OPTIMIZING THIN CLIENTS FOR WIRELESS COMPUTING VIA LOCALIZATION OF KEYBOARD ACTIVITY DURING HIGH NETWORK LATENCY

By

SIVASUNDAR RAMAMURTHY

A DISSERTATION 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

I dedicate this thesis to my parents

ACKNOWLEDGMENTS

I offer my highest gratitude to Dr. Abdelsalam Helal for providing me with the guidance and motivation to complete this thesis. I also thank Dr. Gerhard Ritter and Dr. Randy Chow for serving on my thesis committee. It makes me feel very proud to have people of their stature related to my work.

This thesis would have not been possible without the generous sponsorship of Citrix Systems, Inc., and I extend my sincere gratitude to them.

Last, but not the least, I would to take this opportunity and say thanks to my parents and my sisters. Nothing means more to me than their love, affection, and support.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT x

CHAPTERS

1 INTRODUCTION 1

- Thin Client Architecture 1
  - The Client Server Model 1
  - The Thin Client Model 2
- Advantages and Disadvantages of Thin clients 3
  - Advantages 3
  - Disadvantages 4
- Mobile Computing and Thin Clients 5
  - Relevance of Thin Clients in Mobile Computing 5
  - Benefits of Thins Client for Mobile Computing 6
  - Drawbacks of Thins Client for Mobile Computing 7
  - Real World Scenarios 7
- Need for Optimization 8
- Scope of Thesis 10

2 CLIENT SERVER MODELS FOR MOBILE COMPUTING 12

- Client Server Models with Mobile Aware Adaptations 12
  - Application-Transparent Adaptation 13
    - Coda 13
    - WebExpress 14
  - Application-Aware Adaptation 16
    - Odyssey 16
    - Rover Toolkit 18
- Extended Client-Server Model 19
  - Full Client Architecture 20

Dynamic Client-Server Architecture 20
Mobile Objects 21
Collaborative Groups 21
Virtual Mobility of Servers 22
Thin Clients 23

3 PROBLEM DEFINITION 26

Localization of Keyboard Activity 26
Fundamental Requirements of Keyboard Localization 27
Target Applications for Localization 28
Heuristics: The Definition of a KB Blitz 30
Keyboard Activity Samples 31
Key Types and KB Blitz 31
Latency and KB Blitz 32
Experiments and Evaluations 33

4 THIN CLIENT PROTOTYPE AND TOOLS 35

Microsoft's Terminal Server 35
Citrix's MetaFrame and ICA Technology 36
Citrix Virtual Channel Software Development Kit 37
Utility of the VC SDK 42
Development Environment 43

5 IMPLEMENTATION: THE *KB PRO* SYSTEM 45

Relevance of Virtual Channels 45
VdHook: the Virtual Channel Driver 46
The Shared Header File 46
The Shared Data Segment and Exported Functions 47
Keyboard Hook and Window Enumeration 49
Virtual Channel Communication Functions 51
Latency Detection Scheme 52
KBWin: The Localization Process 53
Interaction with *VdHook* during Entry 54
Interaction with *VdHook* during Exit 55
Localization Mechanism 56
KBServer: The Server Side Component 58
Design Scheme 59
Services of *KBServer* 60
Caret Position Detection Scheme 63
Summary 67

6 EXPERIMENTS 68

KB Pro Benchmark 69
Design Scheme 69
Refreshing Events 71
Keystroke Generation Scheme 72
The Stopwatch and the GUI 73
Network Emulator 76
Design Scheme 77
Bandwidth Manipulation: Future Work 79
Implementation 80
Performance Monitor 81
Experimental Variables 83
Experiment Mechanism 85
Experiment Data and Analysis 88
Time and Communication Data 89
Processor and Memory Measurements 96

7 CONCLUSION 101

Goals Accomplished 101
Future Work 102

LIST OF REFERENCES 104

BIOGRAPHICAL SKETCH 105

LIST OF TABLES

Table Page

1: Server side SDK functions 41

2: Virtual Driver API functions 42

3: Time and communication measurements without *KB Pro* 90

4: Time and communication measurements with *KB Pro* 91

5: Comparison of time and communication measurements (1) 92

6: Comparison of time and communication measurements (2) 93

7: Comparison of time and communication measurements (3) 94

8: Measurements of processor and memory utilization 97

LIST OF FIGURES

Figure Page

1: ICA Client at run time 39

2: Interaction between WinStation Driver and Virtual Drivers 40

3: Excerpts from module.ini file 41

4: *KB Pro* system architecture 67

5: A snapshot of the ICA Client log file 74

6: A snapshot of the GUI for *Bench* 75

7: A snapshot of *Bench's* GUI during a benchmark 76

8: A snapshot of *Bench's* GUI at the conclusion of a benchmark 76

9: A snapshot of the Latency Emulator GUI 81

10: A snapshot of the Performance Monitor GUI 82

11: Graph showing results of benchmark durations 98

12: Graph showing effect of refreshing events in packets exchanged 98

13: Graph showing effect of refreshing events on bytes exchanged 99

14: Graph showing average CPU utilization 99

15: Graph showing average memory utilization 100

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

OPTIMIZING THIN CLIENTS FOR WIRELESS COMPUTING VIA LOCALIZATION OF KEYBOARD ACTIVITY DURING HIGH NETWORK LATENCY

By

Sivasundar Ramamurthy

August, 2000

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

The thin client model is based on the classic client-server model with all client activity being processed at the server. Keyboard and mouse activities at the client are sent to the server, which processes the activities and sends the refreshed display back to the client. This model can be exploited for mobile computing by having powerful fixed servers serve mobile thin client devices with poor computing resources. However high latency in the wireless connection can lead to delay in display of keyboard and mouse activities, thereby affecting the performance of the thin client. This can be remedied by handling client activities locally, which will not only mitigate latency's ill effects but also reduce the volume of exchanged messages passed. My thesis focuses on localization of keyboard activities in thin clients during periods of high network latency.

CHAPTER 1
INTRODUCTION

At the birth of the next millennium, "ubiquitous computing" will become the most prevalent trend among computer users. With the exponential growth in technology and human skills, mobile computing devices characterizing lesser cost and more productivity and utility will be developed and the number of mobile computer users will increase. As this paradigm of computing continues to grow, newer problems and limitations will be encountered and efforts would have to be taken to solve them. Mobile computing, as it rightly should, serves as the main motivation behind my research on thin clients. This chapter provides an introduction to my thesis on localization of keyboard activity for wireless thin clients as a counter measure for high network latency. After a brief introduction to thin clients, the relevance of thin clients in mobile computing will be discussed along with the benefits and drawbacks. The need for optimization of wireless thin clients will be discussed before concluding with the scope of the thesis.

Thin Client Architecture

The Client Server Model

The prevailing models of mobile computing are predominantly based on the classical client/server architecture. This architecture is a logical model and not a physical implementation. It consists of three entities: the client, the server, and a communication

channel, in most cases the network (a client can be invoked on a local server). A client is a process running in a machine (part of a fixed network) that requests service of the server, which is another process running in the same or different fixed network. All the communication between these two parties is done through message passing, which is the most important aspect of client-server computing. The communication channel used in message passing is of two types: connection-oriented and connection-less. While the reliability is greater in the former, the overhead in establishing and maintaining connection is considerably high. In the latter, the reliability is low and might entail the parties of communication to retransmit lost messages. The TCP/IP and UDP protocols are examples of connection-oriented and connection-less communication models respectively. The client-server architecture requires that:

- The client is always connected to the server
- The server is highly reliable
- The Network is fast and reliable

The Thin Client Model

In the traditional client/server model, the full client is placed in the mobile host. A thin client, in contrast, is a dummy terminal that plays no role in processing application specific events. It only handles the display for the thin client process and directs all client mouse and keyboard events to the server. The server does the necessary processing, determines the new display resulting from that event, and sends the update to the client. Upon receiving these messages, the client updates the display on the host machine. Due to the contents of data exchanged, thin client communication is characterized by short messages (short requests and replies) and this allows for good performance even in a

weakly connected network. In addition, thin clients reduce the energy expended by the client machine - a process that runs in a remote machine and displayed locally requires considerably less processing power than a process that runs completely locally. This advantage, coupled with the ability to perform under weak connections, makes this model suitable for mobile computing. Examples of thin clients include the X-terminal and Citrix's ICA Client University of California, Berkeley has come up with their thin client system called InfoPad that is mobile aware: this is done by adding software layers transparent to the thin client application [1].

Advantages and Disadvantages of Thin clients

Advantages

The advantages of using thin clients range over a wide domain of computing issues. As far as system requirements are concerned, thin clients can run on machines with very little RAM and a relatively slow processor. For instance, Citrix's ICA Client requires the equivalent of an Intel 286 processor and access to a minimum of 640k of RAM to operate [2]. The advent of thin clients also means that less-powerful machines that have become obsolete now can be brought back to use and machines built specifically for use with thin clients can be made extremely compact and inexpensive. All the above issues also lead to an advantage that assumes greater importance in machines meant for mobility (like hand-helds, palmtops, etc). Since these machines can be designed with low-speed processors, little RAM, and virtually no hard disk (except for the underlying OS), the battery power requirements reduce considerably. Some of the other benefits of thin clients are [3]:

- Access to Operating System specific applications can be given to thin clients regardless of the client hardware and platform
- Applications can be scaled, deployed, managed and supported from a single location (the server)

The next issue we consider is network connectivity. In the thin client model, the client and the server typically communicate through bursts of short messages; all keyboard and mouse events at the client trigger a message to the server and most messages sent to the server trigger a reply to the client. Since the messages are short (they consist of only keyboard and mouse messages, and display updates), the bandwidth required for communication can be as low as that provided via a modem. For instance, Citrix's ICA Client can function in bandwidths as low as 20kb [2].

Disadvantages

The main disadvantages of thin clients stem from the fact that constant network connection is required for functioning. The most obvious of these is that thin clients are not designed to operate in the disconnected mode. Also, thin clients perform poorly in networks with extreme delays and latencies; display of animation and of user's typing and mouse activities may not be done in a time period that keeps the user comfortable and satisfied with the system. This degradation in performance is highly probable when the communication is via wireless networks with high latency.

Mobile Computing and Thin Clients

Relevance of Thin Clients in Mobile Computing

As more and more computer users realize the advantages of mobile computing and embrace it, needs arise for designing efficient and effective mobile computing systems. Power, machine resources, and network connectivity, the three factors we have taken for granted in the context of fixed machines, assume completely different roles in mobile computing; they are limited, precious, but at the same time, indispensable. Designing efficient mobile computing systems requires total recognition of the importance of these factors.

Power requires most attention since mobile machines are getting smaller and more compact and are fitted with smaller batteries with shorter life. And it is this battery that powers not only the operating system and the applications but also the communication device! It is natural that computing resources like hard disk space, RAM, and CPU speed reduce with the size of the battery. Mobile computing systems need to be developed in cognizance of this factor too.

With the exponential growth in the need for information, Internet connectivity will play a major role in mobile computing. People would like to read and send email, check the scores of their favorite sports, finish office work, etc, while on the move. Also, business consultants might want to connect to the network at the headquarters while working on-site. This need for Internet connection brings in new problems, not only for mobile machines, but also for mobile networks. As for the machine, battery that is already being exhausted on running the application (email client, the web browser, or the word processor) will now have to power the communication system also. This means expending additional power for running the wireless device (like PC-cards, wireless

modems) and for sending messages through these devices! As regards to mobile networks, a need arises for building efficient protocols that deliver messages to the right machines, do it with as little latency as possible, and, most importantly, extract as little energy from the mobile machines battery when the machine sends or receives a message. The next two sections describe the advantages and disadvantages of using thin clients for mobile computing in the context of the three factors discussed above.

Benefits of Thins Client for Mobile Computing

Thin clients, like Citrix's ICA Client, can be employed for mobile computing to execute applications on a remote machine while the mobile machine manages just the display. The amount of RAM required to execute the application is effectively the amount required for the display. The processing power required for these applications is effectively the amount required for running the display. Also, these applications need not be installed locally and this saves hard disk space.

With the decreasing use of the CPU, the RAM, and the hard disk, the battery power required to use the three resources also reduces. This effectively enables the mobile user to increase the life of the battery and work longer before a recharge. These benefits can also be exploited to build mobile machines specifically for use with thin clients. The resulting design can be extremely compact with the CPU size, hard disk space, and RAM just about enough to run the thin client.

The amount of communication usage is also kept to a minimum with thin clients. Even though the client host needs constant connectivity, the messages sent are short and represent only of keyboard, mouse, and display update data.

Drawbacks of Thins Client for Mobile Computing

The drawbacks of mobile thin clients stem from the fact that constant network connectivity is required for functioning. First, wireless communication is at its embryonic stage: wireless protocols are still under development and service providers are able to charge high amounts per packet communicated. Two, wireless communication is inherently slow and the bandwidth available in most commercial wireless Wide Area Network connections is extremely low compared to Ethernet. And given that any mouse or keyboard activity results in a message being sent to the server, running thin client applications involving high volume of user activity might be expensive in two ways. The power supply is continuously drained while sending and receiving messages and the wireless network bill of the user will shoot up!

Real World Scenarios

The following scenarios exemplify the pros and cons of using thin clients for mobile computing:

- Execution of a web browser as a thin client as opposed to a local process

Benefits. The burden of using expensive protocols to communicate with the web server is now shifted to the thin client server and the server sends the thin client only the display and not the data downloaded.

Drawbacks. Network connectivity is required even while browsing "off-line" and high latencies can slow down the display of updates.

- Execution of a word processor as a thin client instead of a local process

Benefits. The documents created by the user need not reside locally; this increases universal access to documents and saves hard disk space on the mobile machine

Drawbacks. Network communication is needed for a process that is typically Internet isolated! Every keyboard or mouse action, the former of which is relatively high for word processors, results in a message being sent to the server with a reply coming back for the refreshed display. This not only induces needless increases in the users wireless network bill but also eats up the valuable battery supply. Also, speed of work is affected when latency slows down display of keyboard and mouse activity.

Common Benefits. The benefits of thin clients in both scenarios include:

- The previously discussed advantages of thin clients, such as centrally deploying and updating applications
- The application's executable need not reside on the client machine and it does not execute locally.
- Eventually, hard disk space is saved, RAM requirements and utilization reduce, and universal access to documents increases

Need for Optimization

For reasons related to cost and loss of performance in slow connections, thin clients are certainly not the perfect system for mobile computing. Assuming that the cost of wireless communication is a secondary issue, a question arises on how thin clients can adapt to perform better under high latency. Low bandwidth's effects can be mitigated based on the thin client architecture; thin clients can be designed to optimize data exchange between client and server. For instance Citrix's ICA Client sends only keyboard and mouse data to the server and the server replies with the new display and this system can perform in connections as low as 20 KBPS. However, no thin client

system can be designed to counter latency's ill effects; latency can delay handling of activities like displaying typing and mouse movements and thus reduce performance levels. The satisfaction experienced by thin client users in networks characterized by high latency and delay may be low. For example, a user working on a document using MS Word on a thin client may not see the display of characters typed within a reasonable amount of time. This slows down the user and as a result increases the thin client session duration! Another example would be when the user downloads web pages with animation or video but finds the thin client too slow in displaying the animation smoothly. These events may lead to the user getting uncomfortable and frustrated with the system. In order to remedy this, mechanisms would need to be developed to counter the ill effects of latency.

One remedy is to localize events that may get affected by various degrees of latency. For instance, no matter how fast the network is, displaying video by processing each and every frame at the server and then sending the updated display to the client will not produce good quality video. But if video is processed and displayed locally, like Citrix's VideoFrame enhancement to the ICA Client, the performance would certainly be much better. While video is an example of a process that should be localized for networks with low delays, even simple events like typing can become tasks to localize in high latency networks. In addition, localization of activities can reduce the amount of communication between the client and the server. With the high cost of wireless connections, this may be highly desirable for mobile users!

As one might have observed by now, localization of events does go against the definition of thin clients. With localization, the thin clients get fatter, which means that

more burdens will now be placed on local resources than before. The client machine that was previously acting only as a display will now have to utilize more CPU and memory to help localization. A localization process needs to be deployed in the thin client and this may not need different versions in order to localize typing on all applications. Problems regarding version updates and control may arise with this installation. These factors can trigger off an argument that thin clients performing localization may not only unsuitable for mobile computing but also kill some of the benefits of true thin clients. However, this may be valid little if the benefits obtained via localization outweigh the drawbacks and the increase in user satisfaction compensates for the loss of thin client benefits. In summary, genuine thin clients are useful since they place very little burden on client machine's resource but their performance can get affected by latency, a common characteristic of most wireless networks. Mobile machines that work as stand-alone PCs rely very little on communication but require more processing and memory resources and application management. In between these two ends of the spectrum is the thin client with a lean localization system that succeeds in maintaining the thin client benefits even while restricting the burden on local resources.

Scope of Thesis

The scope of this thesis is restricted to building a foundation for localization of keyboard activity in wireless thin clients. Given that this subject of research is at the rudimentary stages, only word processing applications will be targeted due to their suitability towards localization. The system developed will be tested with various benchmarks and network latencies and the measurements will be analyzed. The next

chapter is a survey of literature in the subject of mobile computing and chapter 3 describes the problem definition and thought process undergone in its refinement. While chapter 4 provides a brief description of the thin client prototype and the tools used for creating the localization system, chapter 5 is an in depth discussion of the localization system implementation. Before concluding with chapter 7, chapter 6 elaborates on methods developed to experiment with the localization system and to evaluate it.

CHAPTER 2
CLIENT SERVER MODELS FOR MOBILE COMPUTING

Developing software systems that are motivated by mobile computing must be founded on a survey of its literature. This chapter describes client server models for mobile computing that have been strongly influenced by two factors that characterize mobile computing: one, the mobility of the user and the machine; two, the resource constraints such as limited battery life and limited bandwidth [1]. Client server models with adaptations will be discussed first followed by the extended client server models. The chapter will conclude with a brief introduction to thin clients and a case study of InfoPad, a thin client system from University of California at Berkeley. Citrix's ICA Client is not discussed here since Chapter 4 is devoted to describing it.

Client Server Models with Mobile Aware Adaptations

Mobility increases the tension between interdependence and autonomy that is a part of all distributed systems [4]. One can argue that mobile machines should rely on robust servers since they are resource poor. However, the argument for independence also becomes strong since bandwidth available for communication is limited! Given that the physical conditions of mobile computing will never be static, client-server computing must adapt and reassign client and server functionality for better performance. This adaptability can be made without any system support, can be left entirely to the system, or anywhere in the spectrum formed by these two extremes.

Application-Transparent Adaptation

Application transparent adaptations are aimed at making applications work in the mobile environment without modifications. The system is entrusted with the responsibility to take required measures to counteract environmental changes like location and connection. A typical adaptation of this kind is the introduction of a third party that adds mobility awareness between the client and the server. This entity, called a proxy, can be viewed as a client to the server and as a mobility aware server to the client. The designer of this model should decide whether to place the proxy in the server, the client, both the client and the server, or the network. A couple of applications that are based on this model are Coda and WebExpress.

Coda

Coda is a file system for a large-scale distributed computing environment composed of Unix workstations. It provides resiliency to server and network failures through the use of two distinct but complementary mechanisms, namely server replication and disconnected operation [5]. Server replication involves storing copies of files at multiple servers – the replication is usually aggregates of files called volumes. Disconnected operation is a mode of execution in which a caching site temporarily assumes the role of a replication site. The client a cache built based on the user profile. A file system proxy called Venus emulates the file system services in order to hide mobile issues from the mobile computer. It pre-fetches the files when it senses disconnection (data hoarding), logs the client's requests during disconnected mode (server emulation), and synchronizes the local cache with server (reintegration).

In the process of hoarding a file, the proxy inquires all the available servers about the file and receives the most recent version. After modification, the file is stored at all server replication sites that are currently accessible. To achieve good performance, Coda exploits the parallelism in network protocols; for instance, parallel RPC mechanisms are used to simultaneously transmit files to multiple sites.

Consistency, availability and performance tend to be mutually contradictory goals in a distributed system. Two principles guide the design of consistency mechanisms in Coda. First, the most recently updated copy of a file that is physically accessible is always used. Second, inconsistency must be rare and always detected by the system (even though it is tolerable). Coda uses atomic transactions at servers to ensure that the version vector and data of files are mutually consistent at all times.

In order to overcome the data inconsistency problem resulting from partitioned sharing, Coda employs isolation-only transactions (IOT). IOT is a consistency model that uses serializability constraints to automatically detect read/write conflicts. It provides a set of options for automatic and manual conflict resolution, and to incorporate application specific knowledge to detect and resolve conflicts. The IOT mechanism is provided as an optional file system to preserve upward Unix compatibility; it allows any Unix application to be executed as an IOT with its flexible interfaces being an added advantage. Unlike traditional transactions, it does not guarantee failure atomicity and guarantees permanence only conditionally.

WebExpress

Web proxies are employed to enhance web browsing over wireless links without compelling server or browser changes. Web proxy can be used to pre-fetch and cache

Web pages to the mobile client's machine, compress and transform image pages for transmission over low-bandwidth links, and support disconnected and asynchronous browsing operations [1].

WebExpress is a system based on this approach. WebExpress controls the wireless communication and optimizes the protocol in order to reduce traffic volume and latency respectively. It achieves this by employing two processes: one is the Client Side Intercept (CSI) that runs on the same mobile device as the client and the other called the Server Side Intercept (SSI) that runs within the fixed network. The idea is that the CSI would intercept HTTP requests, perform optimizations with the SSI, and reduce data transmissions. The whole process of handling an HTTP request is done as follows. The browser communicates with the local Web proxy, the CSI, over a local TCP connection using the HTTP protocol (the browser has the address of the CSI). The CSI, using a TCP connection, communicates with the SSI using a reduced version of the HTTP. The SSI reconstitutes the HTML data stream and sends it to the CSI, which then forwards it to the Web browser through the local connection.

Some of the optimization methods used by WebExpress involves caching graphics and HTML objects by the SSI and the CSI, protocol reduction in terms of decreased connection establishments, and header reductions with respect to maintaining client information at the SSI throughout the connection. Since WebExpress provides transparency to both the browser and the server, it can be used with any browser. The CSI/SSI protocols facilitate highly effective data reduction and protocol optimization without limiting any of the Web browser functionality or interoperability [1].

Application-Aware Adaptation

Application-aware adaptation enables applications to react to changes in the mobile computing environment via collaboration between systems and individual applications. While the system monitors and notifies resource levels and allocates them accordingly, the applications decide how to adapt when notified of changes. This division of responsibility allows for diversity and concurrency; each application decides how to present data based on resources allocation and at the same time the system monitors resources and decides on allocation. And instead of treating data generically and forwarding them to the applications, support for type-specific operations at the system level may be required to optimize performance. For instance, it may be desirable that the system knows that data from an NFS server differ considerably in consistency requirement from data of relational database records. A couple of systems that incorporate application aware adaptation are Odyssey and the Rover Toolkit.

Odyssey

Odyssey is a system designed with the motivation to successfully adapt to mobility, and application concurrency and diversity [1]. It is an application-aware adaptation, which includes a set of extensions to the NetBSD file system, and allows mobile clients to access data from fixed servers. There are a number of factors upon which Odyssey's design is based. First, it is a realistic possibility that mobile clients can only accept data that are degraded due to dwindling resources; Odyssey defines fidelity as the degree to which the data presented to the client match the server's copy. Second, concurrency of operations can provide more benefit to the user, as in the case of stock

market watching programs running in the background and alerting the user when needed. This implies that a centralized mechanism of managing the machine's scarce resources would be ideal. Third, successful adaptation requires quick detection of and response to changes in the mobile environment that is bound to happen; Odyssey defines this property of the system as agility [6].

Odyssey's approach to adaptation is application-aware and applications decide how best to adapt to changes in resource levels that are reported and notified by the system. The system considers data generically and applications would handle different data types. However, the disparity in logical and physical properties of data types asks for some type-awareness in the system. This requires that Odyssey not only include type specific knowledge in order to perform correct resource management but also awareness of shared data access by concurrent applications in order to handle resources effectively. Thus Odyssey employs two entities: a Warden that supports a particular data type and, its superior, the type-independent Viceroy that centrally manages resources. The relationship between the Warden and the Viceroy is data-centric and fidelity levels influence data management.

Operations on Odyssey objects are performed the following way: Wardens run along with the Viceroy in user space and the unit executes in a single address space via user threads. Operations on Odyssey objects are routed to the Viceroy through an interceptor module in the kernel. The Viceroy and the Wardens communicate via shared data structures and procedure calls. The Wardens are entirely responsible for contacting the server for data as and when needed – the applications never access the server directly.

Applications use the *request* call to communicate resource expectations with Odyssey. A resource descriptor identifying a particular resource and a window of

tolerance on its availability can be specified as arguments in this call. If the request can be handled, the Viceroy registers it and returns a request identifier. This identifier can be used by the Viceroy to notify the application that the resource has left the requested bounds, and by the application to cancel the request [6]. An error code along with the current resource levels is returned if the request cannot be handled and the application is expected to try again with a different window of tolerance.

When resource levels stray beyond the specified window, the Viceroy uses an *upcall* to inform the application, based on which the application issues a fresh request with a revised tolerance window. The request identifier, the requested window of tolerance, and the available level of tolerance are provided as parameters in the upcall. In order to change fidelity, type-specific-operations (TSOP) are used as general purpose escape mechanisms [6]; these calls take as arguments an Odyssey object and the *opcode* of a type specific operation to be performed on that object.

Rover Toolkit

Based of the flexible client server architecture and the mobile object model, Rover provides a framework to construct mobile applications using two approaches, Relocatable Dynamic Objects (RDO) and Queued Remote Procedure Calls (QRPC) [1]. This system can support both application-aware and application-transparent adaptations and is particularly beneficial in the disconnected mode.

In the traditional client-server model, an object resident in a server that needs modification requires the client exchanging messages with the server and the server modifying the object. In the Rover system, the object is imported from the server using a fetch-request, modified the client, and eventually returned to the server. Upon receipt, the

server must resolve conflicts with its version of the same object and then make permanent changes to it. Rover refers to such objects as RDOs.

In the connected mode, the above process can be accomplished using Remote Procedure Calls (RPC). In the disconnected mode, however, it is necessary to use a mechanism that allows for asynchronous importing and exporting of objects. Rover enables this by providing the QRPC mechanism that is a lazy and non-blocking RPC. Here, communication that takes place in the disconnected mode is queued in a stable log at the client. The queued information (objects or requests) is cleared from the log and forwarded to the server upon reconnection. At the server side, every incoming request and outgoing response is logged and maintained by the Rover Access Manager. This enables the server to handle multiple object requests and to redeliver logged responses that were not sent due to disconnection. Additionally, the Access Manager also maintains an object cache, which enables the Access Manager to intercept object requests and handle them locally if possible.

Extended Client-Server Model

Apart from the two adaptations discussed above, another modification to the classical client-server model is the extended client-server model. Here, the static partitioning of the client and server that suit the fixed environment is changed to suit the mobile environment. A server will be able to perform as a client since certain client operations may need to be performed in resource rich servers. Also, the client can become a server if needed to counter the vagaries of mobile connectivity. Mobile computing architectures that are based on this model are thin-clients, full-clients, and dynamic client-server systems.

Full Client Architecture

Since mobile environments are characterized by frequent disconnection, the disconnected operation model or the full client model is utilized to enable continued functioning. The approach here is to place an entity in the mobile platform along with the client that emulates the functioning of the server during the disconnected mode. Upon reconnection, the requests handled during that period are synchronized with the home server using two means. One, a lightweight server is placed on the mobile platform with a replica of the data. During disconnection, this server handles requests on behalf of the main server and resolves data conflicts with it on reconnection. Two, a mobility agent may be employed to buffer requests during disconnection mode and to notify the home server upon reconnection. While Coda and Oracle Lite implement the former, Oracle Mobile Agents implement the latter [1].

Dynamic Client-Server Architecture

Here, the strict definitions of the client and the server are relaxed in terms of location in order to improve performance and availability. Servers or their thin versions dynamically relocate between mobile and fixed hosts. In addition, proxies can be dynamically created and relocated as part of this architecture. This model generalizes both the thin-client and full client architectures [1] and there exits a spectrum of adaptation and design possibilities within it.

Mobile Objects

Mobile Objects are programmable entities that can freely roam the network and allow clients to download the server code for execution. These objects can maintain state information that enables them to suspend execution at any arbitrary point and migrate to another location and resume execution. This information along with awareness of the mobile environment gives the object ability to make intelligent decisions regarding which host to migrate to (especially during network partitions) and what functions to perform at each host. The intention of this approach is to overcome the problems of weak connection and frequent disconnection. The Rover Toolkit, which employs Relocatable Dynamic Objects (RDO), is an example of such a model [1].

Collaborative Groups

In Collaborative Groups, disconnected mobile clients can form a group of servers and clients, collaborate, and exchange information through an ad-hoc network. The adaptation to the classical client-server model is that clients with data can become servers to disconnected clients who need the same data. The functionality of client and server is redefined: any machine that holds the data is a server and any machine that needs the data and can communicate with the server and access the data is a client [1]. The Bayou system is an example of this architecture.

Consistency and availability are two contradictory requirements in any distributed system. The more the number of replicas, the more difficult it is to reach an eventual consistency. The Bayou System is a platform of replicated, highly available, and weakly consistent mobile databases on which collaborative applications can be built [7]. A server

in Bayou is any machine that holds a database and a client is any machine that is capable of contacting the server and accessing the information controlled by the server.

One of the prime features of Bayou is its application-specific conflict detection and resolution mechanism; every write operation is accompanied with dependency checks and merge procedures. A dependency check consists of a query and the expected results to that query submitted to the server. Before committing the write operation, the server must check whether the result of the query and the expected result match; a tentative commit of the write operation is made if they match, else a conflict is detected and the dependency check is said to have 'failed'. The merge procedures submitted with every write are executed in the event of a dependency check failure. The merge procedures may be an alternate set of updates that are apt for the current database state. Merge procedures resemble mobile agents since they are downloaded from the client to the server and are executed in the server [7].

Bayou manages replicated copies of the database and thus arises a need to maintain consistency among the replicas. Servers propagate their writes to other servers using an anti-entropy protocol, which is a synchronization process of updating the replicas. Peer-to-peer anti-entropy sessions lead to eventual consistency among all the replicas. Version vectors or timestamps are used to detect write-write conflicts.

Virtual Mobility of Servers

This approach is based on the notion that servers located closer to mobile clients can be accessed easier as the latency for remote operations will reduce. Relocation of the server in response to client movement may be desired hence. In this model, the relocation of the server is virtual and the server code does not move with the distributed information

system. The client, who is aware of the existence of multiple servers but not necessarily their location, needs to determine the nearest server when visiting new domains. This is accomplished as follows: the client requests a near by machine to provide the services it requires; a consenting machine, referred to as the coordinator, will provide the service using the data in a near-by or local site. This process continues as the client ventures into new sites.

An obvious disadvantage of such a model is the overhead in synchronizing the multiple servers that may be widely spread. As a client roams and interacts with new servers, the interaction among these servers in turn will grow causing the overhead of global communication to sharply increase [1].

Thin Clients

The thin client architecture is developed mainly for dumb terminals or small PDA applications [1]. Except for the display, the model offloads functionality from the client to the stationary hosts completely. Citrix's ICA technology converts any lean or fat client to a thin client and its MetaFrame server provides Microsoft's Terminal Services through it. Other thin client systems include X-Terminals and Berkeley's InfoPad.

InfoPad is a current project in the Electrical Engineering and Computer Science Department at the University of California, Berkeley [8]. The project’s goal is to provide multimedia support in a mobile environment through a portable terminal called the Pad. The architecture is based on the thin-client model and all computations to make the Pad functional are done on a fixed network, thereby placing no computational resources on the Pad. A number of Pads are grouped into cells and each cell has a base station that is connected by wire to the fixed network and through a radio frequency to each Pad in its

cell. All data packets are sent from the backbone network and routed to the destination Pad via the base station. It is interesting to note here that the system does not give too much emphasis on security and that it assumes that the environment is trusted [8].

Software systems that are developed for the InfoPad system must have the above topology and should also guarantee other features like:

- Transparency to enable use of standard desktop applications
- Scalability to support more users and applications
- High frequency of cell locations and optimal allocation of power
- Mobility of the Pad to ensure transparent, smooth cell transition

InfoNet is a software system that has been built satisfying the above requirements of InfoPad. It is composed of six groups of software elements [8]:

- The Application, like speech recognition and video playback, that provides data streams to the Pad
- A Pad server that controls access to the Pad and allocates resources and bandwidth among applications
- A Cell Server, associated with each cell, that allocates resources to the Pads and interacts with other servers if needed
- A Gateway, which is associated with cell, that connects the cell's wireless network to the backbone network, such as Ethernet or Asynchronous Transfer Mode (ATM)
- The Network Controller, which enables creation of connections among the other InfoNet entities
- The Pad, a simple multimedia display device whose computations and processing is done remotely

Some of the basic activities the system must support are related to the following situations:

- Activation of a new Pad: the system must provide means for users to login, request services, suspend operations, and resume previously suspended work. In order to activate the Pad, a new Pad server or an existing one must be connected to the Pad.
- Application initialization: upon execution, the application should create a connection to the Pad with a specified Quality of Service (QOS). In order to do this, connections need to be created among the Application, the Pad Server, and the Gateway.
- Mobility: when Pads move from one cell to another, the transition from the current cell's Gateway to the next cell's Gateway, referred to as handoff, must be handled smoothly.

CHAPTER 3
PROBLEM DEFINITION

Latency in the communication channel used in running a thin client can affect performance in a number of ways. Keyboard events may not be refreshed promptly, mouse movements may be jagged and delayed, and graphics and animations may not appear as smooth as they would be in a faster network. Given that high latency is inevitable in wireless networks, the activities causing the above blemishes become potential targets for localization. While my colleague Cumhur Aksoey chose to perform his research on localizing animated GIF files, my work is restricted to localization of keyboard activity. This chapter defines the problem this thesis attempts to solve and explains the thought process undergone in refining it.

Localization of Keyboard Activity

Optimizing a thin client though localization of activities should approximately balance the exhaustion of the mobile machine's energy with the increased performance experienced. Since user experiences are subjective, it would be appropriate that localizations require little processing and memory resources. One of the localizations that can be performed to enhance performance is handling a phenomenon termed Keyboard Blitz (KB Blitz). A simple definition of a KB Blitz is when the thin client user types in such a speed that display of the characters is delayed for reasons such as a slow network

or a slow server. As part my work, I intend processing keyboard activities locally whenever the network speed does not enable the prompt display of keys pressed. The motivation is that the user will experience better response to her typing and, as we will see during experimentation, the burden on the communication channel will reduce considerably. As a note, future work on this optimization should widen the definition of the KB Blitz to include server overloading and other relevant factors.

The simple problem definition above makes the problem domain too general and expansive for research. Since this is one of the first attempts at localization for thin clients, the target scenario for localization needs refinement. The main goal of this project is not only to display typing activity promptly but to also keep the overheads at the minimum, keeping in focus the theme of thin clients. This goal will be used as the foundation for designing the various features of the localization system.

Fundamental Requirements of Keyboard Localization

The following have been identified as the fundamental features of keyboard localization:

- Except for subtle differences, the display of the localized typing activity must resemble the corresponding display when handled by the thin client server
- Transition from normal mode to local mode, and vice-versa, must have minimal interference on user's activity. The user may be given cues during this period to indicate transition
- The user should be made aware localization through subtle features. This is important since there will be some degradation in terms of typing activity display and some keyboard events will be handled differently than during normal mode

- The localization system should strive to implement as much of the processing as possible at the server side. Only those functions that cannot be done so should be left to the client machine. This is necessary since most mobile machines are resource poor
- All typing activity that can be handled locally shall be done so. Key events that require special processing, like function keys and control combination keys, must result in a refresh and return to normal mode. These key events must be handled by the localization system as much as possible; that is, the refreshing event should take place without entailing the user to type the keys again.
- The frequency of shifting between local and normal modes should be controlled; it must not reach a level where user productivity actually starts decreasing and the client is slower than in normal mode
- When localization terminates, all typing activity handled locally must be refreshed with the server before the user continues using the client

Target Applications for Localization

Given these building blocks, a quick observation of typing behavior on heterogeneous applications would give us more principles to abide by. Typing in word processing applications, or what I have termed "typing applications," is most suited towards localization. There is little editing, multiple lines of text are typed using the enter key, and there is little logic involved in processing a key press. For instance, an enter key pressed in a web browser may result in a download or a movement to an arbitrary location on the page. An enter key pressed in a typing application like MS Word in most cases would result in a move to the next line and the resulting display can be handled reasonably accurately even in local mode. *The key point is that the chances of frequently*

shifting between local and remote modes are less with typing applications; most keys pressed in typing applications actually get displayed and those that need logical processing are few, can easily be enumerated, and the mechanisms to handle them during localization can be designed.

At this point in the discussion, I would like to point out that keyboard activity in all applications could be localized. The localization can be restricted to displaying text at approximately the "correct" place with even key events such as enter, tab, etc resulting in a refresh. But this rigid enforcement of keyboard localization can actually hinder rather than aid performance. For instance, if the user types in a ten character URL in a web browser application and this typing is localized, the amount of time and resource overhead needed to start and end localization might not make localization worthwhile even during high latency. With these additional observations, the fundamentals are further refined as follows:

- Only typing applications shall be targeted for localization and even extensive typing on other applications will be ignored
- The localization system must observe typing behavior of the user and should begin localization only if the behavior matches a KB Blitz (to be defined later)
- The localization display can be multiple lines, with the width of the window being the width of a local line and the height of the window being the number of lines that can be locally typed without a need for a refresh. The width of the localization area is the width of the application window and the height is the part of the window from the typing position at the start of localization to the bottom of the window. This rule will

help reducing the number of transitions between modes, which may be high when the localization is restricted to one line of typing

- Moving the caret within the localization area using the keyboard or the mouse is permitted
- If the user moves the caret to any portion of the application that is not in the localized area, a refresh should occur and the eventual display must reflect the user action. Any mouse click that is not within the localized area should also result in a refresh

Heuristics: The Definition of a KB Blitz

This section describes the heuristic issues involved in defining the KB Blitz, a term very important to the localization system. In order to deem the keyboard activity in typing applications as a KB Blitz, samples of the keyboard activity will undergo two tests. One, the type of keys in the sample must provide a good indication that forthcoming typing can be effectively localized. Two, the speed at which the user types in the keys must be compared to the network latency to determine if localization is necessary for prompt display. The first rule is substantiated by the fact that certain keys cannot be locally handled, and by the principle that frequent transitions between local and normal modes must be avoided. The second rule is to determine if the typing speed requires localization at all! Samples will be put to the second test only if they pass the first one

Keyboard Activity Samples

The main trade-off in testing the typing behavior for a KB Blitz is between the amount of time spent in monitoring the typing behavior and the amount of typing activity that is not localized because of monitoring. In order to make the KB Blitz test efficient and successful, the typing sample used will be the fifteen most recent keys typed by the user. While these keys give a good representation of the user's current typing mood, the sample length is also ideal for testing. It is long enough to ensure the speed of typing and the consistency in the types of keys pressed, and it is short enough to start localization as quickly as possible. It is worth noting that no sample length can make the tests fail-safe since only the user knows when she will shift from a KB Blitz mood to some other behavior!

Key Types and KB Blitz

A typing sample represents a KB Blitz only if it consists of keys that can be processed with this simple rule: the characters they represent can be displayed at the location where the user wants them to be displayed and there is no other logical processing. For most typing applications, these characters are the letters in the alphabet, special characters, numbers, and white spaces. The sample cannot be a KB Blitz if any other key event is interspersed in it. The heuristic is that samples that pass this test give a good hint that the user will continue with the same typing pattern. Thus, once localization starts and the user continues with this pattern, it will end only when the user reaches the end of the localization area. Any sample that fails this test is not a KB Blitz: the user's typing will continue to be monitored and the next sample will be put through the same test.

Mouse events within the typing applications may also be included in the list of events that disqualify a typing sample from being a KB Blitz. This will bring the current definition of a KB Blitz more in accordance to the principle that only suitable *typing* activity will be displayed locally.

Latency and KB Blitz

Latency in the communication channel that is below a certain threshold may not require localization of keyboard activity. Simple tests have shown that any latency of 200 ms and below is enough for the thin client to display typing reasonably quickly. While there is a short time lag between the key press and the display, latency of 200 ms seems to be a good upper limit for the "comfort zone." The localization system will follow this rule and not attempt localization when latency is below 200 ms. However, even if the network latency is above this threshold, localization may not be required if the user's typing speed does not warrant it. For this purpose, the average round trip time for a packet from the client to the server is calculated; this value is deemed to be the response time of the server and half of it is assumed to be the approximate network latency. Since latencies vary constantly in wireless networks, this process may be repeated periodically to keep the response time and latency values updated.

As for the test itself, only samples that pass the key types test will undergo it. The time frame of the sample is compared to the product of the response time and the length of the sample. If the sample's time frame is lesser, the sample passes the test and the user's typing behavior is deemed a KB Blitz. This test is also heuristic since the response time measurement process involves uncontrollable variables that can cause flaws. For instance, the actual response time may be lesser if the party at the server that participates

in the response time measurement is slow to respond to the Pings used in the process. On the contrary, it is also possible that the actual response time of the typing application is higher than the response time calculated.

Experiments and Evaluations

With the principles and rules defined above, the localization system can now be developed as software. There remains one final problem whose solution is undefined and it involves the scheme used to evaluate the localization system. The motivation of the thesis will be unsatisfied if means are not devised to measure the viability of the localization scheme. Experiments would have to be performed and their results will be compared to show the utility of the localization under various conditions. This process requires:

- Creation of an automated process that generates various keyboard behaviors including KB Blitzes. These will be the experiment benchmarks and two basic keystroke patterns would be generated. One would contain continuous typing without any events that might end localization. The other would contain events in frequent intervals that either postpone localization or result in an end to localization. The frequency of these events can be set to reflect various levels of KB Blitzes. Also, the speed of typing simulated should reflect the response time when the client is in normal mode and be a pre-defined speed ("maximum typing speed") in local mode
- The running time of a benchmark is the amount of time taken to generate all the key events in the specified pattern. The running time of the benchmarks would be

recorded for various levels of client latency and these values will provide information regarding improvements the user might experience in terms of client performance

- The amount of communication between the client and the server will be recorded during each experiment and this will show how much communication localization saves
- The amount of local resources eaten up by the localization scheme during the experiments will be recorded and this should provide information regarding the cost of localization

The data collected from experiments may be combined to produce a composite value that reflects the overall utility of the localization scheme. In point 2, it is mentioned that the benchmarks would run on various latencies. This can be achieved via a simple "Network Emulator" that can artificially alter the latency in the communication channel between the client and server machines. A detailed description of this emulator will be provided in the chapter about experiments.

CHAPTER 4
THIN CLIENT PROTOTYPE AND TOOLS

This chapter describes the thin client prototype and tools used for designing and evaluating the proposed keyboard localization system for thin clients. The first topic for discussion is the components that form the thin client prototype, which include Microsoft's Terminal Server, and Citrix's ICA Technology and MetaFrame server. Following that will be an elaboration of the Citrix Virtual Channel SDK and its relevance to the localization system development. The chapter will conclude with a brief description of the development environment.

Microsoft's Terminal Server

Microsoft's Windows NT Server, Terminal Server Edition (now an integral part of the Windows 2000 Server) can deliver windows desktop and applications to any computing device even if it does not run Windows [9]. The goal is to take advantage of the resources provided by a distributed computing environment. All applications/desktops that run on client machines execute completely on the server; the client sends keyboard and mouse data to the server and receives display updates. Each user sees only her individual session that is managed transparently by the server and independent of other sessions. Terminal Servers help in achieving centralized deployment and management of windows-based applications and providing usage for outdates machines that can act as

client sites. Microsoft's RDP (Remote Desktop Protocol) and Citrix's ICA technology are a couple of means by which Terminal Services can be provided to clients.

Citrix's MetaFrame and ICA Technology

Citrix's ICA technology provides the foundation for turning any client device (thin or fat) into the ultimate thin client [2]. The ICA protocol sends only keystrokes, mouse clicks, screen updates, and audio across the communication channel. While client machines need only a processor as powerful as an Intel 286 and 640KB of RAM, the communication consumes only about 20KB of bandwidth. ICA can work with any Win16 or Win32 application and is inherently platform independent.

Citrix's MetaFrame server runs on top of Microsoft's Terminal Server and provides Microsoft's Terminal Services to ICA based clients. The following are some of the main benefits Citrix's MetaFrame [10]:

Enterprise-class management, End-to-End command and control related to:

- Systems: management tools for enterprise-wide scalability, reliability, and security. Examples: Load Balancing services, SecureICA services, Data encryption
- Applications: Rapid deployment and management of applications from a single point for optimum performance and uptime. Examples: Application publishing, Automatic ICA Client update
- Users: Users control desktop. Examples: Program Neighborhood, Client printer management, Local/remote clipboard, VideoFrame

Web application publishing:

- Integration: Integrate applications into standard web browsers. Example: Nfuse application portal technology, Web Portal Wizard
- Personalization: Personalize the applications a user receives. Example: adding scripts and graphics, embed applications within web pages or launch application windows

Flexible Application access on demand:

- Access all Windows based applications
- Access from any Windows or non-Windows based device or appliance. Examples: Java, Netscape Navigator, Unix workstations/Linux
- Connections: Access with any connection, LAN, WAN, Internet, and Wireless. Examples: Novell Netware LANs, TCP/IP, IPX, SPX, NetBEUI protocols, high-speed analog modems

Citrix Virtual Channel Software Development Kit

This section is a summary of the Citrix Virtual Channel Software Development Kit documentation provided by Citrix along with the SDK [11].

Citrix Virtual Channels are bi-directional error-free connections used for exchange of packet data between a Citrix server and an ICA 3.0 compliant client. They can be used to enhance the functionality of ICA Clients; for example, additional virtual channels can be created to support audio and video data streams.

Virtual Channels (VCs) use the Independent Computing Architecture (ICA) protocol. Since ICA is a presentation level protocol that runs over several different transports (like TCP/IP, IPX, etc) protocols for VCs can be developed independent of the underlying transport. ICA VCs are packet-based; if one side writes a certain amount of

data, the other side receives the same data when it performs a read. There is no need for protocols to parse messages and message boundaries, unlike TCP for instance. System software that is part of the ICA Client and the MetaFrame server manage the ICA stream and VC packets are encapsulated in this stream. ICA also provides for error correction; this ensures that data are received in the order in which it was sent. VCs provide different flow control options to enable developers to structure the VCs; for instance, a limit can be set on the amount of data that can be handled at any one time.

Developers determine the contents of VC packets but their size is restricted to 2KB or 2048 bytes. With 4 of these bytes used for the VC name, the user if left with 2044 bytes. This restriction affects the *WFVirtualChannelread()* and *WFVirtualChannelWrite()* functions on the server-side and the *OutBufReserve()* function on the client side.

VCs support 3 downstream flow controls:

- None: any control has to be part of VC protocol
- Delay: Client VD specified; delay occurs in server-side
- Ack: VD specifies buffer size and sends an acknowledgement based on how many bytes in a packet from the server were processed

VC names are ASCII names with at most 7 characters that can be only letters and numbers. While the first three characters form the OEM identifier, the rest is left to the discretion of the user.

VCs are implemented through a client-side VC Driver (VD) and a server-side VC application. All data are sent from a VD to the server via the WinStation Driver (WD) on the client-side. The VD is actually a Dynamic Link Library (DLL) whose functions are called by the WD. Thus, it cannot initiate data transfers but can only wait for the WD

driver to poll it for data and this is done frequently. When data are polled, the WD sends it to the server, where it is queued until the corresponding VC application reads it - there is no way to alert the application upon receipt of messages. All messages sent on a VC from the server side application to the WD are demultiplexed and the data are sent to the corresponding VD. The interaction between the WD and client VDs will be discussed next.

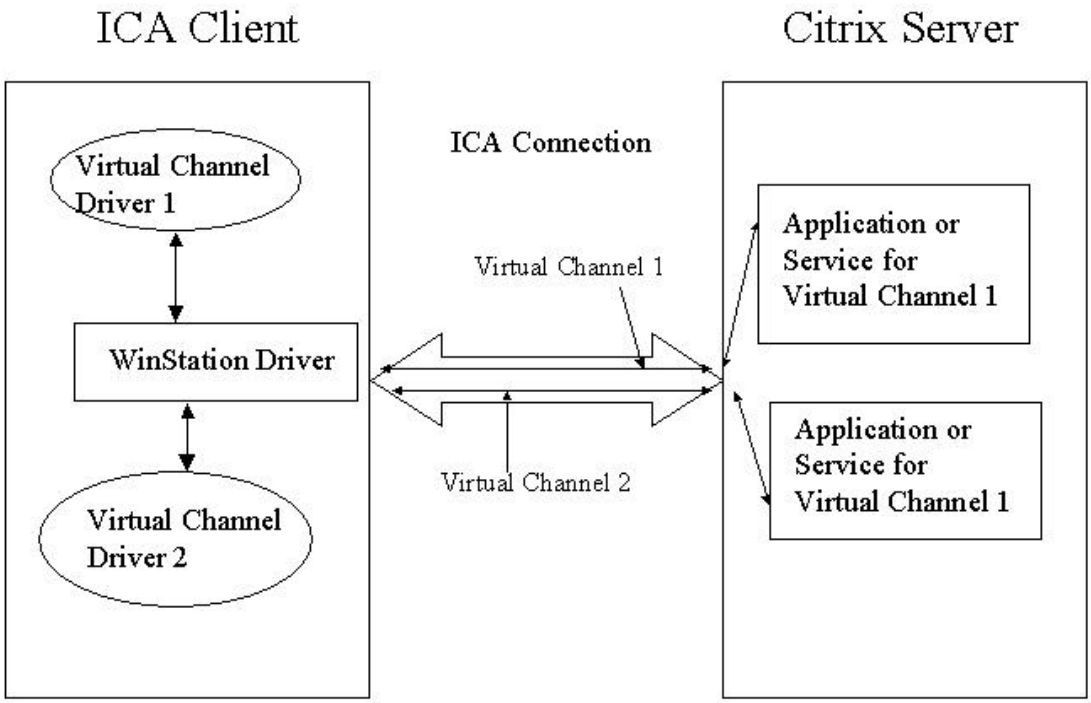

Figure 1: ICA Client at run time

VDs are Win32 DLLs created using the VC API. The developer uses the interface to define what happens when a function from this DLL is called. When the ICA Client starts, the client engine reads the module.ini file to determine which modules to load and how to configure the various VDs. For each VD, the WD calls *DriverOpen()* and important information is exchanged during this function call: the VD gets addresses

of output buffer functions while the WD gets address of the VD's *ICADataArrival()* function. Since the client runtime is single- threaded, VD developers are directed to not to use any blocking calls in the code definition: a blocking call in a function called by the WD will delay the entire client.

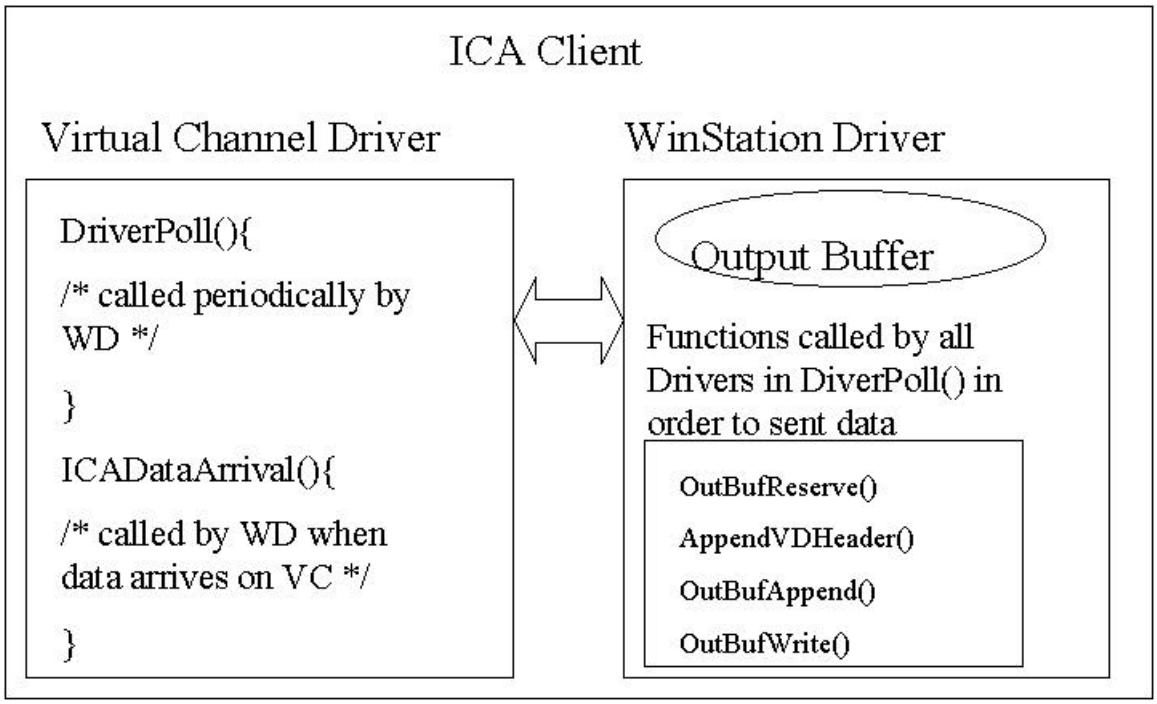

Figure 2: Interaction between WinStation Driver and Virtual Drivers

Once the VC is opened, the communication through it may begin. Data sent from the server application are handled by the driver through the code definition for *ICADataArrival()*. If the client wants to send data, it has to wait for next call to *DriverPoll()*. In the code definition for this function, the VD must use helper functions to send data--it should reserve output buffer, fill it with data, and write the buffer.

The list of VDs to load when the client is started can be provided in the module.ini. This file stores settings for determining VDs to load and the name of the corresponding DLL. The file can also store parameters for the VDs, which the VD can

obtain by making the appropriate API calls. A distributed module.ini file will permit clients to write on to it.

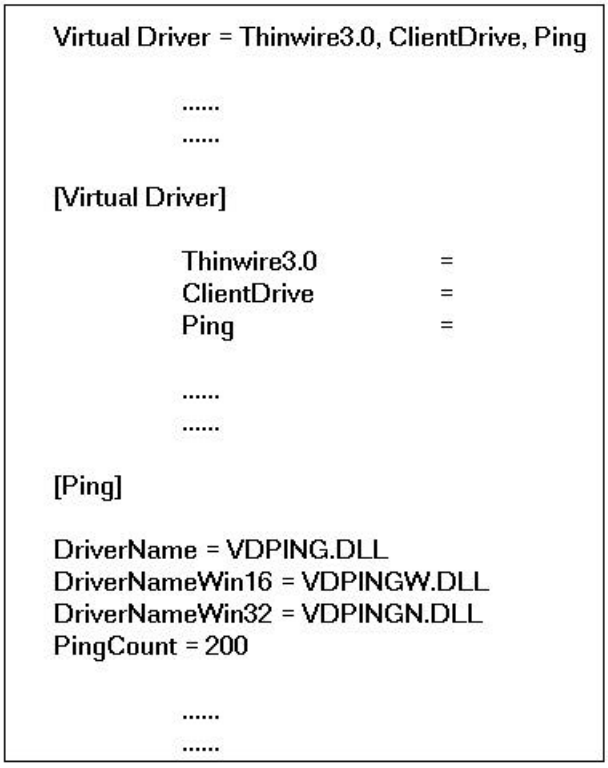

Figure 3: Excerpts from module.ini file

Table 1: Server side SDK functions

WFVirtualChannelOpen()	Obtain and open a VC handle
WFVirtualChannelClose()	Close an open VC handle
WFVirtualChannelRead()	Read data from a VC
WFVirtualChannelWrite()	write data to a VC
WFVirtualChannelPurgeInput()	purge all data sent to server
WFVirtualChannelQuery()	Query information related to a VC

Table 2: Virtual Driver API functions

DriverOpen()	called when the client loads the VD
DriverClose()	called before unloading a VD
DriverInfo()	gives module info about VD
DriverPoll()	called periodically to check for data to write
DriverQueryInformation()	queries specific info about client
DriverSetInformation()	used to send information to the VD
DriverGetLastError()	returns the last error set by the VD
ICADataArrival()	called when data arrive on the VC

Utility of the VC SDK

Utilizing the VC SDK in designing the keyboard localization system will provide the following benefits:

- Since VCs run on top of the ICA stream used by the thin client, there is no need for creating additional communication channels between the client and the server for the localization system. Also, VCs guarantee reliable communication thus eliminating any need to monitor for possible packet losses.
- For the same reason as above, multiple VC server applications can run simultaneously on the MetaFrame server and provide the required service to their respective clients through VCs with the same name; in terms of servers, the VC name can be related to a server's "well known port."
- As VC drivers can be dynamically included to the ICA Client executable, the localization process built using VC drivers can be easily integrated to the ICA Client as an additional functionality.

- Since VC drivers are DLLs, other applications that run completely on the client machine can be made communicate with the ICA Client by adding exported functions to the DLL. As we will see in the next chapter, one of the components of localization is an application that exploits this feature.

Development Environment

For the purpose of development and experiments, two IBM 390 laptops have been deployed to act as server and client respectively. The laptops each use an Intel Pentium II 233 MHz processor with 64MB of RAM and 4 GB of hard disk space. Both machines use Netwave's Air-surfer Pro PC cards for wireless communication on the CISE department's WLAN. In order to remove any interference from other users of the WLAN, the machines will use the PC cards' built in ad-hoc mode to work in a private network with an approximate raw bandwidth of 2 Megabits per second and latency below 10ms. Efforts will be taken to ensure that no other computer is accessing either of these machines during experiments! On one of the machines, Microsoft's Terminal Server Edition was installed and on top of it Citrix's MetaFrame Server, version 1.8. During design and experiments, this machine will be freed of all responsibilities except providing Terminal Services to the ICA Client. The ICA Client runs on the second laptop that is a Windows NT Workstation 4.0 system. Theoretically, the client could run on the same machine as the server; however, measuring the usage of the resources of the thin client machine during experiments would be more accurate if the machine does not run other applications and services. Citrix's Program Neighborhood version 4.20 is used to create

ICA Clients that run Windows NT desktops. The localization system will be designed specifically for Win32 ICA Clients.

CHAPTER 5
IMPLEMENTATION: THE *KB PRO* SYSTEM

This chapter describes in depth the three core components of the *KB Pro* keyboard localization system. They are the Virtual Channel Driver *VdHook,* a process on the client machine, called *KBWin*, to localize typing activity, and *KBServer,* a process on the server side that serves *VdHook*. Since the localization system is for Win32 ICA Clients, all these components are Win32 modules.

Relevance of Virtual Channels

The *KB Pro* system for localizing keyboard activity in the ICA Client befits implementation of Virtual Channels (VCs). VCs are of extreme use since a private communication channel between the client and server machines is crucial. Keyboard activity localized at the client machine can be refreshed with the server through a VC; also, information about the application to localize and the position of the caret in it can be sent by the server to client on this channel. In addition, the VC driver on the client side can be designed to monitoring keyboard activity in the session and detecting a keyboard blitz. Since VCs are implemented on top of the ICA stream used to run the thin client, there is no need for creating extra communication channels. And this design also results in an efficient use of the ICA channel: VC packets for different VCs can be sent together

as one whole message. Finally, the localization process can be integrated into the localization system by defining exported functions in the VD that it can call.

VdHook: the Virtual Channel Driver

VdHook is the driver that runs the communication mechanism for the Virtual Channel "*ctxhook*" on the client side. This Dynamic Link Library (DLL) contains all the basic functionality required to communicate through a Virtual Channel. In addition, the DLL provides exported functions that can be called by other applications after explicitly linking with it. The DLL also contains data in a shared segment in order to let the WinStation Driver and the localization process (*KBWin)* share important information.

The Shared Header File

The first subject of discussion should be the header file defined in its context since it is also used to build *KBWin* and the *KBServer*. Some of the information defined in this header file is:

- It defines the name of the Virtual Channel used by the KB Pro system ("*ctxhook*")
- It defines the application that the DLL should call once a keyboard blitz on a typing application has been detected. This application is the name of the executable for *KBWin*
- Since VCs are packet based and communicate packets with a size restriction of 2048 bytes, the maximum length of text to refresh is restricted to 2000; this number is defined here. Note that this number may be increased or decreased based on how much of the remaining bytes is used

- A latency threshold is defined for localization (200 ms) – network latency below this value does not call for localization
- A series of values that define the communication protocol for the VC. These are used for determining purpose of messages received. (ex: #define Refresh 4)
- A series of data declarations for inter-component communication.
- The average width and height of characters displayed in local mode
- The number of times the virtual channel driver Pings the server to determine network latency
- The frequency of detecting the network latency during the session (300 seconds).

The Shared Data Segment and Exported Functions

Moving to the DLL itself, the first feature for discussion is the shared data segment. This segment is necessary since there are two (three including the benchmark process described in the next chapter) separate processes using the DLL that need to share data crucial to the success of the whole system (the two applications are the WinStation Driver for the client and *KBWin*). Some of the data defined in this segment include

- Data used by the functions called by *KBWin* in order to refresh locally typed text. During refreshing, the same data are used by *DriverPoll()* the next time it is called by the WinStation Driver.
- Data to maintain the dimensions and position of the ICA Client application to localize, and the position of the caret in it. These values are obtained from server by *VdHook* and then are given to the *KBWin* application through exported functions.
- A number of flags maintaining the various states of the localization system.

The next feature of the DLL is the series of definitions of exported functions. These are called by the *KBWin* application for reasons such as registering its window handle, getting the handle of the ICA Client window, getting information about the application to localize, refreshing with the server text typed during localization, and registering its exit with *VdHook*. While the first three are called before localization begins, the last two are called at culmination. Another function defined just for experimentation provides information that helps determining if the client is in normal, local, or transition modes; this is the only function called by the benchmark process described in the next chapter.

Since one of the principles of localization is taking care of refreshing events occurring during localization, the exported function used by *KBWin* to refresh localized text takes in the refreshing event as one of the arguments. Ideally, refreshing events should be generated only after the localized typing activity is refreshed. Consider the situation when the user uses mouse clicks or keyboard shortcuts to save the file during localization; this event should occur only after the localized text is refreshed with the server otherwise the text would not be included in the saved file! If the event is a keyboard event, it is forwarded with the refreshing text to *KBServer* so that the event can be generated at the server with the typing application in focus (and only after the text is refreshed). Mouse events should also take place only after the localized text is refreshed with the server; but unlike keyboard events they may have to be sent to other windows. Thus, the mouse event is generated locally and sent to the ICA session. When a mouse event needs generation, this function uses various global state flags and waits until the *KBServer* acknowledges the refresh; only then does it send the appropriate mouse

message to the ICA Client window. Handling mouse refresh events may also be pushed on to the server and this is left as future work.

The sequence of events that occur after spawning *KBWin* is worth a quick mention in the context of exported functions. In order to let the user type during transition periods, keyboard focus should be given to either the ICA Client or to *KBWin*. However, it has to be given to the latter since the caret position detection process used by *KBServer* requires that no typing take place in the ICA Client! The intuitive solution is to give the focus to *KBWin* as soon as it is spawned; however, typing activity might be lost during the time taken for *KBWin* to show its window! Thus, the server is contacted for application information such as position, dimensions and caret position only after *KBWin* makes a function call to register its handle. *KBWin* will later call another function to get the information provided by the server. During the transition from normal to local mode, audiovisual cues are provided to the user as signs of initiation of localization. It should be added that keyboard activity during transition might get lost even when using the above scheme.

Keyboard Hook and Window Enumeration

The function defined as the “keyboard hook procedure” is next in line for discussion. The DLL registers a keyboard hook only when *KBServer* is running and latency is above 200 ms and provides the keyboard hook procedure to the system. This function should ideally “work” only when the *KB Pro* system needs to watch out for a keyboard blitz on a “typing” application. Since *KBServer* keeps *VdHook* updated with the application the user is working on, the hook function is made to exit immediately if there is no need to monitor keyboard activity. But if the application in use is a typing

application, the function monitors the keyboard activity to check for KB Blitzes. It starts a stopwatch and ends it when a key does not confirm with the "key types" test of the KB Blitz or when fifteen keys (defined as *KBBlitzValue* in the header file) are pressed. If the case is the latter, the key series has passed the first test of a KB Blitz and is put through the second test. The time period is compared with *KBBlitzValue* times the average roundtrip time for a packet (2 * client latency). Localization is triggered if the series is a KB Blitz and *KBWin* is spawned. If the series fails the first or seconds test, the detection counters are reset and process starts again.

In order to ensure that the keyboard monitoring is done only for the ICA Client, it is necessary to obtain the process id of the ICA Client. Since the window handle of the ICA Client session is required and the VC SDK does not provide functions to obtain it, a simple window enumeration procedure is executed when *VdHook*'s *DriverOpen()* function is called, which is at client start time. The scheme used to obtain the handle is trivial; since ICA Client windows contain the phrase “Citrix ICA Client” in the toolbar, the function checks if the text of all windows enumerated contains the above phrase as a sub string! The first handle that passes the test is deemed to be the one for the current ICA Client and the process id is obtained from it. The glaring drawback to using this scheme is that only one ICA Client session should be running. However, this disadvantage can be tolerated at this point of time in research. It is worth mentioning that the ICA window handle is saved as a shared data that can be passed on to *KBWin* for its use.

Virtual Channel Communication Functions

The rest of the DLL contains functions typical of all Virtual Channel Drivers. Some of these, however, perform more that just VC communication tasks. For instance, *the ICADataArrival()* function will set values of shared and/or unshared data according to the type of messages received and the contents of the packet. Some crucial data values set here are the various flags that represent the state of the *KB Pro* system process. For instance, when the DLL is on a watch out for a KB Blitz on a typing application and the server says that the application in focus has changed, the watch for a blitz is ceased (again through shared variables) and if needed restarted based on what type application is in focus. The cease and restart might be needed even when the application in focus changes its dimensions or its position on the session. Situations may arise for an abrupt end to localization due to a message sent from the server. *VdHook* accomplishes this by sending an appropriate message to *KBWin*. Since this message is a windows message that two separate application use for communication, it has to be registered before usage and this is done in the same function. Apart from setting state values, this function also sets information of the various data structures for application information, caret information, etc based on the packet received from the server.

A feature of considerable utility in the *KB Pro* system is giving users the option to refresh localized text and immediately return to local mode instead of the default practice of returning to normal mode. Part of this is implemented in the *ICADataArrival* function. One of the messages sent by the *KBServer* is coded *RefreshAndInfo*, which means that *KBServer* performs a request for refreshing localized text and providing information in for return to localization. Such a request actually originates at *KBWin*, and is then sent to *VdHook* during the refresh function call, and finally sent to *KBServer*. When the user

exercises this option, *KBWin* will not exit as normal but would remain in abeyance while waiting for *VdHook* to receive the required information to restart localization (the waiting scheme is described in the *KBWin* section).

While *ICADataArrival*() is called by the WinStation driver only when data arrive on the VC, the *DriverPoll()* function needs deft handling since it is called in frequent intervals by the WinStation Driver (see section about the VC SDK). When there is no need for sending messages to the server, this function is designed to exit immediately. However, the frequent calls to this function by the WinStation Driver are exploited during periods of transition! The function helps in providing visual cues to the user by making the ICA Client window blink once. By the time *KB Pro* gets out of transition, *DriverPoll()* would have been called a number of times and the window would have blinked accordingly!

Some of the other functionality includes sending pings when the current state of the *KB Pro* system is latency detection. The number of times it sends this ping is defined in the header file. And finally, it is through this function that *KB Pro* sends information to the server for refreshing localized typing and for finding information about the target application for localization.

Latency Detection Scheme

The responsibility of estimating the network latency is given to *VdHook* and not *KBServer*. This is done so since VC drivers can send data only when polled by the WinStation Driver; any Pings sent by *KBServer* cannot be sent back immediately and this may affect the calculation of the average round trip time if done by *KBServer*.

Thus, when *VdHook* is at the latency detection state, *DriverPoll()* sends to *KBServer* a number of Ping packets containing the Ping number and records the send time of each Ping in a global structure (the number of Pings is defined in the header file). When Pings are received from the server in *ICADataArrival()*, the receive time is recorded and the Ping number is decoded to get the send time from the global structure. After all Pings are received, the average of all the round trip times is calculated (response time) and half of this value is assumed to be the approximate network latency. The timing of the entire detection process is also recorded in order to initiate another one after a certain time period elapses (the time period is defined as 300 seconds in the header file). This measure is taken to ensure that latency fluctuations that are highly probable in wireless networks are accounted for. The time check, however, is done only when the client receives a Reset message (non-typing application in focus) from the server. The heuristic is that the user will not require the services of the *KB Pro* system when she moves to a non-typing application, thus making the situation ideal to send Pings and measure the latency. In case a situation arises that localization is triggered and information from the server is required, the request is delayed until the latency detection scheme is complete. Since the probability of this case is low, the occasional delay in localization can be tolerated.

KBWin: The Localization Process

As mentioned in the previous section, this component of the *KB Pro* system is triggered by *VdHook* in order to localize keyboard activity. *KBWin* is made a stand-alone process since I did not attempt adding any additional windows to the ICA Client process,

if it can be done at all. This section describes this component in terms of its interaction with *VdHook* at entry and exit, and in terms of the mechanism it uses in displaying the localized keyboard activity.

Interaction with *VdHook* during Entry

Upon invocation, the process makes a function call to *VdHook* to get the handle of the ICA Client localized (remember *VdHook* maintains this handle as shared data) and creates its transparent process window with position and dimensions such that it exactly covers the entire ICA Client window. Covering the entire window is to ensure that no mouse event will be sent to the ICA window during localization; if the window were to cover just the localized application, there are chances for the ICA Client processing messages in the wrong sequence (recall the example regarding saving files using mouse clicks).

After the window is created, the *main()* function sends a local message to the window. The code to handle this message is designed to ensure that no loss in keyboard activity occurs during transition. Here, a function call is made to *VdHook* to register the window and, as mentioned previously, *VdHook* sends a message to the server for application info and thrusts keyboard focus onto *KBWin*. Then, *KBWin* calls the function in *VdHook* trying to obtain the application information. If this call returns with the information that the server has not responded yet, *KBWin* waits for a short time period and makes the call again. If the function returns with the information, the process continues with its initialization. This design will ensure that all typing activity occurring during transition will be taken care of; while the part occurring before *KBServer* sends the request for application information is handled by the client itself, the other part will

be handled by *KBWin.* And since *KBWin* waits until it gets the caret position information, the keyboard messages sent to it will be queued and eventually processed and displayed at the correct place. Note that processing and correctly displaying keyboard activity is impossible without application and caret information. Incidentally, *KBWin* will exit if the function call to *VdHook* returns the information that *KBServer* could not provide the application information (for instance, if caret position detection fails). At this stage of development, all the typing activity directed at *KBWin* will be lost in this case; this can be remedied by including a mechanism that will process any queued keyboard messages and immediately refresh with the server. After the refresh, *KBWin* can exit .

The information obtained from the DLL is same as the one provided by *KBServer.* They are the dimensions and position of the area *KBWin* should localize and the position of the caret in *KBWin* where a caret should be initially shown and text displayed. In addition, *VdHook* provides through the same function the handle of the localized application window in terms of the ICA session. Since the localization area information provided by *KBServer* is in terms of the session, *KBWin* has to translate the position of the area to suit the display on the client machine. In order to do this, the position provided by *KBServer* is normalized with the top-left coordinates of the ICA Client window.

Interaction with *VdHook* during Exit

Before moving on to the display mechanism, the interaction between *KBWin* and *VdHook* during *KBWin*'s exit will be discussed briefly. If *KBWin* refreshes before exit, it will make function calls to *VdHook* and give it the text to refresh and the handle of the application window that it localized. Along with this, the event that resulted in the end to localization will be sent with pertinent information. The function in *VdHook* handles the

refreshing event as described earlier. Sometimes, the user may stop localization without requesting a refresh and thus the refreshing function will not be called. Irrespective of reason for end in localization, *KBWin* will finally call a function in the DLL to register its exit. Various state values are set in this function based on whether *KBWin* refreshed before exit.

In order to ensure that the user does not interfere with the refresh mechanism employed on the server side, it is important that *KBWin* remains as the active window during this time. For this sake, the refresh function called will wait until *KBServer* acknowledges the refresh on the server side. This is done even when the user exercises the RefreshAndInfo option even though it is not required; it has no affect on *KBWin* in this case.

Now is the best time to discuss *KBWin's* RefreshAndInfo capability. When the user exercises this option, localization is to continue after a refresh; *KBWin* calls the refreshing function as usual and instead of exiting goes back to the logic as followed during the WM_CREATE message! Keys typed during refresh will continue to be directed at *KBWin* while it waits for the application information; they will be handled once the updated information is retrieved from *VdHook*.

Localization Mechanism

Now the focus of our discussion shall completely shift to *KBWin* as a stand-alone process. While the window covers the entire ICA Client, the localized area is defined as follows:

- Its width is the same as the window of the application localized and its height covers the window from the initial y position of the caret to the bottom of the application window.
- Its x position is the same as the application localized and its y position is the same as the initial y position of the caret on the application.

Keyboard events that can be localized are handled and displayed as if *KBWin* was a regular edit box. More importantly, the position of characters displayed on *KBWin* combined with the apt positioning of its window on the ICA Client makes the localized display look very similar when the characters are typed in the application window! This capability of the localization process makes it as transparent as possible to the user. The display cannot be made completely transparent for reasons such as lack of information regarding font and the number of characters to display on one line. Note that MS Notepad expects the user to press <enter> before a new line is created, while MS Word has page size properties. In this regard, *KBWin* works like MS Word by setting the width of a typing line the same as the width of the localized area; a new line is created when the right border is reached. Keys typed are handled as long as the character they represent can be displayed at the appropriate place. If normal typing results in caret movement beyond the localized area, *KBWin* performs a Refresh and exits.

On the contrary, encountering the following keys even when the caret is within bounds will result in a simple refresh and exit:

- syskeys, control keys combinations,
- function keys with/without control or shift key combinations,
- page up, page down.

- Attempts to highlight text

The reason is simple: these keys cannot be handled displaying the "character" they represent. This list may grow or reduce with further research regarding the keys that can and cannot be handled locally.

As regards to the mouse, only left mouse clicks are allowed inside the localized area and they are handled by simply moving the caret to the appropriate position. However, the following mouse activities will result in a simple refresh:

- Left mouse clicks outside the localized area
- Any right mouse click

Finally, *KBWin* will perform a simple refresh when it loses focus (either by keyboard or mouse). No other termination scenarios have been identified at this stage in research.

KBServer: The Server Side Component

KBServer is the process running on the MetaFrame server that compliments *VdHook* in the VC implementation. Ideally, *KBServer* should begin execution as soon as the ICA Client is invoked and ideally be an iconic application throughout the session. Since I do not have the means to link it with the MetaFrame server, this process should be started manually but not necessarily at the beginning of the session. In fact, it is more convenient for the sake of experiments that this application can be started and ended manually. The application can be made iconic but since it is still in the development state, it maximizes at initiation and is manually minimized during experiments. This section

describes the design scheme of *KBServer*, the services it provides, and the scheme it uses to detect caret position in typing applications.

Design Scheme

Upon invocation, the process immediately opens the Virtual Channel used by *VdHook* and purges any outdated data. After determining the session ID, it goes into a daemon mode and the sequence of events during each cycle is described below.

A list of processes running in the session is determined and information such as process id and process name of only typing applications is retained. Then a window enumeration process is run and the ids of the windows enumerated are matched with those of the typing applications. This filtering mechanism enables *KBServer* to get the handles of windows associated with typing applications and maintains them as application information. This is done every cycle since the user will most likely not have the same windows running throughout the session.

Next, the handle of the window in focus is matched with the typing application window handles. If the window is one of these, a message is immediately send to the client that a typing window is in focus and that it should keep its eyes open for a KB Blitz (the handle of the window is also sent) If the window in focus is not associated with a typing application, the message sent will inform the client accordingly. From the second cycle onward, the window in focus is compared with the window that had focus during the previous cycle. If they are different, a message is sent according to which window received focus. If the window handle is the same but is a typing window, the dimensions are matched with the information saved in the previous cycle. If anything is different, a message is sent to the client as if this window just came into focus. These messages may

result in any localization or keyboard monitoring at the client side to cease or reset respectively. This reaction is in harmony with the principle that switching between windows and changes in windows' dimensions and placement should not occur during a KB Blitz.

The next task executed in the cycle is reading on the VC for any requests/information sent from the client machine. *KBServer* returns to the top of the cycle if there are no messages. Received messages could be as simple as a Ping or something more complicated as a request for application information. The other types of messages received are “hello” messages and messages with information pertaining to refreshes.

Services of *KBServer*

For messages requesting a refresh service, *KBServer* matches the window handle in the message with the handle of the window in focus. This is done since refreshing is actually achieved by copying the text to refresh on to the clipboard and then adding events to the keyboard queue to simulate a CTRL + v key press, a shortcut for paste. Note that this is the shortcut for most typing intensive Windows applications. There is a potential problem to this scheme: the paste message will be sent to the wrong window if the window handle of the application in focus is not the one in the refresh packet! Since switching windows during localization is a refreshing event that will be generated after refresh, the chances of the localized window not being in focus during the paste are few. However, there is a possibility that the localized application or any other application will pop up a window to inform the user about problems like fatal errors! However, this problem is simple and has multiple solutions. Windows API provides a function called

SendInput() that can help send keyboard messages even to windows not in focus. However, this requires a version of Windows NT higher than the one currently used in the development of *KBServer*. Also, a display update message could be sent to the ICA Client for each keyboard input and this is unacceptable. Another solution is maintaining a buffer that stores the text to refresh for each "typing" window. Remember the size for this buffer need not be bigger than the designated limit for the size of each refresh--if there is another refresh for the same application, the window had to be in focus just before the localization process and at that time the text in the buffer can be pasted on the application. However, this involves race conditions and the user might have typed something on the window before the paste is made, resulting in a failure for *KB Pro*! The third solution is the simplest and is the one implemented: the localized window is forced to the foreground. This will not affect transparency even if the refreshing event is a shift to another window. Recall that *KB Pro* will generate the refreshing event only after the text is pasted on the application. If the event is a key event, *KBServer* will generate it only after the paste and if it is a mouse event, *VdHook* will generate it only after *KBServer* acknowledges the refresh. In addition, pushing the application window into focus is done for another reason. The application window is made to give up keyboard focus to *KBServer* before localization (see below).

Upon receipt of application information requests (*Info* messages), *KBServer* first matches the handle of the window in the request with the window currently at focus. Note that the chances of another window being in focus are very little after the client has detected a KB Blitz. Once the focus is verified, *KBServer* will make a series of Windows API calls to get the dimensions and position of the target window. And if the position of

the caret is detected successfully (described later), the information is sent across to the client. If any of these steps fail, the client is sent a message that the information cannot be provided. A *RefreshAndInfo* request is serviced by a refresh on the localized application followed by the same steps as when replying to an *Info* message. A display issue was considered when clients localize applications. Since *KBWin* maintains its own caret, having the localized application's caret also appear during localization will certainly reduce transparency and confuse the user. To remedy this, *KBServer* will force itself to be the active window before it sends the client application information; since *KBServer* is an iconic application, this step will not affect display. In fact, the user may notice its icon being highlighted and become aware of localization! And as mentioned earlier, focus is given up when localization ends.

When KB server gets a *StartPing* message, it is actually a message to get prepared to answer forthcoming pings in the same cycle; this is done to prevent any delays between cycles from affecting the latency calculation. After a certain number of Pings are received and answered (the number of pings is provided in the DLL header file), *KBServer* moves to the next cycle.

The other miscellaneous messages are *Hello* messages and *Escape* messages. The former is sent when *VdHook* opens its VC and is handled by a replying with the same message. The latter is sent when localization ends without refresh; this is handled by returning to the localized application keyboard focus that was taken away before localization began.

Caret Position Detection Scheme

Typing applications normally display a caret where the user's typing will be displayed and it is common knowledge that MS Word and MS Notepad windows display carets. Note that these applications, like how all Windows applications with carets should be designed, show the caret only when the window is in focus. However, there are no API calls available that applications can use to retrieve the position of the caret in windows other than the ones it owns (itself and its child windows). Thus, *KBServer* will have to devise means to detect the position of the caret in the application *KB Pro* wants to localize and this scheme is described below.

Once the target application is found to be in focus, *KBServer* performs a sequence of complicated bitmap calculations that can be completed within a very short period of time (below 1000 milliseconds). First the dimensions of the application in focus are retrieved and a pair of structures to create and store bitmaps bearing the image of the window is created. Then a snapshot of the application is taken by saving the image of the application window in a bitmap. After that an event is pushed on to the keyboard queue and when the application gets the event (which it will since it is in focus), the character represented by the event is displayed where the caret is positioned. Assuming that this worked, another snapshot of the application window is taken and the character is deleted by sending another keyboard event for representing the backspace key. The first snap is then compared with the second snap via a bit-by-bit comparison; the comparison should result in a conflict where the character was displayed, and hence where the caret was positioned. If any of the above steps fail (example the first snap shot), the detection process is deemed to have failed. I am aware that there could be a conflict even without the display of a character at the caret position since the caret blinks. However, relying on

the caret blinking requires precise timing for the second snap shot, something too complicated. Also, there is no way to determine if the caret will appear in the first snap shot and not in the second, and vice versa. In addition, some applications may not have a blinking caret! Nevertheless, the scheme used by *KBServer* is not foolproof either since the bitmap and key insertion steps are not guaranteed to succeed. In such scenarios, the caret detection procedure is deemed to have failed. The failure to find the caret position exacerbates to a failure in providing information about the target application for localization, and eventually to a failure in KB Pro. The experiments on KB Pro have shown that this possibility exists but only at a low percentage.

Localization that begins with the above information and with the design of *KBWin* will remain transparent only when the user is appending text to a file! Since *KBWin* displays localized typing on top of the ICA display, localizing typing when the user is typing between lines will produce a display with localized text overlapping previously typed text! Such scenarios will certainly reduce the transparency of KB Pro. Localization can be avoided if there are dependable means available to detect typing between lines of text. At this point in research, the user will have to accept this drawback and either continue localization with a faulty display or end it manually.

For such situations, the user may be provided with an option in *KBServer* to choose between two modes: append and edit. While the former is the default mode, *KBServer* can be designed to choose between two options for the latter: it can inform *VdHook* that the typing application is not meant for localization; or, it can artificially create empty space on the application window using the following algorithm:
Detect the current position of the caret;

Generate a key event representing an enter key press and detect caret position again;

/* *this should give the approximate amount of vertical space eaten up by one line in normal mode */*

Calculate the difference in the y positions of the two caret positions;

/* *obtain the ratio of space eaten up by one line in local and normal modes. the average width and height of the character display in local mode are defined in the s hared header file. The average character height is deemed to be the space eaten up by one line in normal mode */*

Compare the difference to the average character height in the header file and determine a line ratio;

Use the difference in the y caret positions, the ratio, and the dimensions of the application window and generate enter key events in order to create a specified number of lines for localization;

Use key events to push the caret back to the original position;

Detect caret position again and send pertinent information to *VdHook;*

A similar scheme can be adopted to increase the localizable area if the caret position is found to be so close to the bottom of the window. This idea is in harmony with the principle of keeping the number of transitions to the minimum and transparency will not be sacrificed except in edit mode. The amount of space to create can be specified in the shared header file.

This algorithm was implemented in *KBServer* but is not guaranteed to create the required amount of space. Nevertheless, it will be included as one of the features since it

is not detrimental to *KB Pro* even when it fails; this feature will be included during experiments and some of the longer benchmarks might utilize it.

The caret position detection method will not work correctly if there are any other moving pixels on the bitmap above the y position of the caret. The result is the same when the application window is being repainted when the snapshots are taken. It is virtually impossible to guarantee that the window will be repainted before a snapshot is taken. The caret position information given to the client might be nowhere near the actual position of the caret in these cases. Also, as mentioned earlier, a failure in any sub task in the detection scheme will result in the entire process being deemed a failure. These two reasons for failure are accepted as unavoidable flaws of the detection scheme and it is for this reason that *KB Pro* provides an option for the user to exit localization without a refresh.

A question that can be raised here is why doesn't *KBWin* use the same scheme and detect the caret position itself? There are two main reasons for this design: one, creating and manipulating bitmaps may require considerable amount of memory and processing resources and these are very precious in mobile machines; two, making the server do as much of the processing as possible is in harmony with the core principle of thin clients. Needless to say, MetaFrame servers on fixed networks will be much more powerful and there is no dearth for memory or processing power. In addition, the event added to the keyboard queue during caret position detection (to display a character at the caret position) will be processed faster when it is inserted locally (by *KBServer*) than when the client machine does it (*KBWin*). Still, maintaining the leanness of the thin client remains the salient reason why the caret position detection is left to the server.

Summary

All features of KB Pro have been designed and implemented keeping in mind the principles enumerated in Chapter 3. The resources of the Windows API have been exploited to their maximum and, at the same time, its limitations have imposed restrictions such as excluding font information and adopting a caret position detection scheme that is not hundred percent guaranteed to succeed.

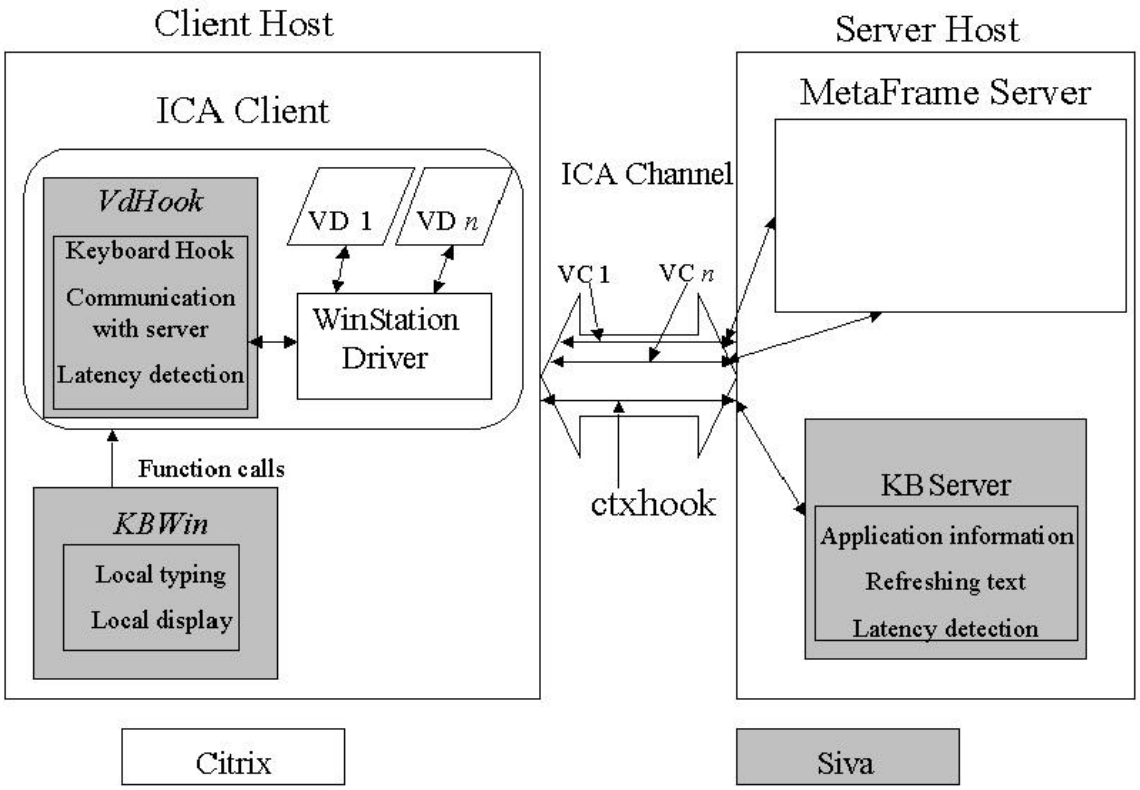

Figure 4: *KB Pro* system architecture

CHAPTER 6
EXPERIMENTS

The main motivation behind the *KB Pro* system is that users of thin clients must be shielded from the harmful effects of high network latency with respect to keyboard activities. The localization system implemented enables prompt display of user's keystrokes and produces a reasonably transparent display. And even thought it maintains the thin client theme, there is unavoidable dependence on the resources of the client machine: the localization process will require hard disk space for installation and RAM and CPU power for execution. As scientists, we would like to measure the utility of such systems by experimenting under heterogeneous scenarios and producing data related to benefits and drawbacks. This chapter first describes a process called *KB Pro Benchmark* developed to produce heterogeneous typing patterns. The next section describes a Latency Emulator built specifically for the ICA Client. The third section is a brief description of the Performance Monitor built into the Windows NT system and its utility in experiments. Then the variables chosen in evaluating the system are enumerated and the rationale behind their choice is explained. The chapter then elaborates on the experiment mechanism before concluding with a section devoted to data collected from experiments and its analysis.

KB Pro Benchmark

Any experimentation with the KB Pro system must obviously involve typing as the main input. However, manually producing key events will not guarantee that the conditions for all experiments will remain consistent; the production of keyboard activity must be made consistent and controlled and thus automatic. For this purpose a process that will produce key events has been designed, called the *KBPro Benchmark* (we will refer to it as *Bench*). *Bench* does more just insert arbitrary key events! It does it such that the series of typing events produced emulates various typing behaviors of humans with awareness of network latency.

Design Scheme

Before further discussing *Bench*, lets recalls from that a KB Blitz is defined as fifteen continuous key events involving letters, alphabets, and special characters that can be processed by simply displaying the characters they represent at the correct place. Also, the implementation scheme uses this standard to decide if a certain typing behavior requires localization. In addition, localization stops when a refreshing event (certain key event or key combination event) occurs; the typing done locally is refreshed with the server, the refresh event is then generated, and the client returns to normal mode. The inputs to the experiments must be able to create scenarios for the client to work in normal, local, and transition modes in order to measure the viability of the *KB Pro* system. These input sets, which will be referred to as benchmarks henceforth, will be produced by *Bench* using the following design scheme:

- At its bare form, *Bench* will produce a series of key events that will emulate the typing of a pre-defined sentence (called *default*): " Welcome to the KBPro

Benchmark created for measuring the performance of the KB Pro system." It is worth noting that this sentence contains 90 characters, though there is nothing special about the number.

- The number of times this sentence is typed can be specified and this will be defined as the *number of cycles*. The product of the number of cycles and 90 will be the length of the benchmark. When the benchmark ends, an event will be produced such that the client will return to normal mode, even if it happens to be in normal mode.
- Running a Benchmark with multiple cycles of *default* will result in the client going into local mode and remaining there until the end of the Benchmark. In order to produce typing behaviors with fewer KB Blitzes, refreshing events can be interspersed in each cycle of the benchmark. The number of refreshing events per cycle can be specified.

At this stage of design, *Bench* will produce benchmarks of various lengths. One missing feature is that these benchmarks produced do not make typing mistakes or do editing. This is acceptable since these events only affect the shift from normal mode to local mode; any of these occurring during localization process can be handled as long as the user stays within the localization area (as defined in chapter 2). This observation results in the following design principles for *Bench*:

- Editing and errors are not that important to measuring the utility of KB Pro.
- A key event series is not a KB Blitz when a refreshing event is interspersed in the series. The reaction of *KB Pro* to this event during normal mode will be the same as when the user does any editing--localization is postponed.

- The event used to disqualify a series from being a KB Blitz can also be used as the refreshing event when the client is in local mode. Thus, the refreshing event can be made a keyboard event to achieve the dual purpose: disqualify a series from being a KB Blitz and cease localization. This adds simplicity to *Bench.*

Refreshing Events

Contrary to the section heading, there is only *one* type of refreshing event and it has been designated as:

- Press of the alt key
- Press of the key representing the character 'f'
- Another press of the alt key

On applications like MS Word and Notepad, this will result in the File menu dropping down and disappearing immediately. However, the event produced by these key presses may not occur properly if the client is extremely slow. Also, the entire event may not be seen during experiments if the key presses occur continuously. Therefore, before steps 2 and 3 *Bench* waits for a time period equal to the response time of the client. This is obtained via a function call to the DLL of a Virtual Channel Driver in the Latency Emulator that measures the network latency of the ICA Client connection. It should be noted here that *Bench* would produce benchmarks only if the Latency Emulator is running. This response time is assumed to be a good representation of the amount of time needed for the display to change.

The number of times the refreshing event occurs in a cycle can be specified (event rate). The effect of this event on the client is either the delay in going into local mode or the end to localization, which is exactly the feature needed to produce heterogeneous

typing scenarios for testing KB Pro. However, the position where these events occur during the cycle should not be fixed since the effects of every cycle on the client would be constant. Thus, a certain degree of randomness needs to be added to the timing of these events. This randomness would add reality to the Benchmarks and at the same time maintain the number of events per cycle. The timing of all refresh events will be determined prior to the execution of the benchmark and will be based on the event rate and the number of cycles. For clarification purposes, the timing will actually be the index of a character on the cycle after which the event should occur.

Keystroke Generation Scheme

Another factor considered while designing *Bench* involves the transition period between local and remote modes. Remember that *KB Pro* handles user activities while shifting from normal mode to local and not during the transition from local to normal mode. Thus, *Bench* needs to be aware of such scenarios and does so by making function calls to *VdHook*. This function will tell if KB Pro is in normal, local, or transition mode. If the client is in transition, *Bench* delays the next key event until the client gets out of transition. While this is not required when shifting into local mode, this feature is implemented anyway in order to simplify *Bench*. Note that the effect of this waiting time on the measurements will be negligible.

In order to make the benchmarks more human, the rate of key event generation was considered. Lag between key events will produce a behavior that may make the visual appearance of benchmarks more authentic. But the purpose is defeated if there is a constant lag between each key pressed! A simple observation of common typing behavior will show that users normally type a set of 4 to 5 keys with very little time lag in between

the keys. However, the lag between the 5th key and 1st key of the next set is more - a distinct pause! The user normally waits for or takes notice of the display of the 5 characters before moving to the next set. *Bench* produces key events to mimic this behavior. If the client is in local mode, the time gap may between sets is appropriate to the response time of the client. During local mode, the gap is a pre-defined value that is apt for immediate display of characters typed.

In summary, the following describes algorithm implemented by *Bench* to produce benchmarks:

```
If there are events in Benchmark
    Randomly determine the timings;
While there are keys to generate{
    If in transition mode
        Wait till mode changes;
    Generate the next key event;
    Wait for time lag between keys;
    If the key pressed is the last of a set of 5
        Wait for time period between sets (time based on client mode);
    if a key event should be generated at this time
        generate it;
    }
```

The Stopwatch and the GUI

Since the built in logging features of the ICA Client and the Performance Monitor are utilized for measuring communication and local resource overheads respectively,

Bench is designed to provide the starting and ending times of a particular benchmark. This not only gives the duration of the benchmark (one of the measurements) but also provides the time bounds of the log file to focus on for obtaining overhead information.

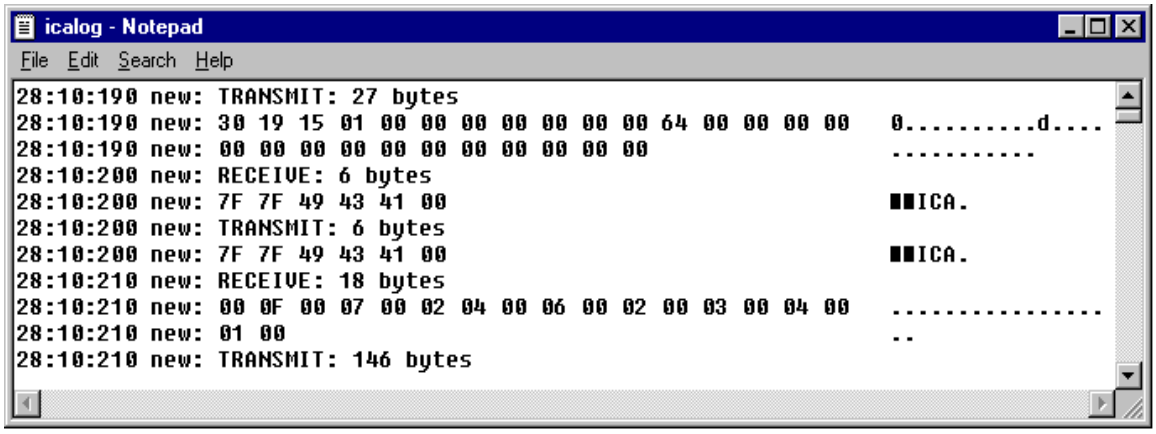

Figure 5: A snapshot of the ICA Client log file

A snap shot of an ICA Client log file is produced above. This file contains information about all packets exchanged during a session, including the number of messages and bytes sent and received. Observe that the first and seconds numbers on each line are the minutes past the hour and the number of seconds past the minute. Since no session will run for an hour during experiments, the hour of the experiment can be ignored. If the starting time of the experiment is 28 minutes and 10 seconds and the ending time is 29 minutes and 32 seconds, all the logging done inside this time period can be parsed to measure communication overheads. For this purpose, a simple file parser was developed to parse a log file between the starting and ending times and obtain values such as number of messages sent, bytes sent, messages received, and bytes received for that time period.

In order to facilitate the specifications of Benchmarks, a simple GUI is provided for *Bench*.

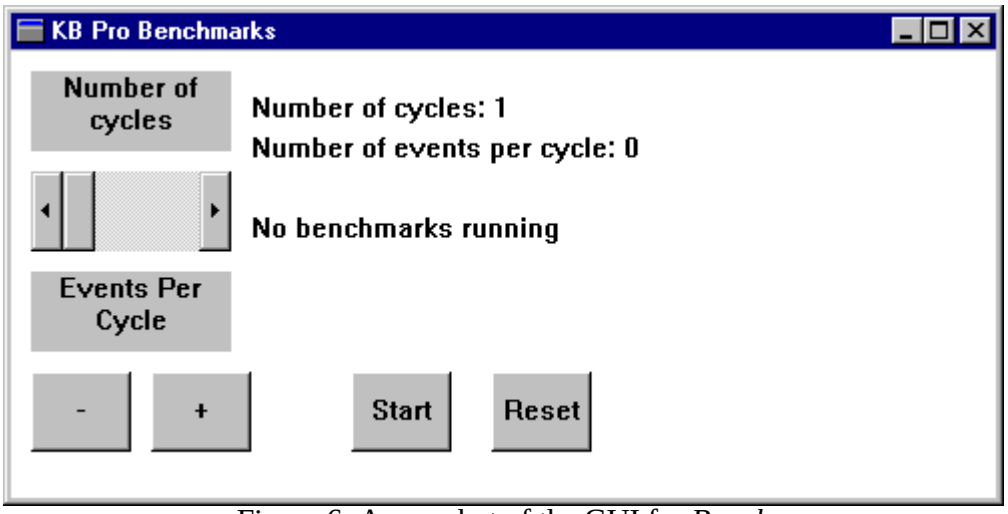

Figure 6: A snapshot of the GUI for *Bench*

This allows for setting the number of events per cycle and the number of cycles to run. And a reset button is provided in case the Benchmark needs premature termination. During the run of a Benchmark, the GUI displays information such as the current character position in the cycle, the current cycle number, the mode of the client, the latency in the network, and the timing of any events during the cycle.

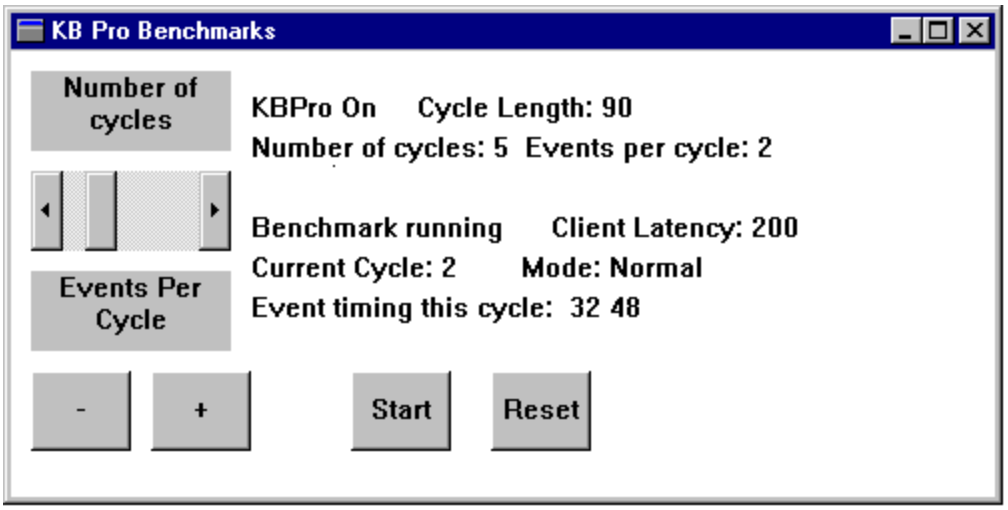

Figure 7: A snapshot of *Bench's* GUI during a benchmark

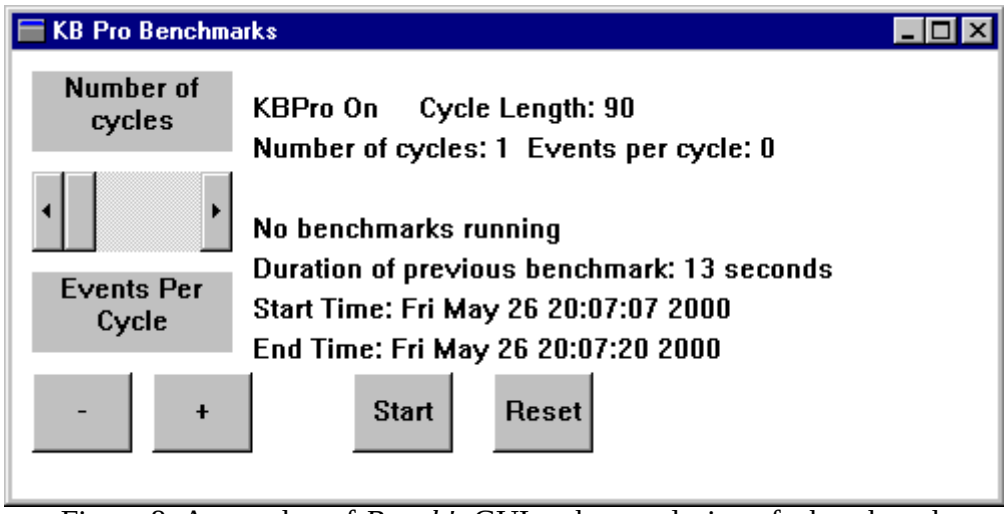

Figure 8: A snapshot of *Bench's* GUI at the conclusion of a benchmark

Network Emulator

The thin client prototype, as described in chapter 3, runs on a wireless private network with a raw bandwidth of about 2 megabits per second and a ping time measurement less than 10 milliseconds. In order to simulate the performance of the thin client in communication channels that are much slower, a network emulator has been developed. This emulator is implemented using two Virtual Channel Drivers (VDs) that

communicate separately with one server side application using two VCs. The server application provides a GUI to set the network latency to particular values.

Design Scheme

The design scheme of the emulator exploits the single-threaded feature of the ICA Client. VD developers are warned against including blocking code in the DLLs since the entire client blocks and thus slows down. This feature along with that of the *DriverPoll()* function being called periodically by the WinStation Driver (WD) provides the perfect framework for artificially slowing down the client. If a VD is made to sleep for a certain time period inside the *DriverPoll()* function definition, the entire client waits for that time period and eventually slows down. This blocking code in one of the VDs delays all other client functionality, including other VDs and, most importantly, the segment that sends messages to the server and receives display refreshes. The delays that occur from this scheme actually occur between the WD and the underlying protocol. While the messages to and from the client machine are delayed only as much as the network latency, the time it takes for the WD to write on or read from the connection is delayed; this effectively increases the latency in communication between the client and the server machines. Since this emulator is implemented using VCs, it can be utilized irrespective of the underlying protocol used by the ICA connection.

As assumption made about VCs here is that the WinStation Driver polls all VDs before sending any kind of packet to the server. The sequence in which the VDs are called may be the sequence by which they are listed in the module.ini file. Thus, if the delaying VD's name is listed last, a packet from any other VD is sent only after the *DriverPoll()* function of the delaying VD completes execution. A drawback to this

scheme is that the client is slow even when there is no communication: however, since most client activities are based on communication, this factor can be tolerated.

This design scheme cannot be proven correct since the frequency of calls made to the *DriverPoll()* function in each DLL cannot be ascertained from the information provided in the VC SDK documentation. For this purpose, another VD is used in order to measure latency. The VD sends packets to the server that returns them immediately and the roundtrip time for the packet is measured, half of which is the approximate latency. This number is a good representation since the client and the server operate on a private network and the server is freed of all other duties except than handling the thin client. And since this emulator is used for experimental purposes, other activities on the client can be terminated. The numbers generated by this VD after inducing delay in the other VD consistently showed that the delay scheme produces network latency very close to the desired value. In addition, it was observed that the response time measured by *VdHook* also closely matched the latency specified in the emulator. The abnormal accuracy in these results may be attributed to two facts:

- The clock resolution in the Windows NT operating system is only 10ms
- The latency on the ad-hoc mode remains consistently below 5 ms, a latency that is virtually 0

The result is that normal network latency cannot be accurately calculated since all Pings would have roundtrip times less than 10ms. Since the roundtrip times for these Pings would be recorded as 0 due to the clock resolution, the detection scheme would deem roundtrips of such packets as 10 ms. Therefore, the latency produced after manipulation is assumed to be a close approximation of the desired network latency.

Bandwidth Manipulation: Future Work

During the development of this emulator, an attempt was made to artificially control bandwidth between the client and the server. The idea was that a tandem of processes (not VC implementations) working on the client and server side would eat up the difference in the current bandwidth and the desired bandwidth through a TCP connection. These processes would continuously exchange a certain number of bytes every second. However, the accuracy of this scheme could not be confirmed since the second pair of applications used to measure the controlled bandwidth started eating the bandwidth used by the bandwidth hogging processes! Thus the bandwidth control feature was removed from the emulator design; fortunately, the latency control scheme succeeded in slowing down the client as needed.

Bandwidth manipulation can be a cause for future work on the Latency Emulator. The bandwidth may be controlled by another VD that writes a certain number of bytes during every *DriverPoll()* function call. And unlike the VD for delays, this VD would be listed first in the module.ini file! However, more information regarding the mechanism of VDs may be required to make this scheme accurate. For instance, the frequency of calls to a VD's *DriverPoll()* should be known; also, the mechanism of how a VC writes may be required in terms of how the data are sent across the underlying connection. Note that a server process for the VC can be implemented to eat bandwidth on server to client communication. Even if this process was implemented and used during experiments, it would have seriously affected the communication measurements of each benchmark. Note that the delaying VD never sends data! The complexity of these issues coupled with time restrictions make bandwidth manipulations beyond the scope of this thesis.

Implementation

As mentioned earlier, there are three components to the client emulator: *VdPoll*, *VdEmulLatency*, and *EmulatorServer* (ES) and they too share a header file. *VdPoll* communicates with *ES* over the VC "ctxpoll" and VdEmulLatency uses the VC "ctxemul" When *ES* is started, a message is sent to *VdPoll* to measure the network latency. *VdPoll* uses the same scheme as *VdHook* in this process; it sends a series of Pings (the number of Pings is specified in the header file) to *ES* and half of the average of the roundtrip times is designated as the network latency. This is a very reasonable estimate since the thin client runs on a private network and the server is free of other duties. *VdPoll* then sends a packet to ES with the latency value after which the user is allowed to specify desired latency values. When a change is requested, ES sends a message to *VDEmulLatency* with the time period to sleep during every *DriverPoll()* call; this number is the difference between the desired latency and the network latency detected by VdPoll. When *VDEmulLatency* gets this message, it decodes it to get the time period to sleep. Actually, the *DriverPoll()* function sleeps for a time period double that of the desired latency since no corresponding delay mechanism can be implemented at the server; it should be mentioned here that the default sleep value is zero when the client is allowed to work without any latency manipulations. Once *ES* communicates this number to *VdEmulLatency*, it directs *VdPoll* to start another series of Pings to measure the new latency, which should be very close to one requested. If the latency measured is within a reasonable range from the desired latency, the change is a success. The value of this range is 10ms, which is not only a tight restriction but also suitable to the clock resolution on Windows NT. If the change is not a success, the user will have to try again. It is

worth noting that the resulting network latency matched the desired latency in a very large majority of tests.

I acknowledge the fact that this Network Emulator induces only constant delays into the ICA Client and not variable latencies within a reasonable range of the latency desired, as may be the real world case in wireless networks. However, the sole purpose of this Emulator is to delay the client to certain values so that experiments can be performed even over fast connections.

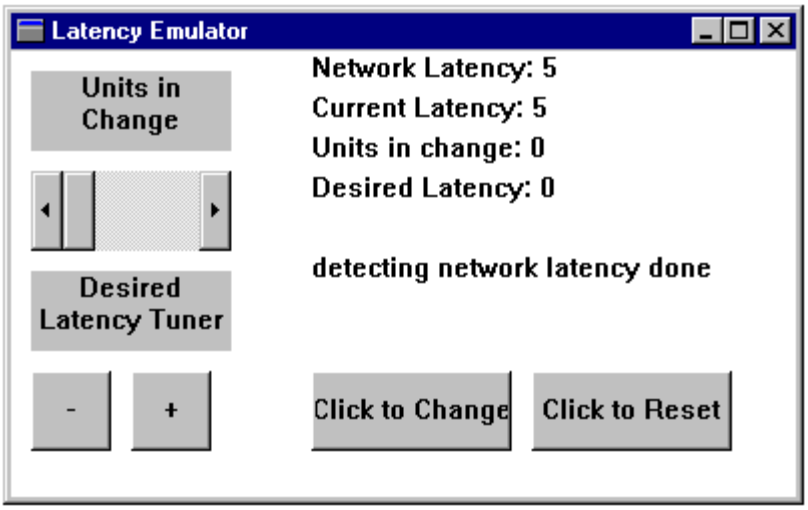

Figure 9: A snapshot of the Latency Emulator GUI

Performance Monitor

The Windows NT Workstation 4.0 operating system comes with a built-in performance monitoring tool called the Performance Monitor (we shall call it Monitor for convenience). This applications runs a GUI by which the user can set counters such as memory and processor usage to be monitored and the display will produce real-time graphs, reports, or log files as chosen by the user. When Monitor produces log files, the data from these files can be used as input for producing graphs and reports. In addition,

time periods within the duration of the log file can be specified to produce data for that period. This feature is exploited during the experiments on KB Pro; since *Bench* produces the starting and ending time of a benchmark run, time periods within a log file can be clearly specified. The result is that processor and memory measurements for that time period can be extracted and added to the other measurements of that experiment. However, we should note that these values are for the entire system during the experiment and not for a particular process. Monitor does provide for specific process monitoring but with the current implementation of *KB Pro*, utilizing this feature is rendered impossible. It should be noted that Monitor does provide for monitoring other systems counters; but they are ignored them due to lack of knowledge regarding their with relevance to the evaluation of *KB Pro.*

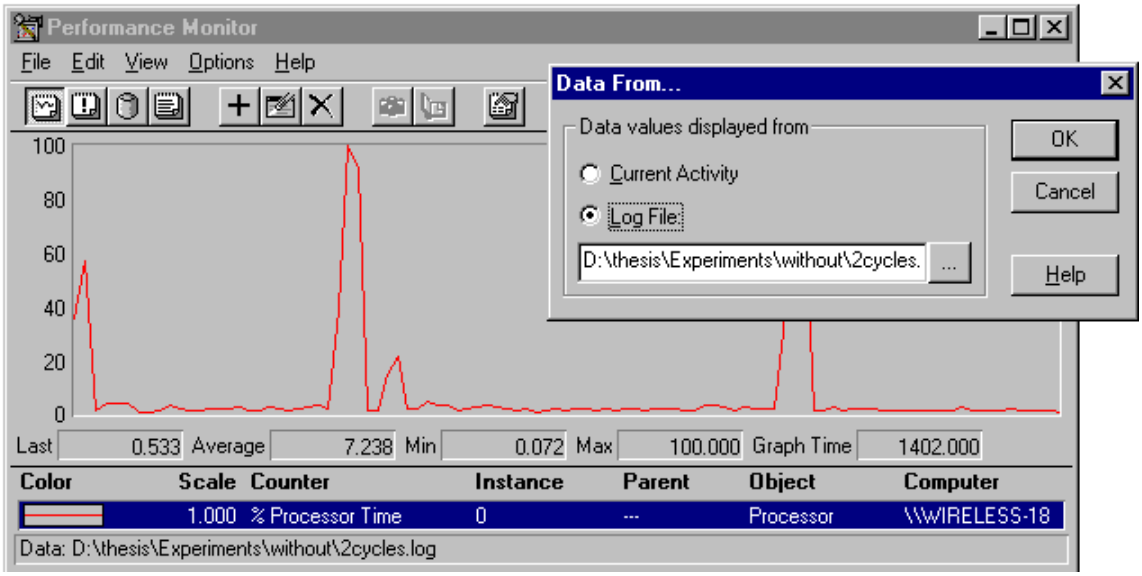

Figure 10: A snapshot of the Performance Monitor GUI

Experimental Variables

Measuring the utility of the *KB Pro* should ideally be done by comparing data collected when the client does and does not use the localization system. While the main motivation of the *KB Pro* system is to improve client performance under high latency in communication, the design of the system is based on two aspects. They are increasing user satisfaction with the thin client and keeping the burden placed on the resources of the client machine and the server at a minimum. The choice of data collected from the experiments should be based on the above two.

Since user satisfaction is subjective, choosing variables that reflect user satisfaction may produce results that are subjective also. However, when the core issue is keyboard activity and thin network latency, *time* can be designated as a measuring scale for user satisfaction (*time is money*!). Personal experiences will show that we type faster in fast telnet connections than in slow ones. When the user immediately sees a set of characters she typed on the screen, she moves on to the next one; if the display is delayed, she either waits or types it in again depending on her patience and/or her knowledge of the delays! Or she might think that the connection has been terminated! One can argue that confident users who are aware of the delay may not wait or retype but this may be the case only when the amount of typing done is little. When the user is typing a document, what she types may require seeing what was typed immediately before. And for all practical purposes, the lay computer user likes to see what was typed as soon as possible.

Adopting this principle, the time required to run a benchmark on a client with KB Pro can be compared to the time required to run the same on a client without KB Pro. Since *Bench* is designed in such a way that the typing behavior generated reasonably

matches typing instincts of humans, the time measurements will be a good representation of the time humans would take to type in the same text. And the difference in these measurements should give us a reasonable index of increase in user satisfaction.

Apart from measurements related to time, measurements on the amount of communication between the server and the client should also provide evidence to the benefits of KB Pro. In fact, it can also be used as means to indicate increase in user satisfaction: if the number of packets exchanged is lower with *KB Pro*, then the wireless network bill will reduce and the user's satisfaction will increase! Intuitively, the amount of communication should decrease when the client is in local mode; each keystroke does not result in messages being sent back and forth on the connection and all the text typed locally is refreshed with one message through the Virtual Channel (VC). Even though there is communication overhead on the VC while starting and ending localization, they are negligible; the measurements on communication are expected to be in support of localization. The communication done will be measured using the logging feature built into the ICA Client that will log all messages received and sent by the client. *It is assumed that the logging includes communication on all VCs since the tutorial of the Program Neighborhood application states that all packets sent and received by the client will be logged*. Since most wireless networks are packet based, the number of packets exchanged is more important than the number of bytes exchanged.

In summary, the following will be measured and compared:

- *Benchmark duration*
- *Number of messages sent*
- *Number of messages received*

- *Number of bytes sent*
- *Number of bytes received*

The remaining choice of variables will provide information with respect to the drawbacks of localization. These are burdens placed on the resources of the client and server machines such as memory and processor usage. The performance monitor built into the Windows NT Workstation operating system will be used in this measurement process. Intuitively, these resources should be used more when *KB Pro* is part of the thin client system and even more when localization occurs during that benchmark. During experiments, the specific counters that will be monitored are:

- *The average percentage CPU time*
- *The average percentage committed bytes in use*

Experiment Mechanism

Conducting experiments to evaluate the *KBPro* involves multiple applications within and outside the ICA Client. First, *Bench* needs to be running on the same machine as the ICA Client and so should the performance monitor, with logging information set. An ICA Client should be running with the Latency Emulator and, in half of the experiments, *KBPro* should be running. Once an experiment ends, measurements have to be taken and some applications may need to be closed or reset. Having a process that automates all the above is beyond the scope of this thesis! Simpler means are employed even while maintaining the control, and thus the authenticity, of the experiments.

The settings for each experiment will be from the following choices:

- Using *KB Pro* and without using *KB Pro* (2 choices): comparison of benchmarks with the same settings in these two groups will provide proof of KB Pro's utility.
- Benchmark lengths of 2,5, or 10 cycles of 90 characters per cycle (3 choices): this will show how effective KB Pro is with different volumes of typing activity.
- Network latency values of 100, 300, or 500 ms (3 choices): 100 ms was chosen to show the effect of *KB Pro*'s presence without any localization. Since even 500 ms latency proved cumbersome, higher latencies were avoided.
- Refreshing event rates of 0,1, or 2 per cycle (3 choices): a refreshing rate of 0 is perfect for *KB Pro* but not too many people work this way! 1 is an ideal setting for *KB Pro* to show its "real world" utility. But 2 events per 90 characters will be an effective setting to evaluate *KB Pro* for typing behaviors unsuitable for typing.

Thus, a total of 3 * 3 * 3 * 2 = 54 experiments will be performed. The experiments will be grouped together based on the hierarchy formed by the sequence of the above settings listing. Experiments are first divided into two basic (a) groups: those that involve *KB Pro* and those that do not. For each (a) group, the experiments are divided into sub groups (b) that use benchmarks of the same length. These (b) groups are further subdivided into groups (c) that use the same network latency: all (c) groups will consist of three experiments based on the refresh event rate settings. Logging in the Performance Monitor is started before the first experiment in a (b) group and is ended when the last one in this group is completed. Before each experiment in a (c) group, the desired latency is induced using the Latency Emulator. Following this, *KBServer* is started and minimized for experiments involving *KB Pro*. It is closed after the last experiment in a (c) group and restarted once the latency is changed to for the next (c) group.

The following processes should be running on the client machine before an experiment begins:

- *Bench*
- ICA Client session
- MS Word in the client session (MS Word will be the default typing application for experiments)
- Latency Emulator in the client session
- The Performance Monitor

The following are done before any experiment starts:

- The MS Word window inside the ICA Client will be given keyboard focus.
- Values are set in *Bench* according to the kind of benchmark desired and the benchmark is started

The following remain constant throughout all benchmarks:

- Client window dimensions (in pixels): horizontal resolution: 640 and vertical resolution: 480
- The MS Word window should be a "blank document" with no previously typed text and it is maximized inside the client display. This provides a maximum localizable area of dimensions (in pixels): horizontal resolution: 590 and vertical resolution: 306
- The user applications that run on the client machine are: ICA Client, Performance Monitor, *Bench*, and a MS Excel window to record experiment results
- *there may be system applications running in the background that are not under user control.*

- The MetaFrame server services only the ICA Client used in the experiments and no other applications execute in it during the experiments.

At the conclusion of each experiment, the starting and ending times of the benchmark are recorded against the settings for the benchmark (number of cycles, event rate, and latency). At the conclusion of a group of experiments using the same benchmark length, the following is done:

- The client is closed and this will make the ICA log file available for use
- Logging is terminated in Monitor.
- The time period recorded for each benchmark is used to set time periods in the log file produce by Monitor; from these time periods, the variables related to processor time and memory usage are obtained
- The same time period is used to parse that section of the ICA Client log file and the values of the communication variables are obtained.
- All these values are recorded against the benchmark settings for that experiment

Experiment Data and Analysis

As mentioned, a total of 54 experiments were performed in order to gather data and evaluate *KB Pro*. These data will be presented in two sections. First, tables will be provided to show the time and communication measurements of all benchmarks. In addition, pairs of experiments with the same settings, one performed without *KB Pro* and one with *KB Pro*, will be compared. This comparison will give testament to the utility of the localization system with respect to time and communication savings. The second section of data presentation will consist of tables containing information about the usage

of memory and processor resources during the experiments. And like the previous section, one-on-one comparisons of experiments with the same settings will be made to show the utilization of local resources with and without *KB Pro*. The section will conclude with a few selective graphs of experiment results.

Time and Communication Data

The first set of data presented here was collected from experiments performed without *KB Pro*. The second set was collected from experiments performed with *KB Pro* integrated with the client. For the sake of comparison, a third set is presented. Pairs of benchmarks, one from the first set and the other from the second, with the same settings are compared against each other with respect to time and communication measurements. This comparison will show the utility of *KB Pro*:

- First, the decrease in benchmark time will give an indication as to the improvement is user experience with respect to time saved in typing
- Second, the decrease in the amount of communication will give an indication in the increase user satisfaction with respect to the wireless communication bill.

Table 3: Time and communication measurements without *KB Pro*

Cycles	Event Rate	Latency (in ms)		Duration (in secs)	Packets Sent	Packets Received	Bytes Sent	Bytes Received
2	0	100		26	400	115	4008	9048
2	0	300		51	401	82	4012	8310
2	0	500		86	399	85	3990	9972
2	1	100		38	405	133	4050	9192
2	1	300		65	413	102	4120	9248
2	1	500		102	411	101	4110	8604
2	2	100		52	422	156	4220	9810
2	2	300		81	424	122	4340	7574
2	2	500		119	417	118	4170	10353
5	0	100		64	998	300	9984	25470
5	0	300		125	998	207	9980	20652
5	0	500		222	1001	221	10012	21858
5	1	100		97	1028	349	10280	23350
5	1	300		163	1024	258	10240	22318
5	1	500		255	1029	253	10290	25336
5	2	100		129	1058	398	10580	23476
5	2	300		201	1056	298	10560	25548
5	2	500		298	1062	295	10620	29246
10	0	100		127	1993	601	19934	47182
10	0	300		255	1996	422	19962	43786
10	0	500		434	2002	432	20020	46300
10	1	100		193	2055	713	20552	48378
10	1	300		326	2060	508	20600	44502
10	1	500		513	2049	511	20490	50444
10	2	100		257	2115	798	21150	50982
10	2	300		401	2117	611	21170	50432
10	2	500		610	2119	606	21190	57274

Table 4: Time and communication measurements with *KB Pro*

Cycles	Event Rate	Latency (in ms)		Duration (in secs)	Messages Sent	Messages Received	Bytes Sent	Bytes Received
2	0	100		25	386	114	3870	10132
2	0	300		31	35	35	368	2704
2	0	500		32	22	31	228	1774
2	1	100		38	402	128	4024	7190
2	1	300		62	114	73	1182	6520
2	1	500		67	89	66	922	7584
2	2	100		51	414	148	4142	7554
2	2	300		85	164	102	1686	11406
2	2	500		116	195	113	2086	14272
5	0	100		63	983	305	9836	25316
5	0	300		68	33	72	346	2154
5	0	500		69	23	67	232	2284
5	1	100		96	1029	358	10292	20818
5	1	300		141	216	176	2232	16042
5	1	500		160	174	155	1814	15750
5	2	100		128	1050	406	10500	24600
5	2	300		215	424	285	4369	33454
5	2	500		244	326	238	3378	29705
10	0	100		126	1990	630	19902	45610
10	0	300		139	72	156	874	12251
10	0	500		146	46	143	508	10728
10	1	100		191	2047	730	20472	48008
10	1	300		277	406	338	4180	34094
10	1	500		315	338	307	3552	37681
10	2	100		256	2112	816	21122	50664
10	2	300		421	434	522	4566	71567
10	2	500		496	658	481	6822	64738

Table 5: Comparison of time and communication measurements (1)

	Cycles	Event Rate	Latency (in ms)		Duration (in secs)	Packets Sent	Packets Received	Bytes Sent	Bytes Received
no KB Pro	2	0	100		26	400	115	4008	9048
with KB Pro	2	0	100		25	386	114	3870	10132
Changes (**increases)**					1	4	1	138	**1084**
no KB Pro	2	0	300		51	401	82	4012	8310
with KB Pro	2	0	300		31	35	35	368	2704
Changes (**increases)**					20	366	47	3644	5606
no KB Pro	2	0	500		86	399	85	3990	9972
with KB Pro	2	0	500		32	22	31	228	1774
Changes (**increases)**					54	377	54	3762	8198
no KB Pro	2	1	100		38	405	133	4050	9192
with KB Pro	2	1	100		38	402	128	4024	7190
Changes (**increases)**					0	3	5	26	2002
no KB Pro	2	1	300		65	413	102	4120	9248
with KB Pro	2	1	300		62	114	73	1182	6520
Changes (**increases)**					3	298	29	2938	2728
no KB Pro	2	1	500		102	411	101	4110	8604
with KB Pro	2	1	500		67	89	66	922	7584
Changes (**increases)**					35	322	46	3188	1020
no KB Pro	2	2	100		52	422	156	4220	9810
with KB Pro	2	2	100		51	414	148	4142	7554
Changes (**increases)**					1	8	8	78	2256
no KB Pro	2	2	300		81	424	122	4340	7574
with KB Pro	2	2	300		85	164	102	1686	11406
Changes (**increases)**					**4**	260	20	2654	**3832**
no KB Pro	2	2	500		119	417	118	4170	10353
with KB Pro	2	2	500		116	195	113	2086	14272
Changes (**increases)**					3	222	5	2084	**3919**

Table 6: Comparison of time and communication measurements (2)

	Cycles	Event Rate	Latency (in ms)		Duration (in secs)	Packets Sent	Packets Received	Bytes Sent	Bytes Received
no KB Pro	5	0	100		64	998	300	9984	25470
with KB Pro	5	0	100		63	983	305	9836	25316
Changes (**increases)**					1	15	**5**	148	154
no KB Pro	5	0	300		125	998	207	9980	20652
with KB Pro	5	0	300		68	33	72	346	2154
Changes (**increases)**					57	965	135	9634	18498
no KB Pro	5	0	500		222	1001	221	10012	21858
with KB Pro	5	0	500		69	23	67	232	2284
Changes (**increases)**					153	978	154	9780	19574
no KB Pro	5	1	100		97	1028	349	10280	23350
with KB Pro	5	1	100		96	1029	358	10292	20818
Changes (**increases)**					1	**1**	**9**	**12**	2532
no KB Pro	5	1	300		163	1024	258	10240	22318
with KB Pro	5	1	300		141	216	176	2232	16042
Changes (**increases)**					22	808	82	8008	6276
no KB Pro	5	1	500		255	1029	253	10290	25336
with KB Pro	5	1	500		160	174	155	1814	15750
Changes (**increases)**					95	855	98	8476	9586
no KB Pro	5	2	100		129	1058	398	10580	23476
with KB Pro	5	2	100		128	1050	406	10500	24600
Changes (**increases)**					1	8	8	80	**1124**
no KB Pro	5	2	300		201	1056	298	10560	25548
with KB Pro	5	2	300		215	424	285	4369	33454
Changes (**increases)**					**14**	632	13	6191	**7906**
no KB Pro	5	2	500		298	1062	295	10620	29246
with KB Pro	5	2	500		244	326	238	3378	29705
Changes (**increases)**					54	736	57	7242	**459**

Table 7: Comparison of time and communication measurements (3)

	Cycles	Event Rate	Latency (in ms)		Duration (in secs)	Packets Sent	Packets Received	Bytes Sent	Bytes Received
no KB Pro	10	0	100		127	1993	601	19934	47182
with KB Pro	10	0	100		126	1990	630	19902	45610
Changes (**increases)**					1	3	**29**	32	1572
no KB Pro	10	0	300		255	1996	422	19962	43786
with KB Pro	10	0	300		139	72	156	874	12251
Changes (**increases)**					116	1924	266	19088	31535
no KB Pro	10	0	500		434	2002	432	20020	46300
with KB Pro	10	0	500		146	46	143	508	10728
Changes (**increases)**					288	1956	289	19512	35572
no KB Pro	10	1	100		193	2055	713	20552	48378
with KB Pro	10	1	100		191	2047	730	20472	48008
Changes (**increases)**					2	8	**17**	80	378
no KB Pro	10	1	300		326	2060	508	20600	44502
with KB Pro	10	1	300		277	406	338	4180	34094
Changes (**increases)**					49	1654	170	16420	10408
no KB Pro	10	1	500		513	2049	511	20490	50444
with KB Pro	10	1	500		315	338	307	3552	37681
Changes (**increases)**					198	1711	204	16938	13763
no KB Pro	10	2	100		257	2115	798	21150	50982
with KB Pro	10	2	100		256	2112	816	21122	50664
Changes (**increases)**					1	3	18	28	322
no KB Pro	10	2	300		401	2117	611	21170	50432
with KB Pro	10	2	300		421	434	522	4566	71567
Changes (**increases)**					20	1683	89	16604	**21135**
no KB Pro	10	2	500		610	2119	606	21190	57274
with KB Pro	10	2	500		496	658	481	6822	64738
Changes (**increases)**					114	1461	125	14368	**7464**

Some of the observations made on the above measurements include the following:

- The amount of packets exchanged for benchmarks of the same length are very close and this is a testament to the control of experiments
- Durations of benchmarks with the same lengths and with no refreshing events are within a very close range, irrespective of latency. This typing behavior is best suited for localization and latency has little bearing on the amount of time required to complete the benchmark
- The duration ranges clearly increase when refreshing events are introduced
- The comparison between benchmarks running with latency below 100 ms shows that the duration, and packets exchanged are within a close range
- There is a consistent decrease in both duration and packets exchanged when latency and event rates increase
- The best results are for benchmarks without refreshing events
- Benchmarks with event rates of 1 and latency greater than 200 ms are realistic typing scenarios when using mobile thin clients for typing applications. The measures for these benchmarks can be marked as the best means to determining the utility of *KB Pro*
- For most cases of benchmarks with event rates of 2, the amount of bytes received is more than when *KB Pro* is not part of the client. The reason for no decrease could be due to the following: as refreshing events increase, the number of times *KBWin* starts and exits during the benchmark also increases. This means that the ICA Client window shifts back and forth from keyboard focus, which might require a repaint of

the window and eventually more bytes being sent by the server. Note that there is clear decrease in number of bytes sent in the same benchmarks.

The results of the experiments, except for the observation above, are in harmony with the anticipated results.

Processor and Memory Measurements

Unlike the communication and time measurements, measuring the processor and memory utilization during the experiments could not be done precisely. Preliminary observations have shown that memory and processor utilization logged by the performance monitor include spikes that sometimes reached 100% processor utilization. This can be owed to the background processes in the system, automated updates of files, etc. Few pairs of experiments, one using *KB Pro* and the other not, were identified without any spikes in either of them. Still, these give only an indication of how much resource *KB Pro* needed and, unlike the measurements on time and communication, definite trends cannot be observed in the results. The graphs shown in the next section have been chosen from these experiments and the only conclusion we can arrive is at is that localization does require *some* memory and processor resources. While this seems obvious, the only doubt in my mind was whether the communication saved by localization offset the processing power required by *KBWin.* Remember, sending and receiving messages does require some processing power! Due to the power of the processor in the client machine, the chances of observing this possibility reduced considerably.

Table 8: Measurements of processor and memory utilization

Cycles	Event Rate	Latency		Without KB Pro Average % Processor Time	Without KB Pro Average % Committed Bytes in Use	With KB Pro Average % Processor Time	With KB Pro Average % Committed Bytes in Use
2	0	100		3.4	39.66	4	42.49
2	0	300		1.56	39.63	6.07	42.85
2	0	500		1.8	39.53	5.2	42.89
2	1	100		3.103	39.52	2.5	42.49
2	1	300		1.41	39.52	3.81	42.8
2	1	500		1.1	39.52	3.08	42.928
2	2	100		2.26	39.52	**54.52**	42.48
2	2	300		1.4	39.52	2.31	42.52
2	2	500		1	39.52	**52.34**	47.75
5	0	100		3.35	39.78	2.86	39.37
5	0	300		**24.92**	39.78	14.48	39.94
5	0	500		**15.165**	39.78	7.25	40.09
5	1	100		2.5	39.8	**38.79**	39.42
5	1	300		1.41	39.78	3.89	39.68
5	1	500		1	39.78	10.22	40.31
5	2	100		2.64	39.8	2.42	39.42
5	2	300		1.56	39.84	3.34	40.06
5	2	500		**11.2**	39.78	2.64	40.23
10	0	100		4.52	37.08	4.68	44.498
10	0	300		**14.03**	37.08	7.61	44.64
10	0	500		8.16	39.78	6.07	44.79
10	1	100		**18.95**	39.85	2.74	44.45
10	1	300		1.63	39.87	12.67	44.717
10	1	500		**10.45**	42.57	3.08	44.89
10	2	100		2.45	39.85	**15.16**	44.37
10	2	300		8.83	39.81	10.115	45.1
10	2	500		**28.79**	42.355	8.61	44.67

Selected results of experiments are presented through the following graphs.

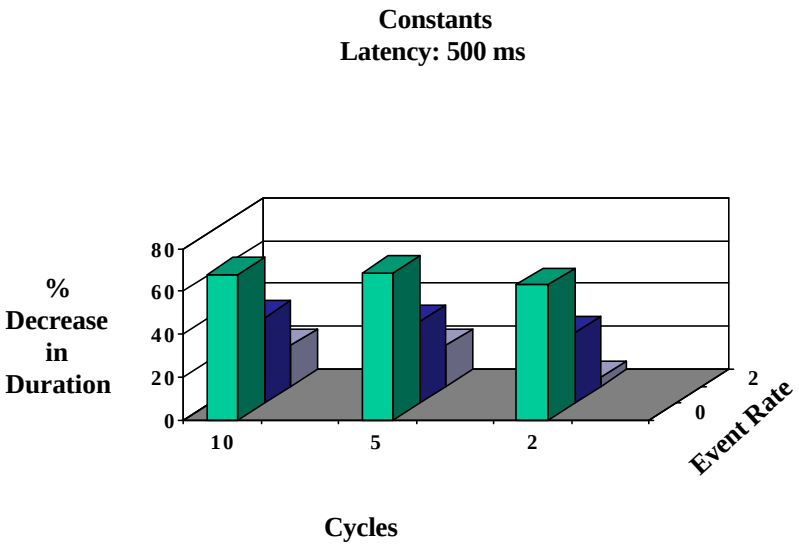

Figure 11: Graph showing results of benchmark durations

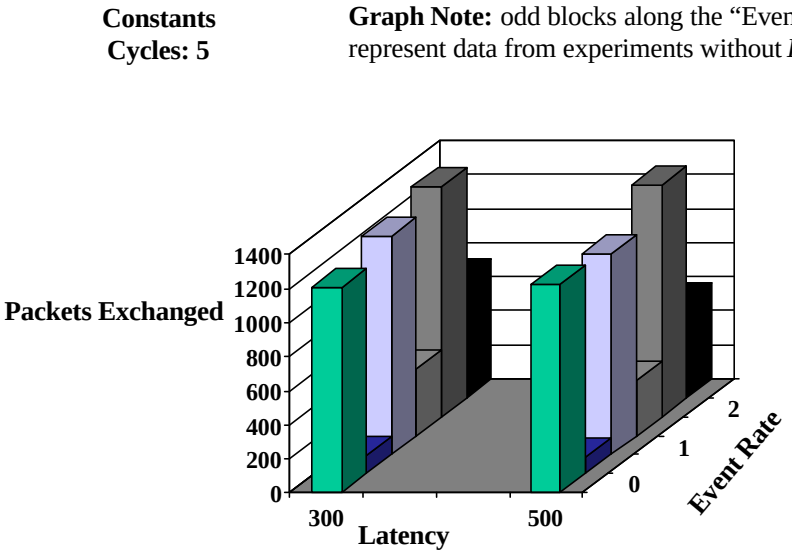

Figure 12: Graph showing effect of refreshing events in packets exchanged

Constants
Event Rate: 2

Graph Note: odd blocks along the "Latency" axis represent data from experiments without *KB Pro*

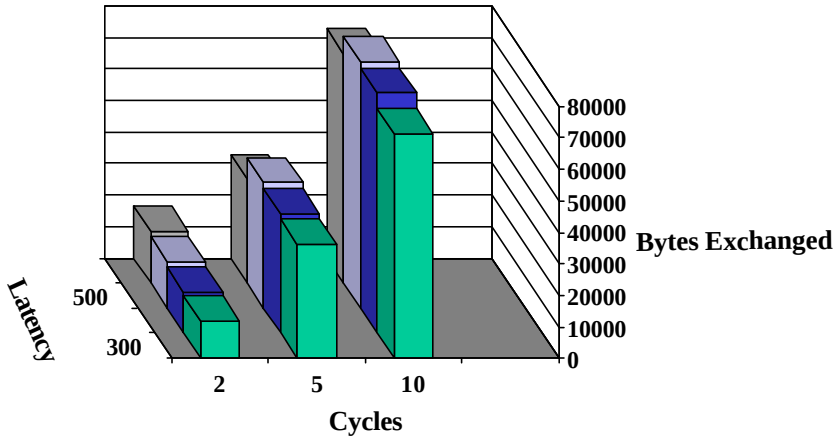

Figure 13: Graph showing effect of refreshing events on bytes exchanged

Constants
Event Rate: 1

Graph Note: odd blocks along the "Latency" axis represent data from experiments without *KB Pro*

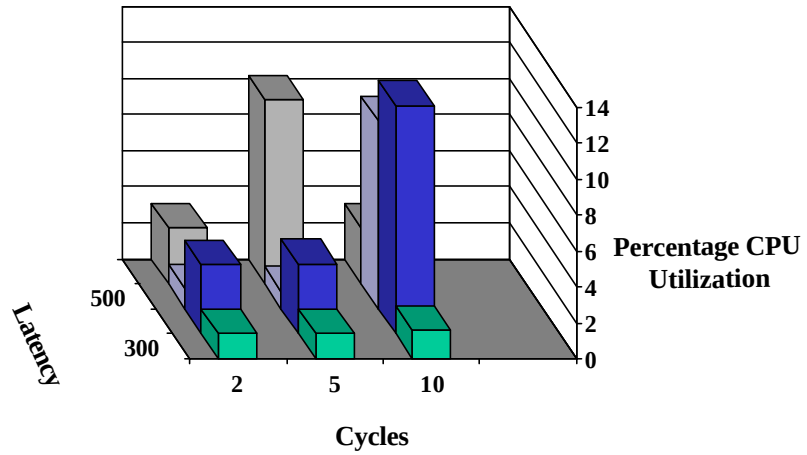

Figure 14: Graph showing average CPU utilization

Constants
Event Rate: 1

Graph Note: odd blocks along the "Latency" axis represent data from experiments without *KB Pro*

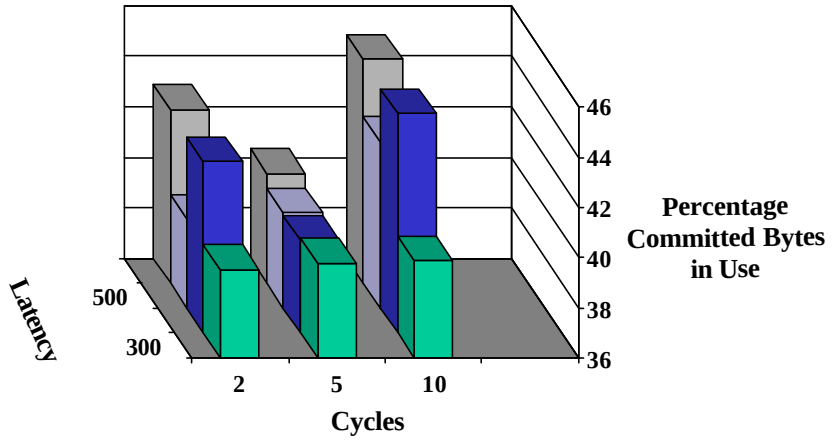

Figure 15: Graph showing average memory utilization

CHAPTER 7
CONCLUSION

This chapter is a brief overview of the goals this thesis has accomplished. It concludes with few notions on future work in the subject of keyboard localization for mobile thin clients.

Goals Accomplished

This thesis has brought into light the potential of thin clients being an ideal computing tool for mobile users; the balance between the benefits and drawbacks of using thin clients certainly appear to lean towards the former. The thesis has surveyed contemporary mobile computing literature with respect to mobile client server computing and has provided the knowledge background required for building systems motivated by mobile computing. It identified localization of keyboard activity during high network latency as a fertile problem for research, especially since high latency is an inherent characteristic of wireless networks. This problem definition is refined to an extent that a localization system can be designed even while retaining its core issues. The thesis then described the thin client prototype, the Win32 Citrix ICA client, and its developmental tools as the platform on which the localization system could be implemented.

The next chapter in the thesis described in depth the implementation of the localization system for the Win32 Citrix ICA client, the bottlenecks posed by the Win32 API in the development process, and the flaws in the final product. Then, the scheme for

experimenting with the localization system to evaluate its utility is elaborated; the tools developed and utilized in the experiments are described and the variables to evaluate the system are enumerated along with the rationale behind their choices. Finally, the experimental method is described before providing the data collected from experiments and their analysis.

Future Work

While the *KB Pro* system exhibits high utility even for its simple design, its drawbacks provide solid grounds for further research on keyboard activity localization. The following are few issues one can ponder and make cause for future work:

- Can the reliability of the scheme to monitor the keyboard activity of the user be increased, especially in improving the chances of lengthening the localization period?
- Can the capability of the localization process increase in order to reduce the scenarios that end localization? Should the versatility of the system increase with respect to heterogeneous applications or should only typing applications be targeted?
- How transparent must the system be to the user? Should the thin client system have complete control of when and how to localize? Or should an interface be provided for the user to exercise these options? Should the control be shared?
- Can more powerful experiments be designed to evaluate the viability of localization?
- Can a reliable mechanism be designed to detect the position of the caret in an application?
- Is there a mechanism for the server side to refresh the text without paste and without sending more than one display update?

- Finally, can *KB Pro*'s components be integrated into the ICA client at the client and server sides?

LIST OF REFERENCES

[1] J. Jing, A. Helal, A. Elmagarmid, “Client Server Computing in Mobile Environments,” ACM Computing Surveys, volume 31, number 2, June 1999

[2] Citrix’s ICA Technology, www.citrix.com/products/ica.asp, May 2000

[3] Citrix’s Server Based Computing White Paper, www.citrix.com, May 2000

[4] M. Ebling, M. Satyanarayanan, “On the Importance of Translucence for Mobile Computing,” First Workshop on Human Computer Interaction with Mobile Devices, May 1998, Glasgow, UK

[5] Carnegie Mellon University’s Coda Project, www.coda.cs.cmu.edu, April 1999

[6] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, K. Walker, “Agile Application Aware Adaptation for Mobility,” Proceedings of the 16th ACM Symposium on Operating Systems, October 1997, Saint-Malo, France

[7] Xerox PARC's Bayou Project, http://www.parc.xerox.com/csl/projects/bayou, April 1999

[8] M. Le, S. Seshan, F. Burghardt, J. Rabaey, "Software Architecture of the InfoPad System," MOBIDATA Conference, Oct 1994, Rutgers University, New Jersey

[9] Microsoft’s Introduction to Terminal Services, www.microsoft.com/windows2000/guide/server/solutions/terminal.asp, May 2000

[10] Citrix MetaFrame for Windows 2000 Services Fact Sheet, www.citrix.com April 2000

[11] Citrix Virtual Channel Software Development Kit Documentation, Version 2.0, Citrix Systems, Inc., May 1999

BIOGRAPHICAL SKETCH

Sivasundar is a native of Chennai, India. Born on October 9, 1976, he attended P.S. Senior Secondary School and graduated with his high school diploma in 1994. After completing his freshman year in India, he was granted academic and golf scholarships to continue his college education in the United States. In Fall 1998, Siva graduated with high honors from Eckerd College in St. Petersburg, Florida, and earned a bachelor's degree in computer science. He immediately went on to pursue a master's degree in computer engineering at the University of Florida in Gainesville, Florida. Upon completing his master’s degree, Siva will join Hewlett Packard in Cupertino, California. A sports fanatic, his passions include golf, American football, and tennis.