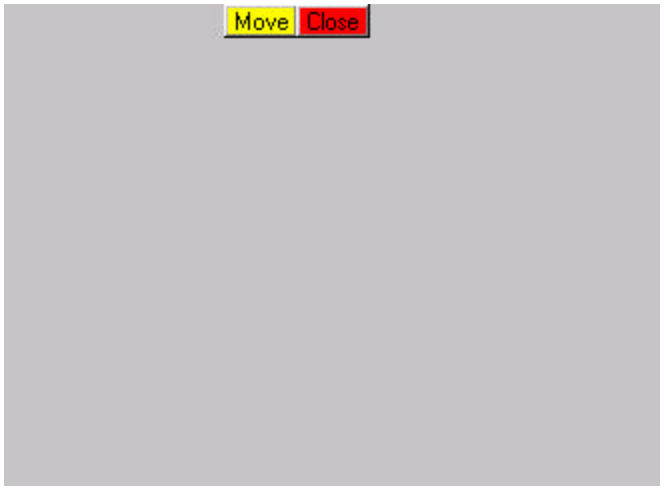


Figure 5-9: An empty thin client image.

5.5.1 Handling of Events

The client consistently monitors the user events related to the Thin Client window. This is accomplished via placing hooks to the operating system messages.

Windows operating system every operation takes place via sending messages to the windows or applications. When a user clicks the mouse, mouse click event is sent as an event to the window just under the cursor, which might be a text box, a label or a button. Each object receiving this message, checks if it has an operation related to that event. If the object has something to do with that event (a button has a function to perform an operation), the code that is related is executed. Then the event might be passed to the next window that is the parent of the current window or be killed at that

point without further forwarding. If the window does not have a function related to that event, then it just forwards the message to its parent.

For the sample application as in Figure 5-2, assume that the user clicked the mouse on the button that is at the bottom. Then a click event message is created by the operating system with the characteristics of the mouse click and it is forwarded to the buttons window, handle 3081794.

That window, continuously monitoring its event queue, finds the click event and checks if it has to process this message. If it has a related operation (e.g. displaying up a message box) it executes that code, and should or should not pass the message to its parent, which is the frame with handle 3343924 in this case. Buttons usually do not forward click events. But this depends on the application; it is the choice of the programmer to either pass the message on or not. Assuming the frame gets this message, and it doesn't have anything to do, it forwards the message to its parent, which is the main application window in that case.

Windows API provides means of access to the messages that travel between applications. *SetWindowLong* function from *user32.dll* gives the ability to substitute a window's property with another property. The programmer, once knowing the window handle can change the window's style (e.g. from regular to border-less), get the parent window's handle. It can also be used to replace the window's procedure that handles the messages. By the call

SetWindowLong (hWnd, GWL_WNDPROC, AddressOf MessageSpy)

After that call, the window calls *MessageSpy* procedure every time an event is added to its event queue. The *MessageSpy* (it can be of any name, programmer just needs

to pass the address of that function to the window) is a user-defined function whose address is passed to the window. The header of the MessageSpy is:

Function MessageSpy (ByVal hWnd As Long, ByVal Msg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long

Hwnd is the handle of the window that has received the message. *Msg* is the message that is to be acted upon and *lParam* and *rParam* contain additional information about the message. Inside the procedure the programmer, can forward the message up on its path by calling the *CallWindowProc*. This will transfer the messages to the parent window [12].

Inside the procedure, the programmer checks the *Msg* variable to get the message type. The messages that the UTCP is interested in are:

WM_KEYDOWN: A key is pressed

WM_KEYUP: The user released the key

WM_SYSKEYDOWN, WM_SYSKEYUP: Same as key down and key up, but for system buttons, like “ALT” or “F10” which have special interpretations.

WM_LBUTTONDOWN, *WM_MBUTTONDOWN*, and *WM_RBUTTONDOWN*: User pressed the left, middle or right button of the mouse.

WM_LBUTTONUP, *WM_MBUTTONUP*, and *WM_RBUTTONUP*: The user released the specified mouse button.

WM_LBUTTONDBLCLK, *WM_MBUTTONDBLCLK*, and *WM_RBUTTONDBLCLK*: The user double clicked that respective mouse button.

If the message intended for the thin client window is one of the types mentioned above, the message is sent to the server along with its parameters.

5.5.2 Server Processing of Messages

The server, constantly monitoring the display of the application, also listens to any kind of transmission from the client. Once a message is received, it is decoded and processed immediately (the types of messages and their formats will be discussed in the following section.)

There are two ways of sending events to an application. Either by using *SendMessage* which sends a message to the application's window, or by *SendInput* which places the event in the event queue of the system to be processed by the operating system. The second method is used in the UTCP because it is a single client system.

The operating system channels the user inputs received from the client to the real application. *SendInput* function from Windows API (*user32.dll*) synthesizes a series of keyboard, mouse, or other hardware inputs and adds them the input stream. And the operating system processes those events as if they were actually created by the user working on the local computer.

For the mouse clicks a position mapping must be performed to transfer the position of the click to the space of the server's display. That has to be done because the resolutions for the client and server are not the same (1024x768 and 800x600), and the relative position of the application's real display and copy in the client can be standing at two completely different places.

The Message Format

Each message that is transferred between the client and the server has a header and a data part. The header is composed of 10-bytes and contains information about the characteristic of the message, and the data part carries the data (e.g. the image). Figure 5-11 shows the decomposition of the header. And table 5-1 gives a listing of message types and what they are used for.

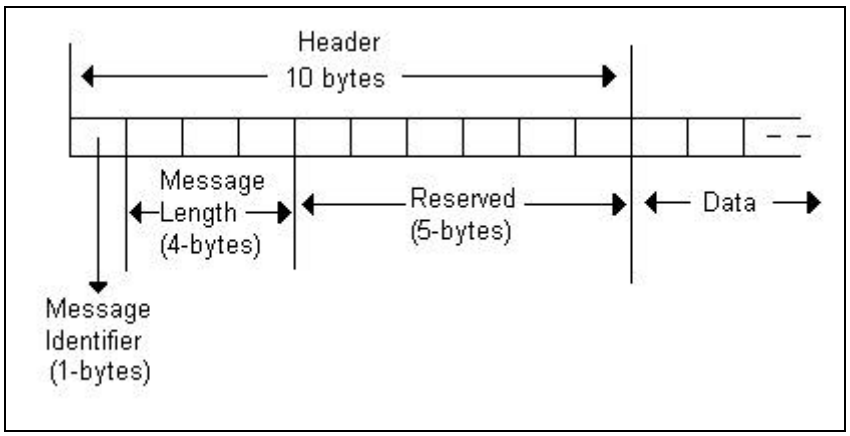


Figure 5-10: The format of the messages transferred between the client and server

5.6 A Sample Walkthrough

This section will demonstrate a sample session of the Ultra Thin Client Prototype with screen shots from the server and the client.

The server manager is started in the server, and the client manager in the client (Figure 5-11 and Figure 5-12).

Table 5-1: The Basic UTCP Protocol

MESSAGE IDENTIFIER	PATH	DESCRIPTION	SIZE
TRN_APPS	Server→Client	Server supplies the list of available applications to the client. The data part is a list of applications, new and running along with handles of the running ones.	Variable
TRN_RUN	Client→Server	The client tells the server to run the application whose name is specified in the data part of the message.	Variable
TRN_HOOK	Client→Server	Client requests to get the display of an already running application. The data part is a 4-byte long value, the handle to that applications window.	14 bytes
TRN_SIZE	Server→Client	Server, either after firing the new application or getting the handle of a running application, gets information about the size of its window. Then it transfers this size to the client. The data part contains the application width, height and the length of the bitmap to hold the display	22 bytes
TRN_READY	Client→Server	This message informs the server that the client is idle.	10 bytes
TRN_TERM	Client→Server Server→Client	These messages indicate that one of the peers has closed connection.	10 bytes
TRN_BPM	Server→Client	Server sends the client the compressed image. The size of the image is not the size of the bitmap transferred since it is compressed.	Variable
TRN_EVENT	Client→Server	Client sends an event to the server. The data part is composed of event's type and parameters of that event.	22 bytes

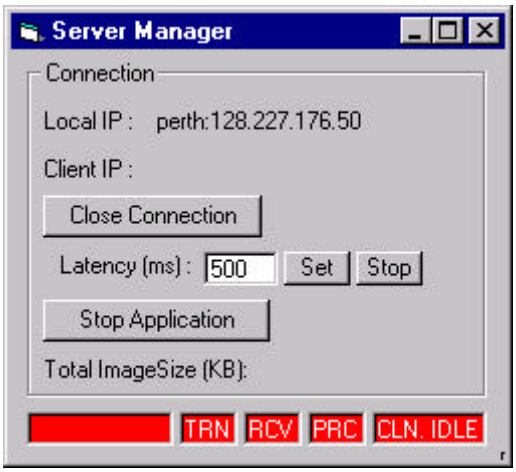


Figure 5-11: The server manager

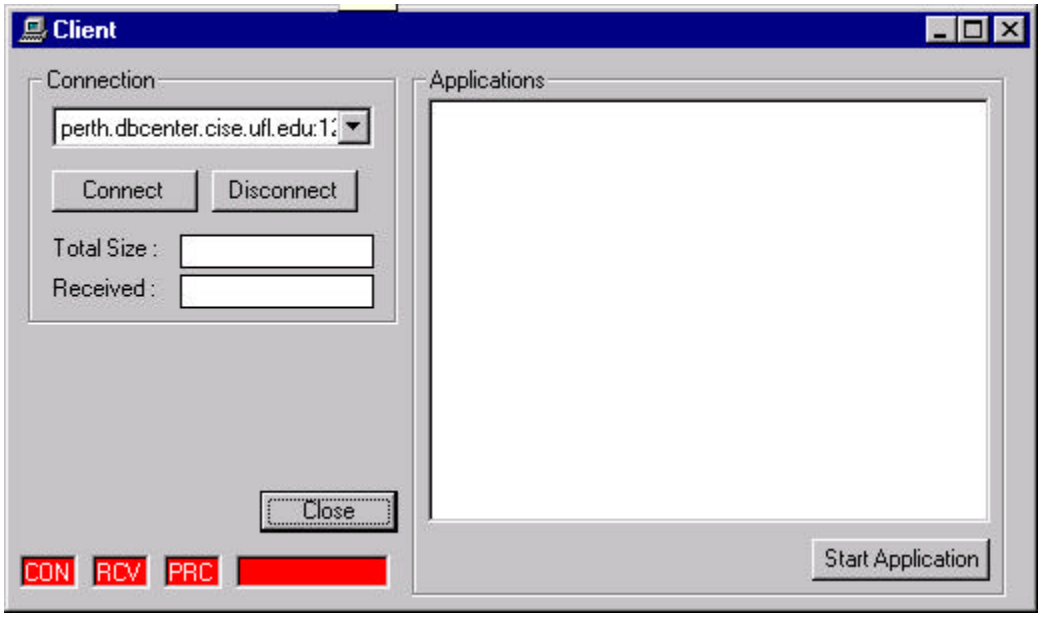


Figure 5-12: The client manager

Then the user presses the connect button at the client, and the server supplies the list of available applications which are displayed at the client's Available Application's list (Figure 5-13).

The user then selects one of the applications (Internet Explorer) and presses to Start Application button or double clicks on the selection. Then the server gets the image

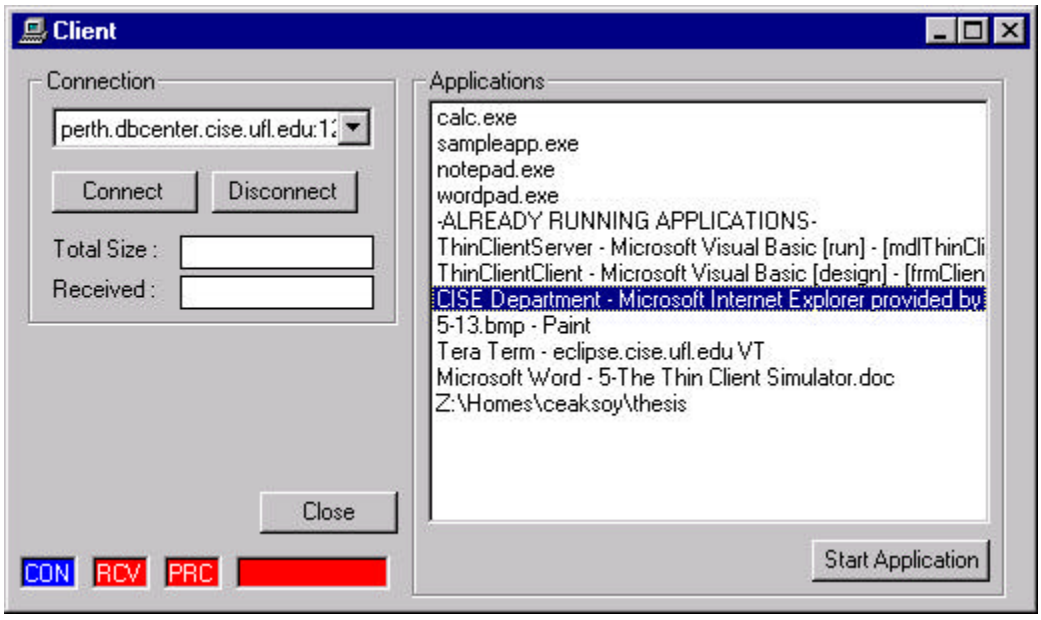


Figure 5-13: Client gets the list of available applications

of the application sends client the size information followed by the first snapshot of the Internet Explorer's window (viewing the home page of University of Florida CISE department) display. The client, then, displays that image on its local window. (Figure 5-14) The user at the client can move and close the local window by the help of the red Close and yellow Move buttons.

Assume that the user clicks on the File menu item in the menu bar. The client manager, capturing that click sends it to the server and the server channels it to the Internet Explorer, which in turn opens the pull-down menu. Right after that, the server manager senses the change of display, grabs the new display and takes the difference with the previous image sent. This results in the image that is seen in figure 5-15. Then this image is compressed and sent to the client; the client then decompress it and pastes over the previous image resulting in the illusion that the user opened the pull-down menu.

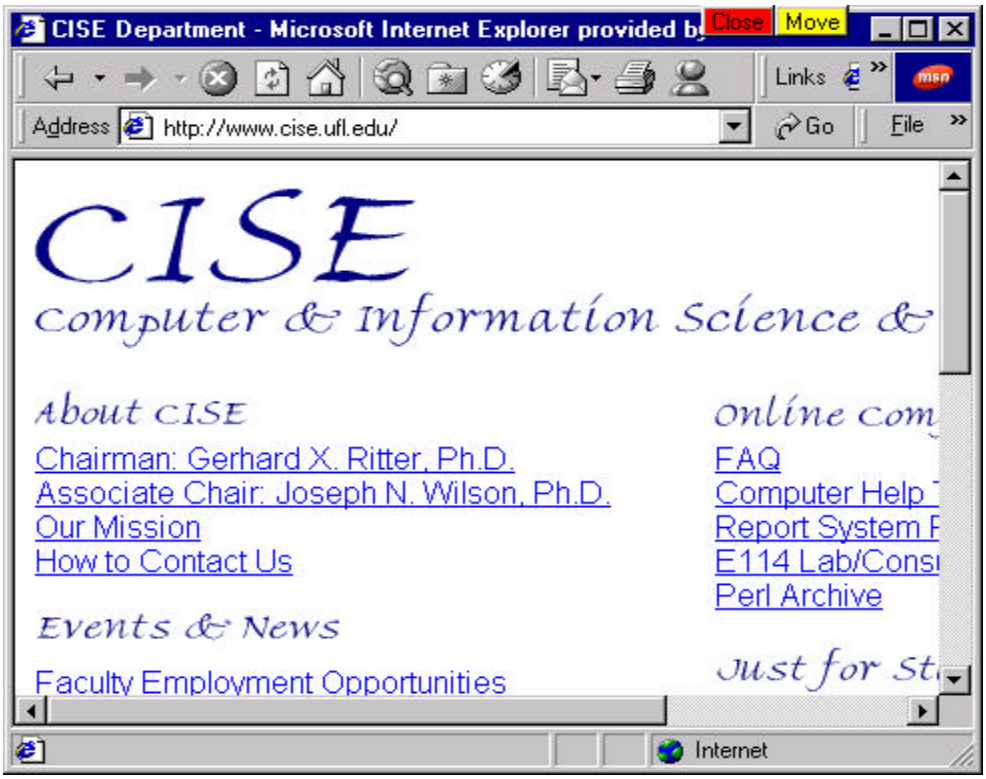


Figure 5-14: Thin client image of the browser

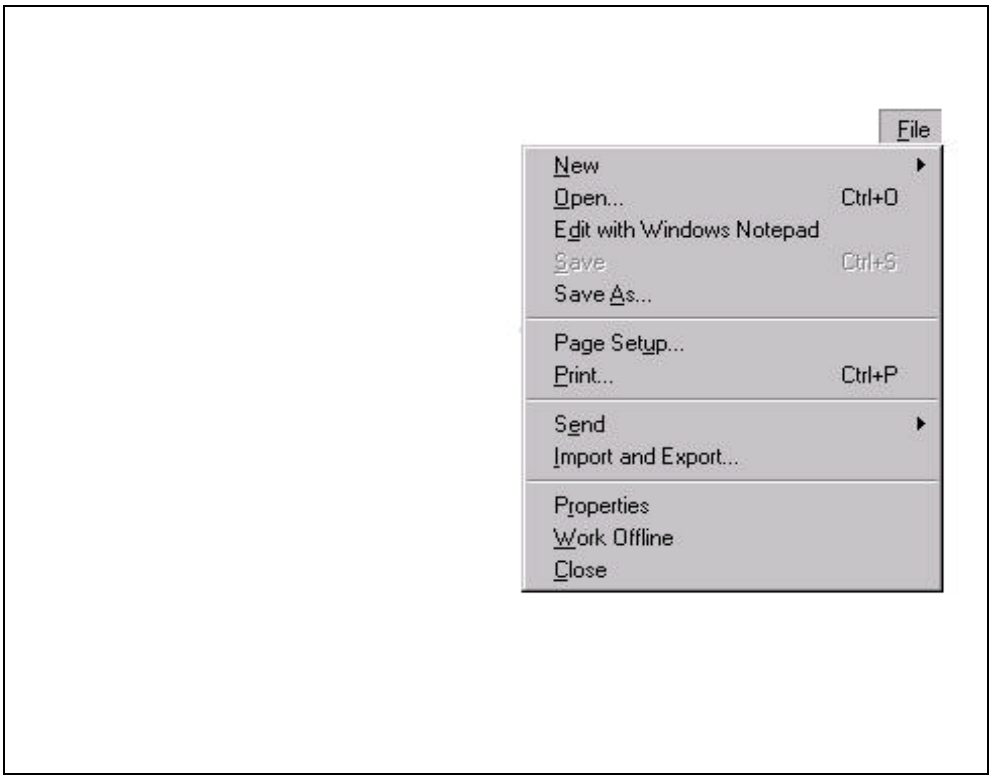


Figure 5-15: The difference image for the pull down menu

If the web page had an active component in it (e.g. an animated GIF in the browsed page) then the server were going to send the display to the client each time the GIF changed its display.

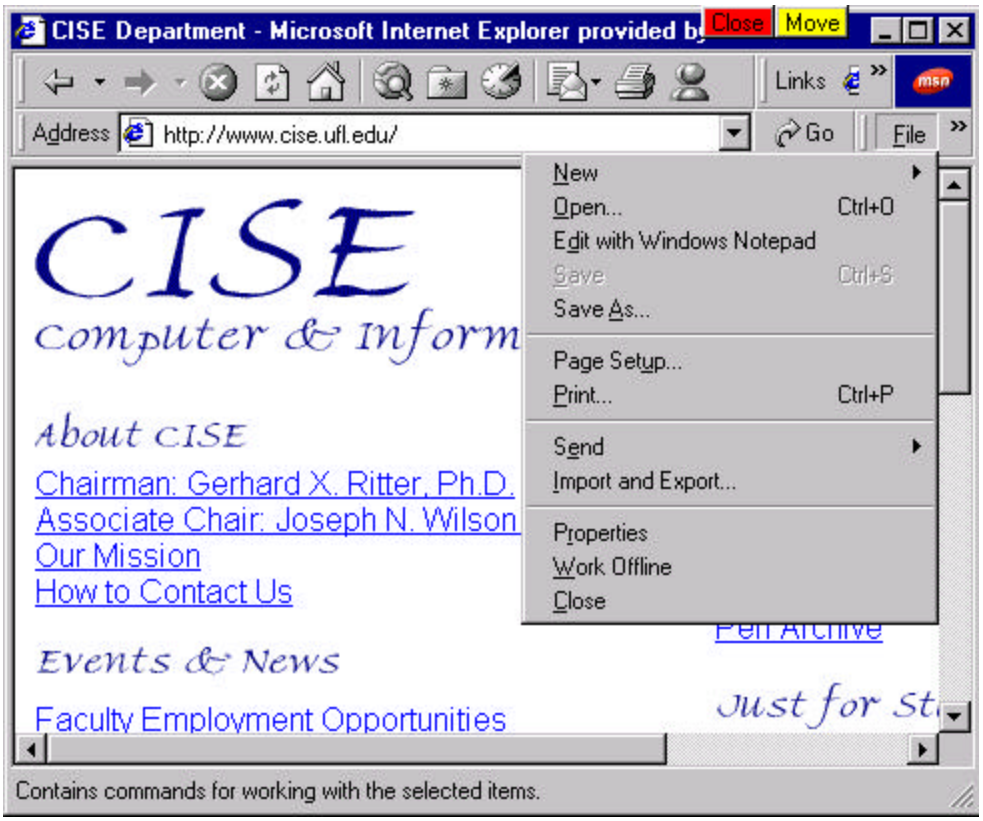


Figure 5-16: Final image of the thin client

CHAPTER 6
OPTIMIZATIONS

The example of a browser viewing a web page that contains a single animated GIF will be used throughout this section to demonstrate how the thin client prototype handles active components. The image of the application (the browser) is given in figure 6-1.

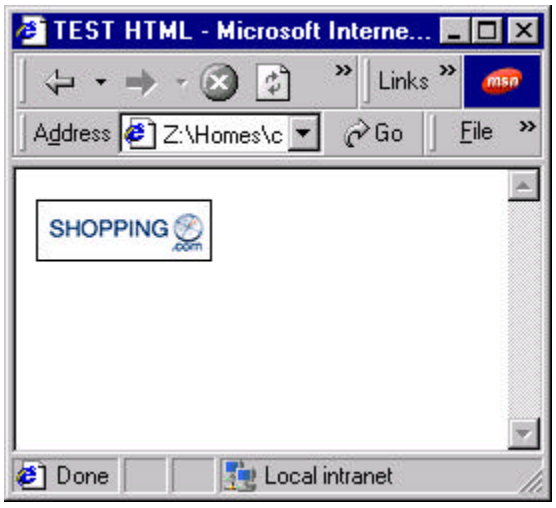


Figure 6-1: Sample browser application with a single animated GIF

The state diagram of the animated GIF is given in Figure 6-2. It is an animated GIF formed of two states.

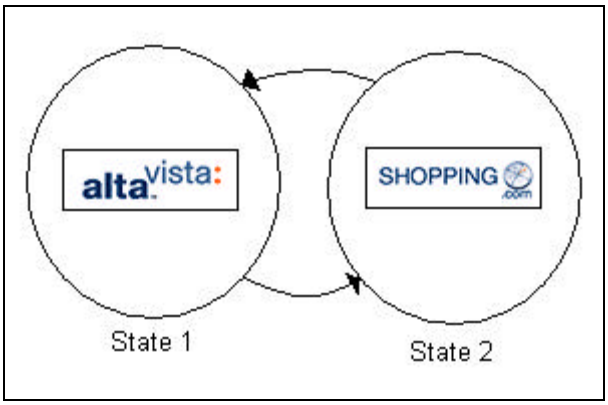


Figure 6-2: The state diagram for the sample animated GIF

6.1 Buffering

To detect the active components, the server has to keep a buffer of successive displays. The buffer, in the Thin Client Prototype is implemented as a linked list, using a dynamic array. Figure 6-3 describes the format of the buffer data structure.

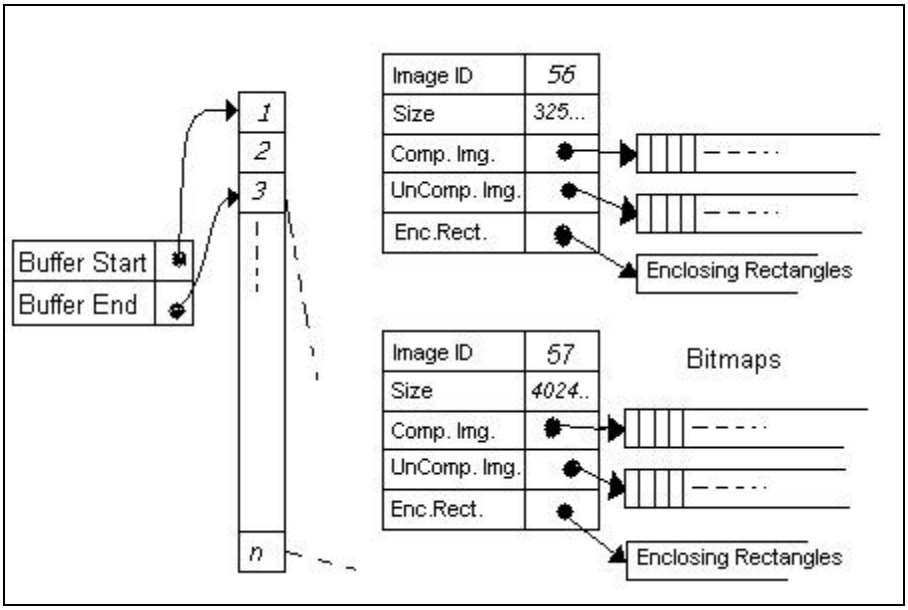


Figure 6-3: Buffer to detect repeating active components

The buffer size is determined by the user through the scroll bar located on the server and the client. This is done by selecting the maximum number of states the program should use to look over and catch a loop. If the user is looking for loops of *m* states, the program initializes a buffer of $n = 2m + 1$ states. So if there is a recurring pattern of *m* states, the program should be able to detect it whenever it repeats itself the second time.

The buffer is implemented as a FIFO queue. Buffer Start and Buffer End are pointers showing the start and end of the buffer, respectively. For the buffer in Figure 6-3, when image *n+1* arrives, the image is placed in the array at position 1, then Buffer End is updated to 1, and Buffer Start to 2.

Every buffer entry has the following fields:

- *Image ID*: A unique Id created for that image. Ids start from 1 and go up to 20000 and then start back at 1 again. Two bitmaps share the same id if they are the same. So before adding an image, it is checked again the images in the buffer to see if it has been added before.

- *Size*: The size of the compressed bitmap in bytes. This field is used to check the equivalence of bitmaps. One can tell two bitmaps are different without comparing the bits but the size of the compressed image. If the sizes are different, it is for sure that the bitmaps are different. If the sizes are equal, then a bit by bit comparison has to be made to check equivalence.

- *Compressed Image (Comp. Img.)*: Pointer to a byte array that holds the compressed screen shot.

- *Uncompressed Image (UnComp. Img.)*: Pointer to the uncompressed screen shot.

- *Enclosing Rectangles*: The purpose of this field will be explained in the next chapter. This field is used in active component extraction optimization.

When the server captures another screen shot, which is different from the previous one, it is added to the buffer. So no succeeding images in the buffer are the same. For an image to be added to the buffer, it has to be different from its predecessor.

The following algorithm is run through the last image before adding it to the buffers:

- The whole buffer is searched to see if the same bitmap is already stored in the buffer. This is implemented by comparing the size of the new comer with every item's size, and comparing the bitmaps if the sizes match.

- If a match is found, then the same *ImageID* is used, otherwise, a new unique id is created and used for the new image.

- A new node is then added to the buffer array and buffer pointers are updated.

The following figure (Figure 6-4) shows the difference image for the browser application viewing the 2-state animated GIF. This image is taken from the buffer, and shows black areas which mean that there are no changes in the bit values of those places. And non-black regions represent a change. (A buffer viewer, seen in the figure, is implemented to visualize the contents of the buffer)

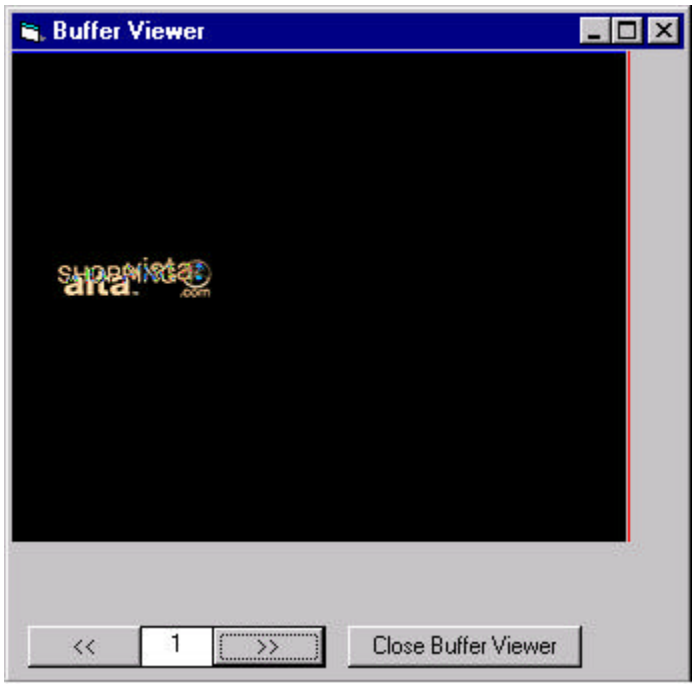


Figure 6-4: Buffer viewer

After several captures the images start to accumulate in the buffer. If nothing in the display of the browser changes (user doesn't interfere) the buffer will have the state in Figure 6-5.

The first image is the whole application display being transferred, which is huge in size (64K bytes). The rest are the images due to the GIF animation, which are smaller in size (having the characteristics of Figure 6-4, run length encoding performs a very impressive compression on the images). As seen in figure 6-5, the fourth image in the buffer has the same id as the second one. That is because they are essentially the same bitmaps. (The compressed bitmap and uncompressed bitmap portions of the array items are not shown in Figure 6-5.)

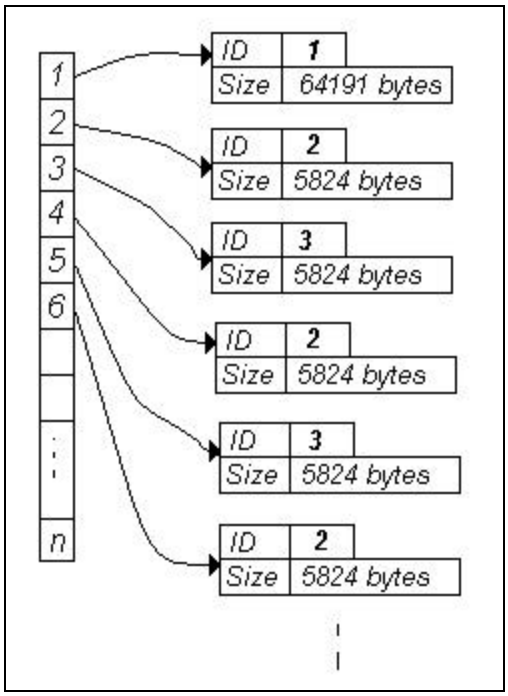


Figure 6-5: The state of the buffer with the 2-state animated GIF

6.2 Detecting Self-Repeating Active Components

Whenever a new image is added to the buffer, the *DetectLoop* algorithm runs through it to check if there is a recurring pattern in the images buffered so far. The algorithm is as follows:

```
k=2
BEGIN: Get last k images' ids;
Then in the buffered images, search for this pattern;
If this pattern is found then {
        Check if there are any more id's in between the last k and the found
pattern;
        If there is any then {
                k = k +1;
                Go to BEGIN;
        }
```

```
        Else {
                Found a recurring pattern composed of k states;
                Output the state and return;
        }
}
End
```

For the example pictured in Figure 6-5, when item 5 is added to the buffer, *DetectLoop* is run through the loop. First the last two ids are selected that is items at slots 4 and 5. This gives us “2 and 3” then the whole buffer is searched for “2 and 3” which returns us slot number 2. Then we check to see is there are any other stages in between the last and the one that has been just found. The number of the beginning slot for the last “2 and 3” is 4, whereas the ending slot for the found “2 and 3” is 3. Which means there is no other states in between and thus, “2 and 3” pattern has repeated itself for the second time. *DetectLoop* determines this and returns telling that a loop structure of two states has been discovered and the image ids are 2 and 3.

After detecting the loop, the server informs the client of the loop structure. The benefit is that: if a deterministic loop is going on in the application’s display, the client can buffer all those images and display them in a timely manner to simulate the real application. This will bring the communication between the server and the client to zero, unless the application’s display runs out of the recurring pattern.

The operations of the client side will be explained in the following section. Having detected the loop and informed the client, the server now monitors the application and makes sure that the display structure changes according to the loop structure it has discovered. It will continue to capture the image of the application on a timely basis (sampling) and add it to the buffer. But this time it does not look for new loops but

ensures that everything is going as expected. For the example in Figure 6-5, after detecting the loop when item 5 is added, the server takes the new image and checks if its id is 2. That condition has to be satisfied to remain in the loop. The image after that has to have the id 3, and the next must be a 2 again. A distraction out of this routine means that the application display has changed and the loop no longer exists. In such case the server informs the client that the application is out of loop and sends the new image. The client, being in the simulation mode (which will be described in the next section), wakes up and returns to the thin client mode, displaying the images that it receives.

6.3 Detecting Non-Deterministically Self-Repeating Active Components

Another type of active component is the non-deterministically self-repeating ones. The term, being explained in section 4-1-1, is used for situations where the application keeps displaying the same displays over and over again, but there isn't any way to extract a pattern of their order. The commercial bar in a web page displaying random ads chosen from a pool of ads stored somewhere in the server is an example. In that case, when the bar has shown each ad for at least a single time, the server will have all the image pool buffered. So when the next ad is displayed, the server can figure out which ad this one is. The benefits from here is that, if the client could also buffer the whole image pool, then the server instead of sending a bitmap every time, can send the id of the image, and the client can pop up that image from its buffer and display it.

The server, for detecting these kind of active components, keeps a counter. Whenever a new image arrives to be added to the buffer, the buffer is searched if the

image is already in the buffer. If this is the case, then the counter –initially 0 - is incremented by one. This counter holds the number of times a buffered image arrives. When a brand new image is added –one that has not been in the buffer – the counter is reset to 0. When the counter reaches a threshold (e.g. the buffer size), it means that the application has not introduced a new display for that amount of samplings. Then the server, if not in a deterministic loop, informs the client that it has detected a non-deterministically self-repeating pattern of N states. Then the client switches to that mode of operation. In that mode, the server just sends the ids of images to the client. That creates huge savings in the number of bytes transferred. In the meantime, the server monitors the new samplings and goes out of the loop when a brand new image arrives. It sends the image to the client, thus switching the client back to the thin client mode.

There can be times when the number of distinct states in an active component are smaller that the number of states in the loop. An example can be the animated GIF whose states are given in Figure 6-6.

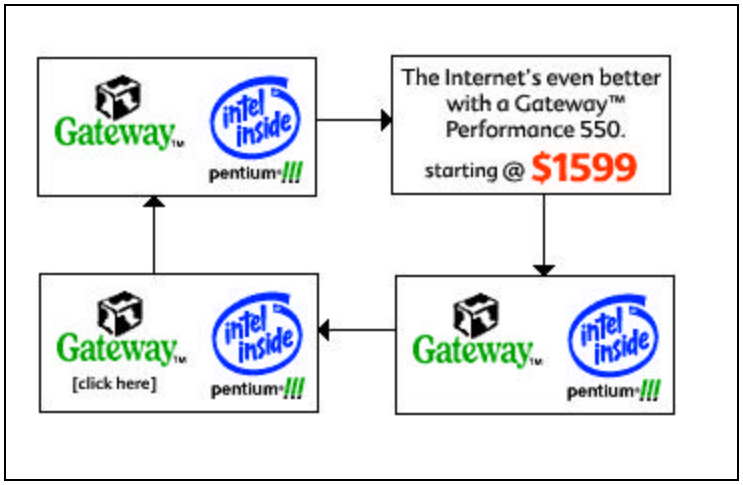


Figure 6-6: An animated GIF of 3 distinct states but 4 loop states

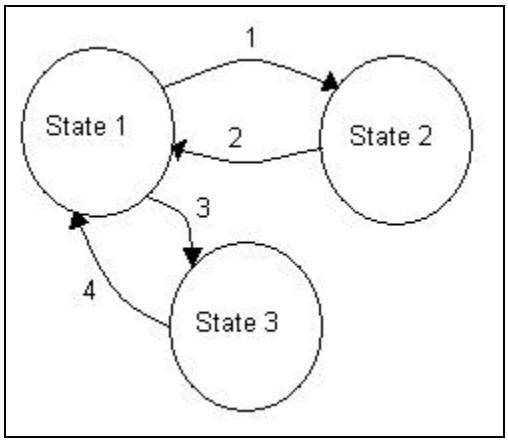


Figure 6-7: the state diagram for animated GIF in Figure 6-6

This GIF has 3 distinct states but the recurring pattern has 4 states. The corresponding state diagram is shown in figure 6-7.

The server in this case detects the pattern of 4 states, but informs the client that there are indeed three different states. The client then buffers only the distinct states. The client's operations are described in the following section.

6.4 The Client Operation Modes

We can describe several types of operation modes for the client depending on the source of the image it is displaying to the user. This classification is based solely on that criterion. The client is at all times connected to the server and passes the user input to the server to get it processed. The client never handles the user input on its own.

6.4.1 Dummy Thin Client Mode

In that mode the client is fully depending on the server to supply the bitmap that is to be shown. This is a regular case, where no active component optimizations have been done. The server, with each capture of the application display, sends the whole bitmap to the client and the client decompresses it and displays.

6.4.2 Active Component Self-Simulation Mode

The client enters this operation mode once the server detects a deterministically self-repeating active component, and informs the client of the number of distinct states. The client then opens a buffer of that size (not the number of states in the finite state diagram of the active component but the number of distinct states). It starts to buffer up the images sent by the server. The client also records the time intervals as it receives those images. For example, if the application was a browser viewing the animated GIF in Figure 6-6 only, and no user interference occurs, then the server, once detecting the self repeating active component informs the client that it has detected a loop of 4 states but distinct state count is three. The client, then, sets aside a buffer of 3, and starts waiting for images. The server sends the first image (state 1) along with its id (1 in this case). The client displays the image, buffers it and records the time (Time information will be used in simulating the active components locally). Then the server sends second image (State 2), which is also stored in clients buffer and the time is recorded. On the third step the server, knowing that the client has the first image (State 1) it just sends a notification saying “Display State 1”. The client again records the time and displays image one from

its buffer. Then the server sends state 3 as the last image and the client stores it in the buffer and records the time once again. Now that the client's buffer is full, the client tells the server that it is ready to simulate the active component locally. Receiving this message, the server passes over the loop structure in a string, which will be "1,2,1,3" describing the order of transformations in Figure 6-7. The client parses the string and starts simulating the active component locally. Using the timing information recorded, the client computes the time interval between the arrival of all images (call them t1, t2, t3, t4). Client, then displays the atate 1 form the buffer, waits for t1 milliseconds, then displays state 2 followed by a wait of t2 milliseconds. Then it again displays the first image (State1) for t3 milliseconds, and refreshes the display with state 4, showing it for t4 milliseconds and then going back to state1. While all this is happening, the server monitors the real application and makes sure that the application is still in the loop. If not, it sends the client "OUT_OF_LOOP" notice, breaking the active component self-simulation mode back into the dummy thin client mode.

6.4.3 Non-Deterministically Self-repeating Active Component Buffering Mode

In this mode, the server detects that the application's display has not introduced any new images for a long time and uses only the ones that can be stored in a buffer and referenced by its id. The server, detecting this sends the client the number of images to store in the buffer. The client, then, sets aside a buffer for that number of images and stores them with the ids the server sends. When the buffer is full, or the server knows that the client has the image in its buffer, the server will just send an id, referring to the id of the image. Then the client will go through its buffer, find the image with the matching id

and display it. This operation mode is not a zero-communication one as the previous mode but the number of transferred bits is a few bytes per image causing significant savings in communication.

6.5 State Explosion

When the application has more than a single active component, an increase in the number of states is observed due to the different frequencies of those active components. If a browser were viewing a web page with two active components in it, the number of total states to capture the whole display states depends on factors on the number of states each active component has and the frequency in which the active components repaint their states. An example web site is shown in Figure 6-8.

Figure 6-8: An example application with two active components

The animated GIF on the left-hand side is a 2-state animated GIF whose state diagram is given in Figure 6-2. The animated GIF on the right hand side is also composed of two states (the state diagram is given in Figure 6-9.)

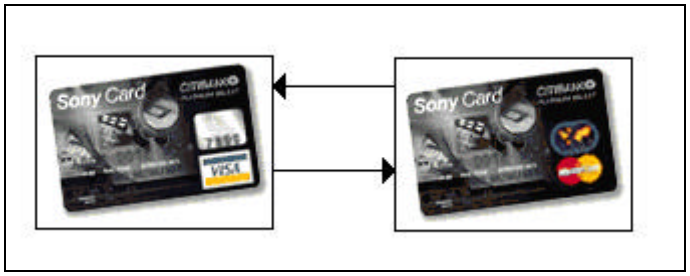


Figure 6-9: the state diagram of animated GIF 2

If the frequencies of the two images were the same (*t*) and they start displaying the first image at the same time, then they would have been completely synchronized. In such a case the two could have been treated as a single active component with 2 states and display change frequency of *t*. This situation is depicted in Figure 6-10.

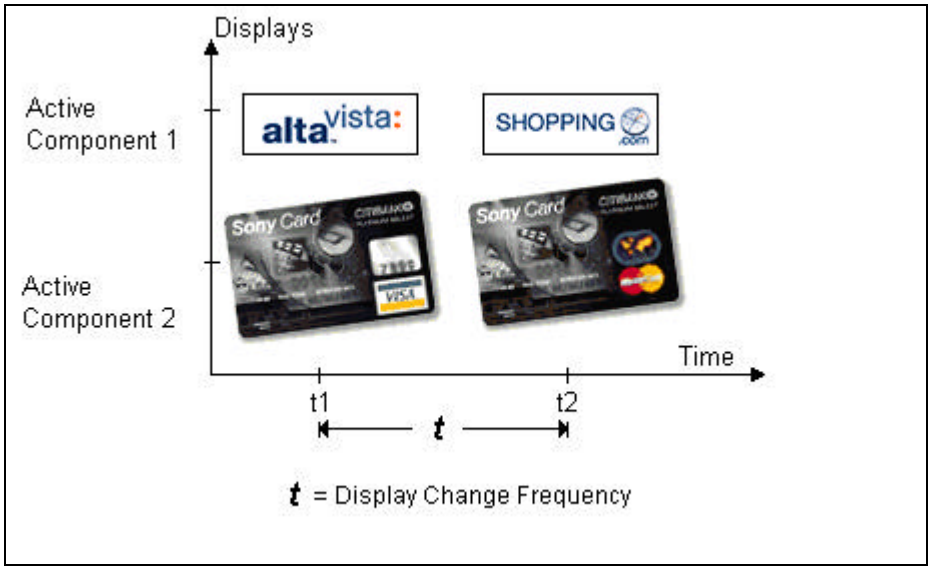


Figure 6-10: Two active components having identical display change frequency *t*

If Figure 6-10 was the case, the server could have found a repeating pattern composed of two states. However, that is not the case in real life. Different frequencies create state explosion, increasing the number of states to capture both active components as a single active component.

Assume, the image change frequencies of the two animated GIFs were, t_1 and t_2, where $t_2 = 2\ t_1$. This is also an unlikely case, where the frequencies overlay so that they coincide after a few iterations. Figure 6-11 depicts this case:

Figure 6-11: Two animated GIFs with frequencies $t_2 = 2t_1$

This frequency schedule constituted 4 states as expected.

6.6 Ghost Images

In a sequence of images having active components in it, the states are not only formed for displaying the active component, but also cleaning what is being displayed in the previous image. These images (or part of the images) will be called ghost images.

In Figure 6-11, the first image at time z_1, contains first state of both active components. But image (screen capture) at time z_2, has the following:

- Drawing for second state of active component one.
- Negative for first state of active component 1
- Negative for first state of active component 2

The last two are called ghost images. The real images are given in Figure 6-12. The images in Figure 6-12 are taken from the middle of the server buffer.

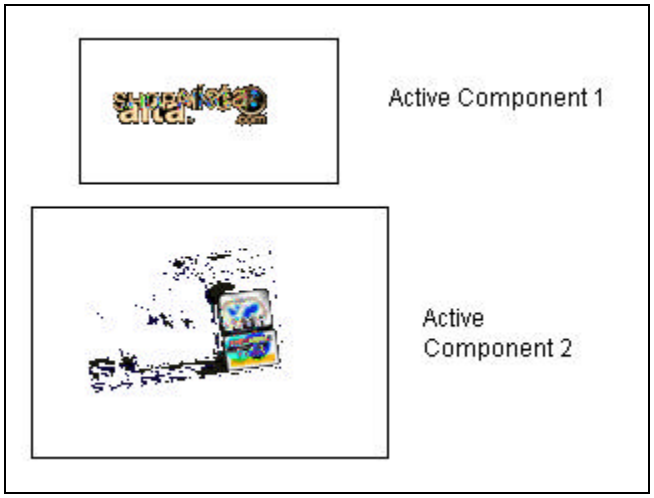


Figure 6-12: The real image states of active components in Figure 6-11

As seen in the image of active component 1, the image has both the parts of "AltaVista" image and the "Shopping" image. In the screen capture of active component 2, the bits for erasing the "VISA" display (at the lower left corner) are seen as changing bits.

So not only the images themselves but also their ghost images contribute to the state explosion. The following section will examine the inter-phase between active components, and describe its effects on the state explosion problem.

6.7 Active Component Phase Shift

Figure 6-11 relies on the assumption that both images start at the same time, displaying their first states. However, in real life, that is nearly impossible, because the drawing of two images on the screen depends on several factors such as the operating system, graphics hardware and the like.

Figure 6-13 shows a situation where active component 2 starts its cycles after image 1 has started.

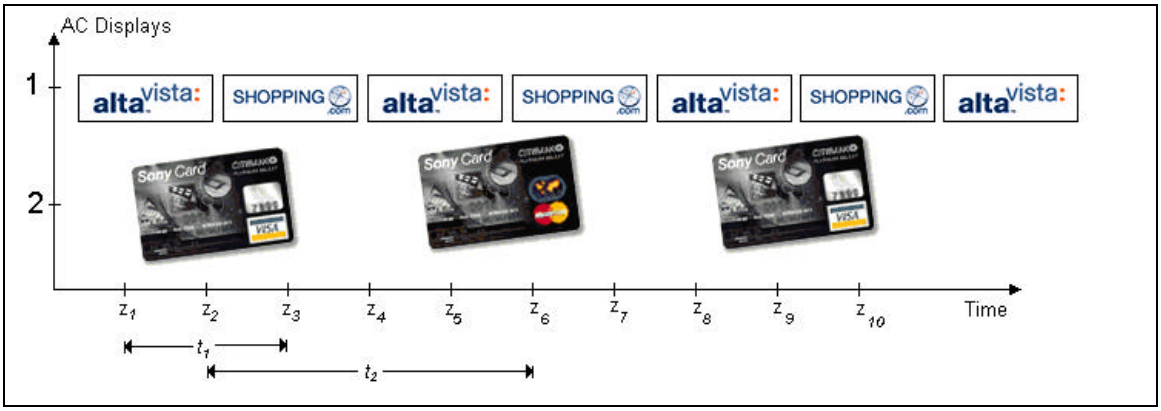


Figure 6-13: Effect of frequency shift in one of the active components on state explosion

A phase shift in one of the active components increases the number of states. Figure 6-13 has 8 states. If we name the state of active components as follows:

Table 6-1: Naming convention for active component images

IMAGE DESCRIPTION	IMAGE	GHOST
Active component 1, state 1:	$AC1_1$	$AC1_{1g}$
Active component 1, state 2:	$AC1_2$	$AC1_{2g}$
Active component 2, state 1:	$AC2_1$	$AC2_{1g}$
Active component 2, state 2:	$AC2_2$	$AC2_{2g}$

They can be numbered as follows:

Table 6-2: Image states for Figure 6-13

STATE	TIME	DISPLAY
1	z_1	$AC1_1$
2	z_2	$AC1_{1g} + AC2_1$
3	z_3	$AC1_2 + AC2_{1g}$
4	z_4	$AC1_{2g}$
5	z_5	$AC1_1$
6	z_6	$AC1_{1g} + AC1_2$
7	z_7	$AC1_2 + AC2_{2g}$
8	z_8	$AC1_{2g} + AC1_1$
9	z_9	$AC1_{1g} + AC2_1$
10	z_{10}	$AC1_2 + AC2_{1g}$

At time z_9, the application displays the same image as in time z_2, creating a self-repeating structure formed by 8 states.

As demonstrated by that example, the number of states formed by combining two or more, few-state active components creates state explosion and the number of states increases geometrically depending on the following factors:

- Number of states of each active component: The frequency of display change for each active component (This might be a varying number, like displaying image one for 200 milliseconds and image two for 300 ms.)

- The phase shift of active components.

- The sampling rate of the application display. (Since the thin client server component constantly gets screen shots of the application on a timely basis, the interval in which those screen shots are taken also has an impact on the number of states seen by the server manager)

In real life applications, the thin client system has no control over any of those components but the last one. The control over last item is also limited in a sense that there is a minimum limit of that capturing interval. Because the server has to process the bitmaps after capturing them, this processing will take some time and the interval can be at most that small, because the server has to finish processing one image before proceeding to capture another one.

All other factors depend on the active component's properties and operating system characteristics. So the thin client system at worst case has to live with that state explosion problem. Several tests were conducted, the "AltaVista" image and the following 6-state "e-mail" GIF. The characteristics of two images are given in Table 6-3.

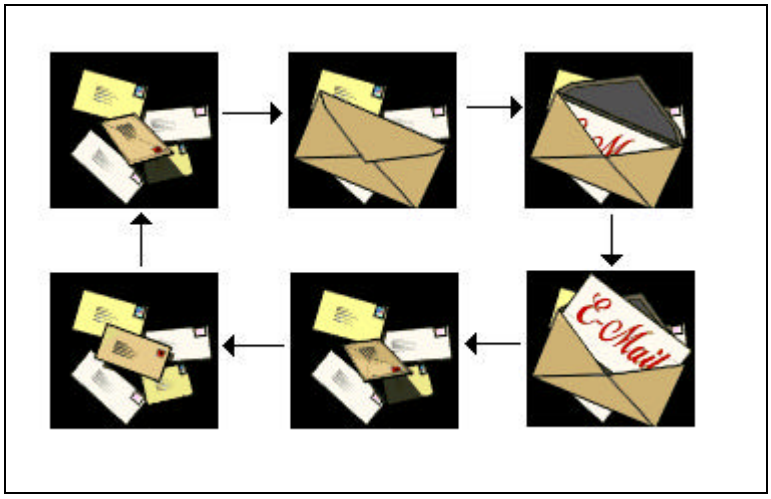


Figure 6-14: An animated GIF of 6 states and varying display change times

Table 6-3: Animated GIFs characteristics

ANIM. GIF	ST. #	STATE 1 DELAY (MS.)	STATE 2 DELAY (MS.)	STATE 3 DELAY (MS.)	STATE 4 DELAY (MS.)	STATE 5 DELAY (MS.)	STATE 6 DELAY (MS.)
AltaVista (Figure 6-2)	2	2000	2000	N/A	N/A	N/A	N/A
e-mail (Figure 6-14)	6	150	150	150	1500	150	1500

.

The experiments conducted with these two images (the application shown in Figure 6-15) showed that the state explosion could exacerbate.

Figure 6-15: The test case for state explosion with two animated GIFs

In that test case a buffer of size 200 images was used, however, it was not possible to capture a repeating pattern. Because, the number of states and the image timings are not identical, and phase shift further increases the number of states. There are states the server manager simply misses, because that state change occurs while it is processing an image. When, after some time, this missed state is captured as a new state, it distracts the buffer structure, and prevents server from detecting a repeating structure.

6.8 A Solution to State Explosion

As stated above, state explosion increases the number of states a client has to buffer, which means inefficiency and will put a considerable amount of burden when resource-poor wireless thin clients are considered. These types of clients will have small

storage and because of power issues, they would not want to use that storage.

The best approach to the state explosion problem is to transfer the active components separately and to let the client simulate them. The server would also supply the image refresh frequency of each active component and their relative starting times so that the same application display can be generated at the client.

In the example of Figure 6-13, with two 2-state active components we had 8 states. However, the best case would be a situation where the thin client transfers the two active components separately, which makes a total of 4 states and their relative timing. This will result in 50% savings. When there are more than 2 active components (generally the case), or the active components have than two states, the state explosion increases geometrically, as seen in the test case of Figure 6-15.

However, the Ultra Thin Client Prototype assumes no contribution from the application. That is, the application (browser in this case) cannot inform the thin client server manager about the content of its display. It cannot tell the server that it has 2 active components in the current display (animated GIFs in that case) with such and such characteristics, or they can be reached through that URL addresses. This will go into a completely different set of issues, where the applications are designed and developed so that they are capable of sensing that they are being executed in a thin client environment and can supply information to the thin client system.

Here, we are assuming no help from the application. Treating the application as a black box makes the improvements and systems to be application independent. What the thin client prototype has is just the bitmap of the applications display. But with some

processing, the active components can be identified in these bitmaps and can be sent as series of bitmaps along with timing information.

Being able to extract the active components from the bitmap image might also contribute in another way, if an application's display is made up of several active components and a region that keeps changing but not in a deterministic or predictable way. That might be the case of a web page that has several animated GIFs on it and has a text box for text entry. Figure 6-16 shows an example application display for that kind of a web page. A sample web based e-mail page, where the left had side has several active components, and the user types in the mail in the text boxes on right. Even if the animated GIF represent a self-repeating structure, as the user types the e-mail, the typed characters distract the repeating sequence of images, thus the server manager will not be able to capture that looping sequence to the left of the screen. However, if the server could extract the active components to the left, then it might tell the client to simulate the left-hand side of the display whereas it keeps sending the textbox's bitmap. This will incur great savings, since a single character constitutes so small when compressed. So the transfer between the server would be just as if the user is typing on a page where no other changes occur. That would be the most efficient thin client implementation: extracting active components without any help from the application, and letting the client simulate them locally, while transmitting non-active-components.

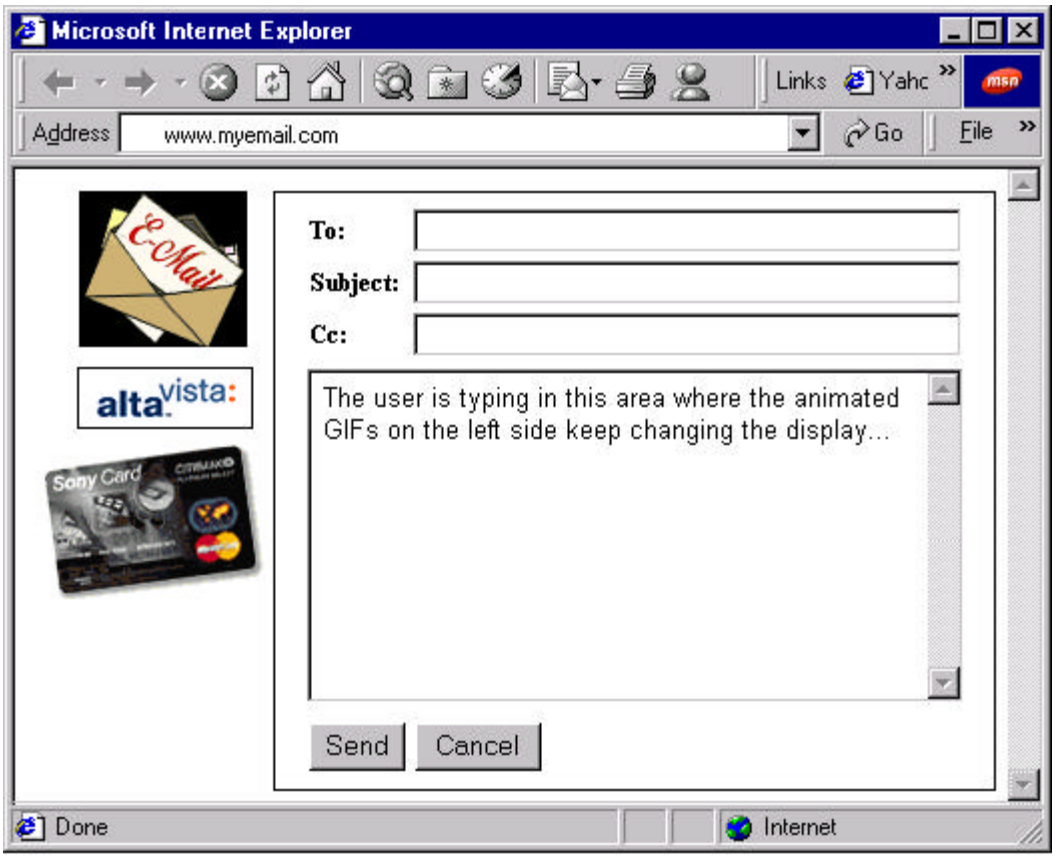


Figure 6-16: An application composed of active components and a text entry window

6.9 Identifying Active Components from Bitmaps

The thin client prototype is dealing with difference images instead of transferring the whole image every time. This makes a good point to catch the active components. Below are three images from the test case shown in Figure 6-15. These are the images that are being transferred to the client.

As seen from the images, active components lie in the middle of an emptiness, which makes it easy to capture. One can use vertical and horizontal scan lines to detect the places of those images, and use information to pick up the active components

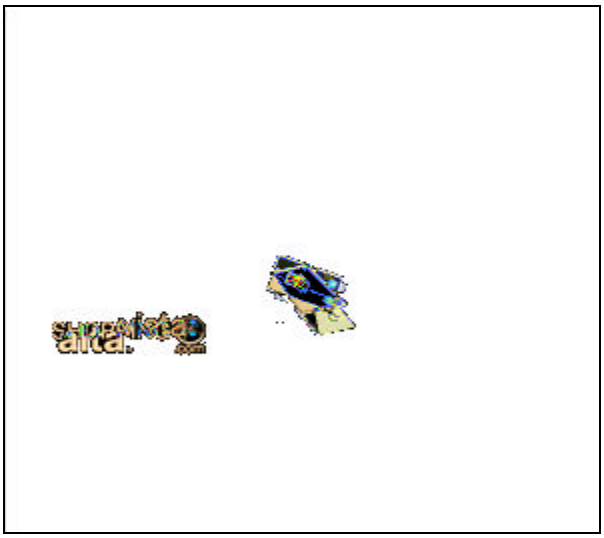


Figure 6-17: A difference image for application in Figure 6-15

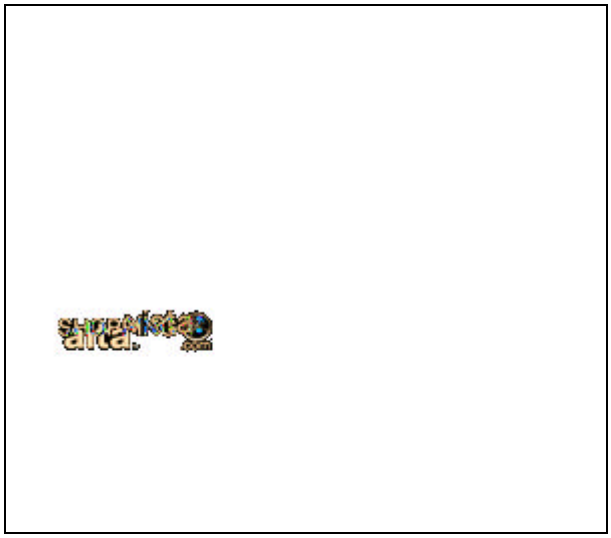


Figure 6-18: Another difference image for application in Figure 6-15

separately. The scan-lined versions of those images are given in the following figures (Figures 6-20 to 6-22).

So the problem becomes the identification of sub-images over a series of images (the image buffer of the server). If a screen capture in the buffer is called I_k, $k = 1\text{-}p$ (p being the buffer size, thus number of images), and sub-images $S_{k(1\text{-}m)}$ meaning that image

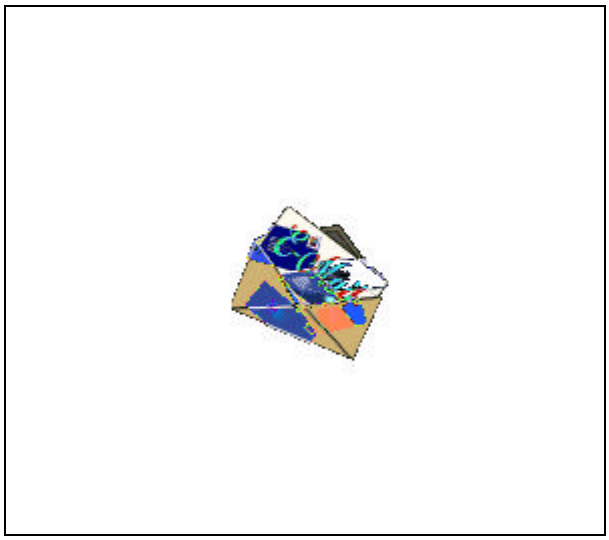


Figure 6-19: Yet another difference image for application in Figure 6-15

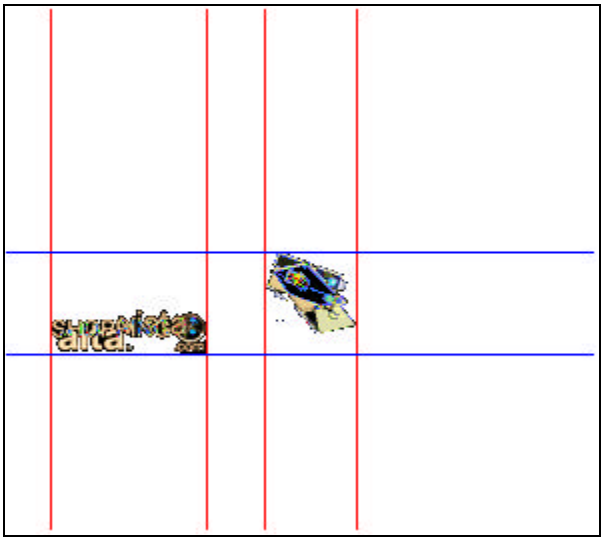


Figure 6-20: Enclosing rectangles for difference image in Figure 6-17

I_k has *m* sub images named as S_{k-1}, S_{k-2}, S_{k-3}…S_{k-m}. If we call the active components as AC_i, *i* =1-*h* (*h* being the number of active components on this series of images), then each active component is formed of several sub-images (states of the active component) which are in several images. Then,

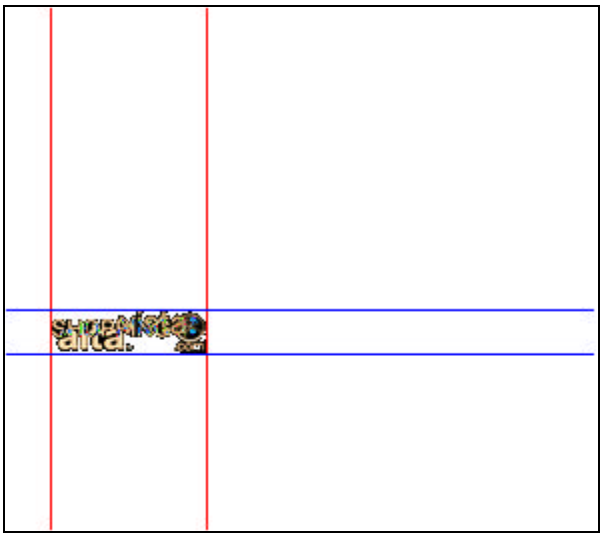


Figure 6-21: Enclosing rectangles for difference image in Figure 6-18

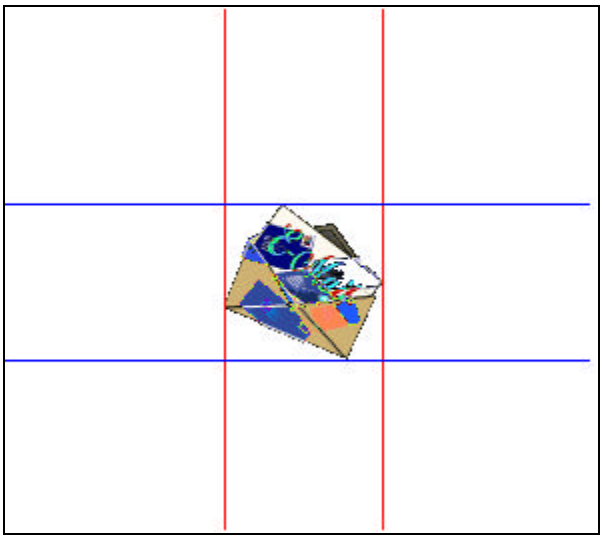


Figure 6-22: Enclosing rectangles for difference image in Figure 6-19

$I_k = \Sigma\ S_{k\text{-}b}$, where b = the number of sub-images present in that image I_k, and,

$AC_a = \Sigma\ S_{i\text{-}n}$, where i=1...p and n=1...m

For example, if a buffer of 6 images is being searched for active components, assuming that there are 3 active components in those series of images, they can be defined as follows:

$AC_1 = S_{1\text{-}1} + S_{3\text{-}1} + S_{6\text{-}1}$

$AC_2 = S_{1\text{-}2} + S_{2\text{-}2} + S_{3\text{-}2} + S_{5\text{-}2}$

$AC_3 = S_{1\text{-}3} + S_{3\text{-}3} + S_{5\text{-}3}$

Each sub-image $S_{k\text{-}i}$ has an ***enclosing rectangle***, a rectangle boundary capturing that sub image. Examples of these rectangles can be seen in Figures 6-20 through 6-22 (the rectangles are the ones formed by vertical and horizontal lines which have a non-empty region inside). I.e. Figure 6-20 has 2 sub-images thus two enclosing boundaries. For effective detection of active components, the size and the position of this enclosing rectangle is the only information available, other than the number of enclosing rectangles in a single image. So the detection algorithm will have the following input and output:

INPUT:

- Image buffer: p bitmap images
- Enclosing rectangles: number of enclosing rectangles in a bitmap, their size and position.

Working on that input, the detection algorithm must be able to detect the active components.

OUTPUT:

- Active components: their states, and timing information for displaying of these states. If there were *h* active components, the output would be:

$$AC_a = \Sigma\, S_{i\text{-}n} \text{ where } i = 1 \ldots p \text{ and } n = 1 \ldots m \text{ and } a = 1 \text{ to } h$$

There are several problems the detection algorithm has to take care of:

- The sub-images (active components) might not be present in all the images in the sequence : $I_k \not\subset S_{k-b}$, where $b = 1$ to h

- The sub-images' enclosing rectangles might not be at the same size and might not be positioned at the same place, although they are expected to be in the vicinity of each other, otherwise they might belong to another active component. Looking at the images gathered from the test case in Figure 6-15 – Figures 6-20 to Figures 6-22 – the enclosing boundaries' size and position changes can be observed. The size of rectangles belonging to the active component of the left, although being same in width, change size in the height. The position of the rectangle also changes, if top-left corner is used as the reference point for describing the rectangle position. The rectangles of active component two represent a more complicated situation. They vary in all aspects: width, height and position. That kind of behavior will be most common in real-life situations. So the algorithm has to be smart enough to handle these types of imperfect data.

- Another problem is that the placement of active components on the display might generate enclosing rectangles for non-existing active components. This case is illustrated in a sample web page (Figure 6-24) and its difference image along with enclosing rectangles (Figure 6-25). As seen in Figure 6-25, scan lines was able to plot accurately the enclosing rectangles for three active components. However, there is the fourth rectangle (the rectangle at the lower right corner) which is a side effect of the three active components. So the algorithm has to be clever

enough to figure out which rectangles contain images and which do not. Some additional processing has to be done to determine the content of the rectangle is full or not. Care has to be taken in that process because the rectangle, just by chance, can contain a small active component in it, which the algorithm might miss if it does not check every pixel in a rectangle.

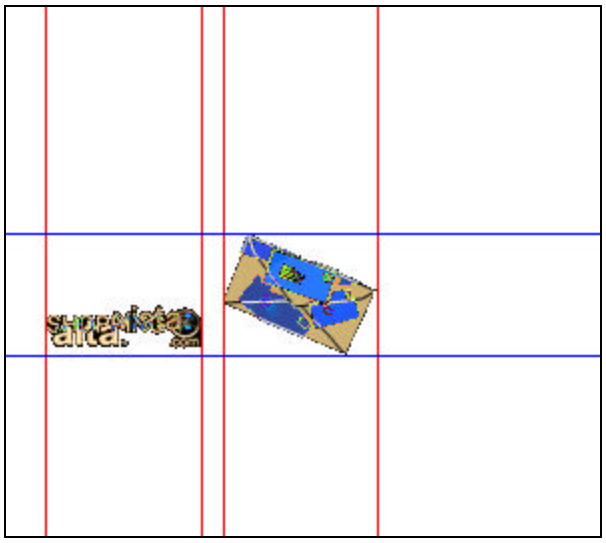


Figure 6-23: Enclosing rectangles for application in Figure 6-15

Figure 6-24: A web page with 3 animated GIFs

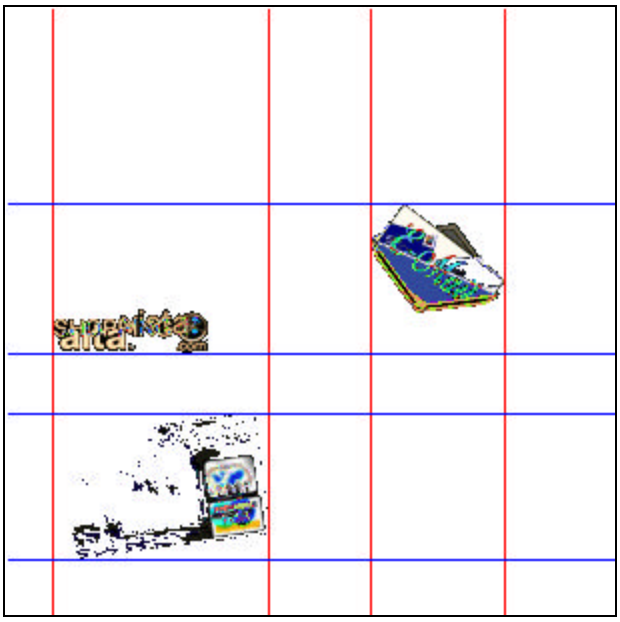


Figure 6-25: Corresponding difference image and enclosing rectangles.

Assuming that an algorithm has been developed to get the active components in a series of images, how would one test the effectiveness of such an algorithm? The loss function described in the next section will determine how well the algorithm performs.

6.10 Loss Function

The loss function's definition is simple. Since network communication has to be kept to minimum, the loss function has to measure unnecessary network usage [18]. The best case will be where only the necessary bits of the image are transferred to the client, whether they are active component bits for client to store and then simulate, or regular images for the client to just display.

The thin client prototype has already dealt with cases of regular image transfers through use of compression, and image id communication. While dealing with

localization optimization, the loss function will only measure the total number of bits transferred to describe an active component to the client.

Without extracting separate active components from the display of the image, the server tries to get a recurring structure out of the whole application display, which results in great number of states, thus cumbersome network transfers because of state explosion problem. As mentioned above the best case would be to send the active components separately (animated GIFs as animated GIF files) so that the client has the full knowledge of those and simulate them locally. However, dealing with bitmaps and extracting active components from the bitmaps, incurs some inefficiency, depending on the algorithm. The ultimate algorithm would be the one to perform as if it knows the structure of active components and transfer them to the client.

So the loss function definition will be:

Loss Function = # of unnecessary bits transferred to the client to be used in local simulation.

An example can be drawn from Figure 6-26. Assume the algorithm uses the gray rectangle on the left to describe a state of that active component. Then all the gray bits will incur a loss in the transfer and the loss function will be total number of gray bytes.

Loss function gives the number of bits that are being transferred, but has not changed or do not belong to that active component.

Figure 6-25, also gives a good example of loss where the two big active components (left-bottom and right top) and the “AltaVista” (left-top) active component have bigger enclosing rectangles than they actually need. So the algorithm has to handle those excess areas and find active component rectangles with a better precision.

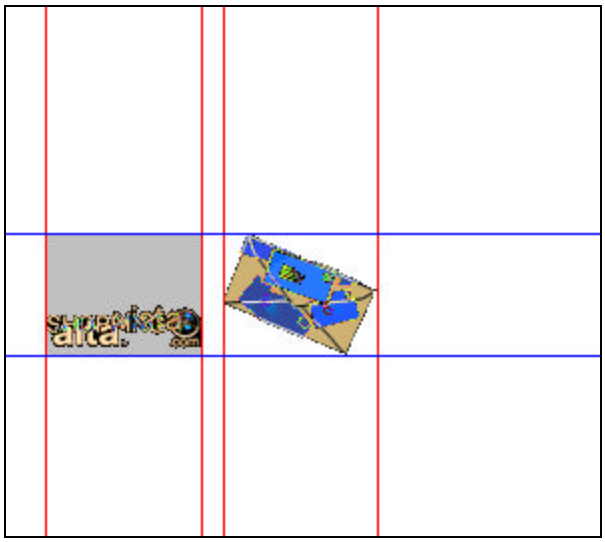


Figure 6-26: Grey rectangle is the enclosing rectangle for left active component in this image

6.11 Active Component Extraction Algorithm

The active component extraction algorithm is composed of 4 steps:

1. Find the “big” enclosing rectangles by scanning the difference images with vertical and horizontal scan lines.

2. Process each “big” rectangle, and find the smallest enclosing rectangle. This step might also cause a reduction in the number of enclosing rectangles in a single image, because this part will neglect empty rectangles like the fourth in Figure 6.25.

3. For each enclosing rectangle, determine which images in the buffer have the same boundaries, forming a list of image ids for each enclosing rectangle.

4. For each enclosing rectangle, pass over the images that have that enclosing rectangle and enumerate the regions, discovering the number of distinct images enclosed in that area.

Step one produces the "big" enclosing rectangles that are shown in figures 6-26 and 6-25. Applying second step gives the smallest possible enclosing rectangles and minimizes the loss. For Figure 6-25, the second part produces the rectangles that are shown in Figure 6-27. As seen in figure 6-27, the rectangle on the lower right corner is thrown away by the second step.

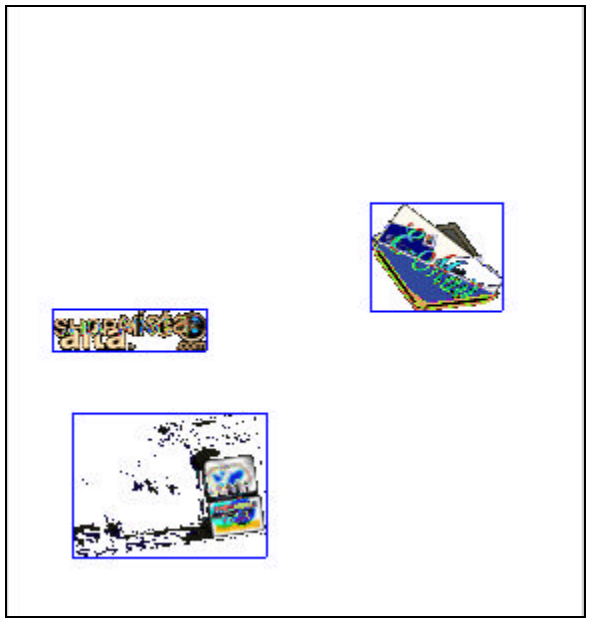


Figure 6-27: "Small" enclosing rectangles for image in Figure 6-25

The small enclosing rectangles are represented with 4 numbers, x1, x2, y1 and y2 as shown in Figure 6-28.

Using the same notation as the previous section, m^{th} sub-image in k^{th} image (S_{k-m}) is represented by four numbers:

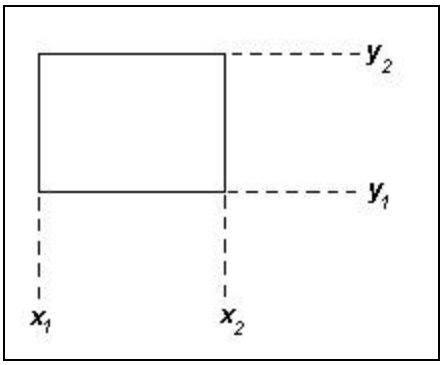


Figure 6-28: Smallest enclosing rectangles are represented by 4 numbers

$S_{k\text{-}m}$ = x1,x2,y1,y2

At the end of step two, all enclosing rectangles for all images in the buffer are known. Figure 6-29 shows what happens if all the enclosing rectangles are drawn on the same image (overlaid).

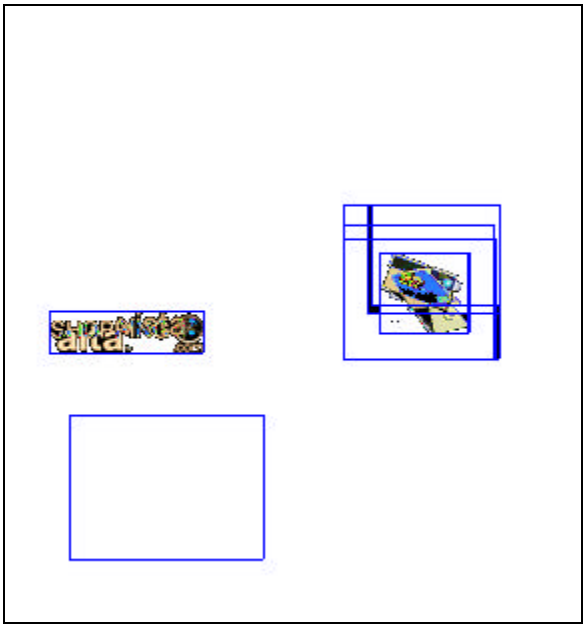


Figure 6-29: All smallest enclosing rectangles overlaid on an image

The rectangles for the two active components on the right fall on the same boundaries every time (these two are 2-state active components) whereas the rectangles of the AC on the right show variations. That is because the difference images of that AC (that is a 6-state AC) are of different sizes. The difference images for that AC are shown in Figure 6-30.

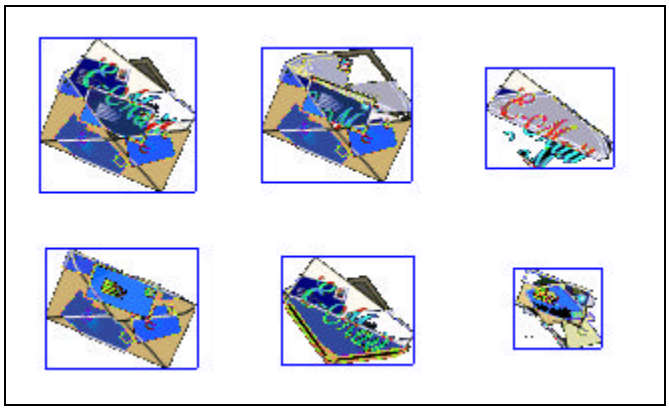


Figure 6-30: Difference images for active component in Figure 6-14

As seen in Figure 6-30, the enclosing rectangles' sizes are different which represents a problem in detecting the active component as a whole.

Another point of view helps handle these kinds of situations. As mentioned above, every image display, in the thin client has a ghost-pair that is required to delete the image from the screen in the following images. In the light of that information, every active component of n-states can be represented as *n* active components of *m* states, where $m<n$.

If the states of an active component are $S_1,S_2\ldots,S_n$ and the corresponding ghost images are $S_{1g},S_{2g},\ldots,S_{ng}$, then the states created by this AC can be represented in following steps:

1. Display S_1
2. Display $S_{1g} + S_2$
3. Display $S_{2g} + S_3$

…..
…
n. Display $S_{(n-1)g} + S_n$
n+1. Display $S_{ng} + S_1$

If the sampling rate was fast enough, another screen capture could have been made between steps 2 and 3, then step 3 would be divided into two sub-steps:

3.1 Display S_{2g} → Clears the display of state 2, and forms a black(empty) image
3.2 Display S_3 → Display state 3 on the empty space.

So depending on the screen capture interval, there can be *n* to *2n* images in the buffer for an AC of *n* states. If the sizes of these state images are different then each state can be captured as an active component itself. For example, state 2 can be extracted from the active component above and can be represented as follows.

1. Display $S_{1g} + S_2$
2. Display $S_{2g} + S_3$
n+1. Display $S_{1g} + S_2$
n+2. Display $S_{2g} + S_3$

This gives a new two state AC, whose image display frequency is composed of uneven intervals. The first image is displayed at time, 1, then the second is displayed at time 2 and then the AC goes into sleep till time n+1 comes. Then it displays the first image followed by the second at time n+2. And again sleeps for n intervals.

This point of view enables the application to extract any kind of active components, whether the enclosing rectangles are of the same size or not. The number of states is not a problem for the thin client unless the timing among them is established in the correct way and the AC is simulated in a way that reflects its original version.

The browser application shown in Figure 6-24 is run through the algorithm, and the following enclosing rectangles are found in a buffer of length 20 (Table 6-4)

Table 6-4: Enclosing rectangles and image numbers for browser in Figure 6-24

RECT.ID	ENCLOSING RECTANGLE	IN IMAGES
1	32,129,29,101	0,2,4,6,8,10,12,13,15,17,19
2	22,99,132,153	0,1,3,4,5,7,8,10,12,13,15,16,18,19
3	169,247,129,206	1,3,4,7,8,9,11,12,15,16,18,19,20
4	187,232,142,182	2
5	169,244,129,196	5
6	183,247,156,206	6
7	169,245,129,189	13
8	181,247,152,206	14

Each rectangle is shown in the following images Figures 6-31 through 6-38.

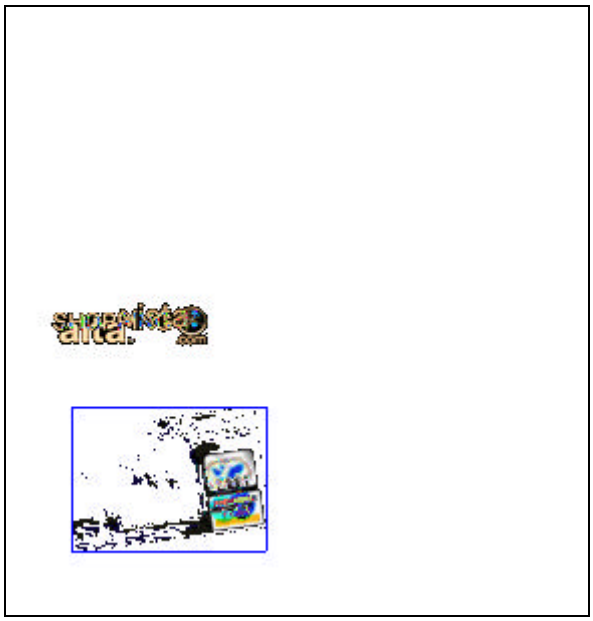


Figure 6-31: Image ID: 0, Enclosing Rectangle 32,129,29,101 (Rect. Id = 1)

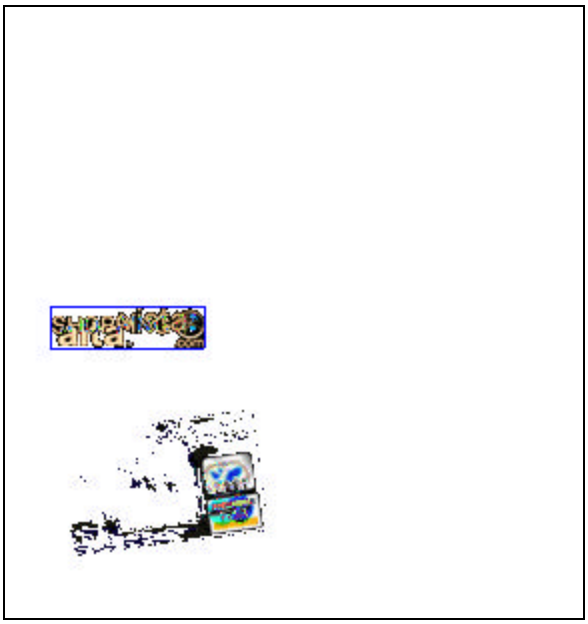


Figure 6-32: Image ID: 0, Enclosing Rectangle 22,99,132,153 (Rect. Id = 2)

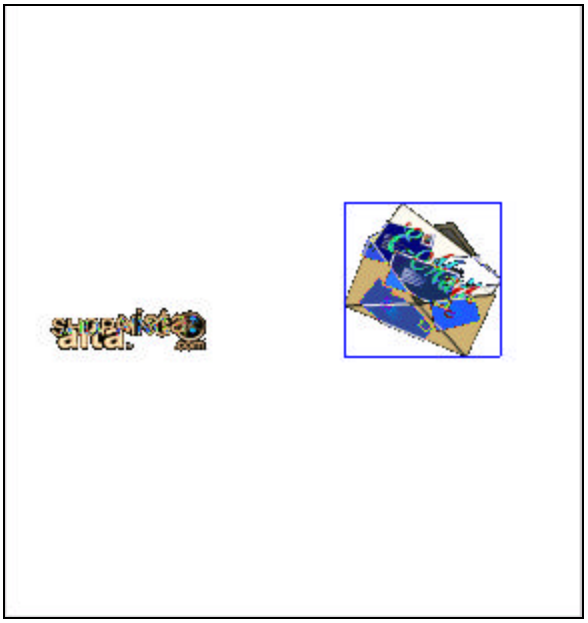


Figure 6-33: Image ID: 1, Enclosing Rectangle 169,247,129,206 (Rect. Id = 3)

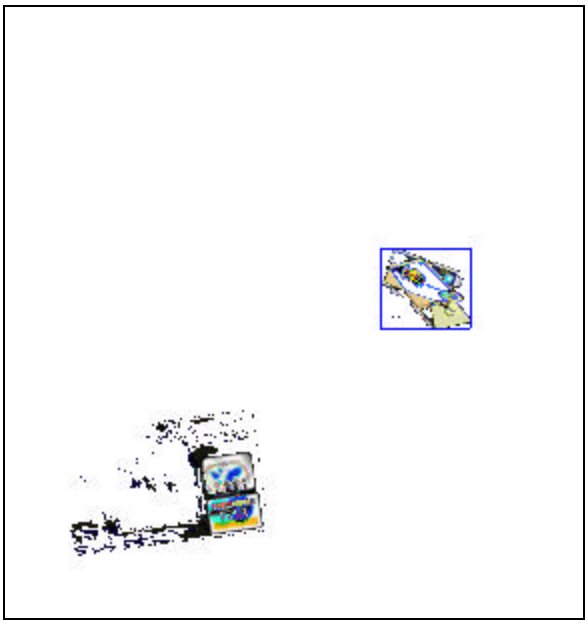


Figure 6-34: Image ID: 2, Enclosing Rectangle 187,232,142,182 (Rect. Id = 4)

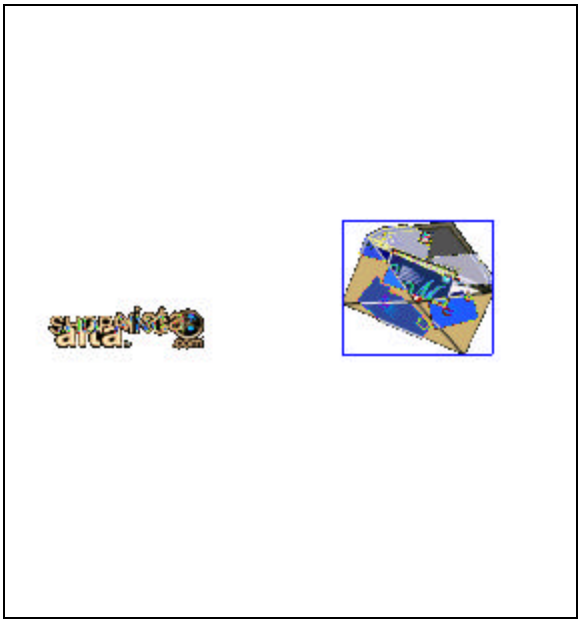


Figure 6-35: Image ID: 5, Enclosing Rectangle 169,244,129,196 (Rect. Id = 5)

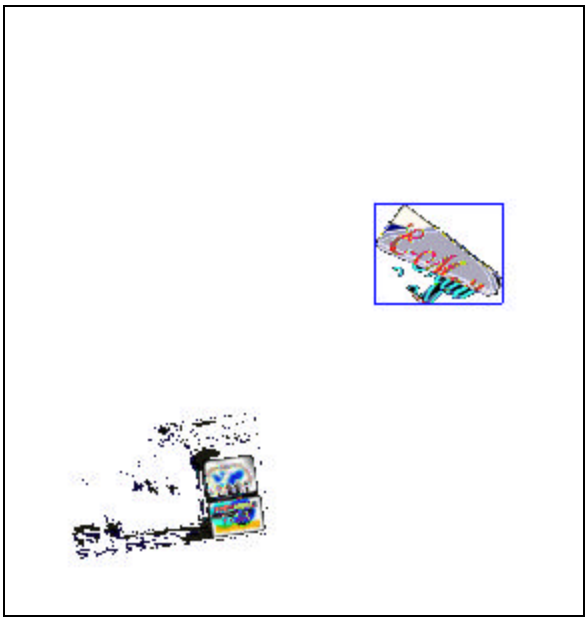


Figure 6-36: Image ID: 6, Enclosing Rectangle 183,247,156,206 (Rect. Id = 6)

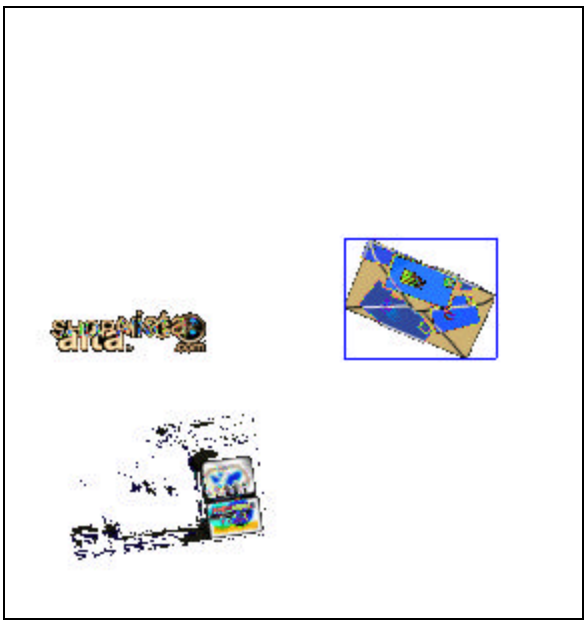


Figure 6-37: Image ID: 13, Enclosing Rectangle 169,245,129,189 (Rect. Id = 7)

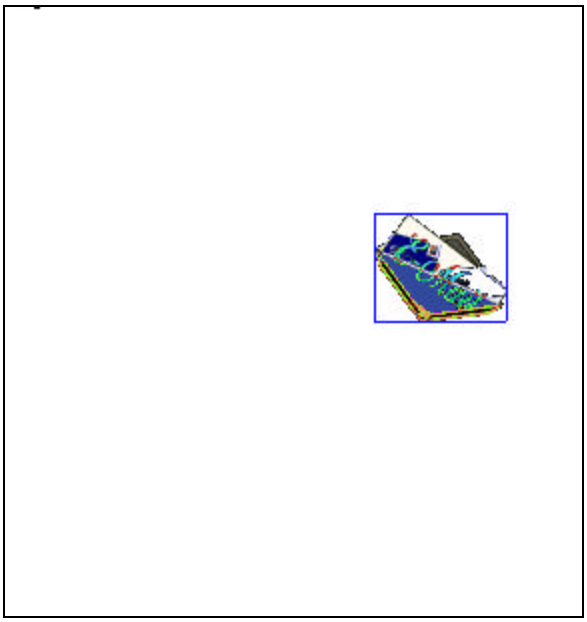


Figure 6-38: Image ID: 14, Enclosing Rectangle 181,247,152,206 (Rect. Id = 8)

As seen from the table, the third active component (the animated GIF on the right hand side) is divided into 6 sub-ACs. The output from step 3 of the extraction algorithm produces Table 6-4. Now the algorithm, for each enclosing rectangle, has to compare the bitmaps in different images to find out if the same bitmaps have been displayed over and over. This is accomplished by first extracting the bitmap that forms the enclosing rectangles and then comparing it with the other bitmaps extracted from the same enclosing rectangles in other images. The algorithm assigns an id number to each bitmap of that rectangle; distinct bitmaps have different ids whereas same bitmaps share the same id. The algorithm is given below:

```
k= 0;
IDGenerator = 1;
For each enclosing rectangle Ri (i = 0 to number of enclosing rectangles-1)
        For each image (Mi) that contains Ri {
                BMPi = Extract the bitmap enclosed by Ri;
                SubImages(k) = BMPi;
                Foz z  =  k-1 to 0 {
```

```
                    If subimages(z)= subimages(k) {
                        subimageId(k)=subimageid(z) ;
                        exit for ;
                    }
                }
                if subimageid(k)=Empty  subimageid(k) = IDGenerator++ ;
            }
```

Running this algorithm through table 6-4 gives the results in table 6-5:

Table 6-5: Enclosing rectangle bitmap ids

ID	ENCLOSING RECTANGLE		
1	32,129,29,101	0,2,4,6,8,10,12,13,15,17,19	Images
		1,1,2,1,2, 1, 2, 1, 2, 1, 2	Ids
2	22,99,132,153	0,1,3,4,5,7,8,10,12,13,15,16,18,19	Images
		1,2,1,2,1,2,1, 2, 1, 2, 1, 2, 1, 2	ids
3	169,247,129,206	1,3,4,7,8,9,11,12,15,16,18,19,20	Images
		1,1,2,2,1,2, 1, 2, 2, 1, 2, 1, 2	ids
4	187,232,142,182	2	Images
		1	Ids
5	169,244,129,196	5	Images
		1	Ids
6	183,247,156,206	6	Images
		1	Ids
7	169,245,129,189	13	Images
		1	Ids
8	181,247,152,206	14	Images
		1	Ids

As seen in Table 6-5, the rectangles contain bitmap images that repeat themselves after a few iterations. The sequence of rectangle 1 says that bitmap 1 is shown in image 0, then it is again displayed in image 2, and in image 4. Then, another bitmap (bitmap id 2) is displayed, followed by the display of bitmap 1 in image 6. This proves that each active component of 3 or more states can be deconstructed into several sub-active components each having fewer states.

The following table gives the results when the buffer size is 80:

Table 6-6: Enclosing rectangles of a buffer of 80 images of application in Figure 6-24

REGIONS		IMAGES AND BITMAP IDS
22,99,132,153	Imgs	0,4,6,8,11,14,17,19,22,25,28,31,33,35,38,41,44,47,49,51,54,57,59,63,65, 68,70,72,75,78
	ids	1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2
169,247,129,206	Imgs	5,8,13,18,21,23,27,29,33,37,42,46,48,50,52,55,57,59,62,65,69,73,77,79
	ids	1,1,1,1,2,3,2,1,1,1,1,2,1,2,1,2,3,2,1,1,1,3,2,3
32,129,29,101	Imgs	3,6,9,12,16,19,23,27,30,33,36,39,43,47,50,53,56,59,64,66,69,72,76,80
	ids	1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2
187,231,142,182	Imgs	24,58,74,80
	ids	1,1,1,1
169,244,129,196	Imgs	1,10,34,67,71
	ids	1,1,1,1,1
183,247,156,206	Imgs	2,11,35,64,68,72
	ids	1,1,1,1,1,1
169,245,129,189	Imgs	6,15,31,39
	ids	1,1,1,1
181,247,152,206	Imgs	20,26,45,49,54
	ids	1,1,1,1,1
187,232,142,182	Imgs	24,58,74,80
	ids	1,1,1,1

The bitmaps constitute a repeating structure, however, the corresponding image ids do not seem to move in a systematic way. The bitmap in the enclosing rectangle keeps changing from “the bitmap at buffer [0]” to “bitmap at buffer [4]”, however, one can not deduct a scheme looking at the image ids. After image 0 comes 4, and 6 and 8 and 11 and 14, and so on. It seems like those numbers are randomized and the image is a non-deterministically repeating active component. The cause of this problem is the sampling frequency, which was explained in chapter 5. Rephrasing the two problems from Chapter 5, will help gain insight in this situation.

- Problem 1: The accuracy of the timer being used

- Problem 2: The minimum screen capture interval possible

By looking at the time data for the first region (22,99,132,153), Table 6-7 can be constructed. Notice that the buffer also stores the timing information.

Table 6-7: Time interval for buffer of application in Figure 6-24 for rectangle (22,99,132,153)

IMAGE ID	TIME ON THE SCREEN (MS.)	BITMAP ID
0	1983	1
4	2033	2
6	1994	1
8	2003	2
11	2007	1
14	1989	2
17	2014	1
19	1993	2
22	2023	1
25	2003	2
28	2003	1
31	1993	2
33	1993	1
35	2003	2
38	2021	1
41	1996	2
44	2003	1
47	1995	2
49	2012	1
51	2003	2
54	1993	1
57	2013	2
59	2003	1
63	2003	2
65	2009	1
68	2008	2
70	2003	1
72	1993	2
75	2004	1

According to table 6-7 the average display time for bitmap with id 1 (state 1 of the animated GIF) is 2004.3 milliseconds whereas it is 2002 milliseconds for bitmap 2 (state 2 of the animated GIF). The bitmap in this region is the “AltaVista” bitmap. It is known

that the display intervals for this GIF are 2000 ms., which means that the ultra thin client simulator can do a good approximation on the intervals of the bitmap. The error rate for state 1 is 0.22% ((2004.3-2000)/2000x100) and 0.1% for state 2. The average error for that experiment is 0.16%.

Table 6-8: Time Interval for buffer of application in Figure 6-24 for rectangle (32,129,29,101)

IMAGE ID	TIME ON THE SCREEN (MS.)	BITMAP ID
3	2483	1
6	2504	2
9	2515	1
12	2503	2
16	2505	1
19	2483	2
23	2535	1
27	2494	2
30	2503	1
33	2494	2
36	2514	1
39	2494	2
43	2525	1
47	2503	2
50	2495	1
53	2493	2
56	2504	1
59	2514	2
64	2504	1
66	2494	2
69	2514	1
72	2494	2
76	2514	1
80	2469	2

Another active component, the third one at the left bottom known to have 2500 millisecond interleaved 2 states (the Sony Card GIF). This active component has the enclosing rectangle of 32,129,29,101 in table 6-6.

The output of the extraction algorithm for that GIF is given in table 6-8. The average display times for this active component is 2509.25 milliseconds for state 1 and 2494.917 milliseconds for state 2. The corresponding error rates are 0.37% and 0.20% giving an average error of 0.29% which is an acceptable error rate.

However, there might be problems due to minimum screen capture interval. The screen captures were made every 500±10 milliseconds for the above experiment. Minimum possible was 390 milliseconds because the processing time (the time to capture the screen, compute the difference, store it in the buffer) was at most 390 milliseconds.

The following section goes through 4 test cases demonstrating and quantifying the benefits of the optimizations discussed in this chapter. One test case demonstrates the inaccuracy and minimum interval time available for high frequency active components, where the change frequency of the active component is faster than the capture rate of the Ultra Thin Client Prototype.

CHAPTER 7
EXPERIMENT RESULTS

This section describes the benefits obtained by applying the improvements mentioned in the previous chapters. For each experiment application's nature, application's sample display and graphs on processing times and bytes transferred will be given.

7.1 Test Case 1

Application: Browser with a single 2-state animated GIF.

Sampling Rate: 500 ms.

Active Component: Animated GIF with 2 states. States displayed for 2 seconds.

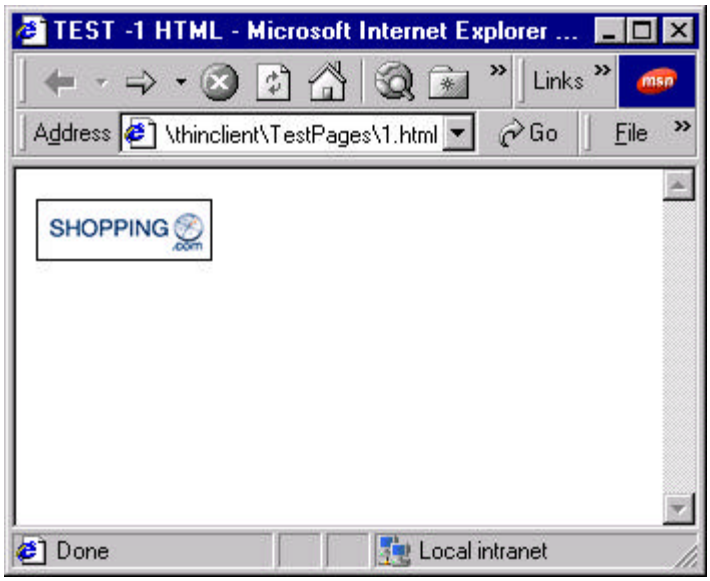


Figure 7-1: Test Case 1 sample screen shot

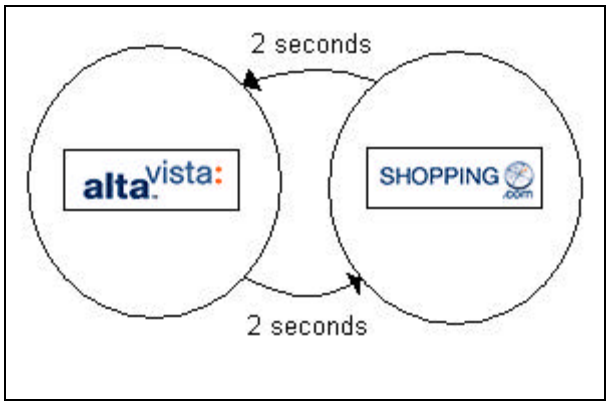


Figure 7-2: Animated GIF state diagram

The application is run through 3 test cases. First, with just thin client prototype, without performing any recurring structure detection and client localization. The results are given in Figure 7-13, the amount of processing and compressing and the bytes being transferred between the client and server are plotted in this chart.

As observed from the graph, the initial screen transfer takes longer (about 75 kilobytes). Afterwards, the server starts sampling at about every 500 ms (500.244 ms on the average), and does some processing to check if the image has changed or not. If the image has changed then the server compresses it and sends to the client. This, as seen on Figure 7-13, happens on every several sampling. Since the difference images are transferred, the sizes are relatively small, around 6K.

But in this case the client always stays in the thin client mode, depending on the server. Running the application with the loop detection algorithm produces the same processing time graph but the bytes transferred is different in this case. The accumulated byte graph can be seen in Figure 7-14. The experiment initially transfers the same amount of bytes for the whole screen. However, after 5 difference image transfers, detecting the recurring structure, the server puts the client into local simulation mode. Then the client

simulates the animated GIF without any communication from the server. The processing graph of server is exactly the same as Figure 7-13 because, the server keeps processing and checking that the application behaves in the recurring structure without breaking the loop.

The gain in this example can be quantified in the following table. If *t* represents the time in milliseconds, the application will be running, then *t*/500/4 is the number of state updates that travel from the server to the client in the non-optimized case.

Table 7-1: Number of bytes transferred in test case 1

OPERATION MODE	NUMBER OF BYTES TRANSFERRED:
Thin Client Mode	75 K + 6K x (t/500/4)
Optimized Thin Client Mode	75 K + (6K x 5)

So if the application were to run for 30 minutes, the bytes transferred would be 240 KB, whereas it is a constant of 105K for any amount of time with the optimized case. In a wireless environment, initiating a transfer every few seconds and transferring 6K would incur a lot of traffic and connection charges. The results are compared graphically on Figure 7-3.

The active component extraction optimization hasn't been applied in that case because there is only one active component. In such a case, active component extraction would produce the same results as loop detection optimized mode.

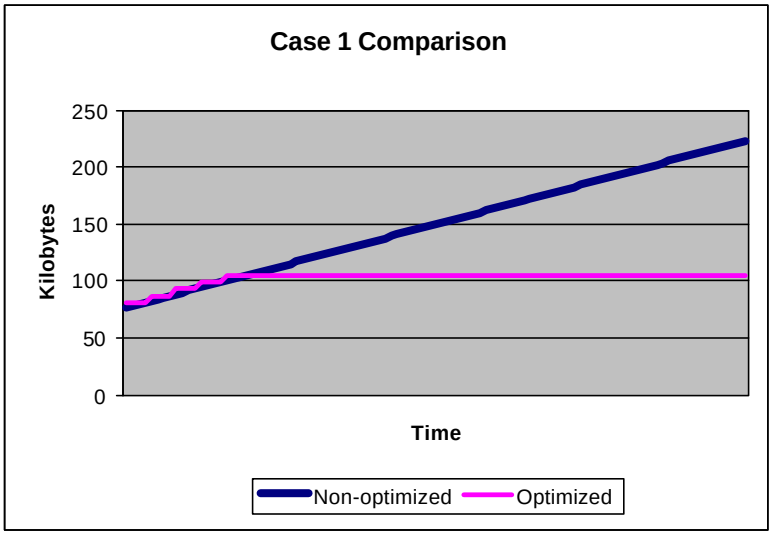


Figure 7-3: Cumulative bytes transferred in test case 1

7.2 Test Case 2

Application: Browser with two 2-state animated GIFs.

Sampling Rate: 500 ms.

Active Component 1: A. GIF with 2 states. States displayed for 2 seconds.

Active Component 2: A. GIF with 2 states. States displayed for 2.5 seconds.

The results of this test case are given in Figures 7-15, 7-16, and 7-17. In Figure 7-15, there are no optimizations of any kind. So the client is fully dependent on the server and the communication in between is required as long as the client is in function. After the initial “whole screen” transfer, the application sends about 5 Kilobytes (5.01 KB) on the average on every sampling which takes place in every ~500 milliseconds. So the amount being transferred is computed by the following formula where *t* is the amount of time the application is being run (in milliseconds).

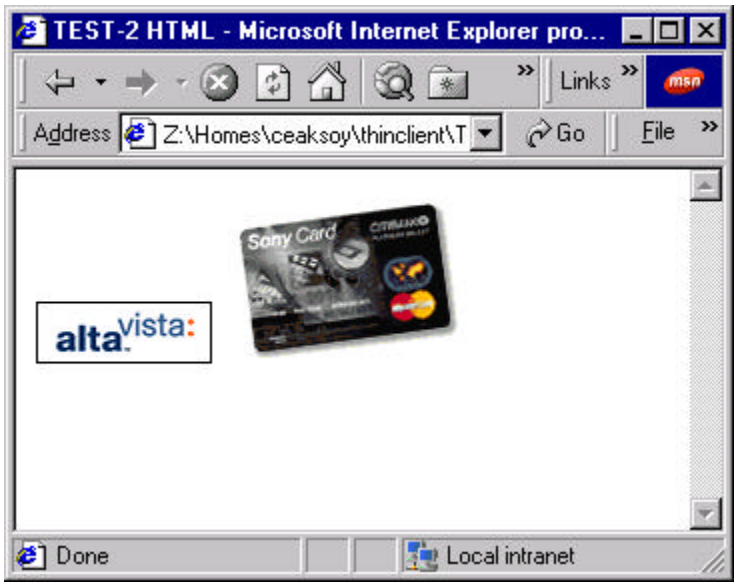


Figure 7-4: Test Case 2 sample screen shot

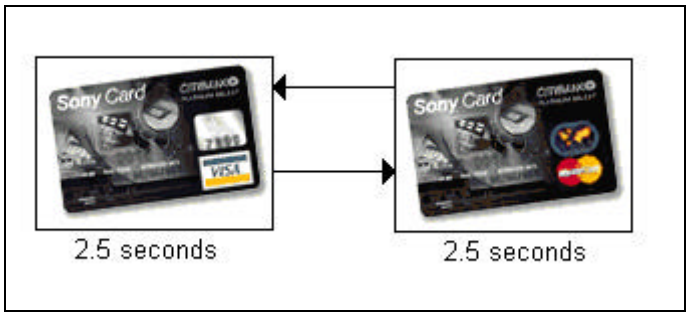


Figure 7-5: Test Case 2 seconds animated GIF state diagram

126 KB + (t/500) x 5K

In the second case, the application, at 79th sampling determines that the display forms a recurring structure of 16 states, with 6 distinct states. And then passes the 6 distinct states to the client and puts client into the simulation mode. The chart for this optimized version is in Figure 7-16. After sampling number 99, there is no

communication between the client and the server. The amount of bytes that is being transferred is 422 KBs and that is the total number of bytes that has to be transferred.

On the third case, active component extraction algorithm is applied and the results are depicted in Figure 7-17. Here after the 12th sampling, the server extracts the 2 active components separately. The first one with a frequency estimation of 2003 milliseconds (0.15% error) and the seconds with 2379 ms (4.84% error). The experiments were conducted with the need for reducing the detection time. If the buffer size were kept larger, the error rates would be smaller than 1%. So in this case, the total amount of bytes transferred is 179 KB. This optimization solves the state explosion problem.

Table 7-2: Number of bytes transferred in test case 2

MODE	BYTES TRANSFERRED
Non-optimized Thin Client	126 KB + (t/500) x 5KB
Optimized Thin Client	422 KB
Active Component Extraction	179 KB

The results are compared graphically on Figure 7-6.

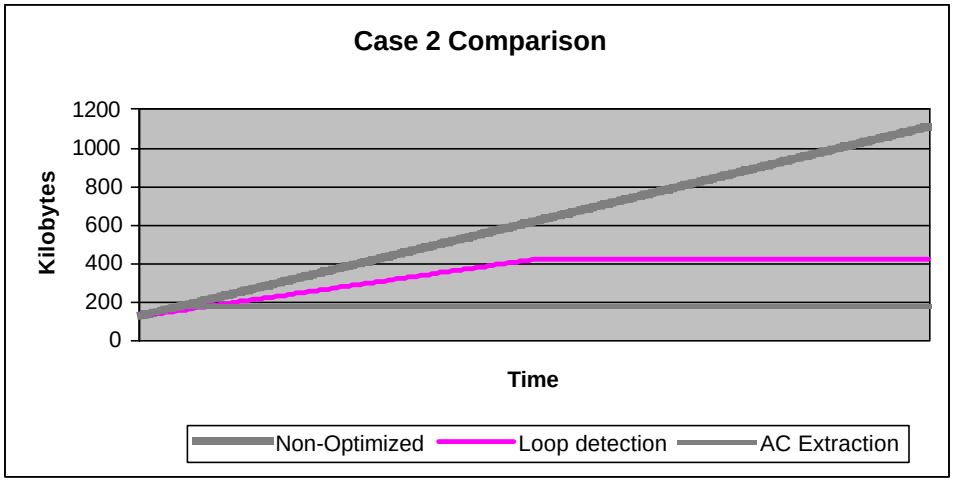


Figure 7-6: Cumulative bytes transferred in test case 2

7.3 Test Case 3

Application: Browser with a single 6-state animated GIF.

Sampling Rate: 400 ms.

Active Component: Animated GIF with 6 states. States displayed for variable amount of times and some are smaller than the sampling rate.

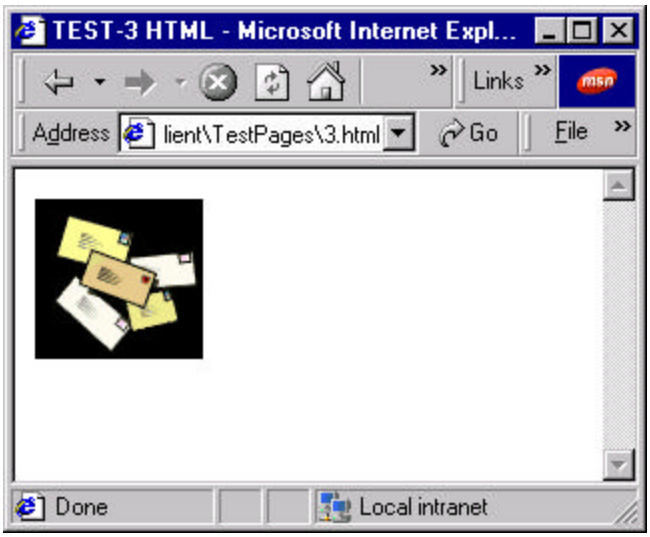


Figure 7-7: Test Case 3 sample screen shot

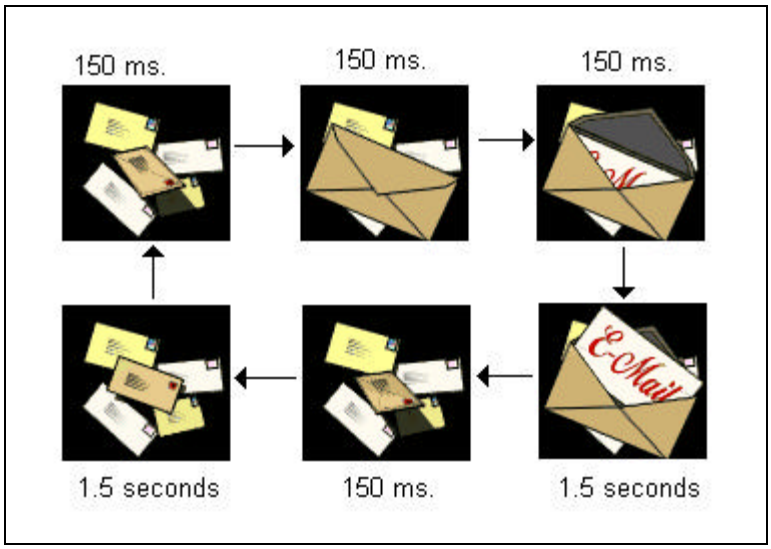


Figure 7-8: Test Case 3 seconds animated GIF state diagram

The results are given in Figures 7-18 to 7-20. Figure 7-18 is the experiment run without any optimizations. Here the initial transfer is 75KB and then an average of 2.5 KB (2446 bytes) is transferred at every sampling which is done every 400 milliseconds.

The important thing about this active component is that some of its states are displayed for a shorter amount of time than the thin client prototype samples the application display. As the following optimized test shows, this leads to state misses. The server misses some of the states that are being displayed for 150 milliseconds because it samples at 400 milliseconds. So the thin client server's view of the application is not perfect. The states that are missed depend on their display quantum and the starting times of the animated GIF and the sampling. For more than 10 experiments conducted with the sampling rate of 400 ms and this test page, the server was able to catch 3 to 5 states out of 6. So the server, depending on the relative timing with respect to the active component, either misses a single state, 2 states or 3 at most. The time the client is being started plays an important role. The loop detection algorithm can be modified to adjust its own timer to be able to capture as much states as possible. The maximum amount of the sampling rate should be able to make a sampling on every image (GIF state). For the above case Figure 7-21 gives the timetable of the active component and gives several sampling rates with variable start times. As seen the upper sampling produces a view of 3 states (states 2,4,6) and the second sampling produces a 5-state view (states 1,3,4,5 and 6). Since the sampling rate is greater than 300 ms. (150 + 150) the prototype should never be able to capture the all of the states and make a perfect simulation. To capture every state, the sampling rate has to be at most the sum of the display time of the quickest state, and the

one that follows it. So that taking a sample in the most quickly changed state, will not lead to a miss of the following state. For this test case, this limit is 300 ms. Taking a snapshot at state 1, we need to take another snapshot before state three is displayed, that is in at most 300 milliseconds.

For a perfect system, the prototype can be implemented in a multithreaded way, where by interleaving the operations a better sampling rate can be achieved. However, there will always be a limit on the sampling rate unless the thin client is either implemented at operating system or graphics display device driver level.

Figure 7-19 – the optimized version – shows number of bytes transferred drops to 0 after 89^{th} sampling. At that state the application was able to detect a loop of three states (states 3,4 and 6). And the client starts to simulate the active component as a 3 state animated GIF. The total amount of bytes transferred is 263 KB. Because of the sampling rate, the simulation at the client side was the same thing the user would be able to see if there was no optimization or localization. That is because the user at the client perceives the world through the sampling rate of the server. So compared to the real application there is a miss, but compared to the thin client mode there is no loss, the user is presented the same thing both with and without the optimization.

With the introduction of active component extraction, the server detects 5 different active components. Each active component represents a state of the active component. (The discussion in previous chapters proved that any n-state active component could be represented as a combination of several 2-state active components). This means that the prototype misses one state (should have been up to three states depending on the sampling rate). State 2 is being missed in that case. The average

frequencies of the states that are being captured are 3606 to 3615 milliseconds. This makes sense, because looking at the timing information, we see that state 1 is being displayed every 3600 ms. and so does all other states. This is because, a single cycle of the active component (displaying each state for once) is 3600 milliseconds (150 x 4 + 1500 x 2). The following table lists the estimates of frequencies of each state.

Table 7-3: Output of active component extraction for test case 3

STATE	FREQUENCY ESTIMATE
1	3607
2	Missed state
3	3606
4	3615
5	3608
6	3606
Average	3608.4
Average Error	0.23 %

So the average error is in the acceptable range and the amount of bytes transferred are 155KB which is nearly half of what the regular optimization was accomplished. And this optimization provides 5 states out of 6, instead of the 3 states of the previous case.

The following table gives a comparison of bytes transferred in each experiment.

Table 7-4: Number of bytes transferred in test case 3

MODE	BYTES TRANSFERRED
Non-optimized Thin Client	75 KB + (t/500) x 2.5K
Optimized Thin Client	263 KB
Active Component Extraction	155 KB

The results are compared graphically on Figure 7-9.

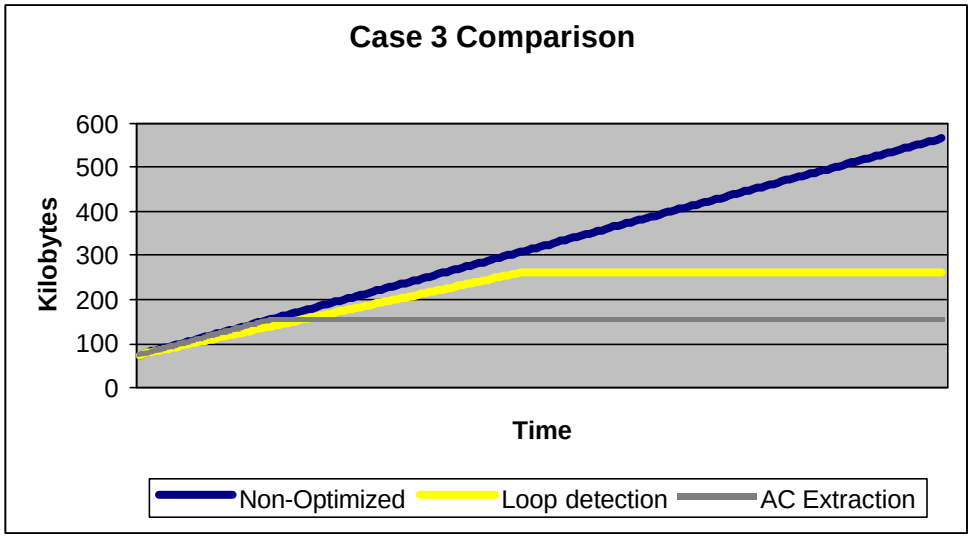


Figure 7-9: Cumulative bytes transferred in test case 3

7.4 Test Case 4

Application: Browser with multiple active components.

Sampling Rate: 400 ms.

Active Components: One from each test case above. The state diagrams can be found in 7-2, 7-5 and 7-8.

This case demonstrates a sample example where the regular optimization cannot find any recurring structure between the screen shots although a buffer of 200 images was used.

Figure 7-22 shows the experiments run with a loop detection optimization. The results are the same as running the application on a non-optimized thin client because the loop detection algorithm cannot be of help due to the state explosion problem. The total number of bytes transferred is 102 KB + (t / 500 * 5 KB).

Figure 7-10: Test Case 4 sample screen shot

However, turning on the active component extraction, the algorithm, after 50 samplings, returns satisfying results. It detects 5 separate active components. 2 for 2-state animated GIFs on the left-hand side and 3 for the 6-state "e-mail" animated GIF. So it misses three states of the multi-state GIF. The following table gives the frequency approximations of the algorithm:

Table 7-5: Output of active component extraction for test case 4

ACTIVE COMPONENT	ANIMATED GIF	ESTIMATED FREQ.	ERROR RATE
1	*"Alta vista"*	1960	2 %
2	*"Sony Card"*	2463	1.48%
3	*"E-mail"*	3609	0.25 %
4	*"E-mail"*	3609	0.25 %
5	*"E-mail"*	3609	0.25 %

The errors on the 2-state GIF are greater than the usual error rates; however, these are still acceptable rates. The total amount of bytes transferred is 357 KB. The extraction algorithm was being run for a longer amount of time to get accurate estimations on

frequencies of the active components. After transferring 357 KB there is no communication between the client and the server.

The results are compared graphically on Figure 7-11.

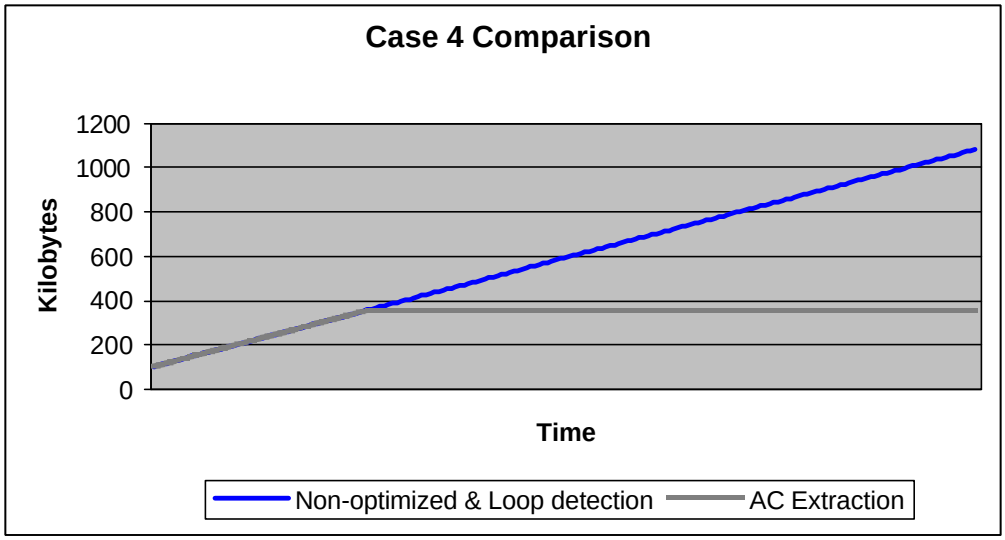


Figure 7-11: Cumulative bytes transferred in test case 4

7.5 Test for Bigger Display Applications

The size of the display effects the thin client's performance in a strong way. The application's used above can be considered as small sized display applications. Figure 7-12, shows how applications' display size effects the performance of the Ultra Thin Client Prototype. The experiments were conducted with application sizes that allow processing time to be below 400-500 ms. range. As seen from the graph the processing time approaches 1.5 seconds for bigger display sizes.

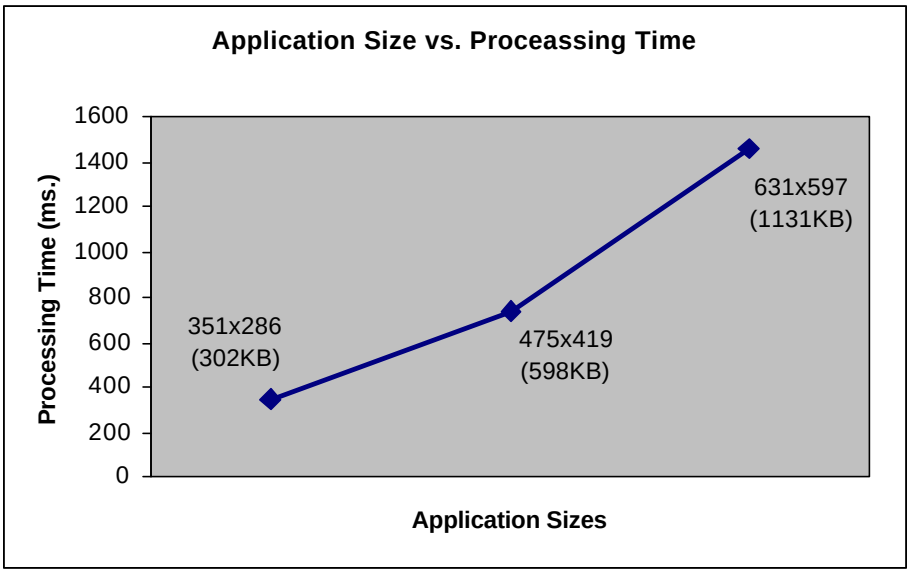


Figure 7-12: Application size versus processing time

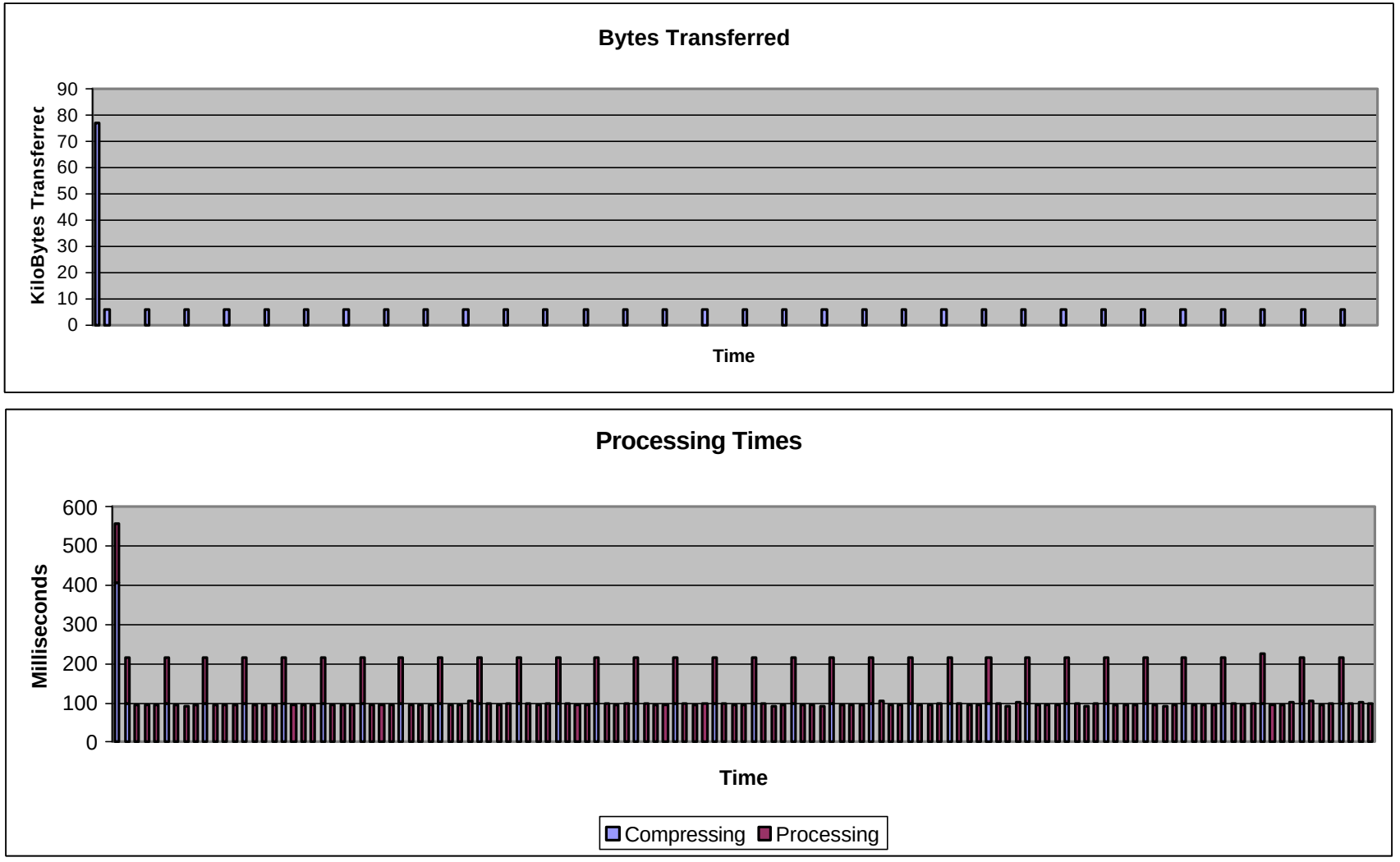


Figure 7-13: Test Case 1 results.

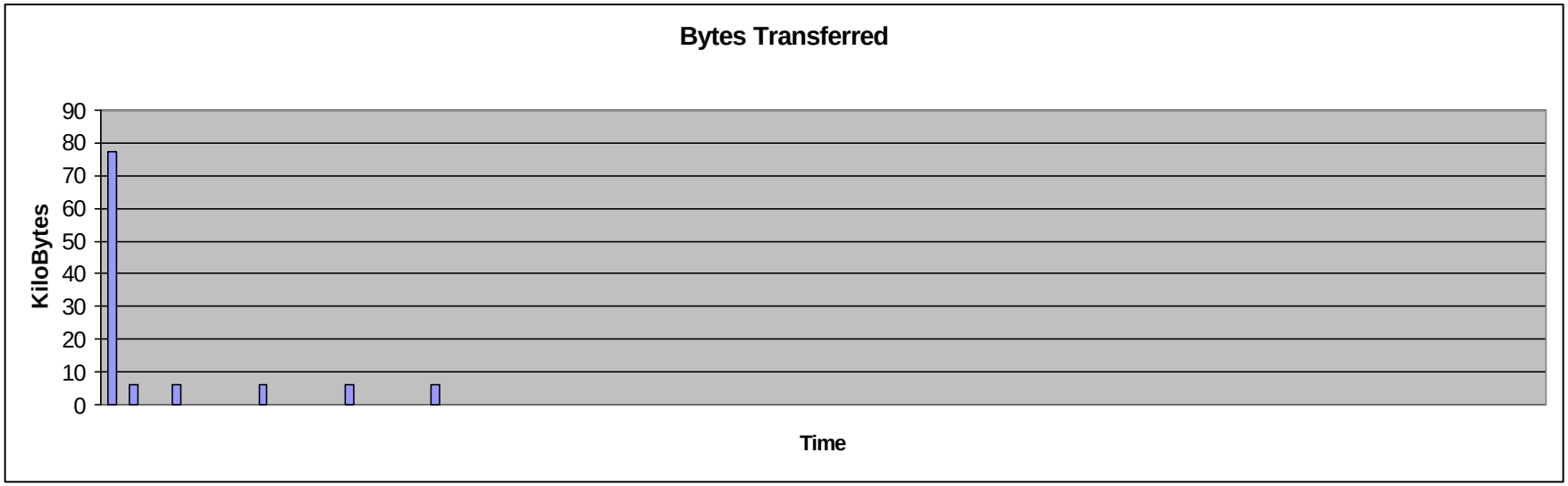
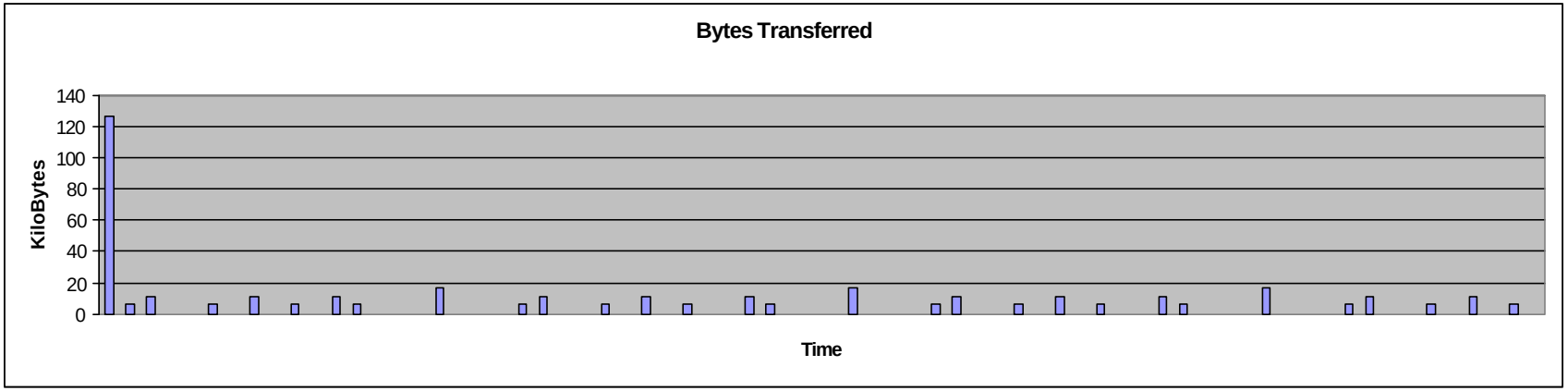


Figure 7-14: Test Case 1 with active component detection



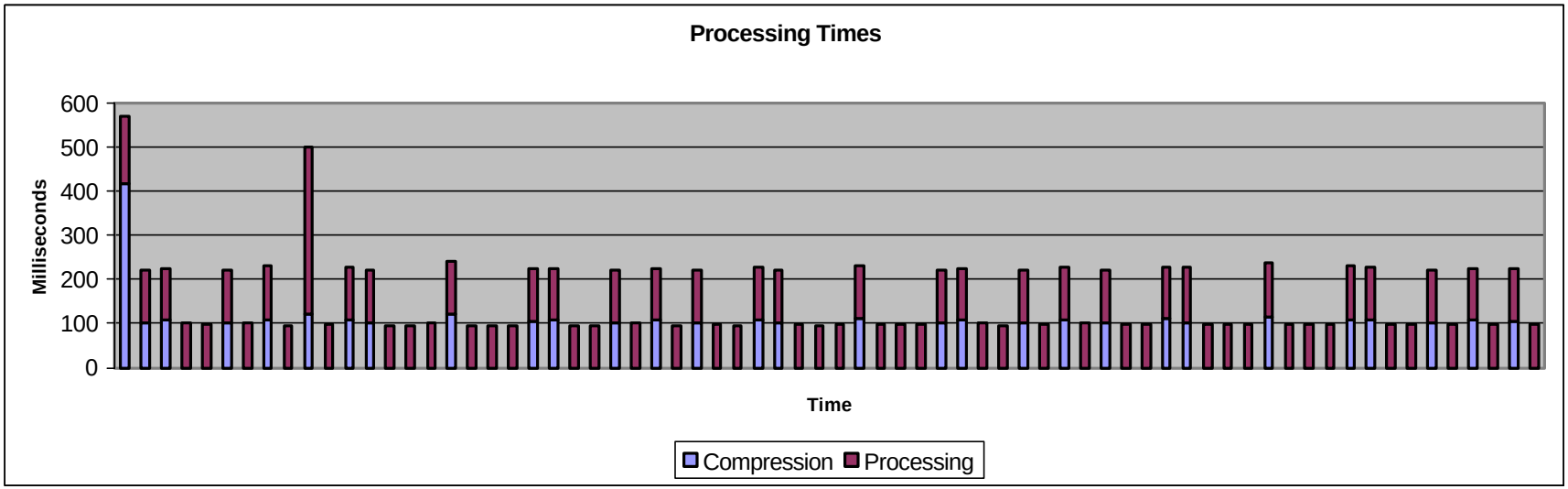


Figure 7-15: Test Case 2 with no optimizations.

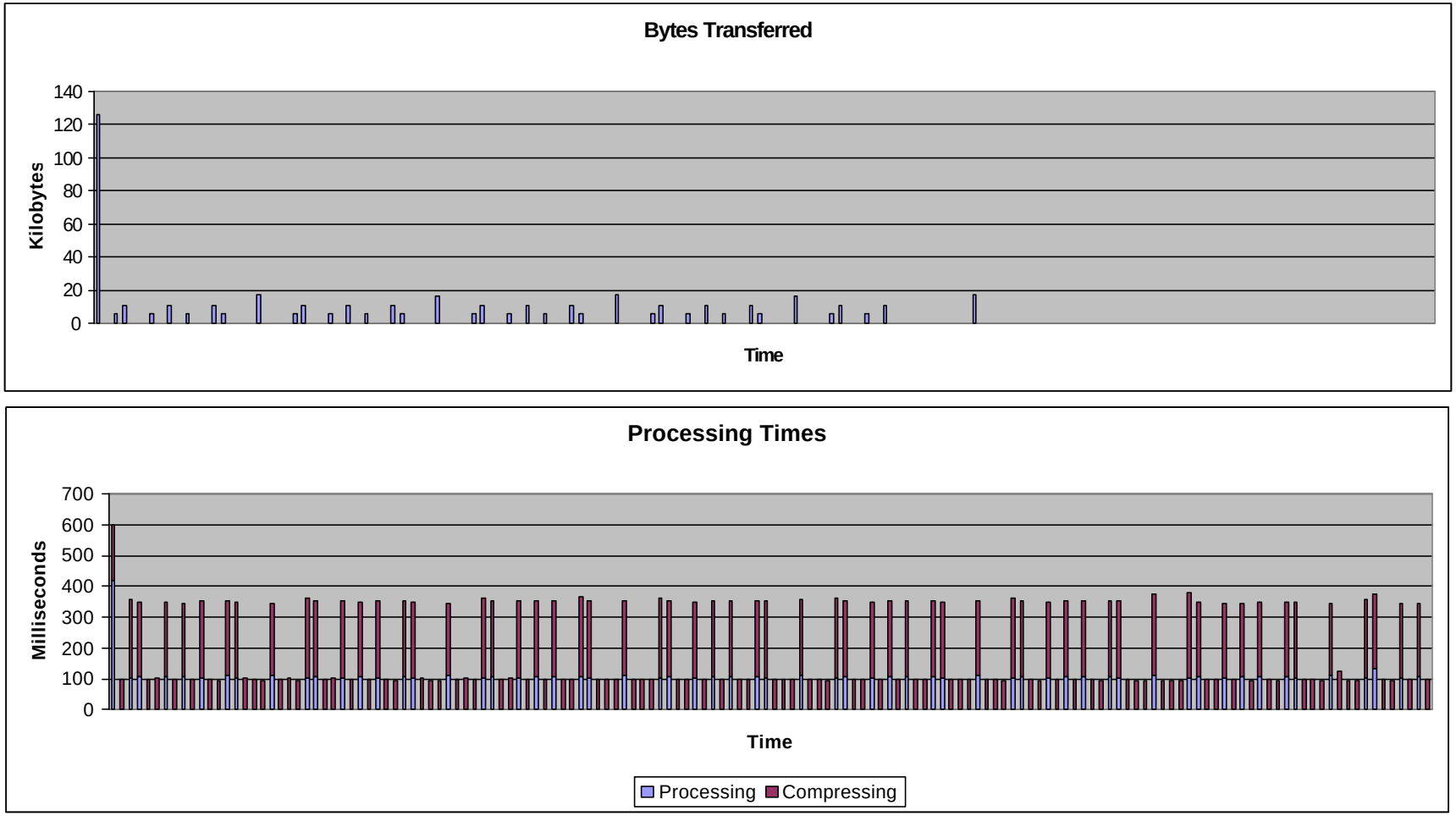


Figure 7-16: Test Case 2 with loop detection optimization

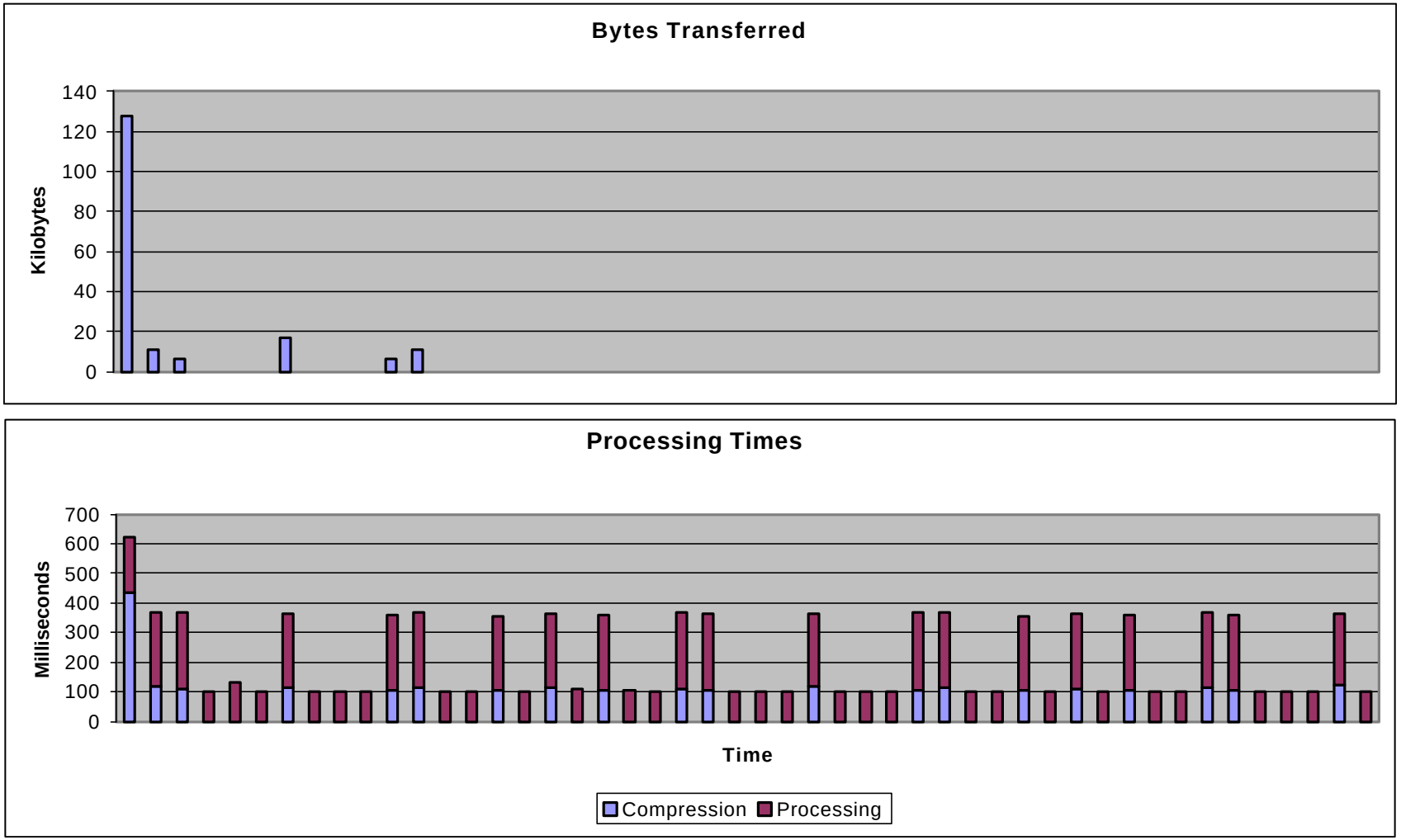


Figure 7-17: Test case 2 with active component extraction

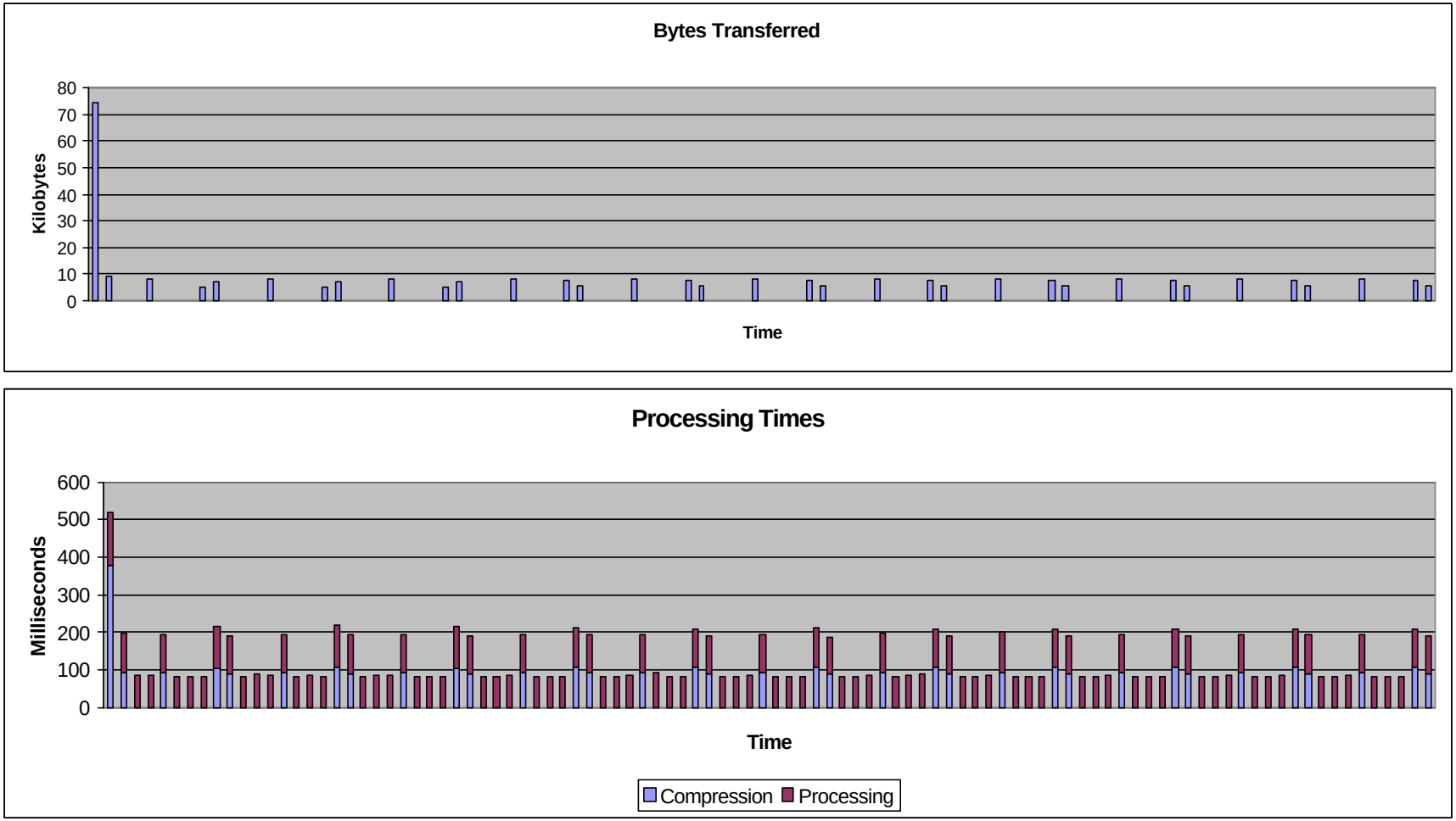


Figure 7-18: Test Case 3 without optimization

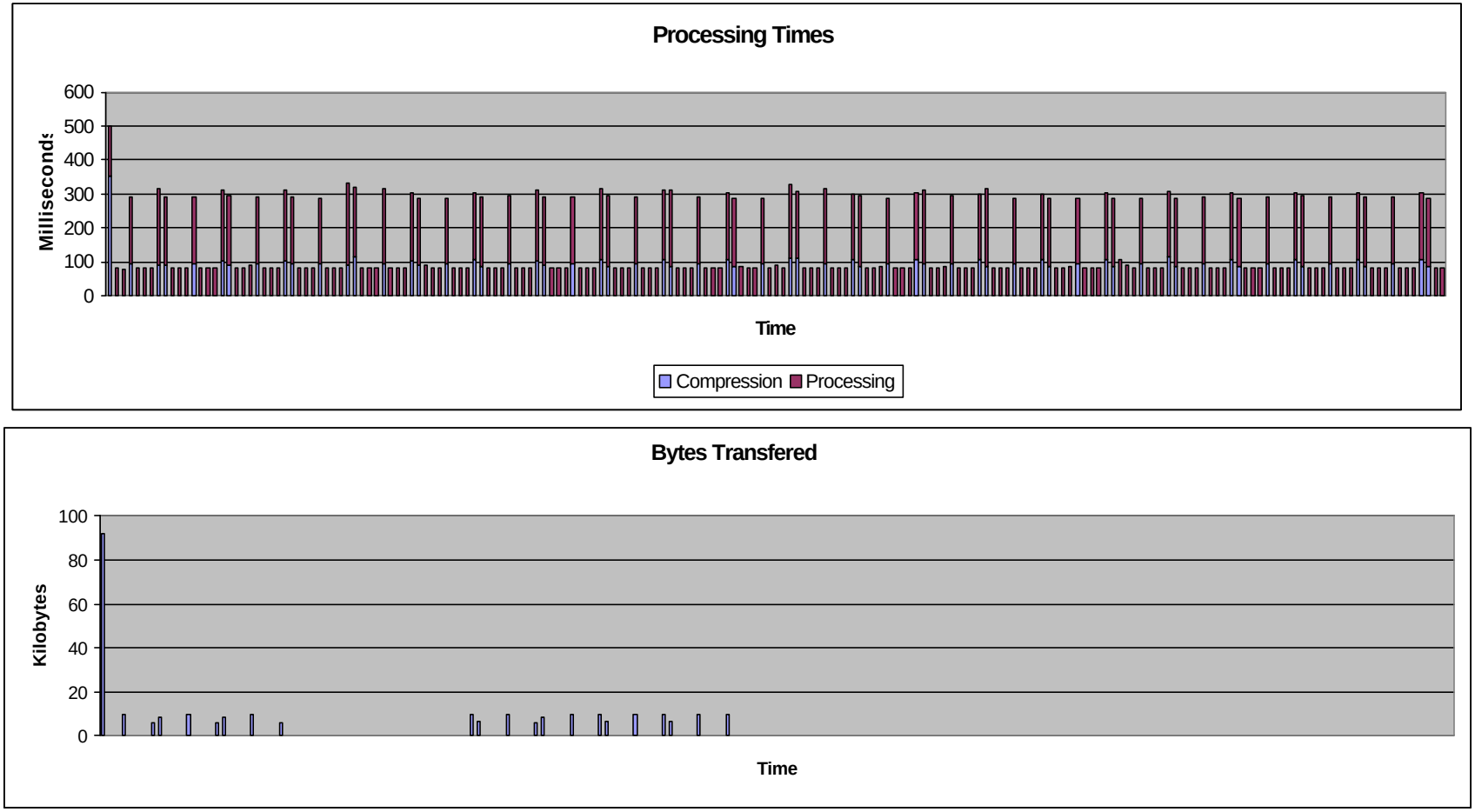


Figure 7-19: Test Case 3 with loop detection optimization

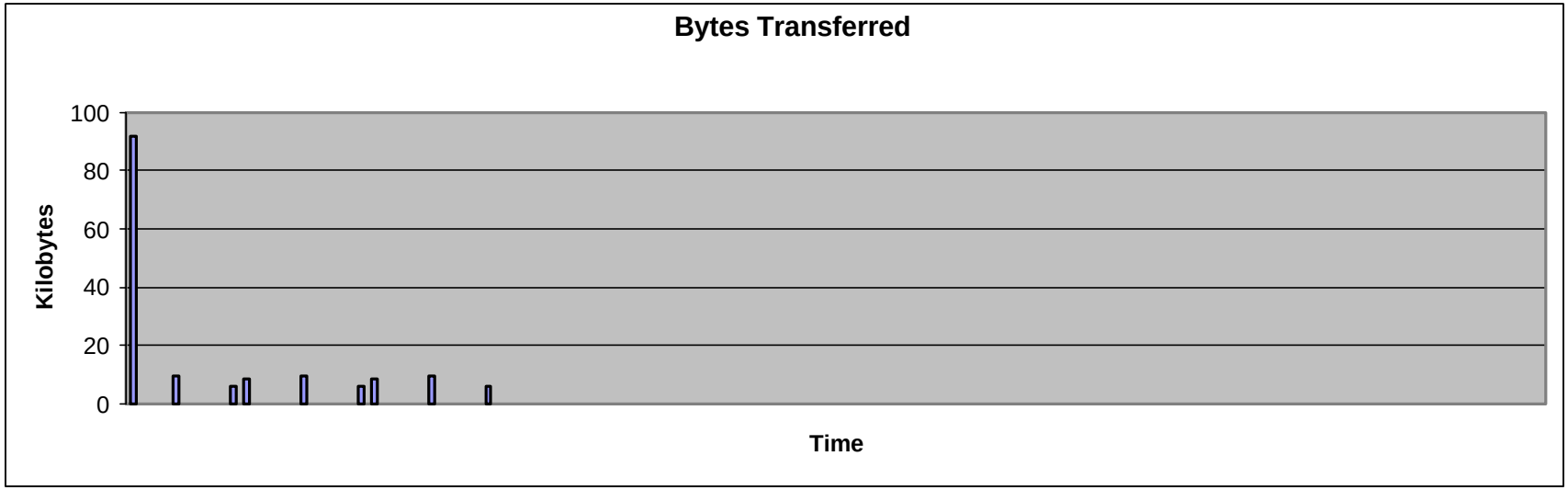


Figure 7-20: Test Case 3 with active component extraction

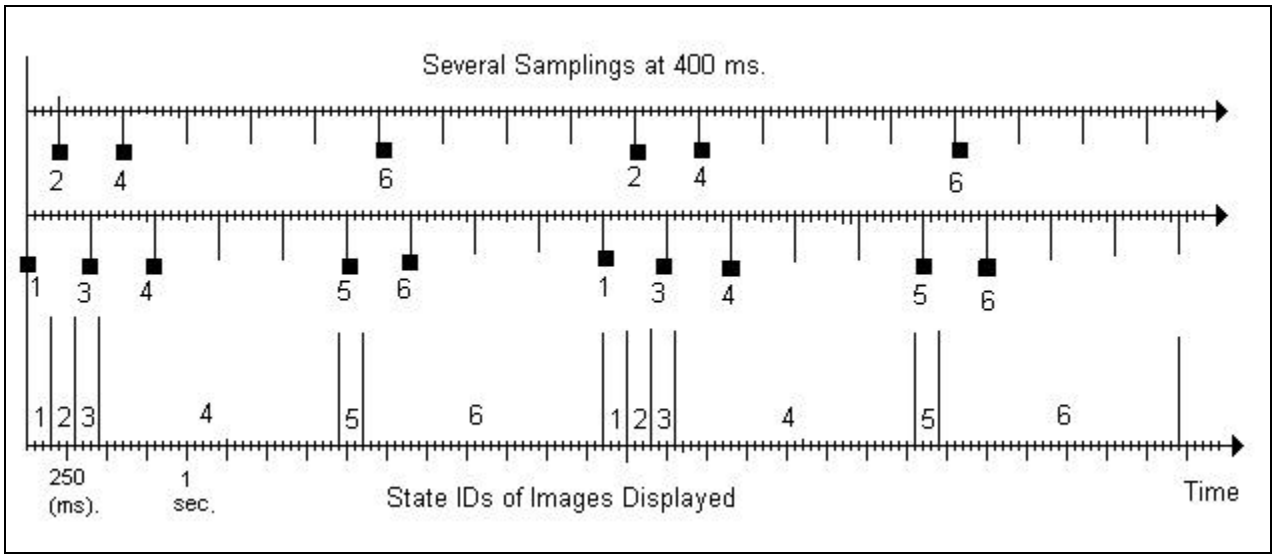


Figure 7-21: Several sampling examples for test case 3

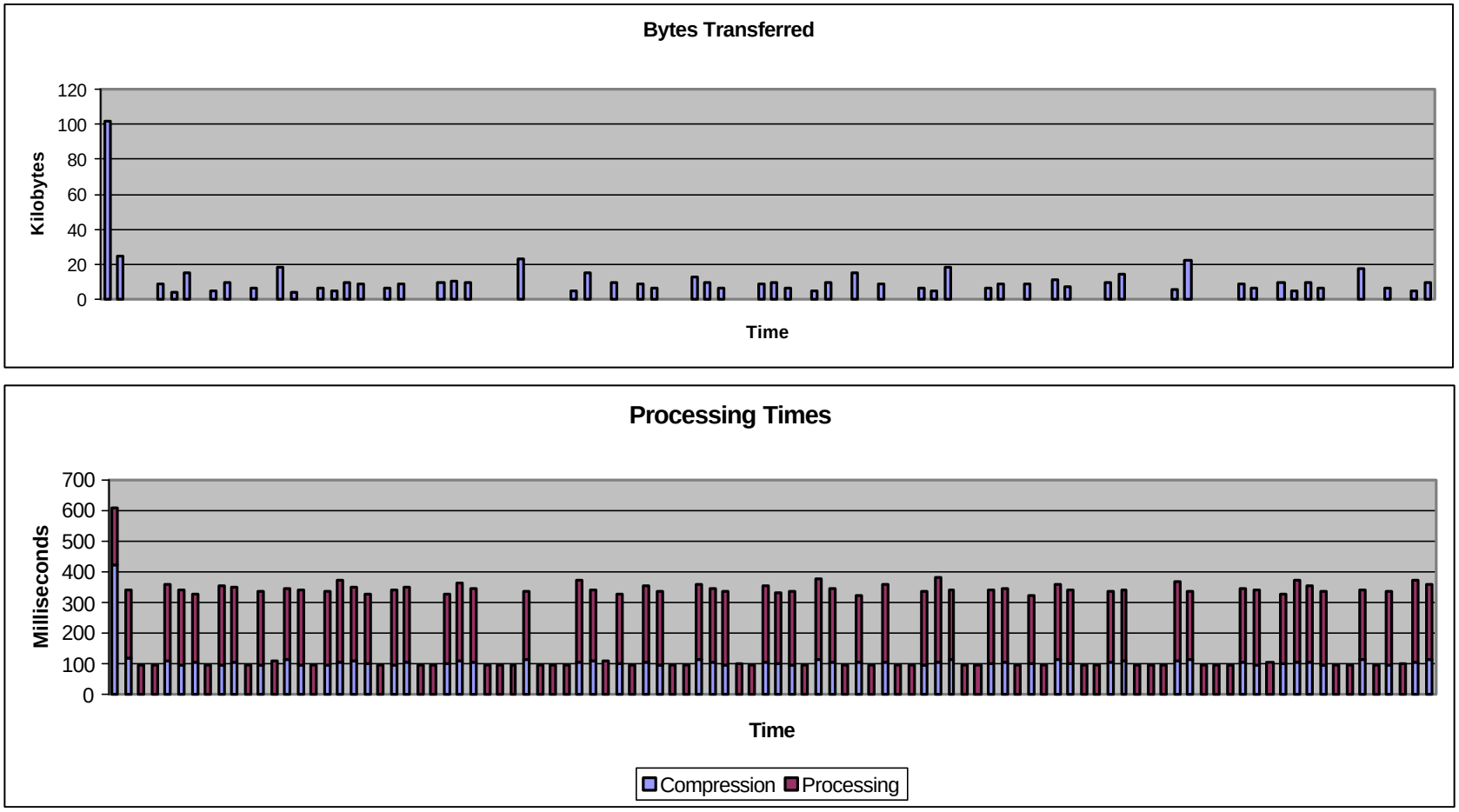


Figure 7-22: Test Case 4 with loop detection optimization

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

The work presented in this thesis, focuses on optimizations of thin client systems when the application includes multimedia components, which are called active components. Mobile thin clients, because of their nature have limited resources and must use them effectively. This thesis focused on decreasing the amount of communication (the network resource) between the server and the client by letting the client simulate some of the application content locally. Active components, which represent a recurring display structure, proved to provide significant savings on the amount of bytes transferred from the server to the client. The total number of bytes transferred has been reduced drastically.

The loop detection algorithm, which works on the whole display of the image trying to capture the repeating displays, provides improvements if the number of active components in an application is small, and state explosion – caused by different frequencies and frequency shift – could still be handled. However, when the number of active components exceeds 2 or 3, the loop detection algorithm simply turns out to be useless because the amount of buffer needed to capture every state becomes unfeasible. In such a case, the active component extraction algorithm, which tries to detect each active component separately, performs superior, capturing the active components quickly

within their first few cycles. Then those active components are passed to the client separately where the client simulates them separately, without any further communication from the server.

However, both optimizations fail if the frequency of a state change in one of the active components is smaller than the sampling rate of the server (which is about 300 milliseconds in our single threaded implementation). The server, in such a case, misses some of the states for those active components depending on the relative timing of the active component and server's sampling.

8.2 Future Work

Further improvements and research can be made on increasing the accuracy and performance of the system. First, performance improvement can be made on the processing time for a screen capture, thus the sampling rate, which will lead to better capture of the states of active components. Because the system incurs some losses for the set of active components that have faster display frequencies than servers sampling, the performance improvement will let the system detect greater number of active components completely. A multi-threaded approach can also be followed trying to capture each state of each active component, by interleaving two or more threads sampling concurrently.

More research can be conducted on the nature of active components and best ways to detect their states from bitmaps. The detection algorithm can be improved to trigger faster sampling of some sub-areas of the application display. Due to the time limitation of the thesis, these ideas were left for further research.

REFERENCES

[1] Cahners In-Stat Group, http://www.instat.com/ , Feb 2000

[2] Citrix Systems Inc, http://www.citrix.com, Jan 2000

[3] Citrix Systems Inc., "Server Based Computing," http://www.citrix.com/products/server_based_computing/default.htm, 1999

[4] "Complete thin-client/server solutions from the Windows NT and network integration experts," http://www.virtualpressoffice.com/vpo/15057/Product/153662.html , Mar 2000

[5] Cruise Technologies, http://ww.cruisetech.com, Oct 1999

[6] Forman, G. H., Zahorjan, J., "The Challenges of Mobile Computing," *IEEE Computer*, Vol. 27, No. 4, pp. 38-47, Apr. 1994

[7] Gartner Group, http://www.gartner.com, Jan 2000

[8] International Data Corporation, http://www.idc.com, Mar 2000

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

[10] Liberate Technologies (formerly Network Computer Inc.), http://www.liberate.com/ , Jan 2000

[11] MaxSpeed Inc, http://www.maxspeed.com, Apr 2000

[12] Microsoft, http://www.microsoft.com, Apr 2000

[13] Microsoft Developer Network, "High-Precision Timing Under Windows, Windows NT, & Windows 95," http://support.microsoft.com/support/kb/articles/Q148/4/04.asp, Mar 2000

[14] Microsoft Developer Network, "Using the Timer Control," http://msdn.microsoft.com/library/devprods/vs6/vbasic/vbcon98/vbcontimercontrol.htm, Mar 2000

[15] Netier Technologies, http://www.netier.com, Jan 2000

[16] Network Computing Devices (NCD), http://www.ncd.com, Feb 2000

[17] Satyanarayanan, M., "Fundamental Challenges of Mobile Computing," ACM Symposium on Principles of Distributed Computing (PODC'95 invited lecture), Ottawa, Ontario, Canada, 1995

[18] Seal, K., & Singh, S., "Loss Profiles: A Quality of Service Measure in Mobile Computing," *J. Wireless Networks* Vol. 2 (1996), pp. 45-61.

[19] Sinclair, J. T., & Merkow, M., *Thin Clients Clearly Explained*, Morgan Kaufmann Press, San Mateo, CA, 2000.

[20] Thin Planet, http://www.thinplanet.com, Jan 2000

[21] Thin Planet, "The Evolution From Host Computing," http://www.thinplanet.com/tech/generic.asp?f=TDnumber&k=s&v=TD19573, Aug 1997

[22] Thin Planet, "Thin Client Architecture," http://www.thinplanet.com/tech/generic.asp?f=TDnumber&k=s&v=TD104331, Aug 1997

[23] University of Strathclyde Computer Centre, "Introduction to X-Windows," http://mech.postech.ac.kr/X/X_course/xc.html, May 1994

[24] Zona Research, http://www.zonaresearch.com/, Feb 2000

BIOGRAPHICAL SKETCH

Cumhur Ercument Aksoy was born on October 28th, 1975, in Ankara, Turkey. He received his Bachelor of Science degree from Middle East Technical University, Ankara, in July 1997, majoring in computer science. He joined the University of Florida in January 1999 and started to pursue a master's degree in computer and information science and engineering. He conducted research at the Computational Vision, Graphics and Medical Imaging Group, Database Systems Research and Development Center and worked as a teaching assistant during his master's degree.

His research interests include mobile computing, thin clients and wireless application protocol (WAP).