# USER-ACTIVITY AWARE STRATEGIES FOR MOBILE INFORMATION ACCESS

A Thesis
Presented to
The Academic Faculty

by

Tae-Young Chang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
April 2008

# USER-ACTIVITY AWARE STRATEGIES FOR MOBILE INFORMATION ACCESS

*To my family*

## ACKNOWLEDGEMENTS

I would like to acknowledge many people for helping me during my doctoral work.

First of all, I would like to extend my sincere gratitude and appreciation to my advisor, Professor Raghupathy Sivakumar, for his assistance, advice, and support during my study. Throughout my doctoral work, he encouraged me to develop independent thinking and research skills. He continually stimulated my analytical thinking and greatly assisted me with scientific writing.

I would like to thank Prof. George Riley and Prof. Chuanyi Ji for serving on my proposal and dissertation committees. I also thank Prof. Magnus Egerstedt and Prof. Umakishore Ramachandran for serving on the dissertation committee. They gave valuable opinions during the preparation and presentation of this dissertation, and I am deeply thankful to them.

I thank the present and past members of the GNAN research group for their warm advice and support. Special thanks to Aravind, Zhenyun, Karthik and Kyu-Han for their collaboration in my research. My thanks also go out to Ashraf, Hung-Yun, Seung-Jong, Ram, Yujie for their friendship and assistance. Sandeep, Sriram, and Cheng-Lin shared their constructive comments on my research, and I wish to thank them. I thank Dr. Yeonsik Jeong for his valuable advice.

My very special thanks to my parents and sisters. Their unwavering faith and confidence in my abilities is what has shaped me to be the person I am today. Finally, I woud like to thank my wife, Yejin, and my daughter, Katherine Yuna for their loving support, encouragement, and sacrifice during my study.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

Information access suffers tremendously in wireless networks because of the low correlation between content transferred across low-bandwidth wireless links and actual data used to serve user requests. As a result, conventional content-access mechanisms face such problems as unnecessary bandwidth consumption and large response times, and users experience significant performance degradation. In this dissertation, we analyze the cause of those problems and find that the major reason for inefficient information access in wireless networks is the absence of any user-activity awareness in current mechanisms. To solve these problems, we propose a new paradigm for mobile information access, which is driven by awareness to user activity.

To tackle the inefficiency problem of read operations in conventional access systems, we classify content into *non-partitionable* single-file content, which cannot be partially accessed in the file-system level, and *partitionable* multiple-file content, which enables an application to access its partial set. Similarly, we also categorize write operations into *content synchronization* and *content generation*. In this dissertation, we present three user-activity-aware strategies for the first three scenarios.

First, for reading *non-partitionable* content, we present an application-unaware strategy called *Cut-Load*, which performs content partitioning in the graphical domain by using user-activity awareness provided by thin-client computing. To improve access performance, Cut-Load selects the best computing mode for requested content, hoards original content in background, and transfers computing modes transparently.

Second, for read operations of *partitionable* content, we propose an application-aware strategy called *Prioritized Fetching*, which performs location-based prioritization for objects and downloads them based on their priority levels. To maximize performance benefits, the strategy also uses an intelligent mix of dynamic object reordering and connection management.

Finally, for bandwidth-efficient content synchronization, we present an application-unaware scheme called *Mimic*, which relies on transferring user activity to the server for file synchronization. To minimize transfer size, Mimic performs optimization for user-activity records at the client, regeneration of input system messages at the server, and verification for the replayed activity.

To evaluate these strategies, we perform *ns2* simulations and real-life experiments using simple prototypes, which are implemented in current operating systems. Then, we show that our strategies can outperform conventional information access schemes in terms of bandwidth consumption and user-perceived response times.

# CHAPTER I

# INTRODUCTION

The rapid emergence of sophisticated mobile computing technology has accelerated the recent growth in the number of nomadic users, who access information using mobile devices. However, the increasing ubiquity of mobile computing poses major challenges to current information access models because of inefficient design. A typical mobile computing environment is one that consists of a mobile host communicating with backbone servers in the wired network through one or more wireless links. In this environment, conventional access models lead to problems of low bandwidth, large response delays, frequent disruption in connectivity, and low capacity shared among multiple users. Some wired links such as dial-up modem lines and ISDNs have the similar problems, however in this work we do not consider such wired networks.

Mobile computing in the broad sense includes a variety of components such as mobile information access, mobility management, database management, mobile communications, wireless networking, and so on. The scope of this dissertation is restricted to *mobile information access* [81]. We consider all the work performed by the mobile host as remote information access, including Web browsing, file transfers, and X Terminal accesses, under the term, mobile information access.

In the past couple of decades, a tremendous amount of research has been done in the area of mobile information access. Mobile file systems, like AFS [30], Ficus [27], and Coda [82, 36], provide the ability of disconnected operation when the mobile host loses network connectivity. Bandwidth-efficient file-transfer technologies, such as Low-bandwidth network file system (LBFS) [58] and operation shipping [43], significantly minimize network traffic between a mobile host and a server. Mobile prefetching schemes, such as power-aware prefetching [102, 83], bandwidth-adaptive prefetching [33], and SPREE [39], reduce access delay by utilizing limited resource intelligently.

Most applications today are accessed by the user using a graphical user interface (GUI), where the user interacts with an application through a pre-defined interface that recognizes user activity and sends user inputs to the application. However, since the underlying access schemes are unaware of the nature of user activity that triggers read and write accesses by the application, those access operations cannot be performed efficiently. For example, consider the scenario where a user wants to view a document from a backbone server. A normal access scheme, which is not aware of the exact needs of the user, would fetch the entire document, irrespective of the parts of the pages that the user finally gets to view. This leads to the retrieval of information that is not ultimately used by the application. This drawback is exaggerated by the fact that the mobile host is connected by a wireless link, where the bandwidth of the link is a precious commodity.

In this dissertation, we propose a new paradigm for mobile information access, which is driven by awareness to user activity. We call this *user-activity-aware* mobile information access. Our paradigm focuses on the two primary operations in mobile information access: retrieving of files from a backbone server, *i.e.*, *read* access, and updating cached content to a backbone server, *i.e.*, *write* access. Since the read and write accesses are the most frequent operations in mobile information access, performance benefits obtained through the optimization of these operations can improve user performance significantly.

To tackle the inefficiency problem of read operations in conventional access systems, we classify content into two types: *non-partitionable* content and *partitionable* content. The non-partitionable content consists of only a single file, and the underlying access scheme cannot perform any partial access in the file-system level. On the contrary, the partitionable content is composed of multiple files, including a main document and other object files, and the application may access only a partial set of the content if necessary. Similarly, we also categorize write operations into two types: *content generation* and *content synchronization.* The former is a write access triggered by a user who wants to generate new content, whereas the latter is executed by an update operation upon the existing content at the server.

In this dissertation, we present three user-activity-aware strategies for three different scenarios, reading non-partitionable content, reading partitionable content, and content synchronization, as follows:

For reading non-partitionable content, we present an application-unaware read-access scheme called *Cut-Load*, which performs user-activity-aware read operations in a graphical domain. For efficient mobile information access, Cut-Load consists of three design elements: *dynamic mode selection*, *opportunistic hoarding*, and *transparent mode transfer*. Through simulations and a prototype, we compare its performance with conventional access mechanisms and show that the proposed middleware brings significant performance benefits both in terms of bandwidth consumption and user-perceived response times;

For user-activity-aware read operations of the partitionable content, we propose an application-aware Web-acceleration scheme called *Prioritized Fetching*, which performs object prioritization and non-greedy object fetching. To maximize response performance, it also uses an intelligent mix of object reordering and connection management. Through *ns2* simulations and a prototype using Internet Explorer, we show that our scheme shows better performance then current browsers in screen response time.

For bandwidth-efficient content synchronizations, we present an application-unaware scheme, called *Mimic*, which relies on transferring raw user activity or differential update selectively to the server for bandwidth-efficient file synchronization. Through a simple prototype, we show that raw-activity shipping can outperform the differential update schemes under many common conditions. We also identify the conditions under which the differential schemes do perform better than activity shipping, but show that the detection of such conditions is straightforward, thus enabling both update schemes to be used in tandem with a mobile file system for bandwidth-efficient file synchronization.

The rest of this dissertation is organized as follows: Chapter 2 elaborates on the problem setting and the challenges faced by conventional information access schemes. Chapter 3 describes the design philosophy and present three user-activity-aware strategies by providing a high-level overview. Chapters 4 presents the details of the user-activity-aware scheme for reading non-partitionable content. Chapter 5 describes the details of the user-activity-aware scheme for reading partitionable content. Chapter 6 illustrates the details of user-activity-aware content synchronization scheme. Finally, in Chapter 7, we conclude the dissertation.

# CHAPTER II

# MOTIVATION AND RELATED WORKS

In this chapter, we first describe the problem setting and the challenges faced by conventional access schemes in improving the performance of mobile information access. Then, we discuss the related works.

## *2.1 Problem Setting*

The target host for this work is a mobile host such as a laptop computer or a handheld device with a wireless network interface. The target mobile host is capable of running applications in a stand-alone fashion. It has enough system resources and battery capacity to run all the applications. The mobile host accesses information from backbone servers using a wireless link. The wireless environment, unlike its wired counterpart, is characterized by limited bandwidth and significant latency. Wireless bandwidth is very limited and expensive. Frequent fluctuations of available bandwidth or disconnections interrupt wireless transmissions, and the performance gets significantly degraded when a user moves in a high speed.

We consider the following mobile information access scenarios in this dissertation:

- **Retrieval of content from a backbone server:** In this scenario, any non-local information access performed by a mobile host results in sending the request to either a backbone or proxy server close to the mobile host. The data retrieved by the mobile host, which acts as a client in this scenario, can be cached locally for future requests.

- **Update of cached content to a backbone server:** As discussed earlier, in order for the mobile host to access data even during network disconnectivity, file systems such as Coda [82] provide a mechanism for accessing and updating the locally cached data. On reconnection with the server, the file system performs synchronization of the local copy with the server to maintain the consistency semantics.

Given the above-mentioned scenarios, the problems we consider are optimization of bandwidth consumption, latency incurred, and overall system utilization, both during information retrieval from the backbone servers and during synchronization of updated information with the server upon reconnection.

## *2.2 Challenges*

Now, we discuss two major challenges, which are faced by conventional information access approaches in achieving optimal performance for user-activity-aware mobile information access.

### 2.2.1 Greedy Content Fetching

In current approaches, content requested by an application is retrieved in entirety from a server, irrespective of whether or not the entire content is eventually viewed by a user. Since the underlying file systems are user-activity-unaware, they are not able to differentiate between essential parts of the content and the unnecessary part that will not be used by the application, and always use a greedy downloading technique to retrieve the entire content. Greedy downloading essentially means that the time to serve a user request is tied to the time to fetch the entire content. It also increases the peak load of the system, thereby bringing down the overall system utilization.

In an ideal scenario, user requests should be served within a short duration, and this should be decoupled from the amount of time taken to get the entire content to the mobile host. This can be solved by partitioning the entire content into smaller chunks so that only those chunks of data that are actually accessed by the user are downloaded with high priority. This is especially efficient when the raw data size of the entire content is very large compared to the amount of the file that the user actually views.

Thus, one of the goals of our work is to incorporate the notion of user-activity-aware content partitioning to enable *intelligent retrieval* of content from the server. This obviously would lead to maximizing user-perceived performance and optimizing precious wireless bandwidth.

### 2.2.2 Non-Linearity in Size between User-Activity and File Update

Here, we are concerned with the second scenario discussed in the problem scenario, namely, synchronization of the updated cache contents with the server copy. Recent file systems use a *differential update (diff*[1]*)* approach to synchronize the local copy with the server, where it transfers only file content differentials that can be used by the server to recreate the copy at the client end. These approaches, under several commonly occurring conditions, may incur considerably more overhead than needed to perform file synchronization.

Briefly, the key reason for large synchronization overhead is that applications today predominantly use application-specific file-storage semantics in lieu of simple text-based storage. For example, Microsoft Office applications, such as Word and PowerPoint, use proprietary file-storage semantics. Other platform-independent document-generating applications, such as Adobe Acrobat, and encryption tools, such as *crypt*, also store files in an application-specific format.

When applications use such tailored formats, even minor changes in the file's user-specific content can result in substantial changes in the binary representation of the file in storage. Hence, there is a high degree of disparity between the magnitude of user-level changes and the change in the underlying file contents, and the actual transfer size during resynchronization phase does not truly reflect the user activity. Given that the network connections are weak, an important problem in a mobile information access framework is the *minimization of file synchronization overheads* if and when copies of files at the client are synchronized with those at the server.

## *2.3 Related Works*

In this section, we summarize the related works in the area of mobile information access.

### 2.3.1 Mobile File Systems

The Andrew file system (AFS) [30] allows clients to *cache* files from the server and supports *close-to-open* consistency semantics wherein writes by clients are visible only to new sessions

[1] *In Unix, the* diff *command compares the contents of the two files and show the difference.*

that are opened after the writes. AFS also allows for *callbacks* wherein clients do not need to explicitly perform cached copy validation, and the responsibility lies with the server to inform the client when a copy is invalid. AFS uses a *full file transfer* method to send updated copies to the server for synchronization.

The Coda file system [82] is based on AFS, but supports disconnected operations for mobile hosts. When the client is connected to the network, it *hoards* files for later use during disconnected operations. During disconnections, Coda emulates the server, serving files from its local cache. Any updates are recorded in the form of dirty "block" logs to be used later during *re-integration.* Logs are periodically optimized by appropriately deleting blocks that have either been rewritten or deleted. During file synchronization, only the log is sent to the server. Coda allows for *trickle integration*, which uses any available bandwidth to trickle updates back to the server, and also provides for *asynchronous* updates wherein local file requests before an update is complete are served from the local copy.

The Bayou storage system [91] is also designed for a mobile computing environment. Updates are propagated using an *epidemic* algorithm that allows for lazy updates. However, the propagation still entails the transfer of full files. The Sprite file system [17] was developed for file-intensive applications. Since Sprite workstations are diskless, Sprite cannot cache entire files and instead caches file blocks of 4KB on its main memory. Sprite writes back all dirty blocks that have not been modified in the last 30 seconds, based on write-back policy. However, when a file server crashes, all processes having opened files with that server must be terminated and restarted.

In [43], Lee *et al.* propose a new technique, called *operation shipping* as an extension to Coda. High-level commands or user operations are logged at the client and shipped to the server for a playback. However, for interactive applications, operation shipping relies on application awareness and hence requires *changes to any interactive application that needs to be served by the approach.*

The low-bandwidth network file system (LBFS) [58] exploits cross-file similarities between files, just as the Unix command `diff` does for text files. It exploits the fact that updated files often contain a number of segments in common with previous versions of the same files. The LBFS file server divides the files into chunks and indexes a large persistent file cache. When transferring a file, LBFS identifies chunks of data that the server already has. Then, the client sends only non-overlapped chunks to the server. The Prayer file system (PFS) [19] also performs differential updates, but requires applications to store data in a pre-specified file format, which thus is not an application-unaware solution. There are several off-the-shelf applications that perform the task of computing the differential between two file copies and patching one of the copies with differences provided as input. Examples include xDelta [34], .RTPatch [73], exeDiff [2], and BSDiff [72]. It is important to note that while the approaches differ in terms of the specific mechanisms, their underlying principle is the same.

### 2.3.2 Thin-Client Computing

An existing model called thin-client computing provides the required abstraction of user-activity-aware mechanisms that we can utilize in the framework. But, we observe that a stand-alone thin-client computing model is not a feasible for mobile information access.

In [10], Chawathe *et al.* design a proxy that can transform data in new formats to old formats to accommodate thin clients. Most software upgrades can then be performed at the proxy as opposed to at the client. In [95], Wong *et al.* analyze a real thin-client product, Microsoft Terminal Services. The authors show the performance analysis of CPU, memory, and bandwidth usage for several types of local applications. The focus of the paper is primarily on resource sharing in a multi-user environment. The paper has limited analysis of the user behavior. In [101] and [99], Yang *et al.* analyze and compare the performance of thin-client computing approaches in a wireline environment. The authors use an internal Web server, and focus solely on Web browsing performance using several thin-client platforms. However, the paper does not use a real network environment, but instead relies on the Cloud network simulator for emulating network conditions. In [40], Lai *et al.* evaluate the Web-browsing performance of thin-client computing in a wireless environment. The authors focus on latency and size of pure transferred TCP data impacted by high packet loss rate over a wireless network. However, their experiments are also executed in the simulated environment emulated by the wired network emulator, even though the network characteristics in a wireless environment are completely different from those in a wired environment.

### 2.3.3 Web Characteristics

In order to obtain optimization techniques related to Web fetching, a lot of research have studied the characteristics of HTML documents and embedded objects included in Web pages.

Bray [5] and Woodruff *et al.* [96] perform a quantitative analysis for the select several million Web pages and provides the basic statistics of byte size, tags, attributes, object file types, links in those pages, using their tools and search engines. Douglis *et al.* [16] quantified the benefit of a proxy cache by using traces collected at two large corporations: AT&T Labs and Digital Equipment Corporation. Through the trace collections and analysis, they conclude that access rate, life time, and modification rate of Web objects depend mainly on content type and domain name. In [6], Breslau *et al.* investigate the distribution of HTTP requests through Web proxy caches and find that it follows *Zipf's* law; the probability of a request for the $i$-th most popular page is proportional to $1/i$. Shi *et al.* [84] have proposed a methodology to obtain the characteristics of dynamic Web objects. Through an analysis of six popular Web sites' content, they conclude that object sizes and freshness times of such dynamic object follow an exponential or Weibull distribution.

### 2.3.4 Web Caching

Web accesses from large population of users typically follow the *Pareto* principle; 80% of total Web content is accessed only by 20% of users, *i.e.*, most users access only a small part of the entire Web content in Internet [80]. As a result, Web servers containing popular content frequently become overloaded and underpowered by repeating the process for the same content continuously. To reduce this unnecessary bandwidth consumption, many Web-cache techniques have been developed.

First, a Web cache can be deployed in the browser level. Most Web browsers, including Microsoft Internet Explorer [47], Netscape Navigator [61], and Mozilla Firefox [56], use private or shared caches to keep the records of accessed Web content, such as HTML documents, images, and videos. A browser-level cache can be deployed easily and show the best access-time performance for the cached content data. However, only limited content can be cached and re-accessed by a local user(s), who uses the browser in the same machine. In addition, the cache should be customized and maintained periodically for the best performance.

Xu *et al.* [98] propose a cooperative client-cache technique, where all the clients generate a large virtual cache by sharing their browser caches in a P2P fashion. It has high scalability since caching and data-lookup operations are distributed across all clients and superclients. Their simulation results show that the proposed scheme show better performance than other proxy-based cache solutions.

In order to support a large number of group users in a large organization (*e.g.*, campus, company, and ISP), *forward proxy caches*, such as Squid [90], Wingate [75], and Privoxy [74], have been developed. These proxy caches store local copies of frequently accessed content and provide them to clients in the group. Mogul *et al.* [54] investigate the potential benefits of data compression and differential update in Web caches and show that the combination of both can yield the best performance in transfer size and time [52]. However, most forward proxy caches are not transparent to browsers, *i.e.*, the users need to explicitly configure their browsers to use the caches. In addition, they may not be able to perform user authentication for specific content, and RFC 3143 [14] discusses problems with these proxy caches.

Proxy caches, called *reverse proxy caches*, can be deployed in front of Web servers by content providers. The reverse caches process HTTP requests on behalf of the main Web servers or pass them to the main servers if necessary (*e.g.*, security issue, dynamic object problem, etc.). Except some products, such as SpiderCache [93] and CacheFlow [4], most reverse proxies cannot cache dynamic objects, which is generated based on ASP [49], JSP [88], or Zope [103], and this dynamic content generally requires a significant amount of processing resource[2].

[2]Generating a dynamic Web page may require up to 60 round trips between Web and database servers,

To distribute compute-intensive or large-sized load to multiple servers, a number of reverse caches may compose a cooperative Web delivery system, called a content delivery network (CDN), where all the reverse caches serve content behind a single IP address of the origin server. Current CDN providers, such as Digital Island [15] and Akamai [3], generally deploy multiple server groups in multiple geographical locations. This edge-origin deployment increases user-perceived browsing speed by minimizing Web-delivery distance, which is calculated as a number of hops or round-trip time. However, the initial deployment of the infrastructure entails considerable expense, and additional maintenance is required continuously. Thus, if necessary, Web content providers outsource the service from CDN providers.

### 2.3.5 Prefetching

Even though bulit-in caches in Web browsers and proxy caches maintains a significant amount of cached objects, objects requested by users may not be available in the caches because of many reasons; new access, limited cache size, and expiration of cached objects. In order to reduce fetching time for these non-cached objects, many researchers have focused on object-prefetching techniques.

In [70], the authors propose a server-based prefetching scheme, where the server makes predictions of user's future Web accesses through a graph-based Markov model and the client decides to prefetch the objects. Using simulations, they show that user-perceived latency can be reduced significantly at the cost of a network traffic increase.

Duchamp [18] proposes a client-initiated prefetching approach, where clients send the records of the hyperlinks included in their accessed pages to let the servers distribute them to all the clients by piggybacking on GET responses. Based on the access frequency, the clients select Web pages that they would prefetch in the caches. Through a real implementation with modifications of Mozilla and `httpd`, the author proves that the proposed approach can reduce response time by more than 50%.

On the other hand, Chen *et al.* [11] propose a cooperative prefetching scheme between

and thus the Web server may become unresponsive frequently [93].

a server and a proxy, which minimizes communication overhead by adaptively utilizing the reference access information in two different levels. In their scheme, the access information stored in the server is used only to access objects that are not qualified for proxy-based prefetching. Using trace-driven simulations, they show that the hit ratios are increased by 5% to 88% compared to proxy- and server-based prefetching schemes.

In [32], the authors propose a context-oriented prefetching technique, which uses keywords in HREFs to capture user-access patterns and neural networks to predict user's future access patterns. Unlike URL-based schemes, this technique does not require the historical references of requested objects but uses its self-learning capability and adaptability. Through experiments with MSNBC and CNN Web sites, the paper shows that it achieves up to 60% hit ratios.

Web acceleration products, such as Google Web Accelerator [26], CacheFlow [4], and Network Appliance's NetCache [62], reduce user response time by using a mix of prefetching and proxy-based caching in current Web applications. However, since those products are optimized for broadband networks and perform prefetching too aggressively, low-bandwidth users may not see any improvement and instead excessive bandwidth consumption can degrade performance of other users or applications significantly [1].

### 2.3.6 Transcoding

Over the past few decades, available network bandwidth for has increased dramatically, however many users in dial-up connections or mobile networks still suffer from narrow bandwidth. However, most Web content has been designed without considering users' various computing environments, and thus the low-bandwidth users often have to wait an inordinately long time to download a Web page. In order to solve this bandwidth-diversity problem, many research has been performed in area of content adaptation.

In [8], [23],and [86], the authors present proxy-based transcoding approaches, where application-specific transcoders convert HTTP responses into different formats better suited for the client. Han *et al.* [28] derive the theoretical conditions of transcoding and present adaptive-transcoding policies for mobile Web browser. These proxy-based approaches can be easily deployed without major modifications in current networks, however in most cases they require lossy compressions that degrade the quality of images or sound significantly.

In [37], the authors propose a scheme that transparently support resource-constrained mobile devices through powerful proxies. The proxy adapts its mechanisms to the dynamic nature of the wireless environment and addresses the limitations of the client devices. The proxy provides filtering and compression of graphical images, converts postscript files to ASCII text, and does static data partitioning.

Gilbert *at el.* [25] propose a new Web-delivery scheme that improves initial response time performance of images using progressive JPEG coding. The scheme also allows to accelerate downloading specific images, to which users explicitly point with interest. Using a Web proxy and browser-side Java applets, the authors implement a prototype for performance evaluation and show that the delivery of the first visible layer can be reduced by up to 80%.

Noble *et al.* [65] also propose an application-aware distillation technique that controls compression ratio of objects in order to adapt to changing network environments. In the paper, they construct a prototype, called *Odyssey*, which control the quality of objects in three modified applications; a video player, a Web browser, and a speech recognition program.

In [35], the authors propose a combination of end-to-end and proxy-based approaches as an ideal solution for supporting mobile hosts. The proxy explicitly requests data from servers that has a resolution matching the present QoS and client capabilities. In [41], the authors propose Puppeteer, a system for adapting component-based applications in a mobile environment. Puppeteer has the advantage of adaptive transcoding execution by a proxy without modifying applications. However, it cannot overcome a quality degradation problem caused by limitations of transcoding. In [22], the authors propose Spectra, a remote execution system, to balance performance, energy conservation, and application quality. Even though it manages resources effectively in a mobile environment, it has a limitation of application dependency. Therefore, it needs newly structured applications for Spectra or modification of current applications.

Currently, a lot of prototypes and commercial transcoding products such as UC Berkeley's Transend [24], Intel's QuickWeb, IBM's WebExpress [29], and Oracle's Portal-To-Go [69] have been developed to improve Web response performance by reducing image quality or size via lossy compression or image transformation. However, these solutions are difficult to deploy since it requires support from non-browser entities.

# CHAPTER III

# CONTRIBUTIONS AND OVERVIEW

To tackle the problem of read operations in conventional information access systems, we classify content into two types: *non-paritionable* content and *partitionable* content. The non-paritionable content consists of a single file, and thus the underlying access scheme cannot perform any partial access in the file-system level. Most application files, such as office documents, come under this category. To the contrary, the partitionable content is composed of multiple files, including a main text and other object files; composite documents such as HTML, XML, TEX, and programming content belong to this category. Therefore, the corresponding applications can access only a partial set of the content if necessary.

Similarly, we also categorize write operations into two types; *content generation* and *content synchronization*. The former is a write access triggered by a user who wants to generate new content, whereas the latter is executed by an update operation upon the existing content at the server.

In this dissertation, we study the inefficiency of user-activity-unaware conventional information accesses under various conditions in wireless networks and present three information access strategies for three different scenarios: reading non-partitionable content, reading partitionable content, and content synchronization.

1. *Application-unaware read access for non-partitionable content*: Notwithstanding the fact that there exist fetch-on-demand versions of some particular applications, it is infeasible to develop a generic data-partitioning technique for non-partitionable content, which cannot be partially accessed in the file-system level. For instance, in Microsoft Word and Adobe Acrobat files, content is stored without any correspondence to the way the user accesses them.

   Content partitioning for non-partitionable content can be achieved in the graphical domain by using the user-activity awareness provided by thin-client computing. This would help fetch only the data that will be shown in the display device, and the user could achieve quick response time that would not be possible if the user were to download the entire content, as is done using conventional content access. Note that this form of content partitioning in the graphical domain can be performed in an application-independent manner because any content can be represented using a common abstraction in the graphical domain. In this dissertation, we use the issues with a default graphical-domain content partitioning technique as the motivation to design and implement a new approach to mobile Web information access, called *Cut-Load*. Cut-Load uses application-unaware content partitioning along with three unique design elements: *dynamic mode selection*, *opportunistic hoarding*, and *transparent Mode transfer*.

2. *Application-aware read access for partitionable content*: Hybrid content, such as HTML, XML, TEX, which consists of multiple text and non-text object files, and programming projects can be partially accessed by fetching object files selectively. We call this content as *partitionable content*. A typical example of partitionable content is a Web page, which consists of a main HTML document, cascading style sheets, javascripts, images, and other multimedia object files.

   When a user is viewing partitionable content on a display device, objects for displaying other screens are unnecessary in the sense that they are not visible to the user at this time. However, in conventional applications that fetch objects in a greedy fashion without considering the locations of objects in the display layout, the process of fetching necessary objects (*e.g.*, on-screen objects) may be slowed down because of competition from the process of fetching unnecessary objects (*e.g.*, off-screen objects). Under these scenarios where different connections may fetch objects on different screens simultaneously, without an intelligent connection management mechanism the multi-object fetching process of current applications does not utilize network bandwidth efficiently.

Thus, we propose an application-aware solution called *Prioritized Fetching*, which performs *object prioritization* as well as a mix of *object reordering* and *connection management*. Basically, the solution differentiates objects from different screens based on the current screen view and allows for downloading the on-screen objects, which are required to render the current screen display with a higher priority. As a result, it reduces response time experienced by users as well as bandwidth consumed unnecessarily by applications. One major advantage of our approach is that it is purely client-side enhancement, and consequently it is easy to deploy since it only requires client-side installation to current applications.

3. *User-Activity Shipping for File Synchronization*: In a distributed file system, the bandwidth usage efficiency of the file-synchronization scheme is important when the clients are connected to the file server through weakly-connected, low-bandwidth links, such as in a wireless environment. An intuitive file synchronization strategy for such environments is one where only the differences between the original and updated files are sent across to the server. However, the performance of this differential update scheme can be improved upon considerably by adopting a raw user-activity shipping. We call it *Mimic* since the strategy is to *mimic* the client-side user activity on the server.

   Similar to the resynchronization phase in disconnected file systems like Coda[82], it supports a synchronization mechanism to update the content server with any writes performed by the user upon reconnection. But, unlike differential approaches, Mimic uses a novel user-activity recording scheme to help in the efficient resynchronization with the content server. We show that despite being completely application-independent, Mimic's record and playback scheme for update performs significantly better than traditional file-system-based approaches such as Coda.

**Key Applications: Word Wide Web and Distributed File Systems**

- The Web is the most widely used application today, and carries about 46% of Internet traffic[20]. Thus, in this work, we consider Web accesses as a representative of read accesses. Note that Web browsers access not only HTML documents but also other Web-friendly documents such as Word, PowerPoint, and Acrobat document through HTTP. Of the solutions presented, Prioritized Fetching and Cut-Load directly apply to Web access optimization.

- Contrary to read accesses, most write accesses are performed directely in the file-system level or indirectly by FTP applications that performs accesses to distributed file systems. Thus, we consider file-update operations in distributed file systems as a representative of write access optimization. The Mimic solution presented directly applies to distributed file system update optimization.

In the next three chapters, we describe the details of the three strategies and evaluate their performance.

# CHAPTER IV

# APPLICATION-UNAWARE READ ACCESS FOR NON-PARTITIONABLE CONTENT

## *4.1 Introduction*

Today, the majority of Internet users perform Web-based information access using a Web browser. Through the continuous integration with various plug-ins, the Web browser has become a unified application to access not only HTML documents but also other Web-friendly documents such as desktop publishing and presentation files [94].

As mobile computing technology has developed, users can browse these Web documents on the Internet from their home, office, or elsewhere. However, in wireless environments, most network applications suffer from low bandwidth, large delays, and frequent disruptions in connectivity. These characteristics lead to major problems with current models of mobile Web information access, such as excessive bandwidth consumption, large response delays, no service for partial disconnections, and inefficient system utilization.

The primary reason for these problems is *the absence of flexible content partitioning* in current file systems and the transfer of the entire content files from the backbone file server through low-bandwidth and less-reliable wireless links. In this context, *graphical content partitioning* is a concept that extracts and provides partial content a user wants to view and can be thought of as a *What-You-See-Is-What-You-Fetch* paradigm. Content partitioning in the graphical domain can be realized without any dependence on applications. This is because any Web content can be represented using a common abstraction in the graphical domain without any relation to the nature of application using the content.

In this chapter, we analyze how an application-unaware content partitioning scheme in the graphical domain would solve the problems of read operations of non-partitionable content with traditional mobile information access systems. We also study the issues that arise with using other simple content partitioning techniques instead of traditional binary-file transfer models. Briefly, the issues include large bandwidth consumption for full-file access and inefficient performance in the case of highly compressed data. We then use the issues with a default graphical-domain content partitioning technique as the motivation to design and implement a new approach to mobile information access.

We design and evaluate a new mobile strategy called Cut-Load[1], which uses application-unaware content partitioning along with the following three unique design elements: *dynamic mode selection*, *Opportunistic hoarding*, and *Transparent Mode transfer*. Dynamic mode selection solves the problem of inefficiency of content partitioning for certain types of non-partitionable content that is not conducive for graphical content representation such as highly compressed multimedia data; Opportunistic hoarding helps decouple the response time that the user perceives from the fetch time of the entire binary file. This reduces the peak load of the system even while fetching binary content for satisfying future user requests; Transparent mode transfer allows the mobile client to switch the current access mode when the user accesses non-partitionable documents in which case traditional binary-content transfer is more efficient than graphical content transfer.

## *4.2 Motivation*

In this chapter, we describe several drawbacks in the traditional model for mobile information access and use them as motivation for designing a new strategy for efficient mobile information access.

To measure the performance of traditional mobile information access systems in low-bandwidth wireless networks, we use a Sprint PCS CDMA2000-1X cellular network in 144-Kbps mode. The client machine used in the experiments is an HP Pavilion N5430 laptop computer with a 850MHz AMD Duron CPU, 128MB RAM, and a Merlin C201 cellular modem made by Novatel Wireless. The client is also loaded with Microsoft Windows 2000 Professional, Office 2000, Internet Explorer 6.0, and Adobe Acrobat Reader 6.0.

[1]The name is inspired from the Unix *cut* command, which cuts files into smaller parts.

We consider Web access as a representative of mobile information access. For performance comparison, we use a single Web browser instance that can view documents associated with all target application types using integrated plug-in programs [94]. Each user access is measures in terms of screen units, which have the same pixel-size as the client area. Hence, a large user access would mean that the user accesses most of the screens of the Web document, and small user access would mean that the user sees only a few screens of the document. The default screen resolution in the client is set as 1024-by-768, and the pixel-size of the client area inside the Web browser window is 1006-by-511.

We also use various document files selected from the Top 50 Internet Sites and their links [13]. In the experiment, we access multi-file Web documents as partitionable content and Microsoft Word and Adobe Acrobat documents as non-partitionable content.

### 4.2.1 Not all content is always seen by users

Users generally decide whether to access specific content based on filename, annotation, or other descriptions of the content, and then mobile clients request to access the binary file(s) of the content they select. However, users generally do not view the entire content of the accessed file. In [63], it has been shown that 90% of users do not scroll down Web pages but simply pick from the options that are visible on the initial screen when a page comes up. It also says that reading computer screens is about 25% slower than reading papers, and even users feel uncomfortable reading online text. As a result, people don't want to read a lot of text from computer screens.

However, in conventional Web-access models, a mobile client generally fetches the entire binary file containing more content than required regardless of the user's intention. Only few applications such as graphic viewers and add-on programs have limited capability for partial access to enable a user to access a fetched part of content before fully downloading it.

Figures 1, 2, and 3 show the total data transfer size for the entire raw data and the amount of data that the user ends up using for 30 Web content files selected from the Top 50 Internet Sites and their links [13]. We have modeled the useful data size based on the

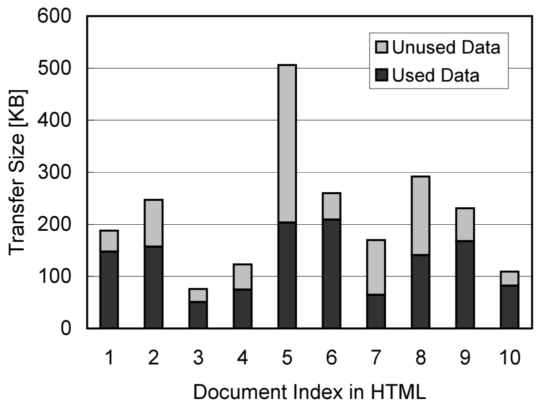


**Figure 1:** Transfer size in HTML.

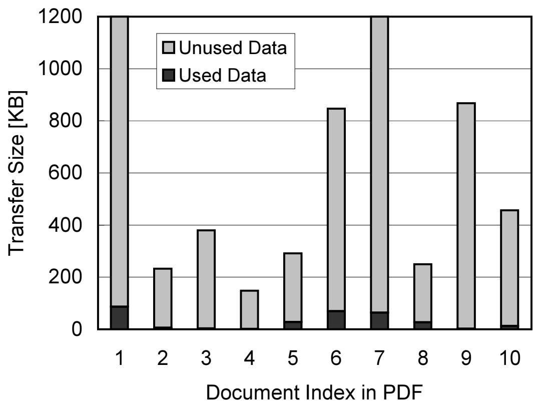


**Figure 2:** Transfer size in Acrobat.

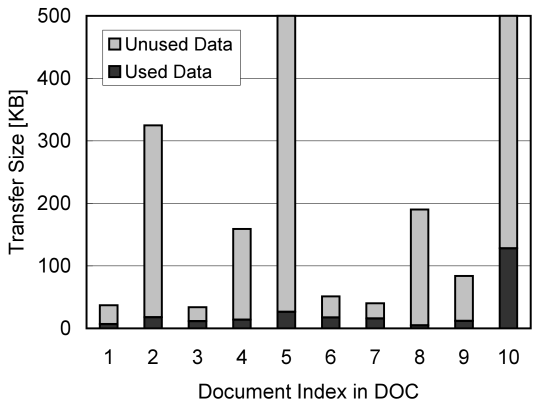


**Figure 3:** Transfer size in Word.

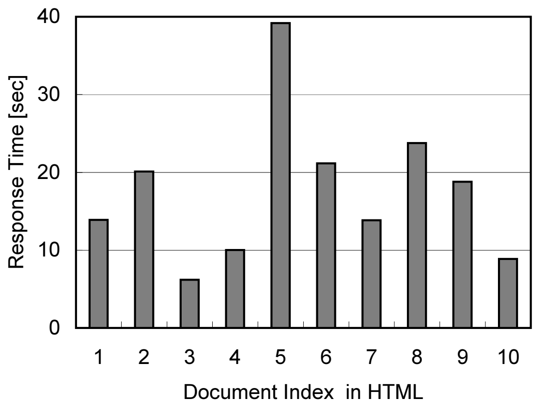


**Figure 4:** Response time in HTML.

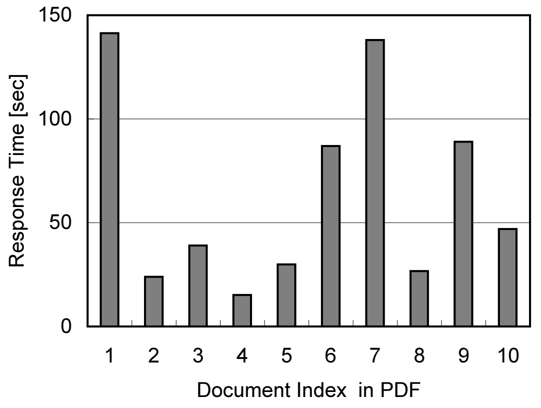


**Figure 5:** Response time in Acrobat.

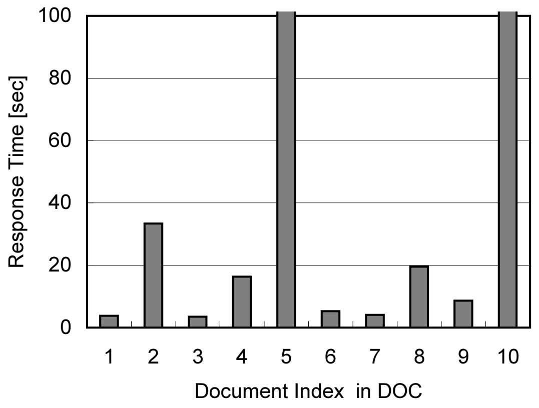


**Figure 6:** Response time in Word.

average number of screens that the user would view from the downloaded content [63].

To get useful access sizes, we measured transfer sizes for the useful accesses in thin-client computing and used the percentage of each screen data with the binary size of the content. We observe from the results that there is a significant difference in transfer sizes between full data transfer performed by current Web-access systems and the necessary content.

Thus one of the goals of our work is to incorporate the notion of application-unaware content partitioning to enable *intelligent retrieval* of content from the server. This would, obviously, lead to optimizing precious wireless bandwidth.

### 4.2.2 Users suffer when response times are large

It has been shown in [59] that users who use dial-up connections wait for about 30 seconds to look at a new Web page, and the situation becomes even worse in extremely bandwidth-limited wireless environments such as wireless wide area network (WWAN). Users don't want to spend a too long time for the Web-content to download completely. Hence, they give up the content download after waiting for a while, which leads to waste of the bandwidth that is consumed to download the data until the user stalls the access.

It is a well-known fact that there is a relationship between computer response time and users' perceptions. In [51], it has been shown that users lose their concentration on their access when the response time is longer than 10 secs, and when it is more 1 min, they lose interest and stop the current access [51]. As mentioned earlier, a mobile client in the traditional Web-access system always waits until the entire content of a document is fetched regardless of which part a user wants to see. Fetching the unnecessary part of content increases the initial response time significantly and makes users impatient. Particularly, in environments where available bandwidth is extremely limited such as in WWANs, it results in extremely poor response time.

Figures 4, 5, and 6 show the response time results for the same select documents from the Top 50 Internet Sites [13]. In the figures, we observe that the time taken to download the Web content is most often greater than the average user-tolerance limit that researchers suggest [51, 85, 59].

To decrease the download time in this environment, various proxy-based transcoding schemes, which convert large-sized image data into bandwidth-efficient types by reducing the image quality, were proposed [66, 7]. However, these are only makeshift solutions in which the proxies downgrade content quality by type-specific conversions. We note that downloading of only useful content can minimize transfer size and response time, without degradation of content quality.

### 4.2.3 Larger file transmissions suffer from frequent disconnections

In mobile environments such as WWANs, users experience highly fluctuating bandwidth conditions and frequent disconnections. This leads to stalls in the download of data from the file server to the mobile client.

Wireless networks are prone to frequent disconnections for reasons, including attenuation of the wireless signal, fading of the wireless channel, interference resulting from other transmissions, and mobility of the client. Particularly, the problem of disconnections impacts the performance of traditional Web-access systems in two ways:

- The partially downloaded non-partitionable (single-file) content until disconnection is generally not usable for serving the user requests for content, and hence the bandwidth expended to download the file is wasted.
- The response time perceived by the user is increased because of the additional time spent in downloading the content again.

We study the impact of disconnection in terms of bandwidth waste for both traditional full-file transfers and for ideal content transfers. We compare the results with ideal values based on user access patterns for HTML, DOC, and PDF files. We calculate the amount of bandwidth waste ($W$) using the average transfer size ($F$) of each document class, the average interval between disconnections ($D$) in seconds, the average bandwidth of the wireless link ($B$), and the formula $W = F^2/2DB$.

Figure 7 shows that the amount of bandwidth waste is significantly smaller in the ideal case than in the traditional full-file transfer case. This is because of the smaller probability of transmission failures caused by disconnections in the ideal transfer as a result of its smaller transfer time. It can also be observed from the results that the impact of frequent

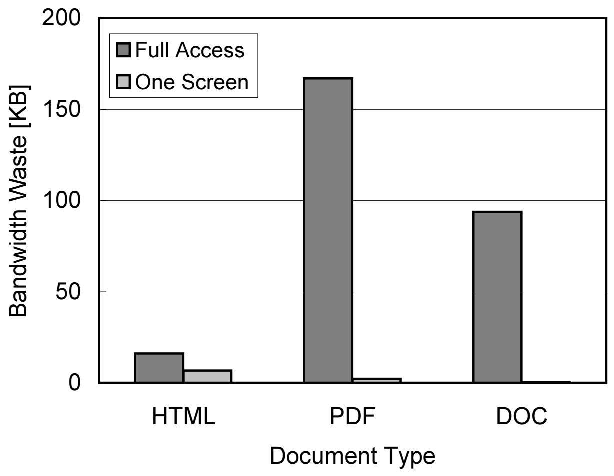


**Figure 7:** Bandwidth wastage.

network disconnections can be alleviated by reducing the amount of content download using the wireless link. Some advanced file systems emulate the previous connected network environment, however they can provide only limited disconnected operations based on locally cached data [82].

### 4.2.4 Greedy transmission makes network utilization inefficient

Traditional Web-access systems use TCP for downloading the binary content files from the file server to the mobile client. TCP uses the best-effort transmission paradigm, which is greedy in using the available bandwidth in the network. This greedy transmission increases the peak load of the system and brings down the overall system utilization.

Response time perceived by the user is influenced by the currently available network bandwidth and the byte size of content. Hence, in traditional Web-access systems, the client resorts to greedy download to minimize response time.

We say that a connection has timed out if the response time for the connection is greater than the specified latency tolerated by the user and the user has stalled the download. When the connection is reset by the user, the bandwidth used in downloading the content till that point is wasted, as mentioned earlier. Figure 8 presents the percentage of connections that times out as a function of the number of users in the network. In the figure, we observe that the time-out rates increase exponentially, whereas the number of users (network load)

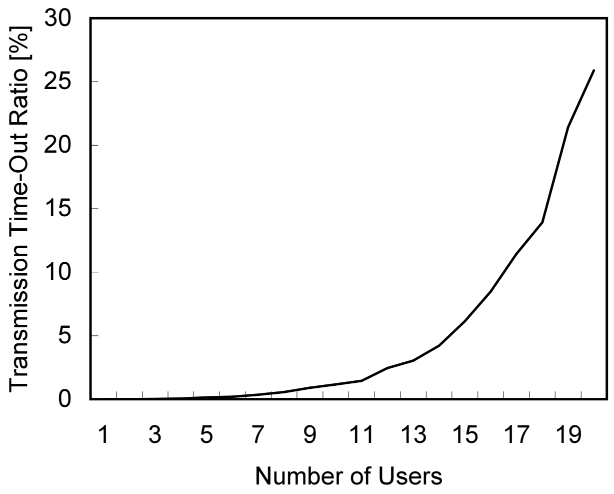


**Figure 8:** Time-out rate.

is linearly increased.

This exponential increase is due to the fact that the data downloads in traditional Web-access systems use greedy transmissions. Therefore, the higher peak load on the system degrades the system utilization and hence decreases the performance of the connections.

## *4.3 Application-unaware content partitioning*

In this section, we present the concept of graphical content partitioning for efficient content access and discuss the issues with the use of pure content-partitioning mechanisms for wireless Web access.

### 4.3.1 Graphical-domain content partitioning

Since current file systems are *user-activity unaware*, they are not able to differentiate between the essential part of the file and the part that will not be used by the application. As a result, a file requested by an application is retrieved in its entirety from the backbone server, irrespective of whether its content is eventually viewed by the user.

Intuitively, this can be solved by partitioning the entire content into smaller chunks so that only those chunks of data that are actually accessed by the user are downloaded by the mobile client. However, notwithstanding the fact there are fetch-on-demand versions of some specific applications, it is infeasible to develop a generic application-unaware data partitioning technique. For instance, Microsoft Word or HTML documents are stored *without* any correspondence to the way the user accesses them. Because it is not feasible for an *application-unaware mechanism* to split the file to correspond to user access in a data-domain, it it necessary to consider alternative content-partitioning mechanisms.

Web content exists in the form of single or multiple binary files while not being accessed. When a user requests access to the content, it is loaded in the system memory that is allocated for the corresponding application in the form of application dependent meta data. Then, the user browses the visual or auditory output that the application converts the meta data into. Using this concept, non-partitionable content can be partitioned in a meta-data domain. However, the meta-data abstraction requires an application-dependent mechanism which is not our objective.

Thus, the only solution to application-unaware content partitioning is at the graphical level, i.e., content-partitioning at the output device level. Using this mechanism, content is abstracted in terms of the different inputs to the graphical user interface of the application. Hence, a Web document is represented as a set of user-viewable screens that feeds an output device. Note that this form of content partitioning in the graphical domain can be performed in an application-independent manner because any Web content can be represented using a common abstraction in the graphical domain.

Now we describe how graphical-domain content partitioning solves the problems with traditional Web access model. We also present certain issues that arise because of the use of pure graphical-domain content partitioning for mobile Web access.

- *Usage to fetch-size ratio:* Content partitioning in the graphical level is highly efficient, especially when the byte size of the entire content is large compared to the amount of the file that the user actually views, which is most often the case. However, for highly compressed multimedia content, it may not be efficient in terms of transfer size because of the performance limitation of recompression in real time . Hence, the graphical content-partitioning mechanism should be used selectively along with traditional full binary-content transfer techniques.

- *Response time:* Content partitioning allows for the quick fetch of user-accessed graphical content because of the smaller byte size of the initial graphical content compared to the entire binary content. This helps reduce user-perceived response time, as the user requests for Web documents are served faster. However, when the user accesses all the content of the Web document, full-binary-file transmission is better than graphical content transfer because of the repeated overhead incurred by graphical content transfer for serving user requests. Thus the graphical partitioning access mechanism should be supplemented with the binary-content transfer so that the mobile client can use the binary-content if the user accesses more than a threshold amount of content from the Web document.

- *Partial download disconnections:* Because of the large transfer sizes in the traditional model, there is a high probability of disconnection during content transfer. This in turn leads to increased response time and bandwidth waste. Since content partitioning brings benefits of transfer size reduction by dividing the accessed content into several parts that can be transferred individually, the probability of network disconnections stalling data transfer is small in the case of content partitioning techniques. However, even small partitions may suffer from transmission failures caused by long-scale disconnections. Therefore, the re-usability of partially downloaded graphical content is essential to serving user requests during disconnections.

- *Greedy fetch problem:* Content partitioning decouples the part that is needed for the initial access from the other part that is not required immediately. Therefore, it can minimize the impact on other network traffic by reducing the initial transfer size. Reduction of the greedy fetching size decreases the peak load duration of the system and minimizes the effect on the performance of other applications in a multi-tasking environment. However, for future disconnected operations, a full binary file may need to be fetched. To minimize the effect of this non-urgent transmission, it is useful to differentiate the greedy transmission for the initial part from the non-greedy or low-priority transmission for the remaining part.

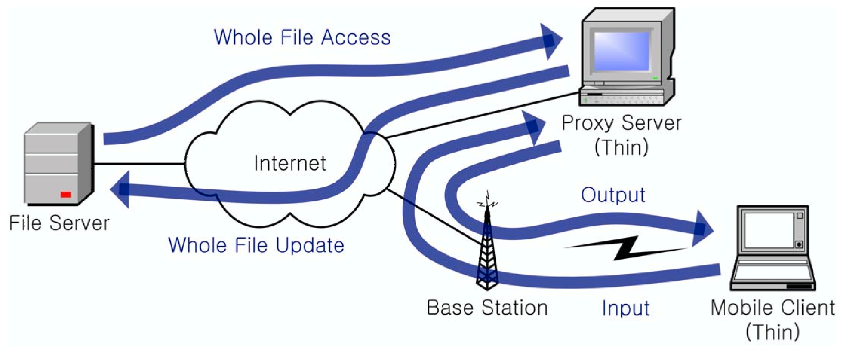


**Figure 9:** Thin-client computing model.

### 4.3.2 Thin-client computing

As mentioned earlier, thin-client computing provides content partitioning by portioning information in a graphical domain, not in a data or meta-data domain. In thin-client computing, the client accesses are directed to the proxy-server called as thin-client proxy, which retrieves the content from the backbone server and serves the request of the mobile client.

By using a multi-user operating system, the proxy-server allows multiple users to log on at the same time. Users can run either the same or different applications in separate, protected sessions on either a single server or multiple servers. Remote applications can be load balanced across multiple proxy servers, which form a server farm. Therefore, distributing the client sessions across multiple servers can increase overall performance as well as provide exceptional fault tolerance.

At a high level, the client software has a device data encoder that encodes keyboard strokes and mouse activity information and sends it to the server. The server has a decoder that decodes the data, and performs the requisite operations on the applications hosted on behalf of the client. The graphic data encoder on the server side encodes any differential in the screen content and sends the encoded information to the client. The graphic data decoder on the client side decodes the data and uses it to refresh its display. Most thin-client computing software uses TCP as the transport layer protocol. Any remote content access such as Web site and remote file attempted by the client results in the content to be fetched by the proxy.

Now we provide results that motivate the use of the thin-client paradigm as a component in our mobile information access framework. We also present results for scenarios where conventional file-system-based mechanism will fare better than thin-client approach. This further justifies our model of intelligently applying the thin-client paradigm for mobile information access.

#### *4.3.2.1 Experimental Setup*

- *Host Configuration*: The client machine used in the experiments has both WLAN and WWAN interfaces. For the thin-client solution, we use Microsoft Terminal Services [48] on the Windows 2000 Professional operating system with a 1024×768 (XGA) screen resolution. The proxy server runs the Windows 2000 Advanced Server operating system, which basically supports Microsoft Terminal Services. Finally, both of the client and the server are loaded with Microsoft Internet Explorer 6.0, Office suite 2000 and Adobe Acrobat Reader 5.0.

- *Network Connectivity*: The client is connected to the network through one of the two wireless interfaces, 802.11b (up to 11 Mbps) WLAN and CDMA2000-1X (144-kbps mode) WWAN. When connected through the WLAN and WWAN interfaces, the round-trip time (*rtt*) between the client and the proxy, as well as between the client and the internal Web server, is about 10 ms and 300 ms respectively. The effective data rates on the WLAN and WWAN interfaces are about 3 Mbps and 40 Kbps respectively[2]. The *rtt* value between the proxy and the external Web servers typically ranged around 90 ms.

- *Test Access Patterns*: While Web browsing, each Web page is fully or partially displayed inside the internal window of a browser. And, the size of a Web page can be estimated as the number of vertical screens having the same pixel size as this internal

[2]Note that although both interfaces come with significantly higher promised data rates, the presence of interference, contending hosts, and fading effects results in the effective data rates to be much lower.

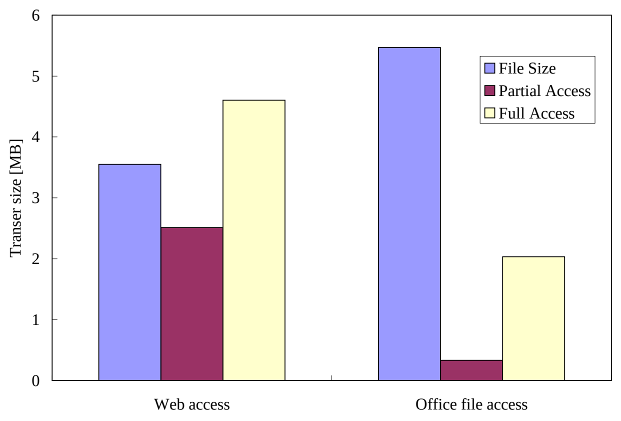


**Figure 10:** Transfer size in WLAN.

window[3]. For the Web browsing and office applications, we employ two different access patterns: (i) In the first pattern, called *full access*, the client reads the whole page or document by scrolling down. We follow the slow-motion benchmarking technique[4] for the full-access pattern [101]. (ii) In the second pattern, called *partial access*, the client views only the first screen displayed automatically at first by the Web browser. All experiments are conducted using 20-minute-long pre-recorded macros and are repeated more than 10 times under the exact same condition by a macro program that captures and replays all user activity in real-time [12]. The macros access about 40 different Websites. We consider only scrolling down activity by pressing a PageDown key. All the caches of Internet Explorer and Terminal Service are turned off to estimate pure bandwidth usage without help of the cache. In actual conditions, the bandwidth usage in both thin- and thick-clients can be reduced by the effect of the cache.

#### *4.3.2.2 Performance Evaluation over WLANs and WWANs*

- *Transfer Size*: Figure 10 presents the total data transfer size results for access of 42 Web pages and 10 office program files that are located in external Web servers[5]. In the figure, it can be observed that the thin-client computing provides better transfer size

[3]Actually, these screens somewhat overlap each other in the experiments.
[4]Briefly, wait till a screen is loaded before scrolling down.
[5]Note that the transfer size will be almost independent of the specific wireless access technology used.

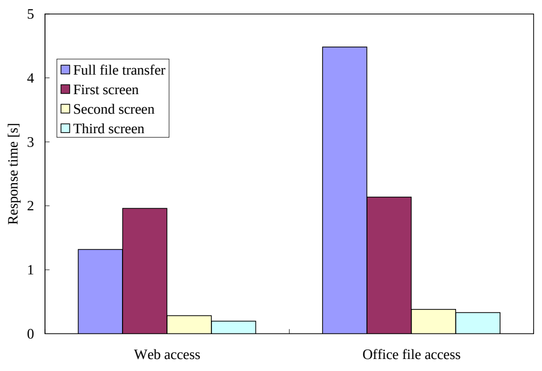


**Figure 11:** Response time in WLAN.

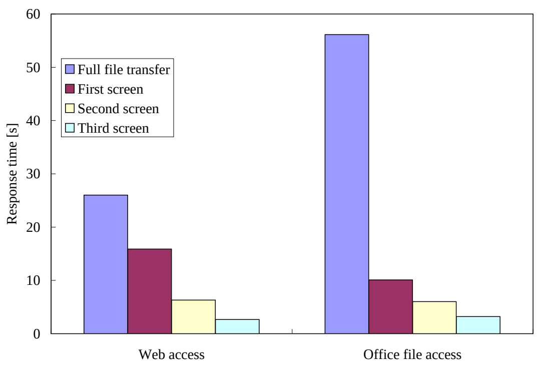


**Figure 12:** Response time in WWAN.

performance when the user performs only a partial access of Web pages. Contrary to the Web browsing results, even when a user accesses office files fully, the transfer size in thin-client mode is much smaller than that of thick-client mode. We conjecture that this is due to the fact that for office files in the thick-client mode, a large portion of the data size is actually consumed by non-critical information such as high-resolution font/image data for enabling rescaling, printing, and editing. Thus we conclude from the above results that the amount of data transferred over the wireless link is optimum for most cases, but not all, when using thin-client mode. We use these insights to use the normal file-system (thick-client) and user-activity-aware (thin-client) approaches depending on the nature of the user access.

- *Response Time*: Figures 11 and 12 present the screen-wise response-time results between the request for data and the completion of the transmission of the screen data in WLANs and WWANs, respectively. In most cases of Web page access over WLANs, the thin client generally experiences larger latency than the thick-client mode over WLANs. The large drop in latency from the first screen to the second screen can be explained by the absence of any fetch time involved from the second screen onward. Essentially, the new content from the backbone server to the thin-client proxy is fetched before the first screen access, and is available locally at the proxy for subsequent screen accesses. On the other hand, in the case of office-file access, the first screen latency is dramatically reduced when a user uses thin mode because of the lower transfer sizes. This benefit is further increased when the network capacity is

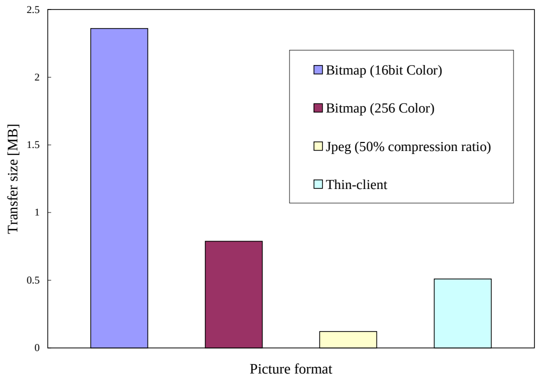


**Figure 13:** Average transfer size for 10 sample pictures.

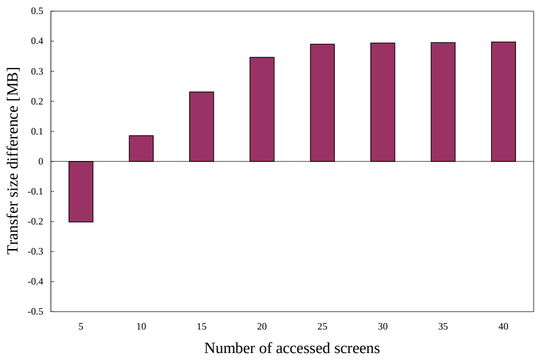


**Figure 14:** Transfer size of localized computing.

decreased as seen in Figures 12 for WWANs.

- *Unfriendly Content*: Figure 13 presents the transfer size when the content being transferred is an image of 1024×768 in pixels. The image considered is a JPEG image with the 50% compression ratio, and the size of the data transfer for the XGA resolution performed by the thin-client computing solution. It can be seen that the thin-client solution consumes more than three times of the size of the JPEG image. The reason for this overhead is that the compression algorithm used by the thin-client solution needs to be fast since it requires to provide real-time service to the user. Hence, its compression efficiency cannot be as good as that of JPEG where the compression is performed off-line. The same reason holds for any image-rich content, such as video or flash files. Hence, thin-client computing is not a good solution for graphically-optimized content that is compressed by an off-line algorithm.

- *Localized computing*: We also test the performance of the thin-client computing approach when the user downloads a document to read and views the document for a relatively long time. We refer to this as localized computing. For thick-client computing, such a scenario would involve only the initial downloading of the file. On the other hand, for thin-client computing, because of the display refresh requirements, some overheads will still be incurred when the client is predominantly idle and is accessing the same document. Figure 14 presents the impact of localized computing on the performance of thin-client computing approaches. We consider access of an Adobe Acrobat file, and the mobile host accesses only the first five screens repeatedly. We observe that using the thin-client model incurs bandwidth overhead even beyond the maximum number of independent screens in the file. The results show that thin-client computing does not show better performance compared to thick-client computing in terms of bandwidth consumption during off-line access.

- *Mobility*: Thin-client computing solutions currently provide no support for disconnected operations. Hence, when the hand-offs due to mobility are slow, or there are disconnections, the impact on client-experience ranges from drastic to catastrophic. In our experiments, when hand-offs are manipulated to incur a large latency, the thin-client software more often than not aborted. Disconnected operations, on the other hand, are positively show-stopping for existing thin-client solutions. This is despite the fact that the thin client does maintain a display cache, and a solution that serves user accesses from the display cache could conceivably by developed. The reason existing thin-client solutions do not support direct reads without going through the proxy from the display cache is that, it is expectedly a complicated process to infer which content to retrieve from the display cache based on user input in the form of keystrokes and mouseclicks.

- *Update of Cached Data* : Typical mobile hosts cache data locally; therefore it is possible to operate even during network disconnections. Any user update during disconnection is performed on the locally-cached copy at first, and is then resynchronized with the server on reconnection. Here we study the efficiency (in terms of transfer size) of normal file-system-based resynchronization approaches, and consider the possibility of using thin-client components for update of cached data.

#### *4.3.2.3 Results Summary*

We observe that for non-partitionable files such as office files thin-client computing performs remarkably better than thick-client computing in terms of bandwidth usage and response time. This serves as a key motivation for the serious consideration of the thin-client model as

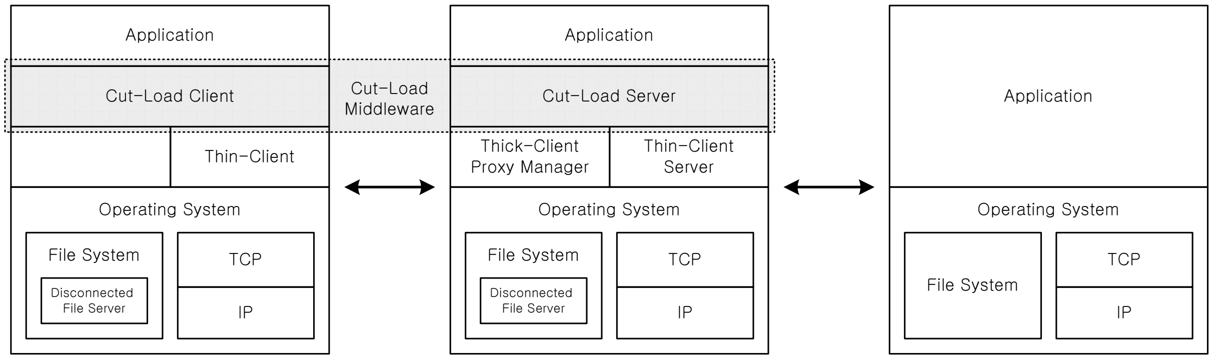


**Figure 15:** Cut-Load architecture overview.

a component in the mobile information access framework. However, thin-client information access performs poorly in terms of bandwidth usage for specific non-partitionable content types, such as highly-compressed images and videos, and partitionable content, such as Web documents. To a less critical extent, the performance of thin-client operations during predominantly localized operations performed by the client is of concern as some basic overheads continue to be incurred by the thin-client display refresh mechanisms.

## *4.4 Cut-Load Architecture*

We use the content-partitioning mechanism used by thin-client computing in the Cut-Load middleware. Cut-Load resides at both the client and the proxy as a middleware and hence it is easily deployable. The client-side middleware transfers a request of content access, manages content cache, and follows an application control message received from the proxy. The middleware at the proxy side decides the best computing mode for a requested content access, transfers objects for caching, and controls applications at both the client and the proxy. Figure 15 presents an overview of the architecture.

Cut-Load operates in one of the following two modes.

- *Normal mode:* In this mode, a Cut-Load client works in the same manner as a traditional Web access client. When a user wants to see content, the client fetches the original content files from the Web server and accesses them by running local applications. Because all required content files are fully transferred to the client, it can provide offline (disconnected) operation with cached files.

- *Dual mode:* A Cut-Load client works in dual mode under specific conditions. In this mode, the client initially operates in thin mode. While a user is seeing the content in the initial thin mode, the client performs hoarding of original content files in the background [6]. When the content file is hoarded completely, the client notifies the user about the change of mode to thick mode. Then, it opens the hoarded thick data using an associated application and moves the current system focus to the application window. After it closes the remote application that the user used before, the mode transfer from thin to thick mode is complete.

To support these mode operations, Cut-Load consists of three basic elements: *dynamic mode selection*, *opportunistic hoarding*, and *transparent mode transfer* to address the issues with the pure graphical content-partitioning mechanism. These elements bring benefits of faster access speed and efficient network bandwidth utilization.

### 4.4.1 Dynamic Mode Selection

Thin-client content access is not always the clear winner in terms of performance [100]. Thus, in Cut-Load, the decision of whether or not to use graphical content partitioning is done dynamically based on several factors, including thin-friendliness, content size, and current network condition, to maximize its performance.

#### *4.4.1.1 Thin Friendliness*

Thin-client computing naturally provides an ability of partial content access in a graphic level, in contrast to conventional computing that performs in a binary-file level. When a user requests to access content, the thin proxy opens an instance of the application associated with the file type of the requested content, like a normal client. Since the application instance has no notion of a thin client, it regards the proxy as the end host, which will show the content in its screen, and immediately generates raw graphic data for the displaying device, such as CRT and LCD monitors. However, the proxy server does not display the

[6]Because the hoarding is performed in an opportunistic non-greedy manner, we call it as *opportunistic hoarding*.

generated raw data, and instead it needs to deliver this raw data to the thin client. For bandwidth-efficient transmissions, the proxy-side software disassembles the raw data into individual graphic objects and compresses them using its own real-time algorithm.

When the requested content consist of one or more precompressed data files, the proxy needs to decompress them by using the original off-line algorithm, which was used for its compression, before generating raw graphic data. Then, the generated raw data is recompressed by the real-time compression algorithm that the proxy-side software uses. However, for some *thin-unfriendly* content types, the recompression process may not show the best performance because of one of the following three reasons:

- *Low recompression efficiency*: In compression algorithms, there exists a trade-off between compression ratio and processing speed. Since for real-time compression algorithms the speed is a more important factor, they generally sacrifice the compression ratio or the quality of the content to minimize the processing time. In the worst case, the size of the recompressed data that will be transmitted to the thin client may become larger than the size of the original content.

- *Wrong selection of a recompression algorithm*: When a client accesses video content that was precompressed by an MPEG algorithm, the proxy server may use not a video-compression algorithm but an image-compression algorithm for recompression. In such a case, the recompression process cannot be performed effectively, and finally this overload affects the server's overall performance for other clients.

- *User interaction or repeatitive playing*: Non-media content also may include thin-unfriendly objects. For example, an HTML document may have a flash or a interactive component that requires user input. A presentation file may include animated objects or sounds. Therefore, documents that have thin-unfriendly objects are also categorized as thin-unfriendly content.

Since the network utilization of thin-client computing is mainly decided by the thin-friendliness of content, one of the decisions the proxy should make for a request to access

**Table 1:** An example of content classification.

| Content type | Thin-friendly | Thin-unfriendly |
|---|---|---|
| Image | bmp, jpg | animated-gif, flash |
| Audio | raw | mp3, wma, ra |
| Video | | mpg, avi, wmv, mov |
| Desktop publishing | doc, pdf | |
| Spread sheet | xls | |
| Presentation | | ppt |
| Graphic | vsd | |

content is whether the target content is thin-friendly or not. In Cut-Load, highly-compressed media content, such as image, audio, and multimedia, is characterized as thin-unfriendly content. Uncompressed image content, such as bitmap, can be recompressed efficiently by the proxy server, and this content is regarded as thin-friendly. Cut-Load uses the normal (thick-client) computing mode to send thin-unfriendly content and the thin-client mode in the other case. Table 1 shows an example of the content classification.

#### *4.4.1.2 Content Pixel Size*

A user in a client gets most information through the screen of a display device, and the size of the screen is the unit of information that the user is able to get at one time. When a user accesses content, he/she decides whether the information of the content is interesting to him/her and whether he/she would continue to see the remaining content, based on the information in the first screen. Therefore, if the content is not what the user wants, the remaining part of content fetched in normal computing is wasted even without being seen by the user.

As a What-You-See-Is-What-You-Fetch (WYSIWYF) paradigm, thin-client computing can be a good solution. Its ability of content-partitioning in the display level enables a client to fetch the display data of only the part that a user wants to see. The benefit that thin-client computing brings depends on the portion of the content part that a user actually sees. As the user sees a smaller part of the content, the benefit from thin-client computing becomes larger. Therefore, pixel size of content is one of the important factors that decide this benefit.

If a user wants to access large-pixel-size thin-unfriendly content, Cut-Load initially accesses it in thin mode and then transfers the current thin mode to thick mode. Since an application generally does not provide the pixel size estimation, it can be performed by preloading the content in an application-specific estimator with user-defined settings, *i.e.*, every file type needs its own estimator. Each estimator can get pixel size information by scrolling-down operations or through a setting of virtually infinite screen resolution. However, the estimation may not be simple because pixel size can be varied by the current zooming rate in an application window. Thus, when the content is preloaded in an estimator, the zooming rate in the main window is fixed as the current setting, and it is assumed that the zooming rate is hardly changed. In addition, an application may have a different method to access the whole content. For example, a Microsoft Visio file may have multiple pages, and a user can move the current page by clicking the label of the page folder.

#### *4.4.1.3 Decision Heuristics*

The first factor that Cut-Load considers to decide the initial mode is the current network connectivity. It is because interactive operations in thin-client computing are based on a strong connection between a client and a server. In wireless networks, its connectivity is represented as received signal strength indication (RSSI) or signal-to-noise ratio (SNR). Cut-Load uses a long-term average SNR value, which directly affects the quality of service in a long term ignoring minor effects of temporary fluctuations. When the current connection is not strong enough for thin-client computing, Cut-Load accesses the content in thick-client computing, regardless of other decision factors.

If the connection requirement is satisfied,it considers thin friendliness of content. If content that a user requests to access is thin-unfriendly, Cut-Load decide the initial access mode as thin-client mode, regardless of other remaining factors that are not considered yet.

The third factors are byte and pixel (optional) size of content. Cut-Load may access thin-unfriendly content in thin-client mode when it has extremely large byte size. The purpose of this operation is to avoid unnecessary bandwidth consumption, and it is because the transfer size of a partial access of thin-unfriendly content in thin-client mode can be smaller than that of a full access in thick-client mode, with a large probability for large-sized document. However, this operation does not support media content, which fundamentally are not suitable for thin-client computing, such as a wrong selection of the compression algorithm.

In order to maximize the rate of successful decisions, Cut-Load requires to choose appropriate threshold values of network connectivity and size for the decision. However, optimal threshold values of byte and pixel sizes are dependent on what a user accesses, and therefore data mining of user access patterns is necessary in the approach. If the current decision is identified as a failure by comparing the total transfer size in thin-data and original-content size, Cut-Load updates the threshold values not to repeat failure in the same condition.

Most threshold values can be acquired only by real sample experiments. However, even if the thresholds are estimated by an extremely large number of experiments, the values cannot be optimal easily because lots of new content is always being created continuously. Therefore, Cut-Load uses semi-optimal threshold values, and it means in some cases the conventional computing models may show better performance.

Cut-Load controls applications externally without any modification of them. A *ExecuteApplication* function chooses a corresponding application based on the file extension or the system command, and generates an application instance with a content file(s). As returned values, the controller receives required various handle values of the application instance. With these values, the controller can detect the content file used by the instance through a *DetectExecutedFile*, close the application instance through a *CloseApplication* function, and do other required operations.

### 4.4.2 Opportunistic Hoarding

By using a combination of both greedy thin-screen access and opportunistic thick-data access, dual-mode operation decouples the response time experienced by the user from the actual fetch time for the thick-content. This decoupling has two positive effects: (1) For users operating over a low-bandwidth link such as in a WWAN this significantly reduces the response time for large data sizes; (2) The decoupling facilitates a non-greedy approach to

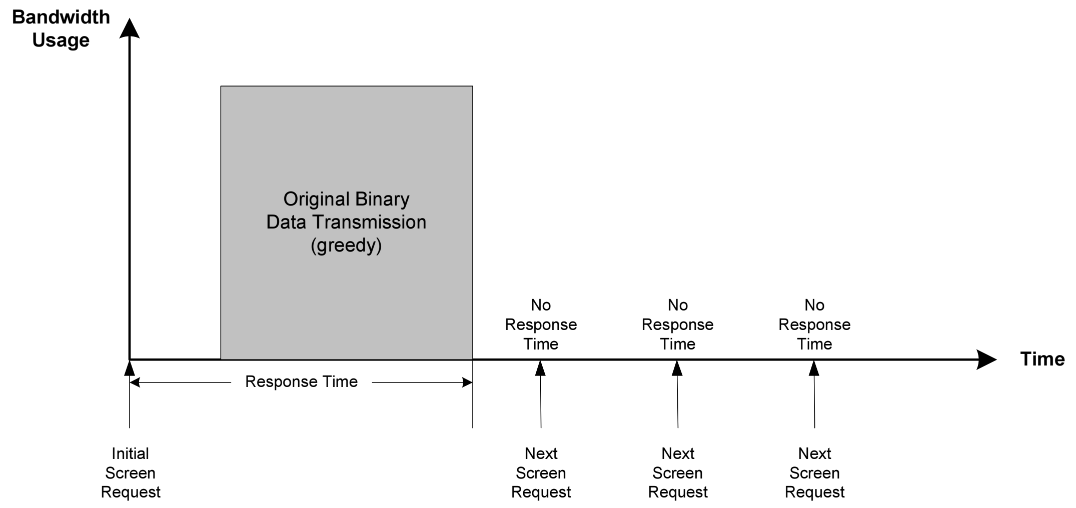


**Figure 16:** Transmission in thick mode.

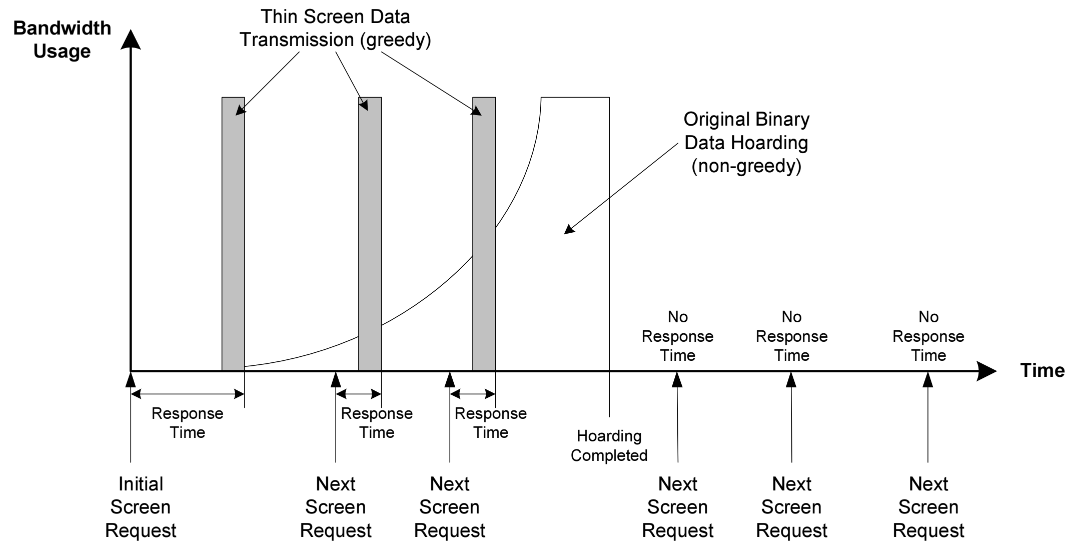


**Figure 17:** Transmission in dual mode.

hoarding. This in turn reduces the peak hoarding rate and hence improves the system-wide utilization.

Opportunistic hoarding is performed when user requests for content are served using the thin-client mode of operation. However, unlike a normal content download that utilizes the entire available bandwidth in a greedy fashion, opportunistic hoarding is performed at an optimal adaptive data rate with lower priority to minimize the impact on the thin-client mode data transfers that use normal TCP with high priority.

Opportunistic hoarding is performed when user requests for content are served using the thin-client mode of operation. But unlike a normal download of content which is performed in a greedy fashion, utilizing the entire available bandwidth, opportunistic hoarding is performed at an optimal adaptive data rate to minimize the impact on the thin-client mode data transfers which use normal TCP.

Thus, we use a novel rate control scheme for performing the opportunistic hoarding. A very low rate for hoarding would keep the user access in thin-client mode even if the user ends up viewing the entire file. On the other hand, an excessively aggressive hoarding rate would affect the response times of users operating in thin-client mode. To achieves a specific bandwidth share, the hoarding mechanism uses a Weighted Additive Increase Multiplicative Decrease (W-AIMD) congestion control mechanism [60]. The mechanism uses weights to achieve a fraction of the bandwidth obtained by the high-priority thin-client flows, which use the normal TCP.

In [45], the throughput of the normal TCP in the congestion avoidance phase can be expected as

$$E[Throughput_{TCP}] = \frac{1}{rtt}\sqrt{\frac{3}{2p}}\ , \tag{1}$$

where $p$ is the packet loss probability.

In the W-AIMD algorithm [60], the average throughput can be expressed as

$$E[Throughput_{w}] = \frac{1}{rtt}\sqrt{\frac{1}{p}\ f_{inc}\ (2f_{dec} - 0.5)}\ , \tag{2}$$

where $f_{inc}$ and $f_{dec}$ are the weighted additive and multiplicative factors, respectively.

When the bandwidth utilization of wireless networks is low, Cut-Load may perform opportunistic hoarding more aggressively by increasing the weight exponentially. When the hoarding starts, the flow is assigned a specific initial weight of $w$, which is much less than one[7]. As the user performs input activity and the proxy server sends screen updates, the weight of the "hoarding" flow is increased by a pre-determined increment. After several increments the weight of the *hoarding* flow reaches one and from then on it would receive the same share of network bandwidth as a normal TCP flow.

[7]This means that the flow would get $\sqrt{w}$ of the capacity that a thin-client flow would achieve under the given network conditions.

### 4.4.3 Mode Transfer

After opportunistic hoarding is completed, Cut-Load performs mode transfer to stop unnecessary bandwidth consumption. Our framework uses both thick-mode of operation as well as thin-mode for information access. When a client accesses content in dual-mode, it operates in thin-client mode initially. It also performs opportunistic hoarding of the raw data file in the background. If the client is still accessing the same content when downloading is complete, the framework changes the operating mode from thin-client to the thick-client mode.

The transfer point of time is decided by the hoarding rate and the byte-size of hoarded data. When the mode transfer point of time is decided, the framework stops the current thin-client operation and notifies the mode transfer to the user. When the transfer is completed, it shows a message in a pop-up window and begins to provide access to the hoarded content file in normal thick mode. In order to provide a seamless user access after mode transfer, the environmental settings and system focus of both sessions in the client and the proxy should be synchronized.

Environmental settings are categorized into system settings and application settings. The system settings include screen resolution, keyboard layout, clipboard content, etc. The value of these settings is obtained by means of query messages to the operating system. The application settings include parameters set up within an application, such as menu bar, zoom rate, view option, etc. Because this information is application-specific and user-specific, the location of the application environment file should be input before it performs synchronization of the application environment.

After the environmental synchronization is performed, the client synchronizes the various types of focuses. Mouse focus is the current location of the mouse cursor, and it can be located anywhere in the entire screen. Keyboard focus means the current position of the keyboard input, and it exists only when one or more text input controls are included in the current window. Screen focus is the screen position of the client area in the document layout. When the system is in thin-client mode and opportunistic hoarding is performed in the background, the client traces all the focuses. Once the new local application is opened, the captured focuses are restored by OS-specific interface functions.

### 4.4.4 Integrated Operation

The three elements presented thus far operate in a concerted manner to realize the solution for performing efficient mobile information access. Figure 18 shows the pseudo code for the integrated operation. The basic disconnected operation in the approach is similar to that of Coda [82]. While a user is accessing thin/thick objects, the client caches all fetched objects as well as prefetches unaccessed objects that have a large probability to be accessed in the future. When disconnection occurs, it emulates network file servers by providing fetched content objects transparently.

## *4.5 Performance Evaluation*

In this section, we evaluate the performance of Cut-Load and compare it with conventional Web access systems and discuss the impact of each design element in improving performance. Then, we present the Cut-Load prototype implemented in Linux and its behavior under different user-access patterns.

### 4.5.1 Simulation Setup

In order to evaluate the performance and profile the benefits, we use the *ns2* network simulator with the *FullTCP* package [42]. We run the simulations with 20 different seeds and use response time as the primary metric for the performance comparison.

To simulate 144-kbps-mode operations and bandwidth fluctuations in cellular networks, we use the 802.11 simulation mode with a 0.5-Mbps bandwidth and three pairs of FTP servers and clients for generations of background traffic We assume that the access point (AP) and all the servers are connected to the network with a 10-Mbps bandwidth and a 10-ms delay. It is also assumed that a Cut-Load client downloads both thin and thick data through a proxy server, which supports W-AIMD with a fixed weight. Except the proxy server, all other servers and clients do not perform W-AIMD in TCP, *i.e.*, the increase factors in their connections are set as one.

The default screen resolution is 1024×768 pixels (XGA), and the pixel size of the client

```
While (Each user access) {
  if (Unfetched content access)
  {
    if (Disconnected)
      Wait for reconnection;
    // connected
    if (Connection is strong)
    {
      if (Content is unfriendly)
      {
        if (Byte size or pixel size is large)
        {
          // dual-mode operation
          Request and display in thin mode;
          Begin hoarding;
          if (hoarding is not completed)
            Wait for completion working in thin mode;
          // hoarding completed
          Perform mode transfer from thin to thick
          Request and display in thick mode;
        }
        else // small size content
          Request and display in thick mode;
      }
      else // friendly content
        Request and display in thin mode;
    }
    else // weak connection
      Request and display in thick mode;
  }
  else // fetched content access
  {
    if (Hoarding is complete)
      Request and display in thick mode;
    else // not hoarded yet
      Request and display in thin mode;
  }
} // End of while
```

**Figure 18:** Pseudo code for the integrated operation.

area in the Web browser is set as 1006×511 pixels[8]. Each user access is measured in the unit of screens, which have the same pixel size as the client area. All user accesses are performed by *PgDn* keystrokes in the unit of screens. The average inter-access interval is assumed as 10 seconds. The internal data processing time is ignored. Thus, when a mobile client asks to access content that is fully fetched or hoarded, its graphic data can be displayed with no delay.

For modeling web and thin traffic, we use the clustering-based model [55], in which byte-size of web content and inter-access interval follow the log-normal distributions, as shown in Equation 3.

$$PDF[x] = \frac{1}{x\sigma\sqrt{2\pi}}\, exp\Big\{ - \frac{\big(ln(x) - \mu\big)^2}{2\sigma^2} \Big\},$$

$$where \;\; \mu = ln\big(E[x]\big) - \frac{1}{2} ln\Big(1 + \frac{STD[x]^2}{E[x]^2}\Big) \;\; and \;\; \sigma = \sqrt{ln\big(1 + \frac{STD[x]^2}{E[x]^2}\big)} \qquad (3)$$

We use the average and standard deviation values of Web document byte sizes from the experiments performed in Section 4.2. We consider the case where the byte sizes of document files follow a lognormal distribution with a mean of 400 KB and a standard deviation 200 KB, and vary the ratio of thin-data size to thick-data size. Figure 19 shows an example when the thin-to-thick data ratio is chosen as 0.1.

### 4.5.2 Impact of thin-screen data size

Figure 20 shows the per-screen-response performance in thick, thin, and Cut-Load clients. In the simulations, wee vary the thin-to-thick data ratio from 5 to 20%. The increase factor of the TCP connection used for opportunistic hoarding is set as 0.1, and frame errors or losses in wireless channels are not assumed. The main bars in the figure indicate the average time spent for the clients to show each screen, and the error bars are used to show the range of one standard deviation.

In the figure, for the initial screens, both the thin and Cut-Load clients show a significantly reduced response time compared with the thick client. And, as the thin-to-thick data

[8]We use the maximum size of the client area in Microsoft Internet Explorer in the XGA resolution.

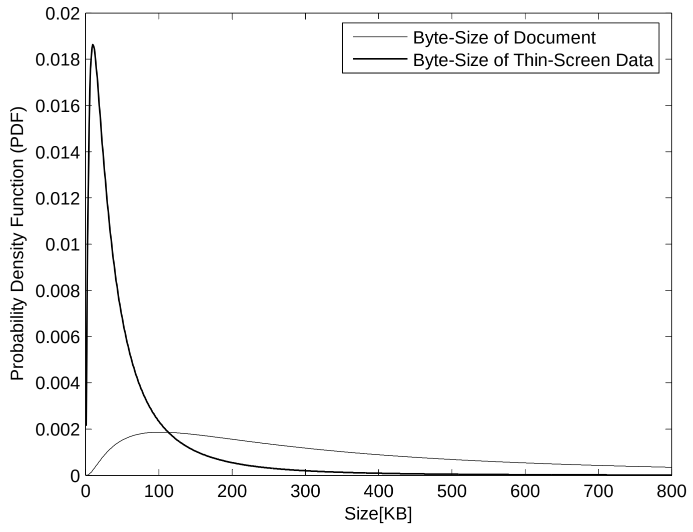
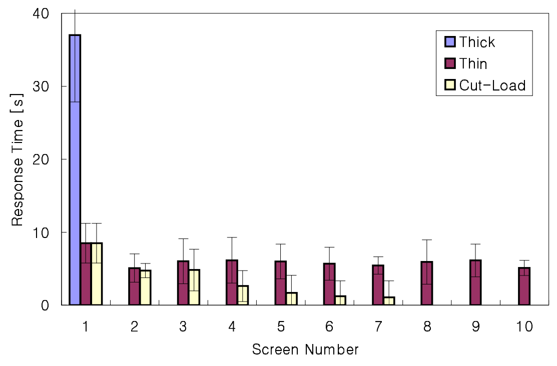
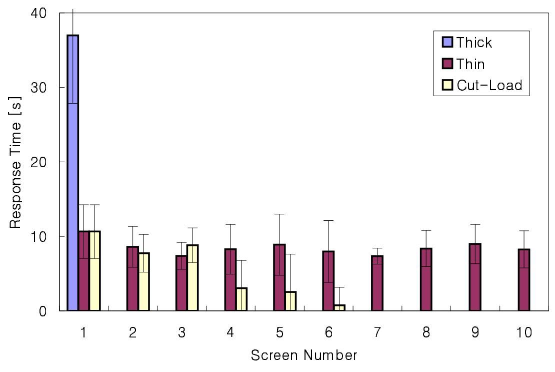


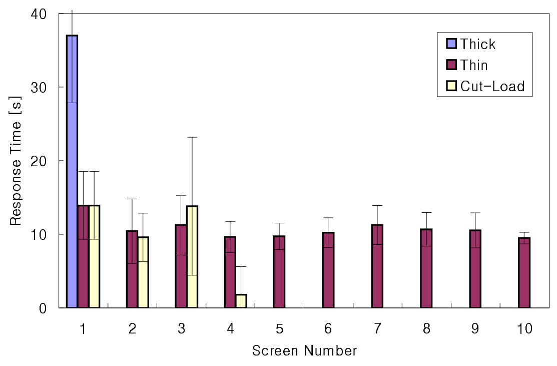
**Figure 19:** PDF of document and thin-screen data size.



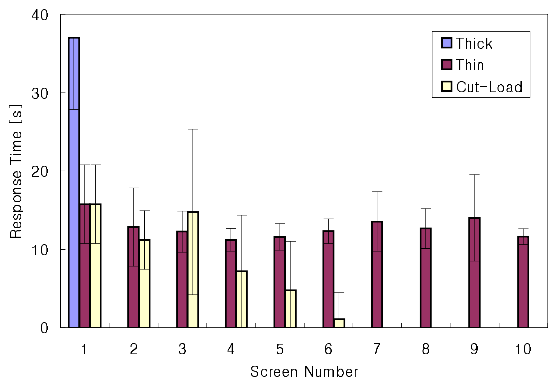
(a) Ratio=0.05.



(b) Ratio=0.10.



(c) Ratio=0.15.



(d) Ratio=0.20.

**Figure 20:** Per-screen response time in Cut-Load.

ratio increases, the performance benefit in both the client becomes less. Thus, the Cut-Load proxy measures this ratio in real-time and decides the best operation mode for the client.

If the ratio is over a threshold value and the dual-mode operation does not improve the performance, Cut-Load chooses the normal mode to access the content.

After accessing the first screen, Cut-Load continues to show similar performance to the thin client until it completes opportunistic hoarding. It is because the W-AIMD scheme for thick data in Cut-Load minimizes the effect on the performance of the greedy thin-data transmissions. In most cases, the response time is around 10 seconds, which is generally accepted as the user-tolerance limit [51].

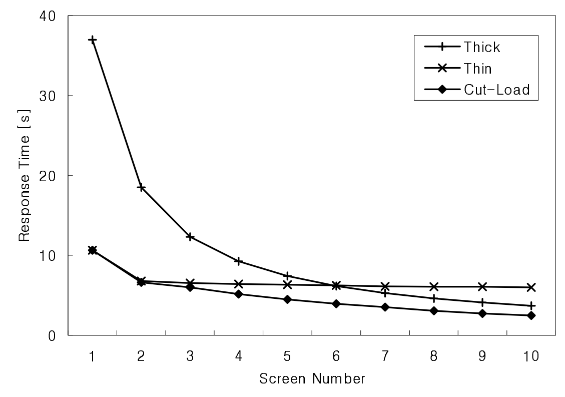
In the figure, it also can be seen that after the access of the third screen, the response time in Cut-Load begins to decrease and becomes zero after the sixth screen. In other words, most hoardings are completed while accessing the third to sixth screens, and after that the Cut-Load client responds to user's access requests immediately.

Another interesting fact is that even if the user accesses the entire screens of the content, Cut-Load may still show smaller cumulative response time than the thick client. In Figure 21, when the ratio of thin data is smaller than 20%, Cut-Load performs better for long or full accesses.

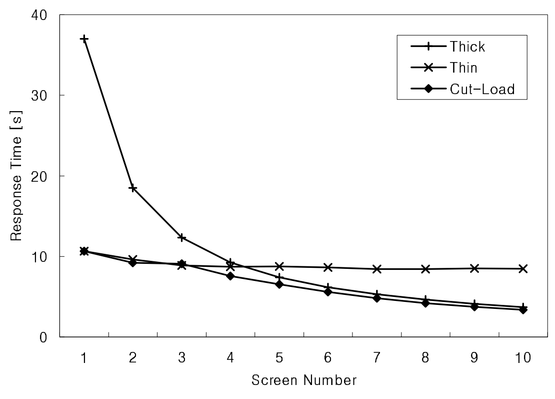
The reason that the Cut-Load client shows better overall performance than other clients is utilization of a judicious combination of thin client and conventional computing models. When a user requests a large-sized thin-friendly document, the Cut-Load client selects the initial thin mode to minimize the initial response time. As the user accesses more screens staying in the same content, the probability that the user will see the whole content increases, and thus the Cut-Load client increases the hoarding rate to reduce the future response time. Because the hoarding is performed in a less-greedy manner, it does not have a significantly negative influence on other users' response time performance.

### 4.5.3 Impact of network characteristics

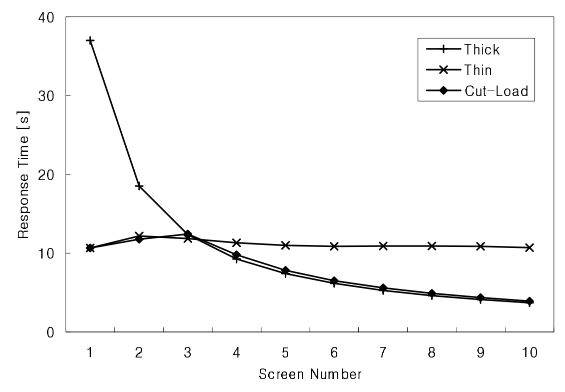
To investigate the impact of network bandwidth, we first vary the wireless bandwidth from 0.2 to 1.6 Mbps with a fixed data ratio value of 0.1 in the simulations. Since the thin mode shows similar performance to Cut-Load before hoarding is completed, we do not compare with its performance.



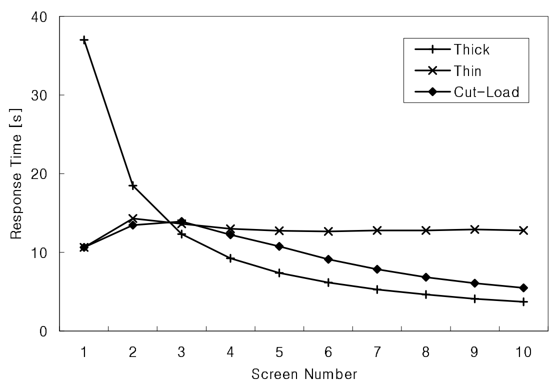
(a) Ratio=0.05.



(b) Ratio=0.10.



(c) Ratio=0.15.

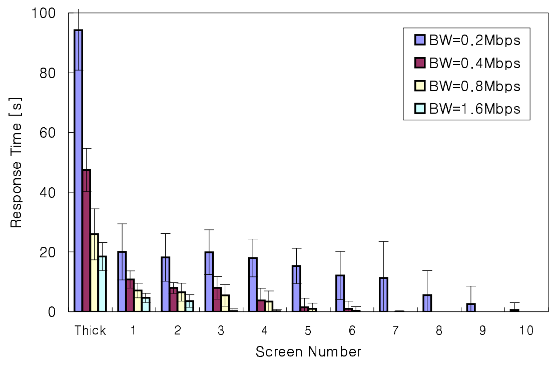


(d) Ratio=0.20.

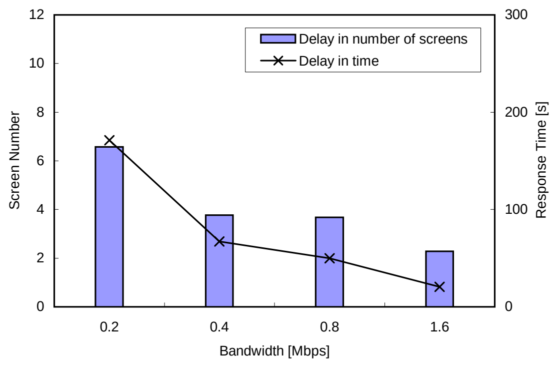
**Figure 21:** Cumulative response time in Cut-Load.

Figure 22(a) shows the per-screen response performance in both the thick client and Cut-Load. Note that the figure shows the thick performance separately on the x-axis because the thick client downloads the entire content at once. Thus, each screen number on the x-axis indicates the current screen focus in Cut-Load. From the figure, it can be seen that the response-time performance is affected directly by the provided bandwidth. And, the performance improvement achieved by Cut-Load is in the same proportion to the increase in bandwidth.

Figure 22(b) shows the hoarding delay in the number of screens, which are accessed in thin mode before the hoarding is completed and the computing mode is transferred to the thick mode. In the figure, it can be seen that the hoarding time is inversely proportional to the provided bandwidth. When the bandwidth is increased to the double, the delay in time is reduced to the half. However, the hoarding delay in number of screens is less affected by the bandwidth, and it is because the fetching speed of thin screens is also increased, *i.e.*,



(a) Per-screen response time.



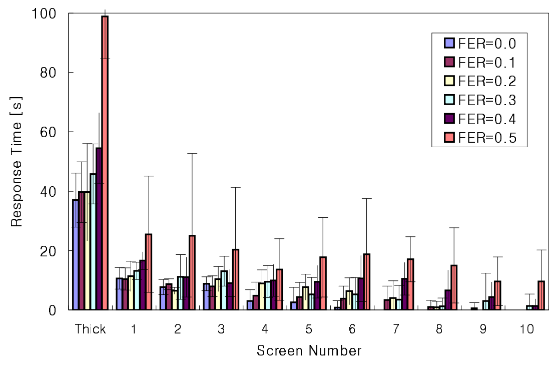
(b) Hoarding delay.

**Figure 22:** Impact of bandwidth in Cut-Load

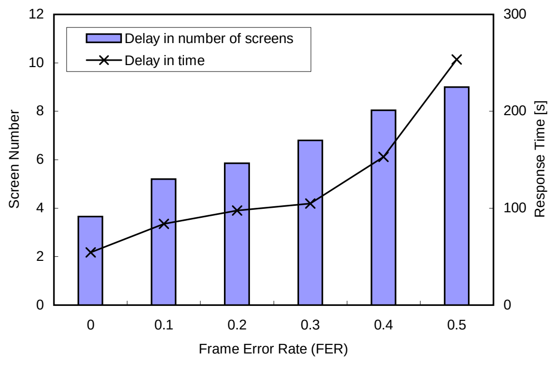
during the same amount of time, more number of thin screen can be accessed as bandwidth is increased.

On the other hand, we vary the frame error rate (FER) in the simulations. Since the 802.11 MAC layer performs retransmissions up to seven times for wireless frame loss, a high FER does not significantly affect the overall performance, unlike a high packet error rate (PER) in the higher layers.

Figure 23(a) shows the response time performance under the various FERs. In the figure, the response time performance is not significantly affected by low FERs, which are considered as normal operating points in 802.11. However, when FER is higher than 0.3, the performance begins to be degraded exponentially.



(a) Per-screen response time.



(b) Hoarding delay.

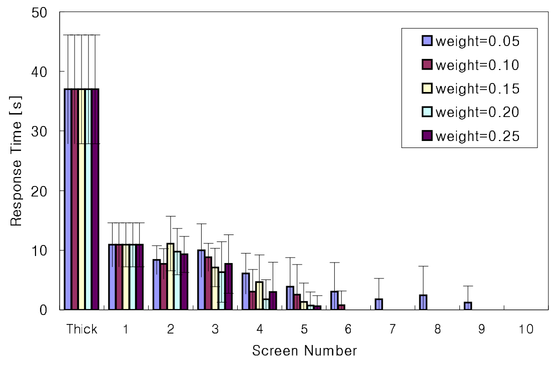
**Figure 23:** Impact of frame loss rate in Cut-Load

Figure 23(b) show the opportunistic hoarding delay in the number of thin screens, which are viewed until the hoarding is completed. In the figure, as the FER increases, more thin screens are accessed until mode transfer is performed. Thus, Cut-Load needs to control the hoarding rate according to the current FER.

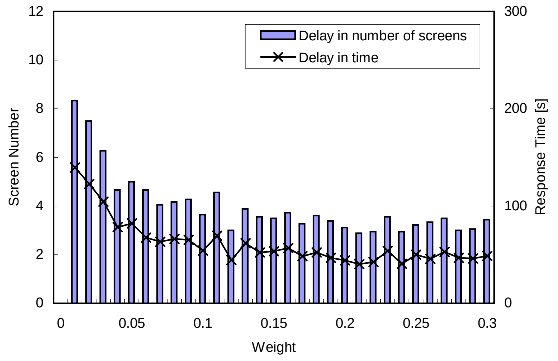
### 4.5.4 Impact of weight

Cut-Load controls the hoarding delay by selecting an appropriate weight used in W-AIMD, and we performed the simulations varying the weight from 0.05 to 0.25. We also assume that the bandwidth is fixed as 0.5 Mbps and wireless frame loss does not occur. The thin-to-thick data ratio is 0.1.

Figure 24(a) shows the response performance in time with various weight values in Cut-Load. Since hoarding begins after the initial screen is displayed, the weight does not affect the initial screen time in Cut-Load. However, after the first screen, the per-screen response time is directly affected by the weight value.



(a) Hoarding delay in time.



(b) Hoarding delay in number of screens.

**Figure 24:** Impact of weight in Cut-Load

Figure 24(b) shows the hoarding delay in time and number of screens. In the figure, it can be seen that the hoarding time decreases as the weight increases, however after 0.1 the hoarding speed is not improved significantly. As shown in Equation 2, the throughput performance in W-AIMD is proportional to the square of the weight. In other words, in order to increase the fetching speed to the double, it is necessary to increase the weight to the four time.

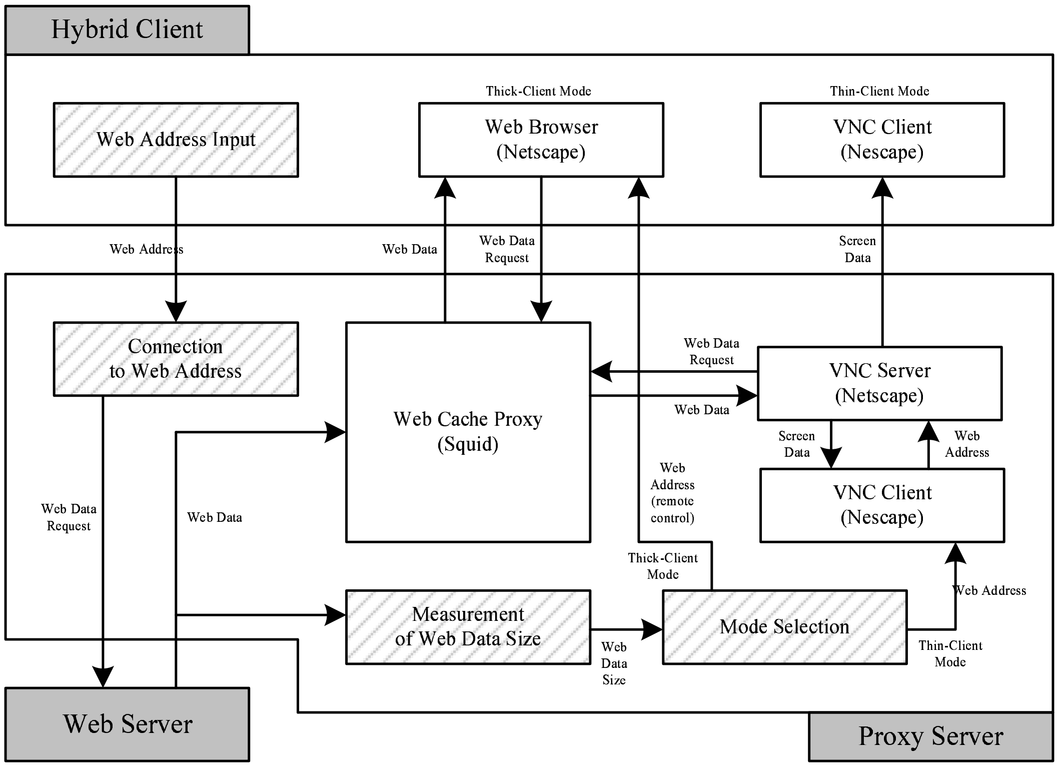


**Figure 25:** Cut-Load prototype.

### 4.5.5 Prototype Strawman

Now we present a strawman prototype of the Cut-Load mobile middleware for Web based accesses. Due to the requirement of an open source Web proxy for the implementation, we implement the prototype on a Linux platform using the Squid Web proxy cache [90]. We use VNC as the thin-client computing solution [78]. The prototype is used as a proof-of-concept and we study the behavior of the prototype in response to different user-accesses. It should be noted that the prototype itself is not tied to any of these operating platforms or thin-client solutions.

Figure 25 illustrates the architecture of our prototype. The client or the mobile host has two virtual panels representing the thick-client mode and thin-client modes. The thin-client mode panel is a VNC client. The proxy is a DELL OptiPlex Gx400 desktop with a 1.8GHz Pentium 4 processor and the Linux operating system. The proxy runs the VNC server and the squid proxy. The Web browser on the client machine in the thick-client panel is configured to redirect all Web accesses to the squid proxy. The mode selector chooses the

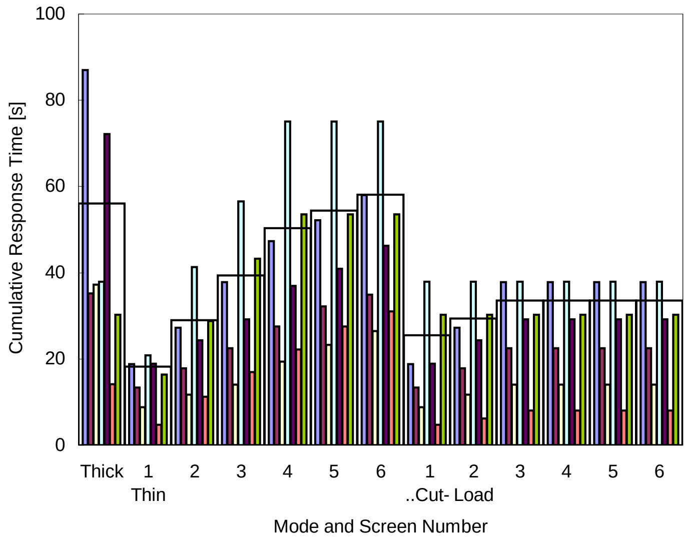


**Figure 26:** Experimental results of Cut-Load prototype.

mode of response depending upon a simple heuristic function and the raw data size of the document to be transferred. The mode selector comes into play only after the requested content is fetched from the appropriate Web server. When the mode selector chooses the thick-client mode, the squid proxy acts in its default mode delivering to the mobile host the Web content. However, if the mode selector chooses the thin-client mode of operation, the squid proxy aborts its regular content delivery process, but instead directs the Web browser running in the context of the VNC server to load the appropriate content. As shown in the figure, the interface with the browser is achieved through Netscape's remote control command-line function. Since the content has already been downloaded by the squid proxy, the load by the browser in the VNC server's context happens without any additional latency. The VNC client panel on the mobile host is automatically updated by the VNC mechanism to reflect the changed browser state.

Figure 26 shows the cumulative response time performance for seven different Web content in thick, thin, and Cut-load clients. In the figure, the transparent bars show the average response time values per screen in each mode. Since the current prototype does not perform W-AIMD-based opportunistic hoarding, we assume that the hoarding process is completed after the third thin access as shown in Figure 22. When the prototype detects partitionable content such as HTML, it chooses the normal thick access mode. In the figure, the 4-th and 7-th accesses are such examples. For other accesses, the heuristic function predicts the dual-mode operation in Cut-Load for the right decision to be made by the dynamic mode selector. Thus, after the third screen, the cumulative response time in Cut-Load does not increase as the same as in the thick client. In the figure, it can be seen that the initial response time in Cut-Load may become longer than that in thin mode, however the long-term average response time becomes shorter after the second screen.

## *4.6 Summary*

Traditional information access systems are not tailored to perform well in low-bandwidth wireless networks. In this section, we study the reasons why conventional client-server models are not optimal for wireless networks and find the reason for the inefficient performance is the operation in the file level and being unaware to user activity. We evaluate the use of application-independent content-partitioning in the graphical domain as an alternative to binary-level file transfers for efficient Web performance for low-bandwidth wireless links.

We found several issues in using pure graphical content-partitioning techniques to serve mobile information access requests. In addressing these issues, we propose and implement a new strategy for mobile information access over wireless links. The proposed strategy uses a intelligent mix of binary file-transfers and graphical content-partitioning along with features such as opportunistic hoarding to reduce the bandwidth consumption as well as response times for Web access. We evaluated the performance of the Cut-Load strategy and proved its benefits for non-partitionable content over traditional Web-access systems. We also analyzed the behavior of the real-life implementation of the Cut-Load prototype.

# CHAPTER V

# APPLICATION-AWARE READ ACCESS FOR PARTITIONABLE CONTENT

## *5.1 Introduction*

In the past couple of decades, a tremendous amount of research has been done on improving read-access performance for multi-file content such as Web documents. Web-optimization techniques such as Web cache proxies [90, 31, 89], persistent HTTP connections [21, 53], and content distribution networks [3, 64] have found widespread adoption. On the other hand, new paradigms such as WAP [67] and BREW [76] have also been developed to address the limitation of Web performance on mobile hosts.

In this work, we consider Web content as a representative of multi-file partitionable content and study the performance of a Web browser under low-bandwidth mobile network conditions. Specifically, we analyze the characteristics of a Web browser with the objective of identifying reasons why they might suffer in bandwidth-limited wireless environment. We find that the current fetching model employed by most commercial Web browsers is not optimal in bandwidth-challenged environments.

We show that the absence of content prioritization and intelligent object fetching mechanisms in current Web browsers leads to increased response times. Web browsers, today, do not prioritize useful data that is viewed by the user over other redundant data in a Web page. As a result, greedy fetching of the entire content of a Web page wastes precious bandwidth and in turn increases user-perceived response time. Further, without an intelligent object-fetching mechanism, the download process of current Web browsers does not utilize network bandwidth efficiently.

To make this problem even worse, many Web pages have become larger, both in pixels and bytes with a large number of embedded objects. For example, `cnn.com` has a main

Web page, which is more than three times the height of a *client area*[1] and larger than 300 KB in bytes, including hundreds of embedded objects. As a result, even with a 100-Kbps bandwidth in a mobile network, the time taken to fetch the entire page can be longer than 20 secs.

In this chapter, we propose a new Web-optimization solution called *Prioritized Fetching* to address the problem of large response time with current browsers. The solution is based on careful consideration of several factors, including the content displayed on the screen viewed by the user, server-side content distribution networks, and the relationship between the HTTP and TCP protocols. Our solution consists of three mechanisms: *object prioritization*, *object reordering*, and *connection management*. One major advantage of our approach is that it is purely client-side enhancement, and consequently it is easy to deploy since it only requires client-side installation to current Web browsers.

To summarize, our contributions in this chapter are:

- Identification of the inefficiencies of current Web browsers by carefully analyzing the interactions of several factors related to Web fetching;
- Proposal of three mechanisms to reduce user response time in an easy-to-deploy fashion.

## *5.2 Motivation*

In this section, we describe drawbacks in a conventional Web access model in low-bandwidth environments and use them as a motivation for designing a new Web access scheme.

### 5.2.1 Web Access Model and Simulation Setup

A typical network model for Web accesses is as shown in Figure 27. In this model, to access a Web page, a user inputs a uniform resource locator (URL) address into a Web browser window. Then, the browser requests a domain name system (DNS) server to translate the URL address into the corresponding IP address. After obtaining this IP address, the browser directly accesses the main HTML document located in the Web server. When load

[1]It is defined as the content area within the main window of a Web browser

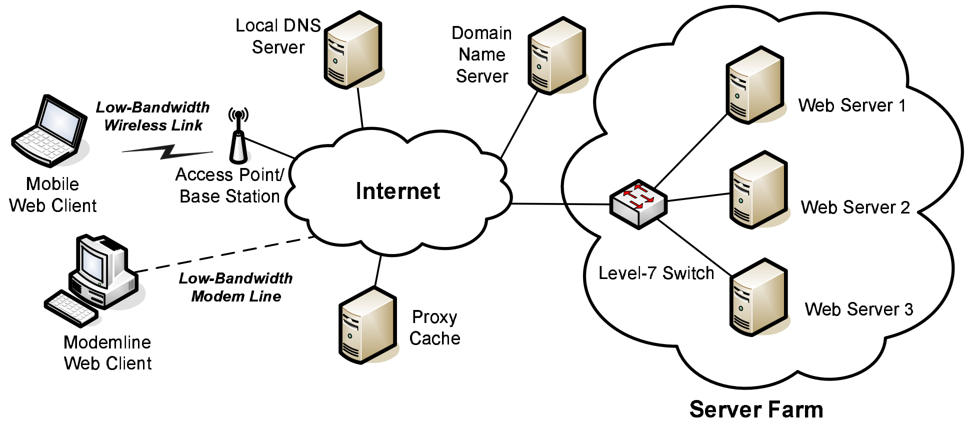


**Figure 27:** Typical network topology for Web access.

balancing is performed among multiple Web servers[2], a layer-7 switch rewrites the domain names of embedded objects defined in the HTML document to distribute requests of objects to multiple servers. Finally, the Web browser performs additional DNS resolutions for other unknown Web servers having those objects and downloads objects from them.

Typical Web browsers can open multiple connections to a single Web server to increase fetching speed. For example, Microsoft Internet Explorer and Netscape Navigator can open up to two and six TCP connections to a single server, respectively [92]. Each opened connection has its own message queue to send object requests, *i.e.*, HTTP GET messages. A parsing engine in a browser inserts object requests to all message queues in a round-robin fashion since the browser is unaware of object and network characteristics.

In this chapter, we consider the *Top 50 Web Sites* [13] as representatives of typical Web pages and measure their Web characteristics. Table 2 shows the statistical results of those Web sites. In the table, a *screen* refers to an effective area for displaying a Web page in the browser window. In the measurements, we used Microsoft Internet Explorer, of which the client area size is 1006×511 in pixels, under a 1024×768 screen resolution. The initial screen is defined as the first part that is shown in the client area when a Web page is accessed. In the measurements, the average number of screens in these pages is 1937/511 = 3.7.

To evaluate the performance of conventional Web access models in a low-bandwidth

[2]They create a server cluster, called a *server farm*

**Table 2:** Web statistics of Top 50 Web Sites.

| | | HTML | IMG | Others | Total |
|---|---|---|---|---|---|
| Byte size per object [KB] | Mean | 31.72 | 2.46 | 12.91 | 225.96 |
| | STD | 35.51 | 5.90 | 9.67 | 186.04 |
| Number in first screen | Mean | 1 | 17.31 | 4.41 | 22.72 |
| | STD | | 15.36 | 6.22 | 16.37 |
| Number in all screens | Mean | 1 | 46.80 | 3.99 | 51.79 |
| | STD | | 28.16 | 6.22 | 30.34 |
| Number of Web servers | Mean | 1 | 5.16 | 1.74 | 5.50 |
| | STD | | 2.90 | 1.20 | 3.38 |
| Width [pixels] | Mean | | | | 998 |
| | STD | | | | 46.49 |
| Height [pixels] | Mean | | | | 1937 |
| | STD | | | | 1119 |

environment, we use *ns2 simulator* [42] with the *Reno-FullTCP* package, which supports bidirectional transmissions. In the simulations, the same network topology as shown in Figure 27 is used with the assumption that a local DNS server has all the required domain information. The bottleneck link is located between the Web client and the backbone network and is configured to have a 100-Kbps bandwidth and a 100-ms link delay. The bandwidth and delay from both the DNS and Web servers to the backbone network is 1 Mbps and 5 ms, respectively.

For modeling Web traffic, we use the same Web characteristics as shown in Table 2. The average processing time of an object in the browser is assumed to be 200 ms, and parsing delay is ignored. We assume that all Web servers support HTTP/1.1 with the persistent connection feature, but pipelining is not considered since it is not faithfully supported by most commercial Web servers [9]. The byte size of a HTTP request message is set as 500 bytes, and the header size of a HTTP response message is ignored. It is also assumed that the cache function of the browser is disabled.

We consider *initial screen-response time*, which is defined as the difference between the time when a Web browser sends a request for a Web page and the time when an HTML document and all embedded objects required for displaying the initial screen are downloaded completely, as a primary metric.

### 5.2.2 Screen Contention Problem

When a user views a screen on a display device, objects for displaying other screens are unnecessary in the sense that they are not visible to the user at this time. However, in conventional Web browsers, the process of fetching necessary on-screen objects (*i.e.*, objects on the current screen) may be slowed down as a result of the competition from the process of fetching unnecessary off-screen objects. We refer to the fact that objects from different screens are competing for bandwidth as *screen contention*.

The main reason for screen contention is the disparity of cumulative transfer sizes among multiple connections. As mentioned earlier, a parsing engine inserts object requests to message queues in a round-robin fashion, which considers only fairness in the number of objects per connections. As a result, some connections having only small-sized objects may finish transmissions of on-screen objects earlier than others and begin to fetch off-screen objects. Under this scenario, different connections may fetch objects on different screens simultaneously.

Another possible reason is directionality in a table structure in HTML. When a multi-cell table is used in a Web page, the internal cells in the table may have a significantly larger height than the client area. In such a case, only after the Web browser fetches off-screen objects located at the end of the first cell first, it begins to fetch on-screen objects located at the beginning of the next cells. Figure 28 shows an example of an object-fetching sequence when Internet Explorer accesses the `amazon.com` page, which consists of one HTML document, five javascript, two flash, and 65 image objects. In the figure, many off-screen objects are fetched or begin to be fetched before downloading all 35 on-screen objects is completed because of the same reason.

We show the effect of the screen contention problem by presenting the simulated object fetching progress in Figure 29. We assume that all the objects are from a single server and the effect of directionality in a table structure is ignored. In the simulations, the initial screen has 18 on-screen objects, *i.e.*, objects numbered after 18 are unnecessary data. As observed from the figure, objects on both the current and other screens are fetched simultaneously because of the screen contention problem. Since fetching the unnecessary objects consumes

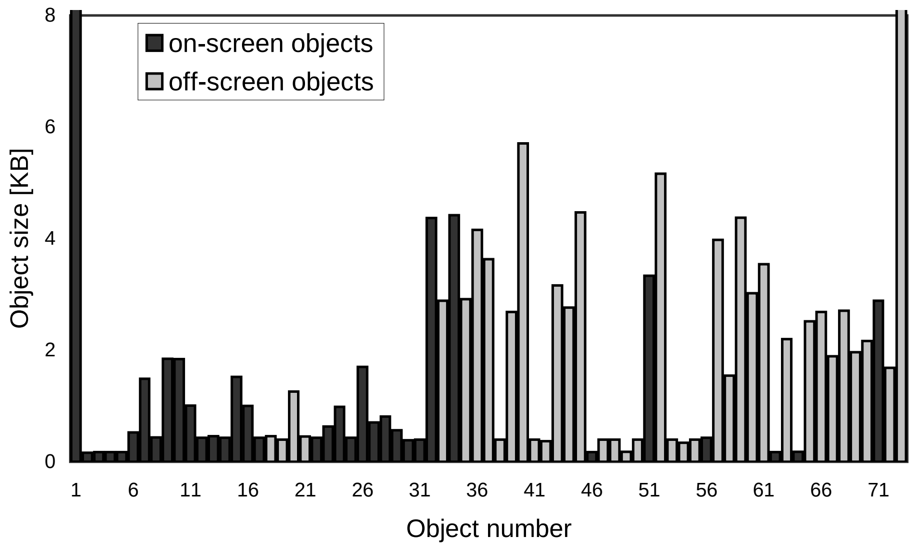


**Figure 28:** Object fetching sequence at `amazon.com`.

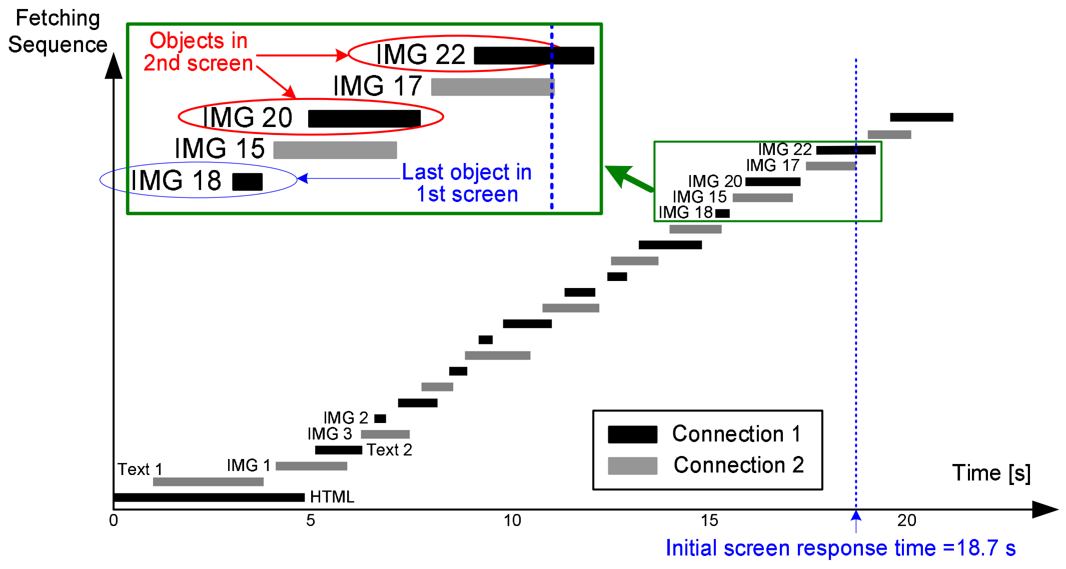
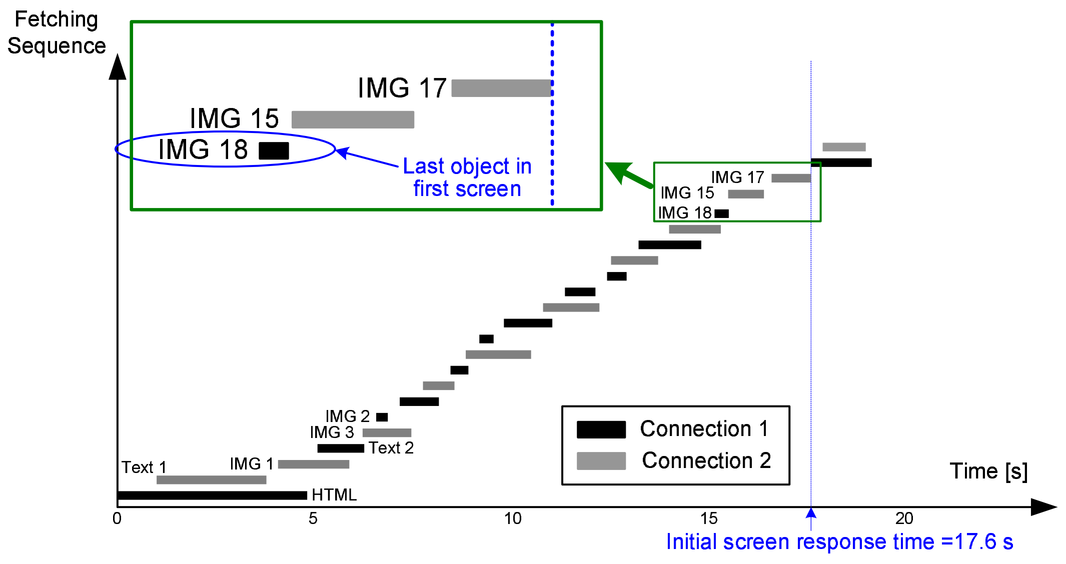


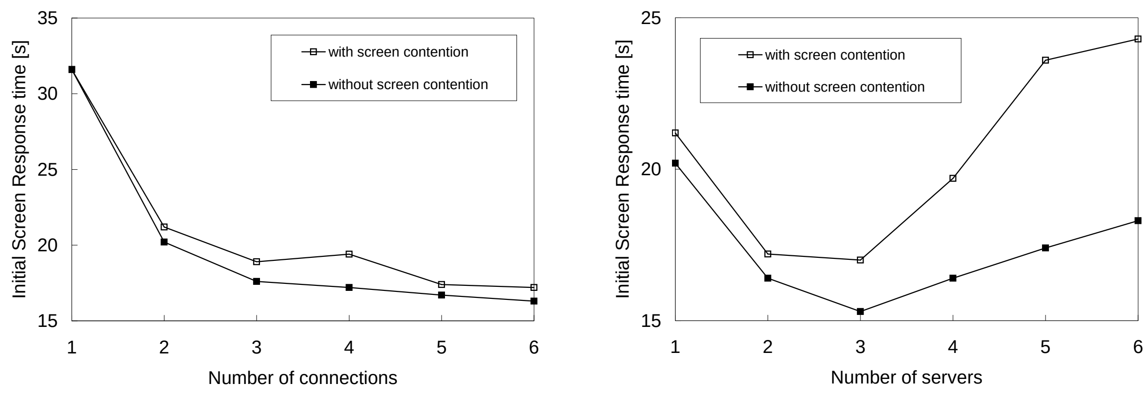
**Figure 29:** Fetching with screen contention in a conventional Web browser.

some portion of bandwidth, the resulting response time for the initial screen is increased unnecessarily. In the figure, two off-screen image objects (IMG) numbered 20 and 22 are fetched in parallel with other on-screen objects located on the initial screen. As a result, the response time for the initial screen, which was measured after IMG 17 was downloaded, is 18.7 secs.

An intuitive solution to screen contention is to prevent unnecessary object fetching. Figure 30 shows an ideal case where contention is eliminated in an ideal browser. As seen



cc



(a) Single server case.

(b) Multiple server case.

**Figure 31:** Impact of screen contention.

from the figure, when the faster connection (*i.e.*, Connection 1) completes downloading all the on-screen objects on the initial screen, it stops fetching and waits for the other connection (*i.e.*, Connection 2) to finish fetching the other on-screen objects. Thus, the remaining connection can obtain more bandwidth and in turn reduce the response time for the initial screen by 1.1 secs.

Figure 31 shows how screen contention affects the initial screen response time under both single and multiple servers scenarios as the numbers of connections and servers increase. In the multiple servers case, we assume that up to two parallel connections are allowed to each server. In Figure 31(a), as the number of connections increases from one to three, both the conventional and ideal browsers show significant performance improvement. However, when the number of connections is larger than four, the performance becomes less affected by it. When Web objects on a single page are distributed to multiple servers by a load balancing technique, the performance is directly affected by the number of servers. In Figure 31(b), both schemes show their best performance as the number of servers becomes closer to three. However, the performance in the ideal scheme is less influenced by the number of servers as a result of its effective prevention of screen contention.

### 5.2.3 Bandwidth Underutilization Problem

In HTTP/1.1, a persistent connection consists of a series of request-response transactions. Given this model, [9] shows that the idle time of a network decreases with an increase in the number of simultaneous TCP connections. In the paper, the authors also show that there exists an optimal number of simultaneous connections (around six in the paper), at which the performance is optimal because of reduced idle time. However, since current Web browsers do not schedule object transfers in a bandwidth-efficient way across multiple TCP connections, they do not always maintain the optimal number of simultaneous connections. In many cases, only a small number (*e.g.*, one or two) of connections are active at any instant. This results in underutilization of the access link, which we refer to as the *bandwidth underutilization* problem.

The above-described bandwidth underutilization problem results in varying levels of performance degradation, depending on the number of Web servers. In the case of a single server, bandwidth efficiency is determined by how much the ending times of transmissions among parallel connections are synchronized. In Figure 30, the last object (*i.e.*, IMG 17) is fetched with no other objects, and thus only a single connection uses network bandwidth toward the end. In the case of multiple servers, the user performance is also affected by synchronized ending among connections to other servers.

The solution to the bandwidth underutilization problem is to schedule GET requests across multiple TCP connections such that all the connections are always active during the fetching process. Figure 32 shows the impact of performing ideal scheduling such that there

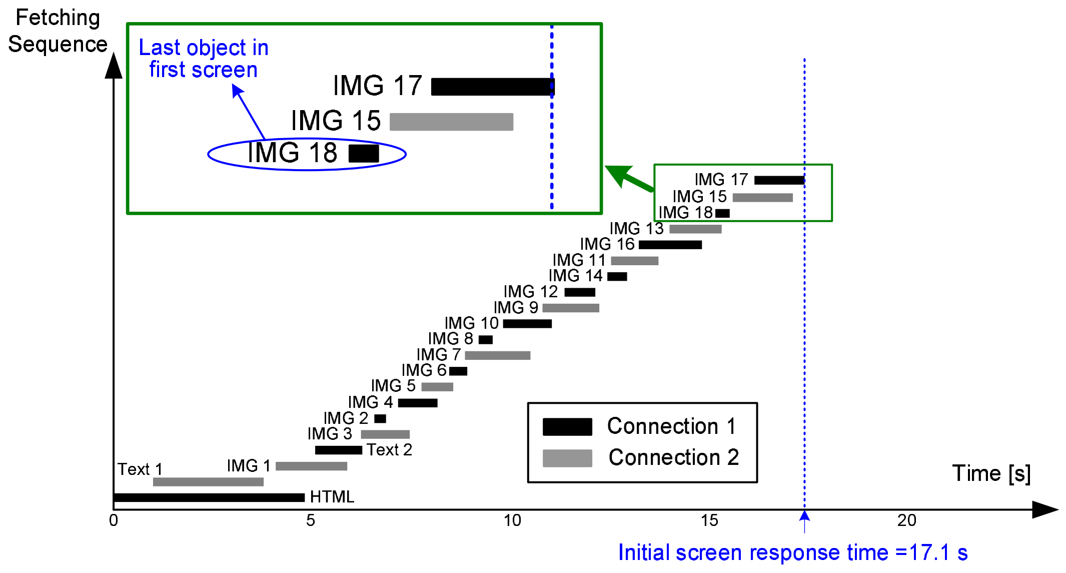
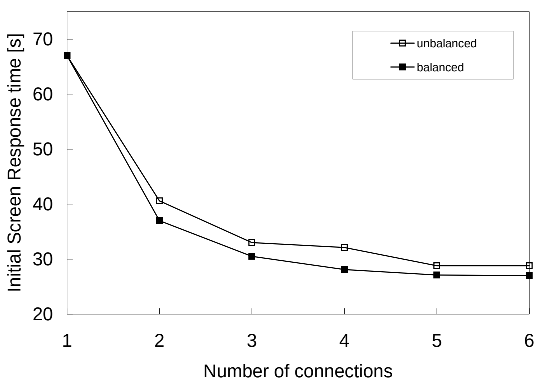


**Figure 32:** Fetching with synchronized ending time in ideal Web browsers.

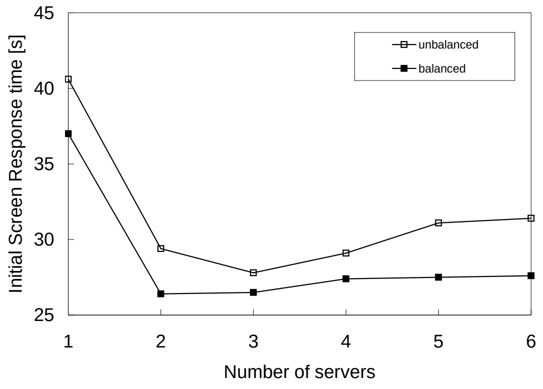
always exist one or more pending requests in every connection. In the figure, when the faster connection finishes fetching all on-screen objects that were in its request queue and has no more objects to fetch, it takes over the unfulfilled object requests from the queue of the other connection and performs fetching. As a result, both the connections can use bandwidth more efficiently, and the initial screen response time is reduced by 1.6 s when compared to that in a conventional Web browser.

In the scenario of multiple Web servers, different objects belonging to the same Web page are delivered by opening TCP connections to the different servers. Commercial Web browsers do not take into account the sizes of objects in scheduling object requests. This invariably leads to the scenarios, in which several TCP connections to Web servers are idle while the other connections are active.

The intuitive solution is to schedule the different object requests across multiple servers such that all the connections are active. Figure 33 shows how bandwidth underutilization affects the response time performance under various server scenarios. Note that the response time in the figure means the transfer time for all the objects of a Web page, including off-screen objects, and thus screen contention does not exist in this scenario. In Figure 33(a), the single server case shows a similar pattern as in Figure 31(a) and improves



(a) Single server case.



(b) Multiple servers case.

**Figure 33:** Impact of bandwidth underutilization.

the performance consistently in the entire range. In Figure 33(b), unlike Figure 31(b), as the number of servers increases beyond 3, both the schemes show stable performance, and the ideal scheme shows up to a 20% performance improvement.

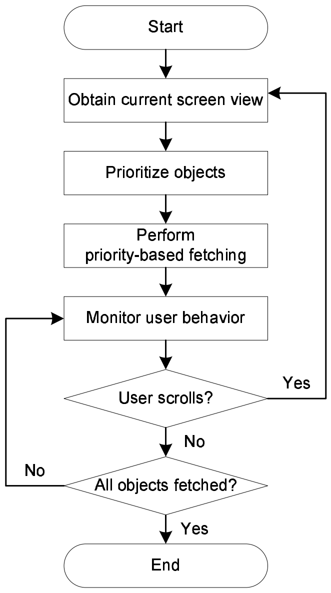
### 5.2.4 Summary

In this section, we have identified two issues with conventional Web browsers in bandwidth-limited networks. First, we observe that contention among objects belonging to different screens within the same Web page can increase user-perceived response time of the initial screen. Second, we identify that network bandwidth can be underutilized because of non-synchronized ending times of transmission. We also observe that in most cases the screen contention and bandwidth underutilization problems affect user performance negatively in a conventional Web model, and show how the ideal browser can overcome these problems and achieve significant performance improvement. Based on these observations, in the next section, we propose a new Web access scheme.

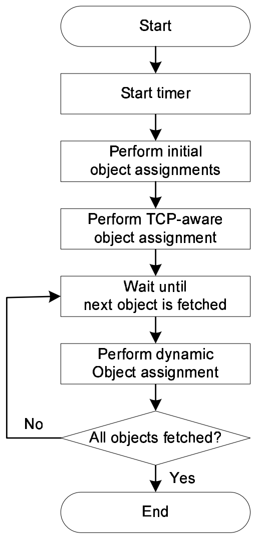
## *5.3 Overview of Solution*

Our proposed solution includes three mechanisms: *object prioritization*, *objects reordering*, and *connection management*. The brief summary of the mechanisms is as follows.

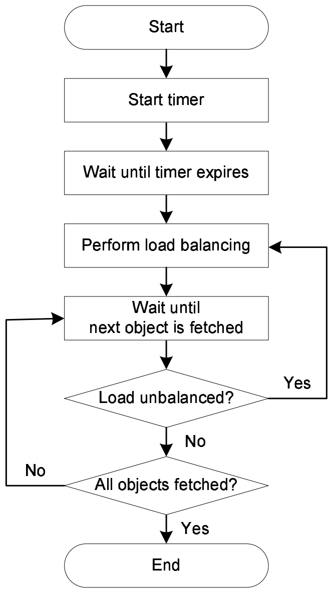
- Object Prioritization (OP) addresses the screen contention problem in a multiple-screen page and provides an optimization solution for fetching objects with varying



(a) Object prioritization (OP).


Start
Start timer
Perform initial object assignments
Perform TCP-aware object assignment
Wait until next object is fetched
Perform dynamic Object assignment
All objects fetched?
No
Yes
End

(b) Object reordering (OR).



(c) Connection management (CM).

**Figure 34:** Flow charts of three mechanisms.

priority levels. Basically, OP is a *What-You-See-Is-What-You-Fetch (WYSIWYF)* mechanism. While giving higher priority to on-screen objects, it gives lower priority to off-screen ones to reduce the user-perceived response time.

- *Object reordering (OR)* addresses the bandwidth underutilization problem when a browser downloads multiple objects from a single Web server. When load on those connections to the server is unbalanced, OR reschedules object requests across connections.

- *Connection management (CM)* addresses the bandwidth underutilization problem when multiple servers are involved in a single Web page. In order to balance load among connections to different servers (*i.e.*, domains), it performs dynamic reassignments for the entire connections.

The three proposed mechanisms complement each other as well as perform optimization with different levels of granularity for Web object fetching on the browser side. One major advantages of our solution is easy deployment, since it requires only client-side modification. In fact, the solution can be implemented as nothing more than an add-on to the current

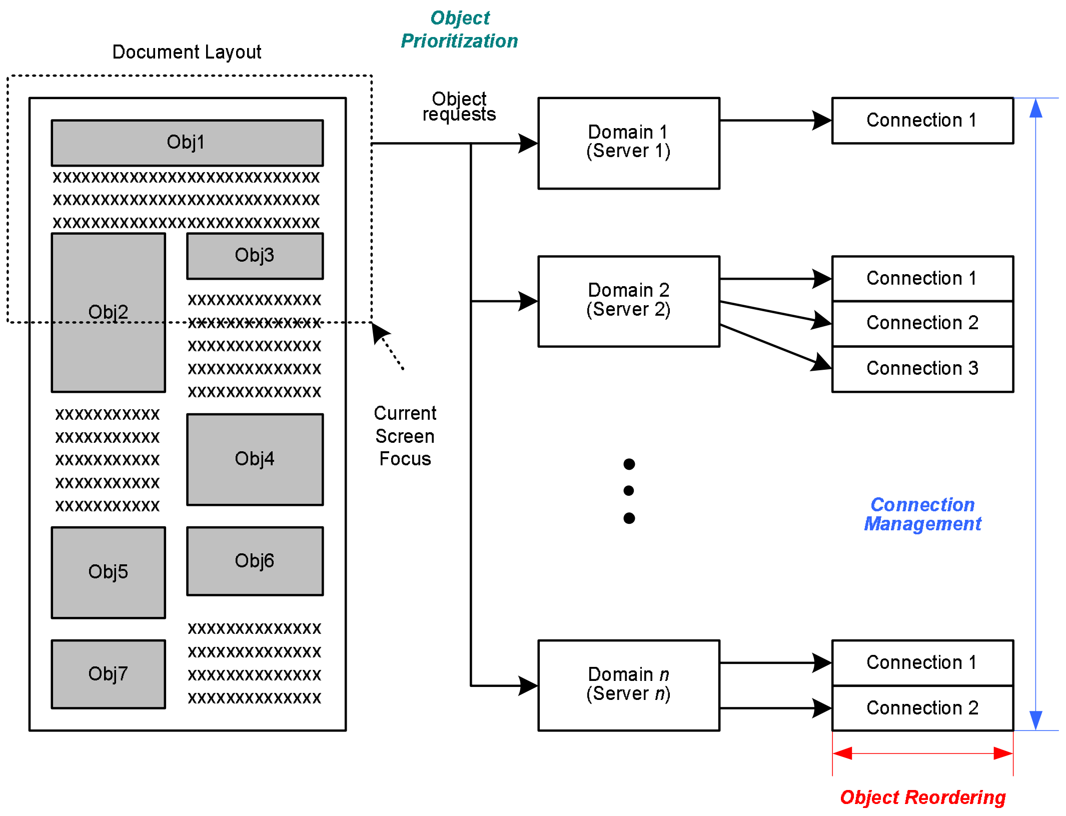


**Figure 35:** Overview of three mechanisms.

Web browsers. Figure 34 shows the flow charts of these mechanisms, and Figure 35 shows where they are located in the entire data flow.

## *5.4 Object Prioritization (OP)*

While a Web browser is downloading a main HTML document, it also scans HTML tags in the part of the document and updates the DOM tree continuously. When it finds an embedded object by scanning, it immediately begins to fetch the object. This on-the-fly fetching mechanism may bring performance improvement in high-bandwidth networks, where the overhead of screen contention is relatively small. However, in a bandwidth-limited client, this overhead affects user performance significantly, as mentioned earlier. Thus, the object prioritization (OP) mechanism differentiates objects based on their locations in the entire document layout and allows for downloading only the on-screen objects, which are required to render the current screen display. As a result, it reduces the response time experienced by Web users.

The basic operation step of OP is as following: (1) When a Web access is requested, it first obtains the initial screen view information in the document layout and prioritizes embedded objects according to their locations in the layout. (2) Then, it performs fetching objects according to their priority levels. (3)When a user scrolls to move to a different view, it performs the above-mentioned process again for the new screen.

OP consists of three components, *initial object prioritization*, *selective object fetching*, and *reprioritization for screen update*. The detailed operations are illustrated in Figure 34(a).

### 5.4.1 Initial Object Prioritization

Generally, a Web-page consists of various types of embedded objects. OP considers text-based files including HTML, javascript, cascading style sheets, and other layout-related files, as the highest-priority objects since these objects play an important role to construct the overall HTML display layout. On the other hand, for other types objects such as image and multimedia objects, OP assigns different priority levels according to their locations. For simplicity, we consider only IMG objects as representatives of objects that do not affect the document layout.

As mentioned earlier, OP performs location-based prioritization for IMG objects. The detection of pixel-location of an IMG object is possible because most HTML document files defines the pixel-size of image objects and a Web-browser can construct the full page layout without downloading these objects. In cases that the HTML document does not specify the pixel-size of an image object that is not fetched yet, OP uses an pre-obtained averaged value based on browsing history.

In order to get the location information of objects, OP scans the document object model (DOM) tree[97]. When it finds an IMG object definition, it searches all the successors in the tree and calculates location offsets from successors to predecessors in a recursive fashion until it reaches the top of the tree. The absolute location in the layout is defined as the sum of all the relative offset. Based on this location information, OP gives highest priority level to objects that are located within the current view in the client area and low priority levels to others.

### 5.4.2 Selective Object Fetching

For the schemes used to fetch objects of different priority levels, many existing schemes such as [38] allocates a small portion of bandwidth for low-priority transmissions. However, these schemes cannot be efficiently exploited in OP due to the following reasons: (1) HTTP 1.1 defines persistent connections, which allows transfer of multiple objects using a single TCP connection. In OP, a priority-based connection doesn't have flexibility to send both high- and low-priority objects. (2) Each TCP connection performs multiple short transmissions. As mentioned earlier, generally IMG objects that account for the half of total byte-size of a Web-page have a small average byte-size (a few packets). In these short bursty on-off transmissions, priority-based connection schemes can't assign a desired portions of bandwidth accurately.

Thus, OP uses a delayed-transmission scheme. When information of new objects are extracted from a HTML document and they are prioritized as high level, OP inserts the corresponding request messages into the already-in-use queues. In this scheme, low-priority objects are fetched only after all the higher-priority objects have been downloaded, *i.e.* after the higher-priority queues become empty.

### 5.4.3 Re-prioritization

When browsing Web-pages, a user may scroll to another view other than the current one before all the on-screen objects in the current screen are fully downloaded. For example, a user may perform *fast scroll* by searching and clicking an internal link to another part in the same page. In these scenarios, the current focus is changed before the downloading of previous screen, and the initial prioritization may not perform efficiently. Thus, a proper mechanism is required to deal with these scenarios.

When the screen focus is moved to a new area, OP removes all the IMG objects that reside in request queues and re-prioritizes them for the newly focused area. For the objects that are currently being downloaded, OP waits for their completeness. The reason for allowing this off-screen fetching is that most Web-browsers, as applications above transport layer following HTTP standards, do not have mechanisms to manage disconnections and re-connections. OP thus keeps the currently incoming transfer and only updates the priority levels of the queues involved.

The provided fetching schemes can be different transport protocols, different parameters in the same protocol, or different starting time. As mentioned earlier, in this work, we consider only adjusting starting time to fetch different screens using the currently existing transport protocols.

### 5.4.4 Analytical Model

OP mechanism can be formulated using the following model. Assume that a Web-page consists of a set of screens, $S = \{s_i | 1 \leq i \leq N\}$, where $s_1$ and $s_N$ are the first and last screens respectively, OP provides a set of priority-based fetching schemes, $P = \{p_i | 1 \leq i \leq M\}$, where $p_1$ and $p_M$ are the highest and the lowest priority levels respectively. Then, OP fetches each screen $s_i$ with scheme $p_j$ in the way that gives the user least response time.

The provided fetching schemes can be different transport protocols, different parameters in the same protocol, or different starting time. As mentioned earlier, in this work, we consider only adjusting starting time to fetch different screens using the currently existing transport protocols.

## *5.5 Objects Reordering (OR)*

For parallel connections to a single server, OR uses balanced ordering of objects to gain benefits in terms of reduced response time. The operations of OR consist of three steps. (1) At the first step, an initial assignment of objects is executed. (2) Then, an optimized ordering of objects is performed by TCP-aware object reordering. (3) After that, it performs dynamic objects rescheduling until all objects are completely fetched. The detailed operations are illustrated in Figure 34(b).

### 5.5.1 Initial Objects Assignment

As we identify in the previous section, conventional browsers perform a round-robin assignment to distribute object requests to multiple connections. This size-unaware assignment may cause unsynchronized ending time among different connections, and as a result increases response time. Therefore, OR performs load balancing among connections using byte-based metric rather than simple round robin in order to synchronize their ending time.

Since larger byte size translates to longer downloading time in Web fetching, OR synchronizes ending time of different connections by distributing same amount of objects to every connection. A more accurate way to synchronize the ending time could be one that also considers the number of objects, the precessing time for each object, and others. That is, the expected ending time is given by $\frac{Size_{Data}}{BW_{Available}} + n * \frac{rtt}{2} + T_{Proc}$, where $n$ is the number of objects and $T_{Proc}$ is the processing time.[3] However, OR simplifies the metric by considering only data size, based on the observation that the first term, $\frac{Size_{Data}}{BW_{Available}}$ dominates over other terms in low bandwidth networks.

Performing OR requires the byte-size information of objects. Since this information is normally not included in HTML documents, OR estimates it by considering both the object's pixel-size included in HTML documents and the object formats such as gif and jpeg. Based on this data size information, OR sends object requests through multiple connections in a balanced way. A time with a $\varphi$ expiration value is used to strike the balance between amount of objects and increased response time.

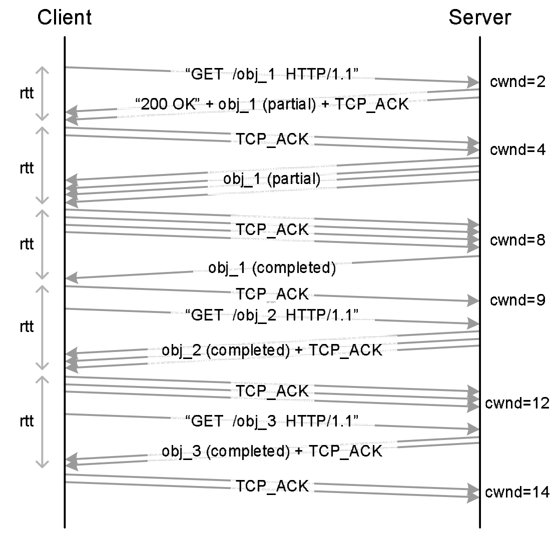
### 5.5.2 Dynamic Objects Rescheduling

Irrespective of initially balanced assignment of objects among connections, due to dynamic behavior of connections, the total fetching time of different connections may still vary significantly. If due to some reasons one connection is delayed, and the other connection is idle, it is possible to reschedule the objects from the busy connection to the idle one, and thus reduce response time even more. Dynamic objects rescheduling runs in an on-demand fashion during the fetching process in order to deal with the dynamic nature of the connections.

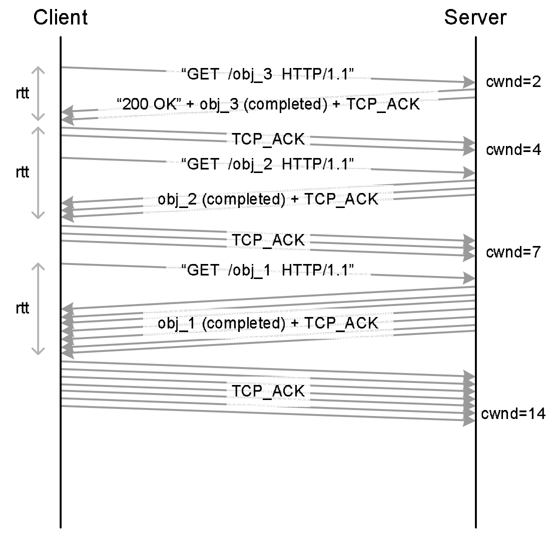
[3]Since fetching each object has to follow the HTTP request-reply handshaking pattern, which wastes 0.5 $rtt$.

### 5.5.3 TCP-aware Objects Reordering

When initial Web objects are assigned to a connection, TCP-aware reordering of the fetching sequence can increase download speed by minimizing the adverse effect of slow start in TCP. Let us assume that there are three different objects with the sizes of seven (obj_1), three (obj_2), and two packets (obj_3) respectively are waiting to be fetched along a single connection. If the TCP connection is newly created and its congestion window size starts at two[4], the total downloading time for an order of obj_1→obj_2→obj_3 is five *rtt*s, as shown in Figure 36[5]. However, if an opposite order of fetching is allocated, it may reduce the total fetching time to three *rtts*. Thus, appropriate ordering the fetching may save response time in the order of *rtt* of the path.



(a) With the initial fetching sequence



(b) With the modified fetching sequence

**Figure 36:** Example of TCP-aware reordering.

TCP slow start can kick in at any time during the downloading process. However, since HTTP and upper layer is unaware of each other's status information, there is no way to take advantage of them without some other cross-layer mechanisms. Thus, the TCP-aware objects reordering scheme only makes use of the slow start phase in the beginning of a TCP

[4]More than 95% of the servers do not perform TCP-JumpStart [71].

[5]We do not consider the Delayed ACK Scheme described in RFC 2581 here for better observation of the change of *cwnd*.

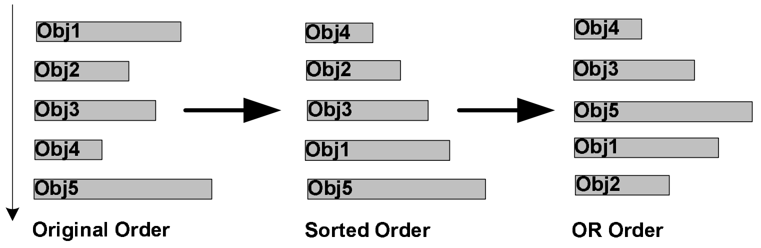


**Figure 37:** Reordering using OR.

connection.

For this rescheduling to take place, appropriate ordering of objects is required. Intuitively, with small objects being put at the end of connection, it is more likely to reschedule objects among connections, and thus reduce response time. Also, ordering objects from big to small also makes rescheduling easily to perform, since small objects can be rescheduled in a *finer* granularity as the fetching process going on.

Thus, both considering the two requirements, OR orders the fetching sequence in a *rats-elephants-rats* fashion. The detailed operation of TCP-aware Objects Reordering is illustrated in Figure 37. First, all the objects assigned to one connection are sorted according to their data size. After that, from smallest one, all objects are inserted from two ends of the queue in round-robin way, and the resulting ordering is a small-to-big-to-small order.

### 5.5.4 Analytical Model

This problem that OR is addressing can be formulated into the following analytical model. Given a screen, assume a set of objects, $O = \{o_i | 1 \leq i \leq N\}$, where the *i-th* server has $o_i$ IMG objects and $N$ is the total number of servers. Assume totally $J$ connections will be opened for fetching objects from this server, and let $C$ denote this connection set.

The output of OR includes two parts. First, an assignment of $O$ to $C$, such that the objects are equally distributed among connections. In other words, the maximum amount of data assigned to the connections is minimized. Let's assume the amount of data assigned to connection to $c_j$ is denoted by $d_j$, and the objective is to minimize the maximum value of $d_j$. Second, an appropriate ordering of these objects on these connections. The fetching order of these objects should avoid the adverse effect of HTTP-TCP interaction, and is easy to reschedule objects among connections.

## *5.6 Connection Management (CM)*

CM addresses the bandwidth under-utilization problem when fetching objects from multiple servers by controlling the numbers of connections a browser can open to different servers. By adjusting the number of connections for each server, CM effectively synchronizes the ending time of downloads in the connections. As a result of the improved bandwidth utilization, the response time is reduced. CM consists of two components, estimation of per-connection load and dynamic connection assignment. The detailed operations are illustrated in Figure 34(c).

### 5.6.1 Per-Connection Load Estimation

In order to estimate the ending time of downloading, CM uses the byte-size information that OR converted earlier. The intuition of CM is to assign more connections to servers with larger data size, while assign less connections to servers with smaller data size. To maintain friendliness to current browsers and compatibility to published standards, the total number of connections in our mechanism is maintained the same as in today's popular browsers. By doing so, CM behaves *friendly* to them. To achieve this purpose, whenever it assigns one more connection to some server, one less connection should be deducted from some other server. Furthermore, CM limits the maximum number of connections assigned to a server to four due to several reasons including the observation made by [9] stating that allocating too many (say, more than six) connections to the same server does not necessarily lead to better performance.

### 5.6.2 Dynamic Connection Assignment

When fetching a HTML document from a server, a browser fetches and parses the contents. Whenever it detects new object information, it estimates the byte size of this object, and starts a timer with $\delta$ expiration value. The setting of this timer requires careful consideration. On one hand, CM needs to collect some amount of object samples in order to achieve improvements. Thereby the $\delta$ should not be too small. On the other hand, CM should not delay object fetching significantly to avoid increasing response time adversely, and thus the expiration value should not be very large such as dozens of ms.

After the expiration of this timer, CM performs the initial assignment based on object information collected so far. During the process of fetching objects, it keeps recording the object information on how much data already received. This information will be used again to adjust the number of connections

### 5.6.3 Analytical Model

This CM mechanism can be formulated into the following analytical model. Given a set of servers, $S = \{s_i | 1 \leq i \leq N\}$, where $N$ is the total number of servers, let $d_i$ denote the total data size of objects from server $s_i$. Given a connection set, $C = \{c_i | 1 \leq i \leq N\}$, where $c_i$ is the number of connections opened for server $s_i$, CM finds a minimized maximum value of $d_i/c_i$.

$C$ is also subject to three other constraints. First, the total number of connections should not exceed $2N$ to maintain friendliness to current browsers. Second, the $c_i$ should not exceed the number of objects in $s_i$. Third, $c_i$ should have a range from one to four.

Let us use $n_i$ to denote the number of objects in server $s_i$. The output $C$ should achieve the purpose described in Eq. 4, and satisfy the constraints denoted in Eq. 5.

$$L_{max} \text{ \textit{is minimized},} \quad \text{\textit{where} } 1 \leq i \leq N \tag{4}$$

$$\sum_{1 \leq i \leq I} C_i \leq 2I \quad \text{ \textit{and} } \quad 1 \leq C_i \leq min\{4, N\} \tag{5}$$

The detailed algorithm is as follows. In the beginning, every server is assigned two connections, and CM computes the largest and smallest values of $L_i = d_i/c_i$. We use $L_{max}$ and $L_{min}$ to denote the largest and smallest values of $L_i$. Assume server $s_j$ has the largest value (*i.e.* $L_{max} = d_j/c_j$), and server $k$ has the smallest value, $L_{min}$. Now, we increase $c_j$ by one (*i.e.* $c_j = 3$), and decrease $c_k$ by one, then compute the new maximum value, $L_{max'}$. If $L_{max'} < L_{max}$, that means the new assignment has a smaller maximum value, then the algorithm will continue to run. Otherwise, if $L_{max'} > L_{max}$, the algorithm stops and resume to previous assignment, since the new assignment results in a larger maximum value.

The algorithm runs whenever an object is downloaded. For the new runs, the metrics considered in Equation 4, *i.e.* $s_i$, are set to be the remaining data size for server $s_i$. Thus, the algorithm will be performed whenever an object-related event happens. For this reason, an adverse effect - fluctuation on the number of connections assigned to servers, may possibly happen. To reduce this fluctuation, we introduce a threshold value, $\tau$. In CM, only when the $L_{max'}$ values between two consecutive iterations is larger than $\tau$, the algorithm will adjust connections set. We suggest a 10% of previous $L_{max'}$ as the $\tau$ value.

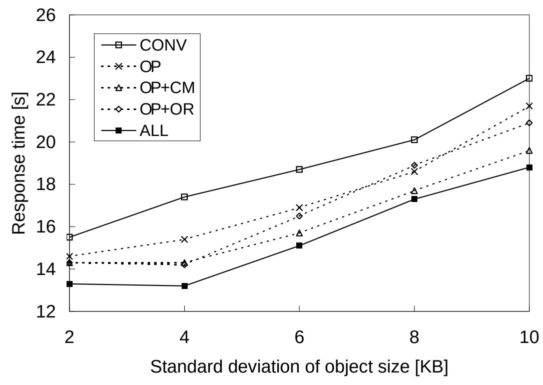
## *5.7 Simulation Results*

In this section, we evaluate the performance of the proposed mechanisms, and compare it with that of conventional Web browsers.

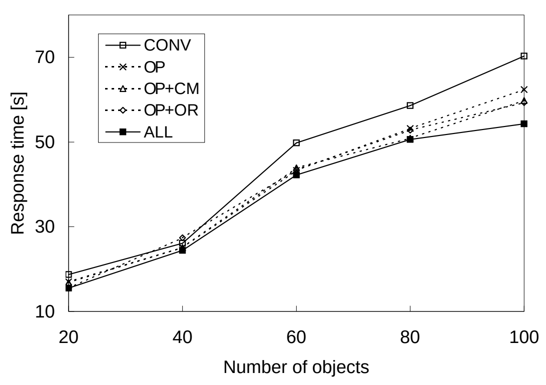
### 5.7.1 Simulation Setup

To evaluate the performances, we use *ns2* simulator [42]. Unless otherwise noted, the network configurations as well as the Web characteristics used in the simulations are the same as described previously. We also use the same network topology as shown in Figure 27 assuming that the local DNS server has all the required domain information.

Response time for the initial screen is used as the primary metric for comparing performances. In this section, we compare five schemes; conventional (CONV), OP only (OP), OP with OR (OP+OR), OP with CM (OP+CM), and all integrated (ALL) schemes. To better explore the impacts of some factors on the performances, we vary some factors in the evaluation. These factors include object characteristics such as sizes and number of objects, numbers of servers involved, numbers of connections opened to a single server, network characteristics such as link bandwidth and *rtt* values, and user's fast scrolling to different screens.



(a) Impact of variance of object size



(b) Impact of number of objects

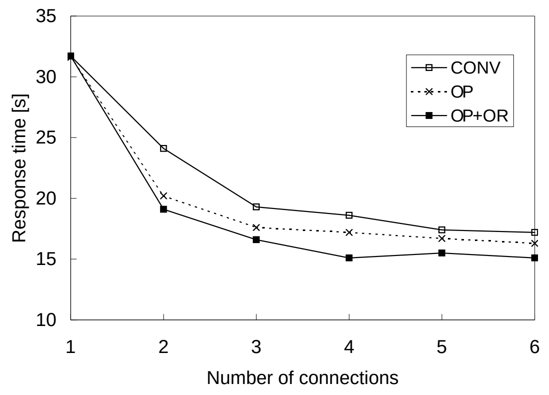
**Figure 38:** Impact of object characteristics.

### 5.7.2 Impact of Object Characteristics

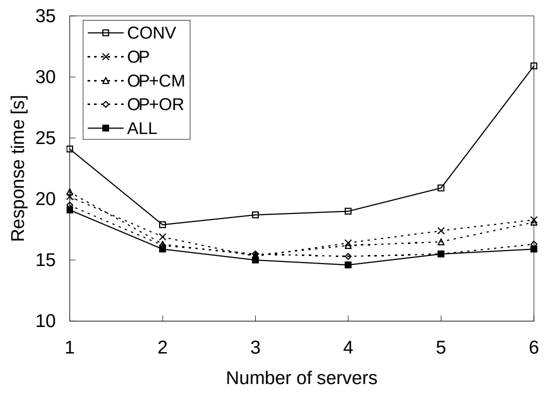
Figure 38 shows the initial response times of conventional browsers and our proposed solution, when the standard deviation of individual object size and the number of objects in the first screen vary. As shown in the figure, when used in combination, the three proposed mechanisms can reduce up to 30% of response time compared to current browsers.

Figure 38(a) shows that as the variance of object size increases, the performance of both the conventional model and our scheme becomes worse. For conventional browsers, the reason is obvious, since larger variance can be translated to reduced bandwidth utilization as described in the Motivation section. Our mechanisms can alleviate this problem, and thus reduce the initial response time. However, since the problem still exists, and becomes more severe when variance of object size increases, the performance degradation is still expected.

Figure 38(b) shows the performance differences between conventional browsers and proposed ones when the total number of objects increases. In the figure, two trends are shown; First, As more objects are included in a Web page, larger response time is expected; Second, the response time reduced by the proposed solution is larger since all of the three mechanisms can gain more benefits.



(a) Impact of number of connections.



(b) Impact of number of servers.

**Figure 39:** Impact of numbers of connections and servers.

### 5.7.3 Impact of Number of Connections and Servers

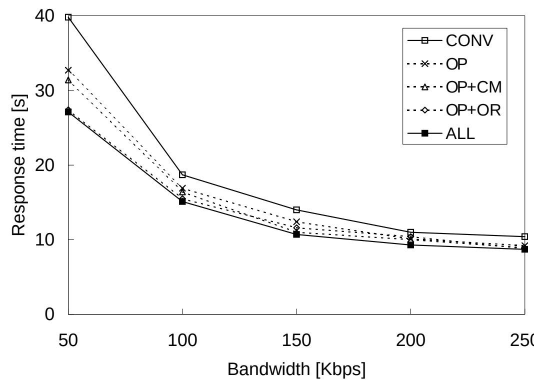
Figure 39(a) shows the impact of number of connections to a single server, and it can be seen that up to 20% of response time can be reduced by using our solution. In the figure, as the number of connections to a single server increases, both the conventional and our solution has smaller response time. However, as this number exceeds four, there are no obvious performance improvements with more connections. This result is consistent with the results presented in other works [9].

Figure 39(b) shows how the initial response time varies as the number of servers for a Web page increases. Two observations can be made from the figure. Increasing number of servers does not necessarily always result in better performance for both conventional browsers and proposed ones. Second, with more servers, our solution can achieve more improvements compared to conventional Web browsers.

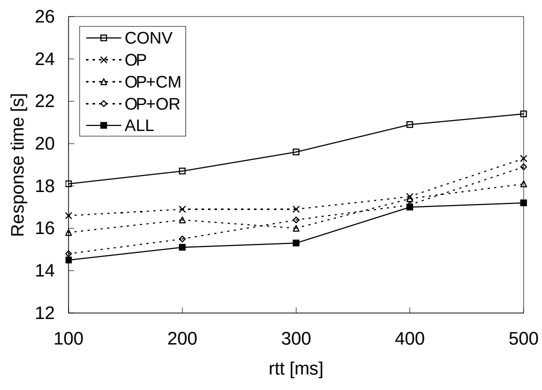
### 5.7.4 Impact of Network Characteristics

Figure 40(a) shows how the initial response time changes under varying bottleneck bandwidth. As shown in the figure, our solution brings more performance improvement for smaller bandwidth. It is because of the fact that smaller bandwidth makes the screen contention problem more severe, and thus our solution can reduce response time more by alleviating this problem.

Figure 40(b) shows how the initial response time is affected by the *rtt* values. Since



(a) Impact of bandwidth



(b) Impact of round-trip time

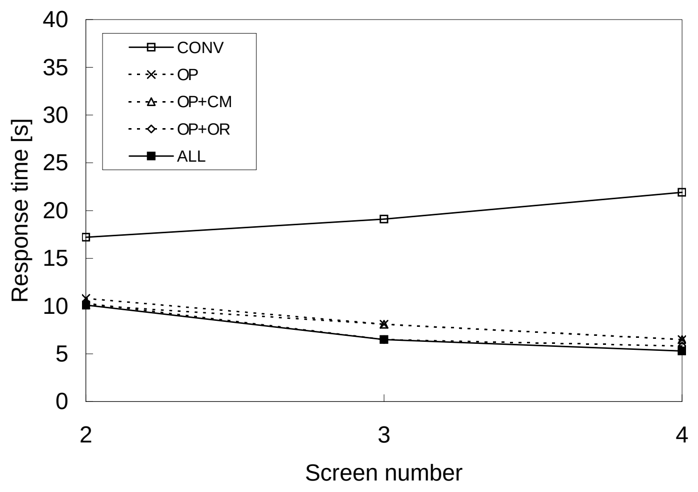


**Figure 41:** Impact of fast scroll.

the major effects of *rtt* come from the request-response behavior of HTTP protocols (*i.e.*, each object is fetched upon the request from the Web client, and thus takes at least one *rtt* to fetch one object) and our solution can alleviate this effect by removing some of these *rtt*s required, our solution sees better performance. As shown in the figure, around 20% performance improvement is achieved by our solution under the *rtt* values considered in the evaluation.

### 5.7.5 Impact of Fast Scroll

Figure 41 shows the response time performance when a user performs fast scrolling. The x-axis of the graph shows the screen to which a user scrolls, and the y-axis is the response time. We assume that scrolling is performed when a Web browser completes downloading of the main HTML document and the entire document layout becomes available.

As seen from the figure, the response time increases when a user scrolls farther away from the initial screen for conventional Web browsers. It is because conventional Web browsers perform greedy fetching without considering the locations of objects on a screen, and thus display of any screen requires downloading of all previous screens. In contrast, our solution has smaller response time as a user scrolls farther away from the initial screen. That is, if a user simply scrolls to the fourth screen, it can experience even smaller response time than any preceding screen! Since OP performs non-sequential fetching and fetches the current screen first, the response time does not depend on the screen number, instead, is determined by the *data size* in the current screen. Consequently, *as less data are located in farther screen (as seen in most popular Web pages), the response time for these screens is less than that for preceding screens.* Thus, we see a 70% reduced response time when the users jump to the fourth screen.

## *5.8 Prototype Implementation*

In this section, we demonstrate the feasibility of the OP mechanism using a simple implementation[6]. In order to prototype the OP mechanism, we designed a plug-in program for Windows operating systems, which allows full control of image delivery and display.

### 5.8.1 Background: Architectural Overview of Web Browser

A Web browser consists of five main components; graphic user interface (GUI), browser control, parsing engine, rendering engine, and memory space called context. Figure 42 provides a high-level overview of a typical browser.

The GUI part of the browser is located at the top level and allows a user to interact with the browser by providing graphical components, including windows, frames, toolbars, and so on. It also initiates or closes a browser instance, and manages system messages delivered by a message queue of the OS.

The browser control performs main functions of the browser; navigation, linking, history

[6]Because of inaccessibility to the source code [47] and high complexity of message queue structures [56], we do not focus on other mechanisms here.

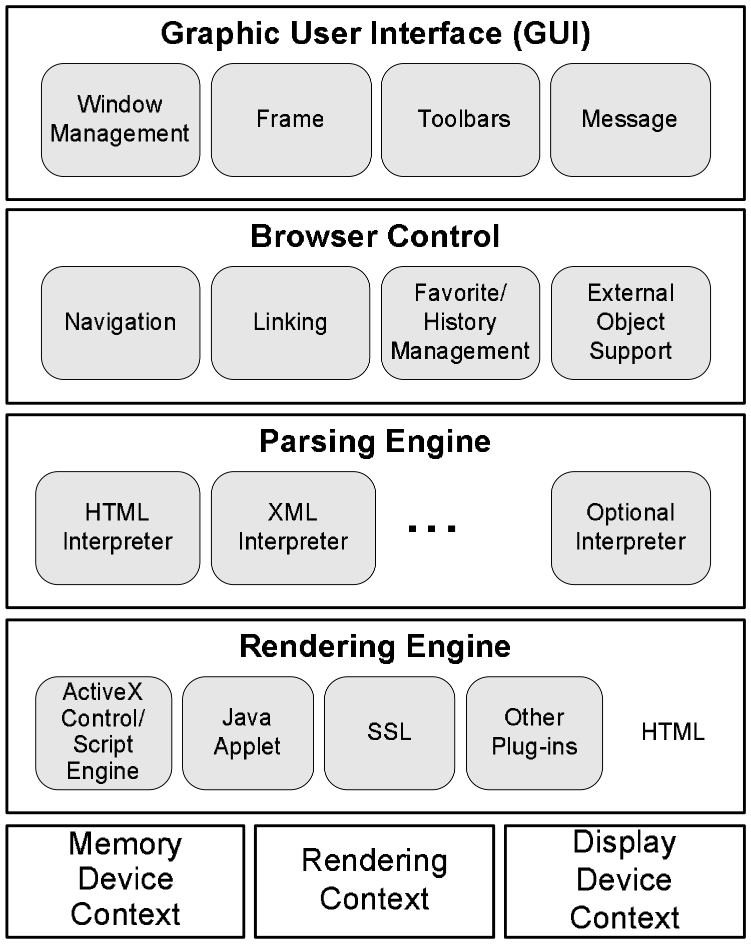


**Figure 42:** Structure of a Typical Web Browser.

and favorite sites management, and support for various other document types such as Word and Acrobat files. The browser control can be overridden by implementing an additional control program without modifications of the source code, and many plug-in softwares are developed in this manner.

The parsing engine interprets HTML documents, fetches embedded objects described in documents, and transforms the original document hierarchy of HTML tags into the display layout structure. It consists of a scanner that reads input streams in real-time, a parser that interprets tags and construct a DOM tree, and a grammar checker that examines the target (mostly HTML) grammar of Web objects.

The rendering (layout) engine displays all graphical Web objects in the client area of a browser window by following formatting information defined by CSS and XSL. Microsoft

Internet Explorer (IE) has a Trident engine (also known as MSHTML) [46] for both parsing and rendering, which also provides a WYSIWYG HTML editing functionality in a manner similar to text editing in Microsoft Word. Mozilla-based browsers such as Firefox and Netscape Navigator use a Gecko rendering engine [57], which is open-source and cross-platform. Opera [68] and Adobe Dreamweaver use a Presto rendering engine. It has a strength in small screen rendering, and many mobile and low-end devices such as Nokia's Symbian-loaded smartphones and Nintendo DS use this browser. Besides, various rendering engines including Ritlabs' Robin, KDE's KHTML, and Apple's WebCore are used in popular Web browsers.

Context is a memory resource that GUI-based operating systems provide to display graphical objects or screen for a specified device or display. Nowadays, many Web browsers follows a dual-context structure, which uses an additional context, called rendering context, which stores partial display information in a browser, to increase rendering speed and reduce resource usage.

### 5.8.2 Prototype Design

Our prototype for object prioritization has been developed as a plug-in program in the WinAPI environment. It can be installed upon Internet Explorer and interacts with the browser control of IE without modification of Windows registry. The prototype performs the following basic operations; on-screen object detection, calculation of object location in layout, and download control. Using this prototype, we will identify the benefit in real-life experiments.

The prototype initially disables the fetching option for all external objects, *i.e.*, *Show pictures* in IE. When a user requests to access a new Web page, it gets the document dispatch from the browser control through *GetDocument*. Then, it gets the *IHTMLDocument2* interface pointer, the element pointer *IHTMLElement* that directs the document body, and the container interface pointer *IHTMLTextContainer* of the body element, sequentially. From the container pointer, it calculates the scrollable area size of the browser window and the current scroll position through *GetHeight/Width()* and *get_scrollTop/Left()*, On the other hand, it requests the image collection *IHTMLElementCollection* to the document interface. Then, it gets the *IDispatch* interface pointer of each image element and the IHTMLImgElement interface from *IDispatch*. Then, it measures the pixel-size and the location of the image element through *get_width/height()*, *get_height()* and *get_offsetTop/Left()*, If the image element has a parent element, it calculates the margin from the parent element and moves the element focus to the parent. After repeating this process until there is no parent any more, it sums all the relative location to calculate the absolute position. Then, it decides whether the element is on-screen or off-screen.

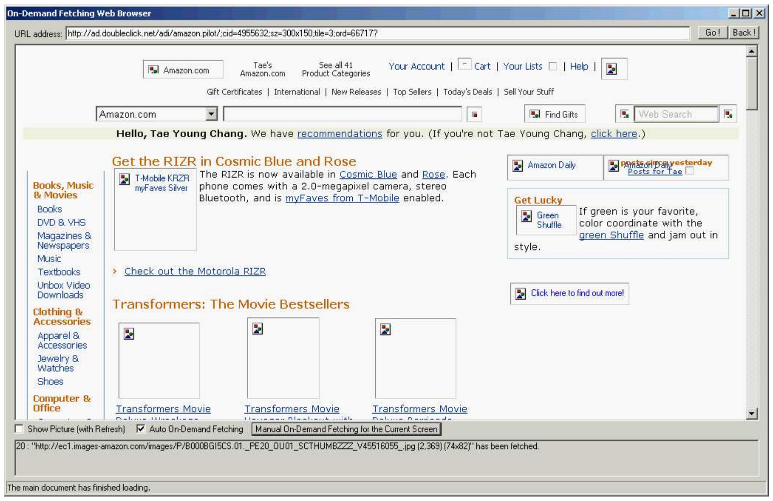
The current version of the prototype performs periodic detection and fetching through *SetTimer()* and *KillTimer()*. If the timing interval is too large, the browser control may not be able to follow user's scrolling speed and update correctly. If it is set as too small, refreshment rate of the client area is too high, and the browser may consume too much system resource by performing unnecessary detection. Currently, we set this value as 1000 ms.

After prioritization, it performs selective delayed transmission for the objects; it fetches high-priority objects immediately and delays low-priority ones until all high-priority objects are completely downloaded. In the current version, we use a simple prioritization scheme that gives on-screen objects high-priority and off-screen ones low-priority. Figure 43 shows the screenshot of browsing amazon.com in the prototype.

### 5.8.3 Performance Results

For the performance evaluation of the prototype, we accessed two popular Web sites; amazon.com and cnn.com. In the experiments, the length of the sites was measured as about 4 and 5 screens, respectively.

Figure 44 shows the transfer sizes for loading the entire pages and the numbers of embedded objects included. In browsing amazon.com, about 15% of the objects in number are non-image, however it counts for a half of the total transfer size in bytes. On the contrary, the off-screen objects in the initial screen account for 30% in transfer size but 50% in number. In accessing cnn.com, this unbalance between the transfer size and the



URL address: http://ad.doubleclick.net/adi/amazon.pilot/;cid=4955632;sz=300x150;tile=3;ord=66717?

Hello, Tae Young Chang. We have recommendations for you. (If you're not Tae Young Chang, click here.)

Get the RIZR in Cosmic Blue and Rose

The RIZR is now available in Cosmic Blue and Rose. Each phone comes with a 2.0-megapixel camera, stereo Bluetooth, and is myFaves from T-Mobile enabled.

› Check out the Motorola RIZR

Get Lucky

If green is your favorite, color coordinate with the green Shuffle and jam out in style.

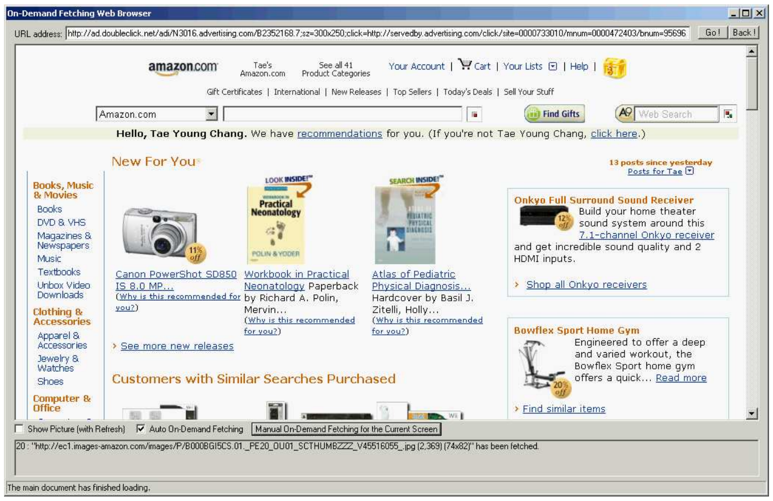
Transformers: The Movie Bestsellers

[ ] Show Picture (with Refresh) [✓] Auto On-Demand Fetching

```
20 : "http://ec1.images-amazon.com/images/P/B000BGI5CS.01._PE20_OU01_SCTHUMBZZZ_V45516055_.jpg (2,369) (74x82)" has been fetched.
```

```
The main document has finished loading.
```

(a) Before fetching embedded objects



URL address: http://ad.doubleclick.net/adi/N3016.advertising.com/B2352168.7;sz=300x250;click=http://servedby.advertising.com/click/site=0000733010/mnum=0000472403/bnum=95696

Hello, Tae Young Chang. We have recommendations for you. (If you're not Tae Young Chang, click here.)

New For You

13 posts since yesterday
Posts for Tae

Canon PowerShot SD850 IS 8.0 MP... (Why is this recommended for you?)

Workbook in Practical Neonatology Paperback by Richard A. Polin, Mervin... (Why is this recommended for you?)

Atlas of Pediatric Physical Diagnosis... Hardcover by Basil J. Zitelli, Holly... (Why is this recommended for you?)

› See more new releases

Onkyo Full Surround Sound Receiver

Build your home theater sound system around this 7.1-channel Onkyo receiver and get incredible sound quality and 2 HDMI inputs.

› Shop all Onkyo receivers

Bowflex Sport Home Gym

Engineered to offer a deep and varied workout, the Bowflex Sport home gym offers a quick... Read more

› Find similar items

Customers with Similar Searches Purchased

[ ] Show Picture (with Refresh) [✓] Auto On-Demand Fetching

```
20 : "http://ec1.images-amazon.com/images/P/B000BGI5CS.01._PE20_OU01_SCTHUMBZZZ_V45516055_.jpg (2,369) (74x82)" has been fetched.
```

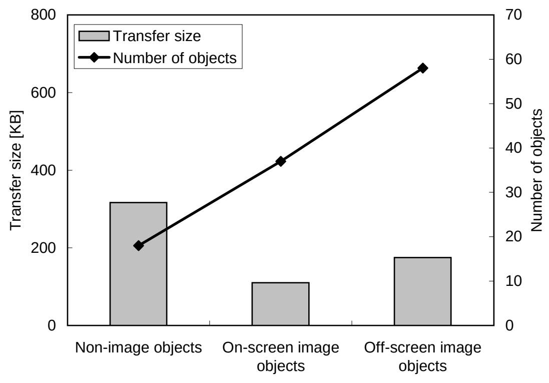
```
The main document has finished loading.
```

(b) After fetching all on-screen objects

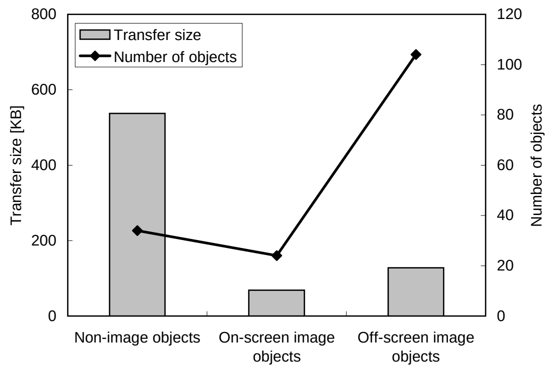
**Figure 43:** Screenshot of Browsing amazon.com in Prototype.

number of objects becomes more severe. In Figure 44(b), the number of the off-screen objects amounts to more than 60%, whereas their aggregated transfer size takes only 17%.

Figure 45 describes the initial and full response time performance for the Web pages. In the figure, the darker area shows the additional delay to fetch the entire objects after displaying the initial screen completely. When browsing amazon.com, the prototype reduces the initial response time by 5 secs; the greedy fetching browser wastes 5 secs to download off-screen objects before the initial screen is fully displayed. In the case of cnn.com, the performance benefit increases to 40%. In the figure, it can be seen that the performance

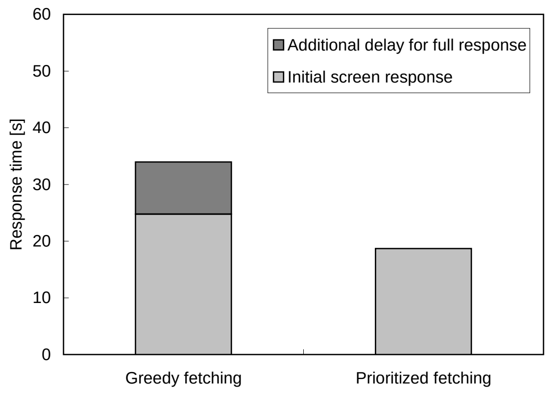


(a) amazon.com

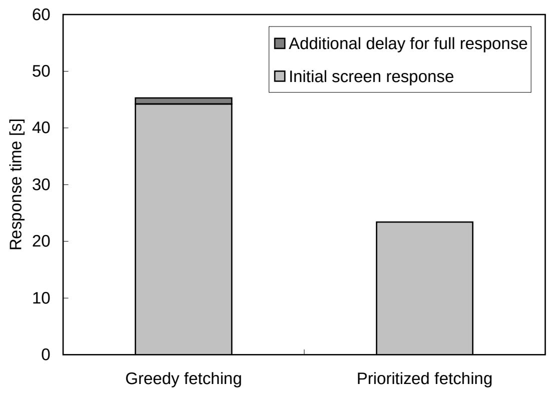


(b) cnn.com

**Figure 44:** Transfer Size and Number of Objects in Prototype.



(a) amazon.com



(b) cnn.com

**Figure 45:** Response Time in Prototype.

degradation caused by screen contention is affected not only by the transfer size of the off-screen objects but also by their numbers.

## *5.9 Summary*

In this chapter, we first explore the reasons why conventional access models for partitionable content are not appropriate for mobile hosts. We identify two reasons, screen contentions and bandwidth underutilization, which result in large user-perceived response time. To address this problem, we propose a new access scheme, which uses an intelligent mix of object prioritization, object reordering, and connection management. Using simulations with the Web parameters obtained from the Top 50 Web Sites, we evaluated the performance of our scheme and proved its benefits over conventional access models. We also implemented a simple prototype of the object prioritization algorithm and measured the performance benefit in real-life experiments.

# CHAPTER VI

# USER-ACTIVITY SHIPPING FOR FILE SYNCHRONIZATION

## *6.1 Introduction*

Distributed file systems allow for transparent file sharing across networks, either between multiple users or between multiple computing devices belonging to the same user. In either type of file sharing, file synchronization refers to the process of synchronizing a file at the file server with the changed copy of the file at a client, after an update session. Especially when the clients are connected to the file server through low-bandwidth links such as in a wireless environment, the bandwidth usage efficiency of the file synchronization scheme in a distributed file system is more important.

An intuitive bandwidth-efficient strategy of file synchronization for such environments is one where only the *binary differences* between the original and updated files are sent across to the server and the original file at the server is thus patched with the differences. The performance of such an approach is obviously better than that of a full-file transfer in conventional file synchronization schemes. *Low-bandwidth network file system (LBFS)*[58] and *Prayer file system (PFS)*[19] are examples of a file system that uses such a strategy. However, under commonly occurring conditions, these approaches still incur considerably larger overhead than essential to perform file synchronization.

Therefore, in [43], the authors propose an *operation shipping* strategy. In this mechanism, the mobile client logs high-level user operations and ships them across to the server. The surrogate then re-executes the operations upon the old copy of the file to re-generate the corresponding updated files. The authors demonstrate that the transfer size for the commands is typically much smaller than that in conventional file systems, and hence motivate a new paradigm for file synchronization. However, they focus only on interpreted user operations such as shell commands in *Unix* or application-specific operations, not uninterpreted raw inputs such as keystrokes and mouse-clicks. Thus, operation shipping has a limitation that it can be applied only to shell-based applications or applications that are modified to perform the interpretation of user operations in an application-aware way.

In this paper, we present a new file synchronization approach that performs raw user-input shipping for unmodified common desktop applications, which interact with operating systems by exchanging messages in graphical user interface environments. The approach we present *records* raw user-activity such as keystrokes and mouse-clicks at the client, *replays* the shipped record of the raw activities at the server, and *verifies* whether the regenerated file is identical to the updated file. We refer to this approach as *Mimic* since the strategy is to *mimic* the client-side user activity on the server. Mimic is not a complete file system, but is meant to be an add-on to an underlying file system such as LBFS[58] or Coda[82], to reduce transfer sizes during file synchronization.

We use a simple prototype to show the considerable bandwidth usage benefits it can provide when used appropriately. Using the prototype with a group of applications, we also identify specific conditions under which raw activity shipping does not provide better performance than the differential update scheme, and hence present an integration strategy that involves both update mechanisms working in a loosely coupled fashion.

## *6.2 Motivation*

The objective of this work is *to exploit conditions under which the encoded user activity for a file update is much smaller in size than the changes to the file itself.* Mimic is fully application-unaware, does not require any changes other than to the file system itself, and applies to interactive applications. Mimic achieves its applications-unaware property by shipping raw activity such as key-strokes and mouse-clicks as opposed to application-aware cases.

Now, we present the key factors that contribute to the motivation in the specific context of interactive applications. The identification of the factors serves to highlight the commonality of the types of file updates, for which raw activity shipping will deliver better performance than differential update (*diff*). We classify file updates for which activity shipping will deliver better performance into two types: (i) updates where small amounts of user-activity results in large amounts of content being *added* to the file; and (ii) updates where small amounts of user-activity results in large *changes* to the content of the file without necessarily increasing the size of the file itself. In the rest of the section, we use experimental results with a graphical document processing applications, namely Sun Microsystems' OpenOffice.org writer, to highlight the inefficiency of the differential update scheme for the above classes of updates, and therein motivate the need for a better file synchronization mechanism.

- The first class of updates is relatively straightforward to understand. An example of an update that results in a large amount of content being added to files with minimal user-activity is a *copy-and-paste* operation. In the experiment, we copy a paragraph and paste it onto another location in a 54-KB Word-format document using keystrokes and mouse-operations. In Table 3, the binary difference between the updated and original files amounts to approximately 1 KB. However, the activity itself, in terms of mouse-clicks and locations, when encoded amounts to only 0.084 KB.

- The second class of updates is relatively more non-intuitive, particularly in the context of interactive applications. In Table 3, for the activity *insert-a-line* by keystrokes, a single line of text is inserted into a 54-KB document. The additional textual content added is approximately 87 characters. However, the *diff* amounts to about 1.4 KB of data. This phenomenon is also observed for the activity *insert-a-paragraph* by keystrokes, where a 5-line paragraph is typed into the document. While the textual content added is approximately 400 characters, the *diff* amounts to approximately 2 KB. The interesting observation to be made in the above results is that while the file size or textual content itself does not change dramatically, there is considerable change in the binary representation of the file despite the minimal user-activity.

- The reasons behind the observation is the complex file structures used by most interactive graphical applications to provide the best interface and services to the user. In the specific case of the Microsoft Word document format, the application maintains each document in the form of *six* different streams [50]. The *main* stream contains

**Table 3:** Comparison between *diff* and activity size.

| **seq.** | **Activity description** | **User activity** | **File size** | ***diff* size** | **Message size** |
|---|---|---|---|---|---|
| 0 | ( initial state ) | | 54272 B | | |
| 1 | Insert a line | 87 keystrokes | 54784 B | 1365 B | 204 B |
| 2 | Insert a paragraph | 432 keystrokes | 55296 B | 1719 B | 768 B |
| 3 | Copy and paste a paragraph internally | 6 keystrokes + 11 mouseclicks | 55808 B | 1021 B | 84 B |
| 4 | Change the font type of a paragraph | 7 mouseclicks | 57344 B | 920 B | 48 B |

the document header and all textual information in the document, while the *data* stream contains information about all non-OLE objects (such as figures and tables) in the document, including their physical addresses in the binary representation of the file. The *object* stream similarly maintains information about all OLE objects (such as imported figures and spreadsheets). A *table* stream maintains the structure of the document itself, including the locations of the different objects in the visual representation of the document, and the relative dependencies between the objects (e.g. a figure and wrap-around text). Finally, two *summary* information streams maintain miscellaneous information about the document including timestamps.

In this complex representation of a Word document, a single line of text insertion thus has the following impact: The main stream is updated to contain the modified textual information. In addition, the data and object streams are updated to reflect the new addresses of the different objects in the binary document file, with their positions altered due to the insertion of the line of text in the main stream. Moreover, the table stream also is updated to reflect all changes to the visual representation of the document in terms of the relative and absolute locations of the different objects in the document. Finally, any resulting changes in other attributes of the file including timestamps results in updates to the summary streams. Such a multi-fold update of the binary representation of the document file, when a single line of text is inserted, accounts for the substantial size in the *diff* despite the minimal user-activity. Note that such complex, and application tailored, binary representations of content is not specific to only Word, and is true for most interactive applications today. In Section 6.6, we consider other applications.

Finally, in Table 3, for the activity *change-font-type*, a paragraph is highlighted, and a font change is effected exclusively through mouse operations. It can be observed that for such "meta" operations, the size of the activity itself is much smaller than that of the *diff*. The reasons are again similar to those described earlier in terms of updates to multiple streams within the file structure.

- In summary, updates for which small magnitudes of user-activity result in large additions or changes to the file's content render activity shipping an attractive option for file synchronization.

In the next three sections, we present a raw activity shipping strategy for file synchronization called Mimic. Mimic is specifically targeted toward reducing transfer sizes for files updated using interactive applications. The goal of the Mimic design is to achieve such reduction in transfer sizes while being fully application-unaware, and requiring no changes to applications.

## *6.3 Architecture of Mimic*

In this section, we present a high level overview of Mimic architecture as well as key optimization mechanisms, and describe how it loosely integrates with the underlying file system.

### 6.3.1 Overview of Mimic

Mimic is a fully application-unaware file-synchronization strategy that consists of components at the client and the server respectively. At a high level, the Mimic client captures input activity messages and converts them into a bandwidth-efficient representation on a per-file basis. During file synchronization, the client generates the fingerprint and error correction code for file verification and ships the activity record to the server The Mimic server

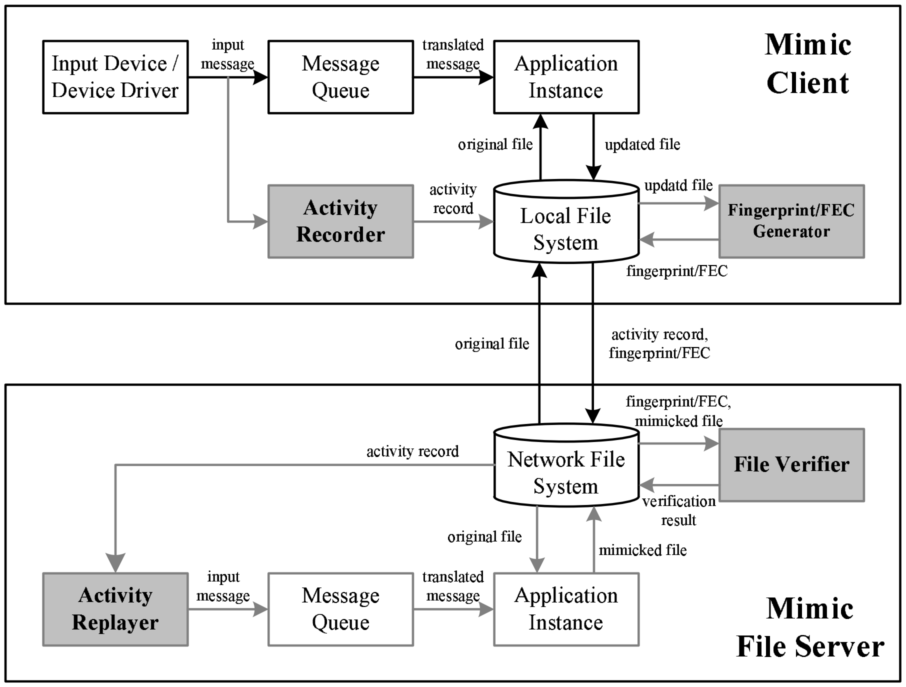


**Figure 46:** Overview of Mimic architecture.

then replays the recorded activities to regenerate the updated files at the client, verifies the regenerated file by using the fingerprint, and corrects errors if they exist. The realization of the above functionalities in Mimic are done with the goals of reducing the transfer file size, minimizing latencies involved, and incurring minimal overheads in terms of usage of system resources. Figure 46 shows the overview of the Mimic architecture.

Briefly, the primary components of the Mimic approach are:

- *Record:* This component is responsible for the effective capturing of raw activity at the client end, classifying the activity, and maintaining per-shared-file records.
- *Playback:* This component is responsible for replaying the activity records in the fastest manner possible while ensuring correctness.
- *Verify:* Finally, this component is responsible for verifying whether the replay based re-creation of an updated file is correct. This component includes both forward error correction (FEC) to correct non-repeatable actions such as timestamps, and detecting any errors that arise due to improper playback. Mimic uses a verification scheme [77]

that is identical to the one presented in [43].

We elaborate on the realization of the above three mechanisms in detail in the next two sections.

### 6.3.2 Message Optimization

Recently, various types of user input devices such as digitizer, touch screens, and microphones for voice recognition are used that are not limited to a keyboard or mouse. However, the majority of them are used only in limited device-specific applications, and most common GUI-based applications use only keyboard and mouse input. Thus, we focus on the input event messages associates with only the keyboard and mouse.

GUI-based interactive applications are event-driven in that they do not generate explicit function calls to obtain input such as keystroke and mouse-click. Instead, they wait for the system to pass input to them. All input is delivered in the form of OS-specific messages to the corresponding application windows in the application. In order to route these messages to an appropriate window procedure, the system posts them to a common system message queue. Then, in a first-in-first-out (FIFO) fashion, the system de-multiplexes the messages based on the handles of the destination windows, and posts them to the message queues of the threads that created the destination windows.

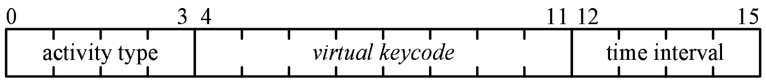
In such a GUI-based environment, not all user input has correlation with the file update. For example, mouse movement without button pressing is redundant in the view of the file state. In order to remove these redundant user-activity messages, the message record hook in Mimic classifies messages according to input activity types, and captures only useful messages for updating the application state.

However, even though a useful message is hooked, its message structure can still contain redundant information and might not be optimized for bandwidth efficiency. Figure 47 shows the structure of the hardware event message in Microsoft Windows operating systems, called EVENTMSG[50]. In this structure, each member is 4-byte long, and hence the total size of each message is 20 bytes.

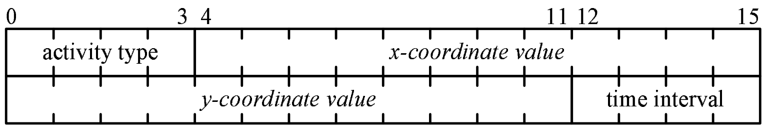
When a keystroke event is triggered, the virtual key-code of the key that was pressed

```
typedef struct {
        UINT message;     // message type
        UINT paramL;      // message content
        UINT paramH;      // message content
        DWORD time;       // posting time [ms]
        HWND hwnd;        // destination window handle
} EVENTMSG, *PEVENTMSG;
```

**Figure 47:** EVENTMSG structure in Microsoft Windows operating systems.



(a) Keyboard activity.



(b) Mouse activity.

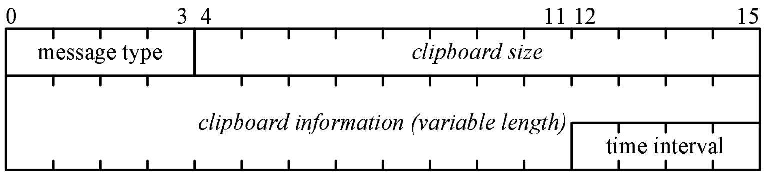
**Figure 48:** Activity descriptor (AD) structures in Mimic.

or released is stored in the high byte of the low word of the *paramL* member, and its other bits are not used. In addition, all other redundant information such as the scancode and repeat count is stored in the *paramH* member. The *hwnd* member is also unnecessary since the keyboard focus[1] designates the target window instead. In most cases, the delivery is performed in a short time, and the *time* member is not much informative. Thus, only the *message* member and the low word of the *paramL* member are essential to deliver a single keystroke event, and all other parts are not necessary.

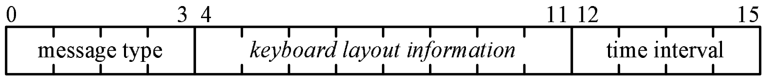
If a mouse event happens, the x- and y-coordinate values of the mouse cursor location are stored in the *paramL* and *paramH* respectively. However, the vertical and horizontal resolution of the display output usually does not exceed $2^{12}$=4096 pixels, and as a result the space of 2.5 bytes in each member are wasted. The timestamp and window handle information is also not necessary due to the same reasons.

In Mimic, the client converts a useful input event messages into a bandwidth-inefficient

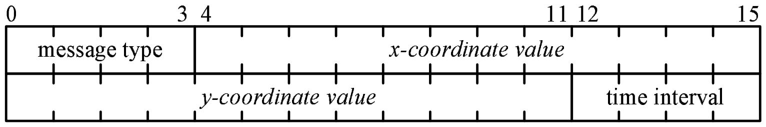
[1]In order to deliver user input to a correct application window, GUI-based operating systems maintain keyboard and mouse focuses. Usually, the keyboard focus is on the top window that is currently active, and the mouse focus is on the window under the current location of the cursor.



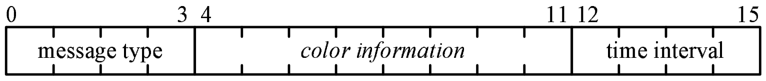
(a) Clipboard content change.



(b) Keyboard layout change.



(c) Screen resolution change.



(d) Color depth change.

**Figure 49:** Meta descriptor (MD) structures in Mimic.

format called an *activity descriptor (AD)*, The AD consists of three members; *activity type*, *activity value*, and *time interval* with the previous message arrival. Since only useful input events[2] are focused in Mimic, the total number of the target activity events can be less than $2^4$=16. Figure 48 shows the structures of the ADs for keyboard and mouse input event. The length of the AD varies depending on the activity type. The *activity value* member in an keyboard AD that describes a keystroke event includes a 8-bit virtual keycode, whereas that in a mouse AD includes two 12-bit coordinate values.

In a GUI-based application, the same input message can be interpreted differently according to the current environmental setting. For example, when a user pushes the key 'A' on the keyboard, the actual interpretation can be totally different in another language after the user changes the keyboard language layout outside the application. Therefore, the Mimic client captures all the changes of environmental setting while recording user activity.

In Mimic, the client also captures the messages delivering the information of the system configuration change such as clipboard content, keyboard layout, screen resolution, and color

[2]In Microsoft Windows, WM_KEY*, WM_SYSKEY*, WM_LBUTTON*, WM_RBUTTON*, and WM_MOUSEWHEEL messages are included.

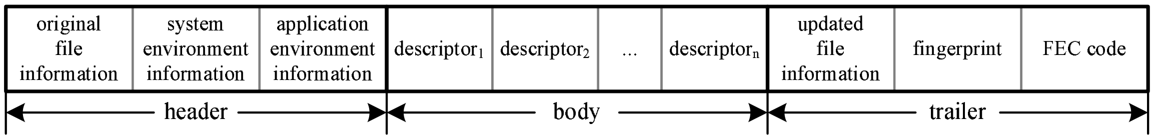


**Figure 50:** Activity record (AR) structure in Mimic.

depth. Then, it generates a structure called as a *meta descriptor*. Figure 49 illustrates the structure of the various meta descriptors.

Once the Mimic client begins to operate with a file, it sequentially records activity and meta descriptors in the body of the per-file record structure called an *activity record (AR)*. Figure 50 shows the structures of the AR. We defer the detailed description about the header and trailer of the structure to the next section.

### 6.3.3 Integration with File System

Mimic requires interfacing with both the underlying file system and window manager at the client, and with the window manager at the server end. The interfacing with the window manager, however, does not require any changes to the operating system, and instead is achievable through standard interfaces that most window managers export. Figure 51 shows the interfaces between the file system and the window manager.

Mimic does not require any interfacing with the file system at the server. We refer to the coupling as *loose* because Mimic currently relies only on *informative* callbacks from the file system that is essential for its operations, and requires minimal changes to the file system design and logic.

The interface between Mimic and the underlying file system consists of six function calls (*open()*, *close()*, *rename()*, *delete()*, *finish()*, and *synch()*), all exported by Mimic, and invoked by the file system. The first five functions are *informative* in nature, and require no logic change inside the file system other than their mere invocation. The sixth function is used by the file system to preferentially use Mimic for the synchronization process, but falls back to its native synchronization mechanism if Mimic indicates a failure in its synchronization attempt. We now present the interface in two stages, based upon which aspect of Mimic are the functions relevant.

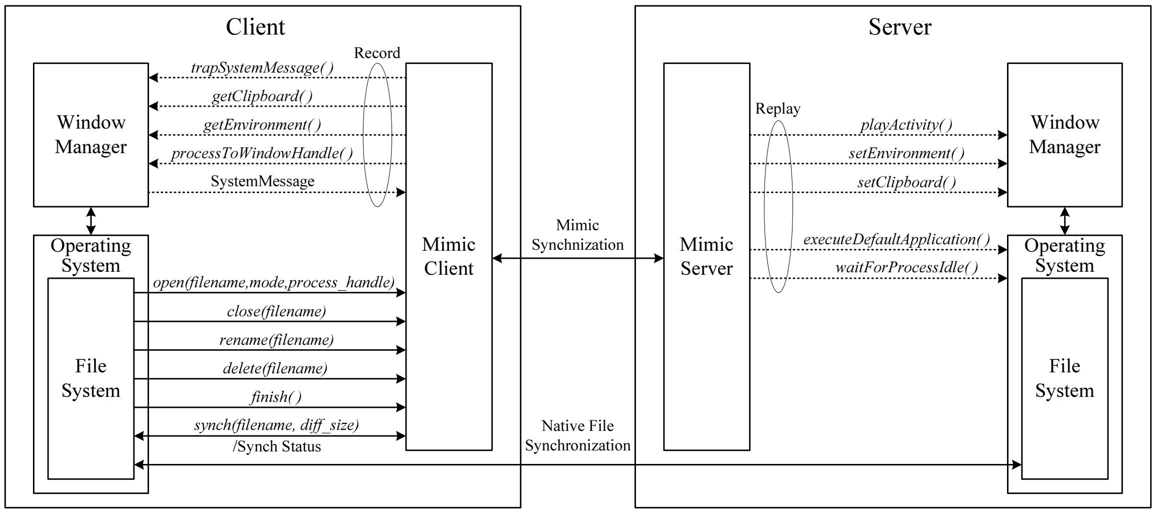


**Figure 51:** Mimic-file system-window manager interfaces.

- The first set of functions which are part of the interface help in the recording component, and consist of *open( )*, *close( )*, *rename( )*, and *delete( )*. All four functions are purely informative in nature, and are used by the file system to *inform* Mimic about the corresponding actions. The *open( )* function is used by the file system to inform about the opening of a *shared* file, and its parameters consist of the filename, opening-mode (read, write, or append), and the process-handle for the process that is performing the open. Mimic uses this information to initialize its recording activity pertaining to that file. In Section 6.4, we explain the filtering process that enables the Mimic client to not maintain records for files not being updated interactively, opened in read-only mode, etc. The *close( )*, *rename( )*, and *delete( )* functions all consist of the filename as the parameter (*rename( )* has both the old and new file names), and are used by Mimic to update the activity records corresponding to the file being closed, renamed, or deleted.

- The second set of functions that Mimic requires the file system to use consists of the *finish( )* and *synch( )* functions, which are used during the actual file system initiated synchronization process. For every file that needs to be synchronized, the file system preferentially calls the *synch( )* function with the filename, and the *diff* size as parameters. The return value for the *synch( )* call indicates to the file system whether or not the raw activity shipping process was successful. Only in the event of a failure does the file system initiate its native synchronization process. We elaborate on the specific conditions under which Mimic will return an error in Section 6.5. Finally, the *finish( )* function is used by the file system to indicate the termination of the synchronization process, after which simple clean-up operations are performed.

Due to the purely informative nature of five of the functions (except *synch( )*), no changes are required in the file system logic when the functions are called. The functions are merely invoked by the file system when it is performing an open, close, rename, delete, and completion of synchronization respectively. The only change in the logic required when the *synch( )* function is invoked is a check for the return value of the function call, and conditionally invoking the native synchronization process.

## *6.4 Client Operations in Mimic*

The Mimic client performs activity logging and shipping through three phases. In this section, we describe the details of operations performed in each phase.

### 6.4.1 Decision of Mimic Operation

When a shared file is opened through a file system call, the file system invokes the *open( )* function. When the Mimic client receives the call, it investigates the attribute of the target file. If it is read-only or the client does not have a permission of write-access on it, the access is performed without Mimic operation.

Even if a file is accessed in write mode, the application associated with the file extension may not have an ability to update the file. Graphic viewers and multimedia players are such examples. Mimic is also not adoptable when the corresponding application opening the file is not able to regenerate the file at a server side by raw activity shipping. For example, the applications associated with the file extension at the client and the server may not be identical. In order to provide information of the default access modes and file regeneratability of applications, both the client and server maintain *a table of the applicable file extension.* Based on this application information table, the client can determine in which mode it perform the access for a file with write permission.

### 6.4.2 AR Header Generation

Once Mimic begins to operate with a file, it generates an AR. If the corresponding AR is already opened, Mimic infers an error and aborts maintenance of the AR for the file. For the file identification in the future playback process, the client writes logical name structure information of the original file including filename, extension, location, attribute, and size in the header of the AR. In the case that there is no notion of global name hierarchy in specific file systems such as Sun Network File System, it records the structure in the view of the server.

System environmental synchronization between the client and the server is indispensable to accurate raw activity shipping. For later environmental synchronization at the server, the Mimic client captures the state parameters of the initial system environment in the header of the AR. Through a *getEnvironment( )* call, the client catches keyboard information including language layout and key state such as *Caps Lock*, and screen information including resolution and color depth. The initial content of clipboard buffer is captured through a *getClipboard( )* system call, and hence the length of the AR header is varied according to the clipboard content size.

In most GUI-based applications, multiple instances of a single application may share a single main process or thread to utilize system resource efficiently. A thread associated with a file usually maintains a main application window including the client area and multiple side windows including the non-client area such as menu bars and pop-up windows. Through a *processToWindowHandles( )* call to the window manager, the Mimic client collects the mapping information between the process/thread handle and the corresponding set of window handles, and then registers it on a table called *window-handle table* with the corresponding filename. Multiple window-handles could potentially map onto a single filename due to the complex windowing structures used by interactive applications today. After the mappings are put on the table, the Mimic client becomes able to sort the user input that affects the state of the application instance opening the file.

Simultaneously, the Mimic client detects the execution environment of the application being executed, such as main window size/location and current language setting, and writes them in the header of the AR. After this initial detection, the Mimic clients do not detect the change of the application configuration any more. Instead, all the changes are reflected automatically by recording and replaying the user activity, because all the corresponding input messages that finally change the environment also have the destination of the window handles belonging to the application instance.

If the Mimic client is unable to retrieve the window handles because the file is not being processed with an interactive application, the Mimic client simply ignores the *open()* call. This *implicitly* ensures that Mimic does not handle such files. Note that any updates to such files will be handled by the native file synchronization strategy of the underlying file system.

### 6.4.3 Message Capture

After the execution of the application, the Mimic client installs an *input message hook* procedure that captures input system messages delivered to the corresponding windows of the application having help from the window handle table. After that, it also creates both a *input message monitor* that selects only useful messages for the file update and record them in the body of the AR.

Basically, the input messages that perform environmental change has a window handle value of the main system window. However, only the minority of the messages focusing on the main window is correlated with the environmental change, and even installing a input message record hook on the main system window makes the operating system have groan under a heavy burden. Therefore, the Mimic client installs a different type of message record hook called *an internal message record hook*, which captures internal system messages delivered to all application windows other than raw input messages. Then, it creates a *internal message monitor* that records the changes of system environment parameters in the AR body.

In Mimic, the client receives all dequeued input event and internal system messages from the queue from the message record hooks through the *trapSystemMessage( )* interface, and pass them to the corresponding message monitors. When the input message monitor receives a input message, it looks up the corresponding filename in the window-handle table. If successful in the lookup, the monitor appends the message in the AD structure into the body of the corresponding record sequentially. If the window-handle of the input message is not found in the table, the Mimic client skips the record phase, and directly queues the message onto the corresponding thread-queue. On the other hand, the internal message monitor captures only the messages delivering the information of the system configuration change.

Among internal messages, there exists a special type of message that needs to be handled uniquely in Mimic is a *paste* message, as the message requires to be capture along with the corresponding *clipboard* information. Thus, the internal message monitor, upon detecting a system paste message, obtains the clipboard information through the *getClipBoard( )* window manager function call, and forwards the content to the message monitor, which then appends it to the file activity record. Although Mimic handles clipboard-based paste operations in the above manner, in Section 6.6, we show that raw activity shipping is efficient only for those paste operations for which the content is copied or cut from the same file, and thus the clipboard content is not essential.

### 6.4.4 Termination

Mimic supports a disconnected data access, which means that after a connection to the server is established for file access the connection may not remains open until the application is closed. For the best performance in disconnected operations, Mimic performs the *write-on-close* cache update, which writes data back to the server only when the file is closed. This update policy also has an advantage that a server manages system resource efficiently by reducing the frequency of playback.

During the termination phase, the client removes the dedicated message record hook. Then, it collects information necessary for file verification at the server, and writes the verification information on the trailer of the AR. For the first part of the trailer, the client calculates the size of the updated file. Then, it generates the fingerprint of the file using the Message-Digest 5 (MD-5) algorithm with a 128-bit keyword [79] and the forward error correction (FEC) code using the Reed-Solomon (RS) coding [77].

When the Mimic client receives a *synch()* call from the file system, it goes through a two stage check to see if it can proceed with the synchronization. It first checks to see if a AR has been maintained for the file. It then checks to see if the size of the AR is less than that of the *diff* the file system will have to send as part of the native synchronization process. If the answer to either of the two checks is negative, the Mimic client returns a *SYNCH_FAIL* message back to the file system. The file system will then proceed with its native synchronization strategy. If both checks are successful, the Mimic client ships the corresponding AR to the server.. When the Mimic client attempts the synchronization, it forwards the result of the server side verification process as the return value for the *synch()* call.

### 6.4.5 Bookkeeping

The *rename()*, *close()*, *delete()*, *finish()* interface function calls invoked by the file system triggers appropriate bookkeeping operations in Mimic. For the *close()* and *delete()* calls, the actions involve closing and deleting the corresponding ARs respectively. The *rename()* function call, however, is handled differently. If the rename is for a file created since the last synchronization, Mimic renames the filename in the corresponding AR. However, if the rename is for an already existing file, Mimic disables the AR maintenance for the file, and allows the native file synchronization process to handle the file. Note that the AR is a record of the *incremental* activity performed since the file was opened. Hence, if a AR is maintained for a file $f_a$, and $f_a$ is later renamed to $f_b$, the server *cannot* recreate the updated $f_b$ by replaying the AR for $f_a$ on the original copy of $f_b$.

Finally, when the *finish()* call is received from the file system signifying the end of a synchronization session, the Mimic client cleans up its data structures by deleting all the ARs, and clearing up its window-handle table.

## *6.5 Server Operations in Mimic*

Each playback at the Mimic server requires the AR corresponding to the file being synchronized since the last synchronization.

### 6.5.1 Decision of Playback Operation

The first operation that the Mimic server performs is to read the file information of the AR header and checks whether it can perform the playback without any problem. First of all, the target file for updating should be available. If the file does not exist, it stops the whole process and notifies the client the failure of the activity shipping. If it is being accessed by another client, it follows the concurrent update policy of the server. When the server concludes that the target file is ready, it calculates the required resource for the playback and checks its system resource status. If enough resource is not available, it enqueues the invoked playback process in a job queue and waits for the turn. After all the requirements are satisfied, it generates an independent virtual session, in which it will update the file.

### 6.5.2 Initialization

When the virtual session becomes ready, the Mimic server synchronizes the initial system environment. In order to regenerate the identical system configuration, it reads the record header and decodes the system environment information. Each recorded system environment parameter is converted into a corresponding system message[3] that sets up the individual system setting. Then, it sets up the current system setting through a *setEnvironment* function and verifies whether the settings are identical to the information through a *getEnvironment* function.

Once the environment is set, the Mimic server reads the application environmental setting information from the header and invokes the interactive application corresponding to the file extension through the *executeDefaultApplication()* interface to the operating system with the identical setting specified in the header. After the application instance is generated, the server moves its main window on top of the desktop and shifts all the input

[3]In Microsoft Windows, WM_*CHANGE and WM_*CLIPBOARD are such system messages

focuses to the main window.

Because each file associated with the record is opened in an independent virtual session, the Mimic server does not use the window-hand table that is used in the client. In other words, there exist no other application instances or their windows, and thus all input system messages are delivered to the application instance automatically.

At the final stage of the initialization process, the Mimic server executes two *descriptor-to-message converters*; an activity descriptor converter and a meta descriptor converter. Then, it installs an *input message playback hook* procedure that inserts converted input system messages to the system message queue, and an *internal message playback hook* that performs system environment update by delivering regenerated internal system message.

### 6.5.3 Activity Playback

In the replaying phase, the mimic server reads descriptors sequentially one at time from the record body. When each descriptor is read, the server passes it to the corresponding converter and inserts the converted system message into the system message queue by the corresponding message playback hook through the *playActivity()* interface of the window manager.

A critical issue with the performance of the raw activity shipping process is the replay time at the server. At worst, the replay time for a file can be even much longer than the real time taken by the user to update the file at the client, and obviously this is undesirable. On the other hand, overspeed playback can result in *mis-interpretation* of user activity in the replay process. Inputs to an application can be thought of to change the *state* of the application. Therefore, certain inputs might be relevant to the application only for particular states, or might be interpreted differently for different states[4].

Consider an example with the following sequence of user-activities: A user pushes the left mouse button on the font size menu in a Word application window, and then pushes the button on the number of the desired font size. The *correct* application state between those two inputs is "waiting for a new input with the font-size pop-up window open". However, if

[4]In Excel, before each SEND.KEYS macro function is performed, building a small delay using the WAIT function is recommended due to the same reason[50].

the playback of the second input is performed before the application thread responds fully to the first message by completing the update of the application state, the replayed activity of the second input is regarded as a meaningless mouse-click on a vacant space in the menu bar and ignored. Then, the remaining all other activities will not be executed correctly.

Thus, the challenge is to replay the records as fast as possible without introducing errors due to activity mis-interpretation. Our strategy addresses this challenge by explicitly monitoring the process state of a process control block after every message playback through the *waitForProcessIdle( )* interface to the operating system. Only when the relevant process is idle does the Mimic server playback the next AD to minimize the overall replay time. In Section 6.6 we show how the latency performance is thus quite reasonable given the bandwidth usage benefits.

On the other hand, Mimic supports *sequential multiple updates* on a single file, especially for the disconnected operation. Let's assume that a user performs several updates on the same file while disconnected. When the client is reconnected, the server receives multiple update profiles for the same file at once. In such a case, based on the timestamps of the file information, the server decides the order of those updates and performs the updates sequentially. And, only after file verification for the previous update is finished, the server can begin to perform the next update upon the newly updated file.

### 6.5.4 File Verification

After all descriptors in the record are played back, the corresponding application process is terminated, and the verification process is begun. The verification process consists of three phases; file size check, fingerprinting, and forward error correction (FEC). Since this verification process is identical to that in [43], and hence we summarize the process in brief.

The first step is to check the size of the updated file. Since file sizes can be expected to be random with respect to user input, it is quite improbable for two different files to have the same size, and most errors can be expected to be detected at this stage.

In the second step, the Mimic server compares the fingerprint of the newly updated file at the server with the fingerprint sent by the client through the AR. The collision probability for the MD-5 algorithm with a 128-bit keyword is $O(2^{-64})$[79], which is smaller than the collision probability of the CRC used in TCP headers[87].

If the fingerprints are not identical, the Mimic server performs a FEC process using the Reed-Solomon (RS) coding[77], which has advantages of minimal overhead and effective burst error correction. In order to maximize the error correction speed, the original and generated files are divided into multiple blocks with the same size and their last blocks are zero-padded. Then, each error correction is performed in the unit of a block. After performing FEC, the server decides whether the FEC process is successful or not by comparing both fingerprints again. If the file-size check or FEC fails, the Mimic server returns a *SYNCH_FAIL* message to the Mimic client, which then propagates it to the local file system.

## *6.6 Performance Evaluation*

In this section, we use a simple prototype of Mimic to evaluate its performance against that of the differential update scheme.

### 6.6.1 Prototype Description

The current prototype is implemented on the Microsoft Windows platform in order to evaluate its performance with Microsoft Office, which currently dominates the office suite market[44]. However, all the implemented basic operations are performed in an OS-transparent way.

#### *6.6.1.1 Client*

At the start of the activity recording, the client software obtains file information such as filename, attributes, extension, size, and timestamps. It then copies the original file at the server into the local directory at the client. After that, it detects the corresponding application associated with the file extension, and opens the copied local file through a process of the application. If the process is already running another file of the same file extension and multiple instances of the application are sharing a main process, it automatically opens a thread instead of a process. After the process (or thread) is created by the client software,

**Table 4:** Application index for experiments.

| Application Index | Content | Feature | Application |
|---|---|---|---|
| A1 | text-based | keyboard-intensive | Word |
| A2 | text + other | keyboard and mouse | PowerPoint |
| A3 | structured text | keyboard and mouse | Excel |
| A4 | graphics | mouse-intensive | Visio |

the operating system returns the handle values of the process (or thread). Based on this value, the client module collects handles of the corresponding application window(s).

The activity record (AR) header has the following structure. The *original file information* consists of two fields; filename/extension of variable length and file size of two bytes. The *system environment information* includes two 2-byte fields for horizontal and vertical resolution values, 2-byte color depth field, and 2-byte language identifier field for the input language. The *application environment information* has two 2-byte fields for horizontal and vertical size for the main window and two 2-byte fields for horizontal and vertical location for the main window.

In order to capture input system messages, we use the *JournalRecordProc* hook procedure, which WinAPI provides. The target messages include WM_KEY*, WM_SYSKEY*, WM_LBUTTON*, WM_RBUTTON*,and WM_MOUSEWHEEL messages. The captured messages are converted into corresponding descriptors and recorded in the body.

The prototype performs the *write-on-close* update scheme. When the process is terminated, the recorder generates the AR trailer and saves the AR in the local directory. The AR trailer includes three parts as mentioned earlier. The *updated file information* includes 2-byte file-size field. The *fingerprint* has 16-byte keyword, and the *FEC* has variable length according to the size of the target file.

After the AR is saved, the client software generates the *diff* using *xDelta* software[34] and compares the size of both the AR and *diff*. If the AR size is smaller, it transfers the AR file in the network directory at the server. The filename of the AR file includes the filename of the original file and the recording sequence number (for the case that the client performs

**Table 5:** User activity index for experiments.

| Activity Type | Activity Index | Size of Change | | | |
|---|---|---|---|---|---|
| | | **A1** | **A2** | **A3** | **A4** |
| Initial file state | | 5 full pages with 176 lines | 8 full slides | 83 rows | 1 page with 10 small graphs |
| Insert | I-1 | 1 line (0.6%) | 1 new blank slide | 1 row (1.2%) | 1 new blank page |
| | I-2 | 1 paragraph (3%) | 1 paragraph (6%) | 10 rows (12%) | 1 small graph (10%) |
| | I-3 | 1 full page (20%) | 1 text slide (17%) | 50 rows (60%) | 1 large graph (17%) |
| Modify | M-1 | 1 paragraph (3%) | 1 paragraph (6%) | 1 row (1.2%) | 1 small graph (10%) |
| | M-2 | 1 full page (20%) | 1 slide (17%) | 10 rows (12%) | 1 large graph (17%) |
| Delete | D-1 | 1 paragraph (3%) | 1 paragraph (6%) | 10 rows (12%) | 1 small graph (10%) |
| | D-2 | 1 full page (20%) | 1 slide with a large picture (17%) | 50 rows (60%) | 1 small graph (17%) |
| Copy and paste | C-1 | 1 paragraph from the same file (3%) | 1 paragraph from the same file (6%) | 1 row from the same file (1.2%) | 1 small graph from the same file (10%) |
| | C-2 | 1 full page from the same file (20%) | 1 page from the same file (17%) | 10 rows from the same file (12%) | 1 large graph from the same file (17%) |
| | C-3 | 1 page of an external file (20%) | 1 page with a picture from an external file (17%) | 1 row from an external file (1.2%) | 1 small graph from an external file (10%) |
| | C-4 | 1 picture from an external file | 1 picture from an external file | 10 rows from an external file (12%) | 1 small picture from an external file |
| Meta data | F | Change the font type of a paragraph | Change the font type of a slide | Change the font type of a row | Change the font type of a graph |

multiple recording for the single file). For the case that raw activity shipping fails, it leaves the *diff* file in the local directory temporarily.

#### *6.6.1.2 Server*

The playback procedure begins when the server software detects the arrival of a new AR file. Initially, it checks whether the original file is available and has the same file-size as described in the AR header. If both checks succeeds, it synchronizes the system environment based on the AR header. Then, it copies the original file and opens the copied file with the associated application. And, it also synchronizes the application environment by adjusting the size and location of the main window. If it detects an arrival of a *diff* file, it performs differential update and remove all unnecessary files and terminates the update process.

The prototype has two options of playback speed control; constant interval based and CPU-utilization based. However, in the current prototype the delay for calculation of the

CPU utilization in a process is relatively large because of its loose integration with the operating system. Thus, for the performance evaluation we choose the constant interval based scheme that sets different time interval per message type.

After all playback is completed, the server closes the application and begins to perform file verification assuming that the user updates the file explicitly while recording (by pushing the *save* button or so on). When the playback fails, it copies the *diff* file from the local directory at the client and performs differential update. After the update is completed, the server removes the AR and fingerprint files at both sides, the *diff* file(s), the copied file at the client, and the original file at the server. Then, it terminates the process.

### 6.6.2 Experimental Setup

We consider wireless wide area networks (WWANs) as a representative of weakly connected networks in our experiments. The client is connected to the network through a CDMA2000-1X cellular network. The measured effective data rate on the WWAN interfaces is about 17 Kbps, and the round-trip time between the client and the server is about 300 ms.

The mobile client used in the experiments is a HP Pavilion N5430 laptop computer with an 850 MHz AMD Duron CPU, 128 MB RAM, and a Sprint PCS Merlin C210 WWAN interface card. The server is a Dell Dimension 4400 desktop computer with an 1.6 GHz Intel Pentium IV CPU, 256 MB RAM, and a 3COM 10/100 Mbps 3C905CX-TXM NIC, and it runs the Windows 2000 Advanced Server operating system. For all the experiments, we use a 1024×768 (XGA) screen display resolution.

We use the Microsoft Office suite of applications for the experiments. Specifically, we use Word, PowerPoint, Excel, and Visio for word-processing, presentation, spreadsheet, and graphic editing, respectively. Table 4 shows the task labels that are used later in this Section for convenience. Table 5 describes the different operations and the corresponding index which will be used in the subsequent results.

We primarily focus on the transfer size overheads and latency as the metrics for the comparison. Note that the total synchronization time of Mimic is composed of three components: the transfer latency for the records, the playback time, and the verification time.

In the experiments, it can be estimated as

$$sync\_time = \left(\frac{record\_size + other\_overhead}{bandwidth}\right) + playback\_time + verify\_time. \quad (6)$$

The synchronization time of *diff* is composed of the time taken by *diff* to generate the differential patch, the transfer latency for the patch, and the time take at the server to use the patch to recreate the updated file. Thus, its total synchronization time is calculated as

$$sync\_time = patch\_gen\_time + \left(\frac{patch\_size}{bandwidth}\right) + merge\_time. \quad (7)$$

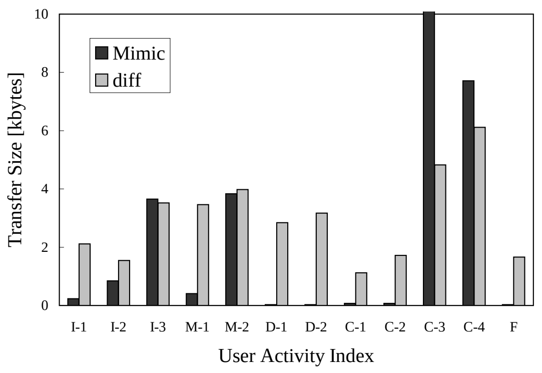
Windows operating systems uses Object Linking and Embedding (OLE)[50], which provides means for integrating objects from diverse applications. An object is a block of information that could come from a word processor, a spreadsheet, a graphic content, an audio clip, or an executable program itself. To provide compatibility with other applications, the content copied to the clipboard is stored in the form, which supports OLE taking much larger space than the original content in the memory. However, when the object is integrated with an application file, it undergoes compacting whereby the size of the file becomes significantly less than that in the memory. We assume that all data except *diff*, which is already compressed by its own algorithm, are compressed before being transmitted. In the experiments, we assume that there is no environmental change while recording and replaying.

### 6.6.3 Transfer Size Performance

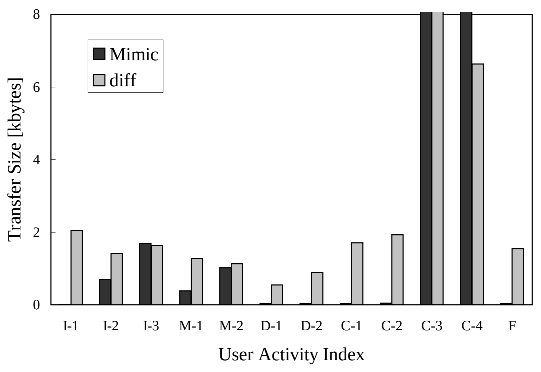
#### *6.6.3.1 Impact of User Activity Type*

Figure 52 presents the experiment results of the transfer size for various user activity types.

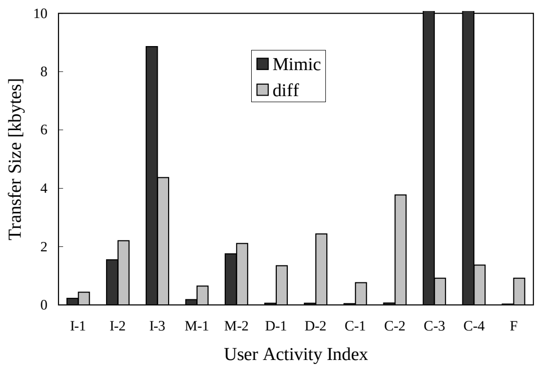
- *Insert:* In Figure 52, it can be seen that the transfer size in raw activity shipping is usually equal or smaller than that for *diff* when any insertion is performed. It is because each user input activity is translated into more complicated operations within the applications, and as a result the size of binary change becomes significantly larger than the input activity size.



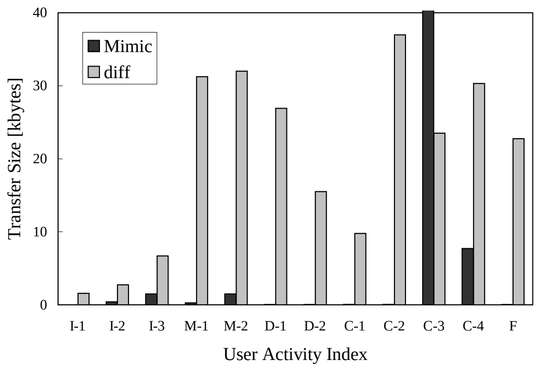
(a) Transfer Size for A1.



(b) Transfer Size for A2.



(c) Transfer Size for A3.



(d) Transfer Size for A4.

**Figure 52:** Performance comparison in transfer size.

The figures also show that the update size in raw activity shipping is always proportional to the magnitude of insertion in any application, while the *diff* size is unpredictable in text-centric applications (A1,A2). As explained in Section 6.2, word-processing and presentation applications have much more complicated file structures in order to embed various types of external objects. Hence, regardless of user activity size, each insertion affects the entire file structure. For example, the location of an insertion is one of the main factors that decides the magnitude of file structure change. When a character is inserted in the first page of a document, its *diff* overhead can be several times larger than that of a similar insertion in the last page of the document.

On the contrary, the *diff* update size in graphic and spreadsheet applications (A3,A4) shows more modest performance improvements for raw activity shipping. It is because those applications employ relatively simpler file structures and each insertion is translated simply as an addition of objects, paving the way for reasonable overheads when using *diff*.

- *Modify:* Files can be modified by changing either the attribute of the content such as font type, or the content itself. In the following experiments, we consider only the latter. This modifications consist of a combinations of multiple deletions and insertions. The former type of file modification is classified as meta data change, and explained later.

  In Figure 52, it is shown that the overhead of Mimic is smaller than that of *diff* for most modification operations. However, the *diff* size in (A1,A2) does not increase linearly with the amount of activity as is the case with the record size for the same reason provided for the effect of insertions.

  On the other hand, the modifications in (A3,A4) show quite different results from those of insertions. Especially in (A4), the *diff* update size of modifications is even larger than that of insertions, while Mimic's record overhead has the same size. This is because in graphic applications the size of an object is proportional to the complexity of the object.

- *Delete:* When a user performs a delete or copy operation, the file structure can be dramatically changed even for a few number of deletes. Content deletions can be categorized into two types, full paragraph deletion and partial paragraph deletion. Generally, within the file structure, file contents are stored in paragraphs with each paragraph having its own content attributes. Therefore, if only some parts of a paragraph are removed, the file size reduction is not as large as that in full paragraph deletions. However, both cases are considered in the experiments.

  Figure 52 shows that the record overhead of Mimic is significantly smaller compared to the *diff* size in all the applications. This is because the deletion operations incur relatively large changes in the file structure while Mimic's record overhead is proportional only to the activity size.

- *Copy and Paste:* Copying content means converting the content, which exists inside the file itself or externally, into another form that have compatibility with any other applications and loading it into the memory as a form of meta data. Once the content is copied into the clipboard, it can be

In the figure, Mimic's overhead for internal copying and pasting (C-1,C-2) is considerably smaller than that of *diff* in all the applications. The reason is as follows. When an object that exists inside the same file is copied, raw activity shipping does not have to send the content of the clipboard to the server, and instead it lets the object to be copied automatically at the server through the user activity itself. Therefore, it eliminates the clipboard overhead as seen in Figure 52.

However, if the object is copied from the external source (C-3,C-4), Mimic's record overhead is significantly increased in all the applications. This is because Mimic captures and compresses the content of the clipboard as binary data, and then transmits it to the server along with the AR. However, the copied objects that follow the OLE format have much larger size in the memory than the originally generated objects. In this scenario, *diff* usually shows better performance than raw activity shipping due to large overheads for transferring the clipboard content.

- *Meta Data:* A user may change the configuration of textual information such as the font size or font type instead of the text content itself. In the experiments, all the modifications to meta data change the file structures drastically even though the file size change itself is small. It can be seen that raw activity shipping shows much better performance than *diff* in all the applications. This is because changes to meta data are similar in nature to simple text insertion (both affect the entire file structure), with potentially a larger impact.

#### *6.6.3.2 Impact of Application Type*

- *Word-processing:* Word-processing applications are basically text-centric programs that are able to embed various types of external objects such as pictures, graphs, tables, and equations. Hence, the file structure consists of a large number of data objects and their corresponding links. In Figure 52(a), *diff* in (A1) shows somewhat unpredictable performance results in (I,M) than raw activity shipping due to its complicated file structure. Again, for activities (D,C-1,C-2,F), the improvement brought about by Mimic is considerable. However, when a user copies a picture from another program, due to the clipboard problem discussed earlier, the overhead of Mimic is dramatically increased.

- *Presentation:* Presentation programs are popular applications for visual presentation where a user intends to present various types of data. Copying objects from other applications is one of the most important functions in addition to text editing. In Figure 52(b), in most cases of text editing and meta data change, Mimic shows equal or better performance than *diff*.

- *Spreadsheet:* A spreadsheet file consists of hybrid objects that are numerical values and graphs. In the experiment, we consider only the former type of data. Figure 52(c) shows the transfer size per user activity in Excel. Again, several activities including (D,C-1,C-2,F) result in Mimic exhibiting large benefits in the transfer file size.

  However, Mimic in (A3) does not show the best performance for insertions and modifications (I,M) due to lots of redundant messages. Basically, a spreadsheet file visually consists of lots of cells, and a user usually moves the cursor to another cell using mouse-clicks. This is the reason why Mimic's record size in (A3) is relatively much larger that that in (A1,A2) even though (A3) is a text-based application.

  While the external copy activities are detrimental to the performance of raw activity shipping as usual, the results for the activity (C3) is also interesting. Based on the results in the figure, it can be observed that when the degree of user activity is a substantial portion of the actual content of the file (20% in these experiments), Mimic's performance will start equaling that of *diff*, and sometimes even become worse.

- *Graphic:* In Figure 52(d), the performance of Mimic is *always significantly better* that that of *diff*. This is because *graphic editors* manage various complicated objects that include large number of environmental data. For example, when a user draws a circle,

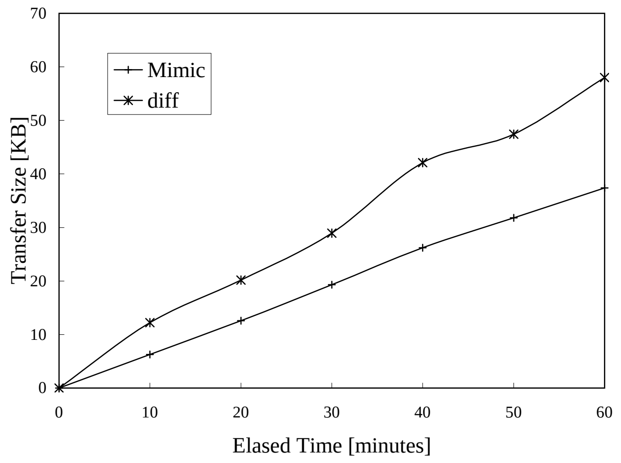


**Figure 53:** Transfer size for large-scale user activity.

the circle has many parameters related to its setting such as display priority with a respect to the other objects or grouping with the other objects. Further its binary level complexity is much higher than text based data.

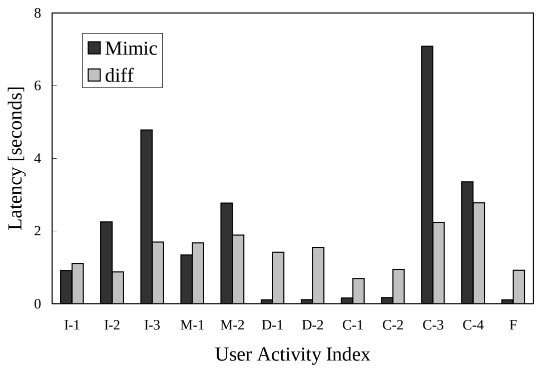
#### *6.6.3.3 Impact of Update Interval*

Figure 53 shows the overhead results with different update intervals when a user accesses multiple files (A1,A4) spending the same time per file, and performs various input activities. The input rate is about 200 operations per minute for (A1) and 85 operations per minute for (A4). The dominant operations used are insertions.

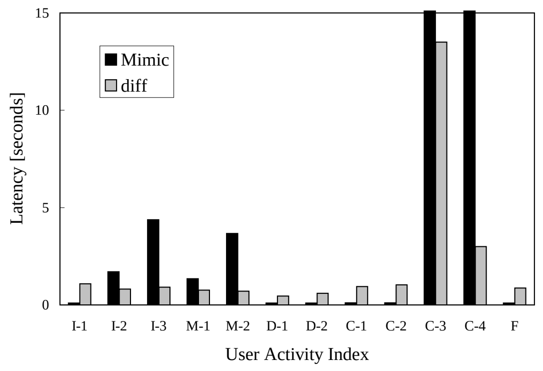
It can be observed that both Mimic and *diff* overhead increases in proportion to the update interval for mixed activities. This is because the overall overhead is dominated by the transfer size of (A4), whose overhead is increases almost linearly with a larger interval. Thus, as the interval becomes larger, the overhead difference is also increased linearly. In the experiments, Mimic reduces the size of overhead by about 40%.

#### *6.6.3.4 Summary*

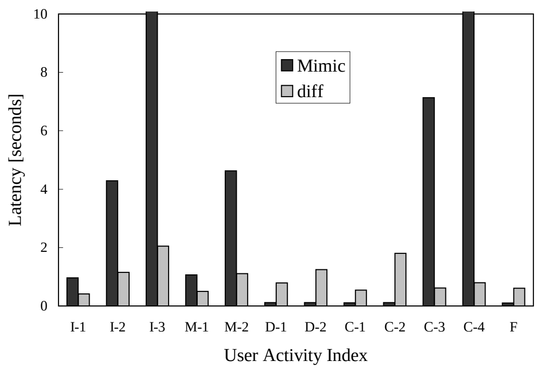
In Figure 52, it can be seen that the transfer size in Mimic is generally equal or smaller than that in *diff* except when copying from outside the file. Overall performance in *diff* and Mimic can be characterized as follows.



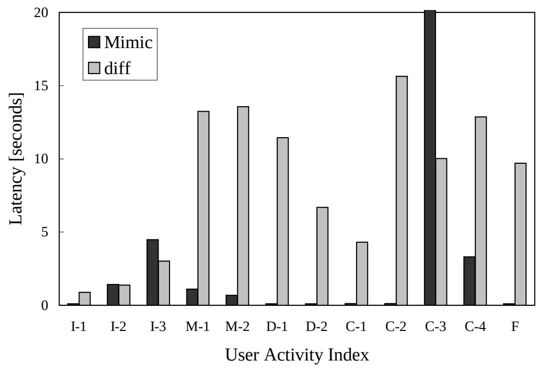
(a) Latency for A1.



(b) Latency for A2.



(c) Latency for A3.



(d) Latency for A4.

**Figure 54:** Performance comparison in latency.

Generally, the record overhead of Mimic is proportional only to the activity size. However, there are some exceptions, in which even Mimic's overheads do not seem to be proportional to the magnitude of user activity, such as copying and pasting from an external source. Similarly, delete or modify operations can incur smaller overheads than their insert counterparts for the same magnitude of user activity.

Equally interestingly, the single line insertion (I-1) in *diff* consumes more bandwidth than a single paragraph insertion (I-2) in Figure 52. This phenomenon is again due to the impact of application-specific storage semantics as discussed in Section 6.2.

### 6.6.4 Latency Performance

The synchronization latency of Mimic generally depends on the playback performance, which is decided by overall system capability such as CPU processing power. In the experiments, we assume that a user types about 200 characters per minute. The CPU idle check based playback mechanism in Mimic results in a maximum replaying speed of approximately 90 times the original speed in the experiments. If the server is equipped with more processing power, the maximum playback speed can further be increased, and the latency thus reduced. Figure 54 is the latency results for all the applications when the synchronization is performed over a WWAN.

#### *6.6.4.1 Impact of User Activity Type*

In the experiments, for small insertions, deletions, internal copies, and meta data changes (I-1,D,C-1,C-2,F), Mimic performs better in terms of latency. Even though the latency in Mimic includes its playback time besides transmission time, its total update time does not exceed that of *diff* because the benefit of small transfer size for those operations is larger than playback overhead.

However, for the remaining types of activities such as large insertions, modifications, and external copies (I-3,M,C-3,C-4), Mimic performs worse than *diff* in latency in (A1,A2,A3). Especially, moderate insertions (I-2) in Mimic shows larger latencies even though its overhead is smaller because the playback latency becomes relatively large. Hence, if the transfer size itself in Mimic is already larger than that of diff, Mimic cannot show better latency performance.

This brings out an interesting trade-off: Can increase in latencies be tolerated if it reduces the total transfer size? For WWANs especially, where users may have to pay on a per-MB basis, this is arguably so.

#### *6.6.4.2 Impact of Application Type*

In the experiments, the latency performance in Mimic follows the trend of the transfer size performance approximately because the latency is closely relevant to the transfer size.

Finally, in Figure 54(d), Mimic shows overall better performance than *diff*, and it is because of both the lower transfer sizes, and the faster replaying speed used by Mimic. Hence, Mimic brings significantly larger benefits for such a mouse-centric graphic application that has a large ratio of file change to activity size.

### 6.6.5 System Overheads

Recording and CPU utilization monitoring are important components in Mimic. However, Mimic subsystems consume only a negligible portion of the system resources. In our experiments, the average CPU utilization for the recording process is less than 1%, and memory usage is about 4 MB. This is small compared to the amount of the resource used by an application, such as Microsoft Word that can occupy up to 20 MB of memory and up to 50% of CPU utilization.

### 6.6.6 User statistics and expected results

In order to quantify the performance improvement that can be experienced by real users, we collected short-term user-activity the records from 447 document updates performed by ten different people. We modified the Windows registry, which stores setting and options for the operating systems, such that all documents of the target formats (*e.g.* Word, PowerPoint, Excel, and Visio) are opened by our recorder. Then, the recorder installs a journal record hook to capture input system messages, opens the associated applications, and gets the process and thread information of the application instances.

Table 6 shows the statistics of insertion, modification, and deletion activity measured from the experiments. We use the same activity indicies used in Tables 4 and 5. In the table, *N/A* means that the test was not performed by the users for this condition. *Other* indicates a different amount of activity size, and *Combined* means a mix of different activities. The statistics show that major update activity performed by the users are insertions and modifications. Deletions and other update activity account for less than 20% of the total activity.

To estimate the performance improvement in real-life, we obtain the overall transfer-size reductions by calculating the sum of the products of the per-activity transfer-size reductions shown in Figure 52 and the portions of each activity. Table 7 shows the expected performance improvement in percentage. In the table, it can be seen that Mimic reduces the total transfer size from 20% (in A3, Excel) to 89% (in A4, Visio).

We also measured the frequencies of other activity types, which do not take a major

**Table 6:** Statistics of insertion, modification, and deletion in the test.

| **Activity Type** | **Activity Index** | **Application Index** | | | |
|---|---|---|---|---|---|
| | | **A1** | **A2** | **A3** | **A4** |
| Insert | I-1 | 4.0% | 1.7% | 2.3% | 6.8% |
| | I-2 | 23.8% | 11.3% | 15.5% | 16.7% |
| | I-3 | 4.4% | 16.4% | 10.6% | 14.9% |
| | other | 2.6% | 9.5% | 4.0% | 12.7% |
| | **subtotal** | 34.8% | 38.9% | 32.4% | 51.1% |
| Modify | M-1 | 18.9% | 17.1% | 1.5% | 6.2% |
| | M-2 | 26.2% | 22.0% | 34.2% | 15.4% |
| | other | 5.6% | 10.1% | 18.1% | 7.0% |
| | **subtotal** | 50.7% | 49.2% | 53.8% | 30.6% |
| Delete | D-1 | 0.5% | 0.6% | 0.0% | 0.2% |
| | D-2 | 1.0% | 1.8% | 0.2% | 0.4% |
| | other | 0.3% | 1.0% | 0.3% | N/A |
| | **subtotal** | 1.8% | 3.4% | 0.5% | 0.7% |
| Combined | | 12.7% | 8.5% | 13.3% | 17.6% |
| **Total** | | 100% | 100% | 100% | 100% |

**Table 7:** Expected transfer-size reductions from insertion, modification, and deletion.

| **Activity Type** | **Activity Index** | **Application Index** | | | |
|---|---|---|---|---|---|
| | | **A1** | **A2** | **A3** | **A4** |
| Insert | I-1 | 88.8% | 99.4% | 47.8% | 99.3% |
| | I-2 | 45.1% | 50.8% | 29.8% | 84.7% |
| | I-3 | 4.0% | 0.3% | 0.0% | 77.9% |
| Modify | M-1 | 88.2% | 70.0% | 71.4% | 99.2% |
| | M-2 | 3.5% | 10.1% | 16.8% | 95.3% |
| Delete | D-1 | 98.9% | 94.5% | 95.9% | 99.9% |
| | D-2 | 99.1% | 96.6% | 97.7% | 99.8% |
| **Average** | | 42.0% | 47.6% | 19.8% | 89.0% |

portion but affect the entire content significantly. In the test, the average frequency of internal, external copy-and-paste and meta-data change varies according to the application types. For example, as shown in Table 8, for A2 (PowerPoint), the users perform internal copy-and-paste seven times on average, whereas they perform less than one time in A4

**Table 8:** Frequency of copying and meta-data change in the test.

| Activity Type | Activity Index | Application Index | | | |
|---|---|---|---|---|---|
| | | **A1** | **A2** | **A3** | **A4** |
| Internal Copy and Paste | C-1 | 2.05 | 4.79 | 0.89 | 0.06 |
| | C-2 | 0.17 | 2.35 | 2.13 | 0.38 |
| | other | 0.01 | N/A | 1.08 | N/A |
| | **subtotal** | 2.23 | 7.14 | 4.10 | 0.44 |
| External Copy and Paste | C-3 | 0.09 | 1.76 | 0.04 | 0.01 |
| | C-4 | 0.01 | 2.51 | 0.00 | 0.00 |
| | other | 0.00 | N/A | 0.00 | N/A |
| | **subtotal** | 0.10 | 4.27 | 0.04 | 0.01 |
| Meta data | F | 0.84 | 5.25 | 2.26 | 1.67 |

**Table 9:** Expected transfer-size reductions from copy-and-paste and meta-data change.

| Activity Type | Activity Index | Application Index | | | |
|---|---|---|---|---|---|
| | | **A1** | **A2** | **A3** | **A4** |
| Internal Copy and Paste | C-1 | 1047 B | 1663 B | 716 B | 9729 B |
| | C-2 | 1648 B | 1883 B | 3709 B | 36901 B |
| External Copy and Paste | C-3 | 0 B | 0 B | 0 B | 0 B |
| | C-4 | 0 B | 0 B | 0 B | 0 B |
| Meta data | F | 1630 B | 1514 B | 886 B | 22701 B |
| **Average** | | 3795 B | 20339 B | 10540 B | 52517 B |

(Visio). Thus, to maximize the activity-shipping performance, it is very important to customize the execution parameters according to the running application.

To estimate the performance improvement from these activity types, we again calculate the expected transfer-size reductions based on the results shown in Figure 52. Table 9 shows the expected transfer size reductions in bytes, which are calculated as the sum of the products of frequencies and transfer-size reductions of those activity types. In the table, it can be seen that the copy-and-paste and meta-data change in graphical applications (A4) affects their transfer performance significantly, and Mimic performs the update optimization for such applications.

## *6.7 Summary*

In this chapter, we consider the problem of file synchronization when a mobile host shares files with a backbone file server in a network file system. We show that differential file-synchronization schemes incur substantially more overheads than necessary. We then propose an application-independent approach called *Mimic* that relies on transferring user activity records to the server, where the new file is recreated through a playback of the user activity on the old copy of the file. We show that Mimic performs much better than the differential update in most scenarios in terms of the transfer file sizes. The trade-off is that the latency incurred by Mimic due to its replay mechanism can be larger than the overall latency incurred by the differential schemes. We also identify some conditions under which Mimic incurs more transfer size overheads than the differential scheme. Despite the trade-offs, we conclude that raw activity shipping can be used in tandem with the differential update scheme to substantially improve file synchronization performance, especially when the bandwidth available on the network connection is low and expensive.

# CHAPTER VII

# CONCLUSIONS

In this dissertation, we investigated the inefficiency problems of operations in conventional information access under mobile environments. Since most conventional access schemes are designed only for static high-bandwidth wired networks without considering other various computing environments, they do not utilize network resources efficiently in low-bandwidth networks, and as a result their user-perceived performance is unnecessarily degraded in such a environment.

We found that the reason for the inefficient performance is being unaware to user activity and propose a new paradigm for mobile information access that is driven by awareness to user activity. Our paradigm consists of three user-activity-aware strategies for three different scenarios: reading non-partitionable content, reading partitionable content, and content synchronization.

First, for reading non-paritionable content, we present an application-unaware read-access scheme called Cut-Load, which perform read operations in a graphical domain. Cut-Load uses a intelligent mix of binary file transfers and graphical content-partitioning along with features such as opportunistic hoarding to reduce the bandwidth consumption as well as response times for information access. Through *ns2* simulations, we evaluated its user-perceived performance and proved its benefits for non-partitionable content over traditional access systems. We also analyzed the behavior of the real-life implementation of the Cut-Load prototype.

Second, for reading partitionable content, we propose an application-aware acceleration scheme called Prioritized Fetching, which performs a mix of object prioritization, object reordering, and connection management. Using simulations with the Web parameters obtained from the Top Fifty Web Sites, we evaluated the performance of our scheme and proved its benefits over conventional Web-access models. We also implemented a simple prototype of the prioritized fetching algorithm and measured the performance benefit in real-life experiments. One major advantage of this scheme is that it is easy to deploy since it only requires client-side installation to current applications.

Finally, for content synchronization, we present an application-unaware scheme, called Mimic, which intelligently selects the transfer mode between raw-activity shipping and differential update for bandwidth-efficient file synchronization. Through a simple prototype in Windows operating systems, we show that raw-activity shipping can outperform the differential update schemes under many common conditions. We also identify the conditions under which the differential schemes do perform better than activity shipping, but show that the detection of such conditions is straightforward, thus enabling both update schemes to be used in tandem with a mobile file system for bandwidth-efficient file synchronization.

In conclusion, we believe that the contributions of this work lie not only in proposing a new user-activity-aware paradigm and strategies but also in indicating new directions that future researches can build upon. We also believe that future researches can bring more benefits based on this work addressing other problems in mobile information For future works, we are interested in designing network protocols that are optimized for multi-object transmissions as well as remote applications based on thin-client computing.

# REFERENCES

[1] Alshanetsky, I., "Google Web Accelerator and the dangers of prefetching," May 2005. (Date Accessed: Mar. 2006).
`http://ilia.ws/archives/46-Google-Web-Accelerator-and-the-dangers-of-prefetching.html`

[2] Baker, B., Manber, U., and Muth, R., "Compressing Differences of Executable Code," in *Proceedings of the ACM SIGPLAN Workshop on Compiler Support for System Software*, Apr. 1999.

[3] Beck, M., Arnold, D., Bassi, A., Berman, F., Casanova, H., Dongarra, J., Moore, T., Obertelli, G., Plank, J., Swany, M., Vadhiyar, S., and Wolski, R., "Logistical Computing and Internetworking: Middleware for the Use of Storage in Communication," in *Proceedings of the 3rd Annual International Workshop on Active Middleware Services (AMS)*, pp. 12–21, Aug. 2001.

[4] Blue Coat Systems, "CacheFlow." (Date Accessed: Mar. 2006).
`http://www.cacheflow.com`

[5] Bray, T., "Measuring the Web," in *Proceedings of the Fifth International World Wide Web Conference on Computer Networks and ISDN Systems*, pp. 993–1005, May 1996.

[6] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S., "Web Caching and Zipf-like Distributions: Evidence and Implications," in *Proceedings of IEEE INFOCOM 1999*, pp. 126–134, Mar. 1999.

[7] Brewer, E. A., Katz, R. H., Chawathe, Y., Gribble, S. D., Hodes, T., Nguyen, G., Stemm, M., Henderson, T., Amir, E., Balakrishnan, H., Fox, A., Padmanabhan, V. N., and Seshan, S., "A Network Architecture for Heterogeneous Mobile Computing," *IEEE Personal Communication Magazine*, vol. 5, pp. 8–24, Oct. 1998.

[8] Brooks, C., Mazer, M. S., Meeks, S., and Miller, J., "Application-Specific Proxy Servers as HTTP Stream Transducers," in *Proceedings of the 4th International World Wide Web Conference on Computer Networks and ISDN Systems*, pp. 539–548, Dec. 1995.

[9] Chakravorty, R., Banerjee, S., Rodriguez, P., Chesterfield, J., and Pratt, I., "Performance Optimizations for Wireless Wide-Area Networks: Comparative Study and Experimental Evaluation," in *Proceedings of ACM Mobicom 2004*, pp. 159–173, Sept. 2004.

[10] Chawathe, Y., Gribble, S. D., Hodes, T., Nguyen, G., Stemm, M., Henderson, T., Amir, E., Balakrishnan, H., Fox, A., Padmanabhan, V. N., Brewer, E. A., Katz, R. H., and Seshan, S., "A Network Architecture for Heterogeneous

Mobile Computing," *IEEE Personal Communications Magazine*, vol. 5, pp. 8–24, Oct. 1998.

[11] Chen, X. and Zhang, X., "Coordinated Data Prefetching by Utilizing Reference Information at Both Proxy and Web Servers," *Proceedings of the ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 2, pp. 32–38, 2001.

[12] Chosen Software Inc., "Journal Macro." (Date Accessed: Feb. 2003).
`http://www.keyboard-macro-recorder.com`

[13] comScore Networks Inc., "comScore Media Metrix Top 50 Online Property Ranking," Jan. 2005. (Date Accessed: Mar. 2006).
`http://www.comscore.com/press/release.asp?press=547`

[14] Cooper, I. and Dilley, J., "RFC 3143 - Known HTTP Proxy/Caching Problems," June 2001. (Date Accessed Mar. 2007).
`www.ietf.org/rfc/rfc3143.txt`

[15] digital ISLAND Communications. (Date Accessed Mar. 2007).
`http://www.digitalisland.co.nz`

[16] Douglis, F., Feldmann, A., and Krishnamurthy, B., "Rate of Change and other Metrics:a Live Study of the World Wide Web," in *Proceedings of USENIX Symposium on Internet Technologies and Systems*, pp. 147–158, Dec. 1997.

[17] Douglis, F. and Ousterhout, J. K., "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software- Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.

[18] Duchamp, D., "Prefetching Hyperlinks," in *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, pp. 12–23, Oct. 1999.

[19] Dwyer, D. and Bharghavan, V., "A Mobility-Aware File System for Partially Connected Operation," *ACM Operating Systems Review*, vol. 31, pp. 24–30, Jan. 1997.

[20] Ellacoya Networks, "Ellacoya Data Shows Web Traffic Overtakes Peer-to-Peer (P2P) as Largest Percentage of Bandwidth on the Network." (Date Accessed: Oct. 2007).
`http://www.ellacoya.com/news/pdf/2007/NXTcommEllacoyaMediaAlert.pdf`

[21] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1," 1999. (Date Accessed Dec. 2002).
`www.ietf.org/rfc/rfc2616.txt`

[22] Flinn, J., Park, S., and Satyanarayanan, M., "Balancing Performance, Energy, and Quality in Pervasive Computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, July 2002.

[23] Fox, A. and Brewer, E. A., "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation," in *Computer Networks and ISDN Systems*, pp. 1445–1456, 1996.

[24] Fox, A., Gribble, S., Chawathe, Y., Brewer, E., and Gauthier, P., "Cluster-Based Scalable Network Services," in *Proceedings of ACM Symposium on Operating Systems Principles*, Oct. 1997.

[25] Gilbert, J. and Brodersen, R., "Globally Progressive Interactive Web Delivery," in *Proceedings of IEEE Infocom 1999*, pp. 1291–1299, Mar. 1999.

[26] Google Inc., "Google Web Accelerator." (Date Accessed: Sep. 2006). `http://webaccelerator.google.com`

[27] Guy, R. G., Heidemann, J. S., Mak, W., Page, T. W., Popek, G. J., and Rothmeier, D., "Implementation of the Ficus Replicated File System," in *Proceedings of 1990 Usenix Summer Conference*, pp. 63–71, June 1990.

[28] Han, R., Bhagwat, P., LaMaire, R., Mummert, T., Perret, V., and Rubas, J., "Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing," *IEEE Personal Communications*, vol. 5, pp. 8–17, Dec. 1998.

[29] Housel, B. C., Samaras, G., and Lindquist, D. B., "WebExpress: A Client/Intercept Based System for Optimizing Web Browsing in a Wireless Environment," *ACM/Baltzer Mobile Networks and Applications (MONET)*, vol. 3, no. 4, pp. 419–432, 1999.

[30] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R., and West, M. J., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, Feb. 1988.

[31] IBM Inc., "IBM Websphere Edge Server." (Date Accessed: Aug. 2006). `http://www-306.ibm.com/software/webservers/edgeserver`

[32] Ibrahim, T. I. and Xu, C.-Z., "Neural Nets based Predictive Prefetching to Tolerate WWW Latency," in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, pp. 636–643, 2000.

[33] Jiang, Z. and Kleinrock, L., "An Adaptive Network Prefetch Scheme," *IEEE Journal on Selected Areas in Communications*, pp. 358–368, Apr. 1998.

[34] Jones, E., "XDelta for Windows." (Date Accessed: Mar. 2007). `http://xdelta.org`

[35] Joshi, A., "On Proxy Agents, Mobility, and Web Access," *Mobile Networks and Applications*, vol. 5, pp. 233–241, Dec. 2000.

[36] Kistler, J. J. and Satyanarayanan, M., "Disconnected Operation in the Coda File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, 1992.

[37] Kunz, T. and Black, J. P., "An Architecture for Adaptive Mobile Applications," in *Proceedings of the 11th Annual International Conference On Wireless Communication*, July 1999.

[38] Kuzmanovic, A. and Knightly, E. W., "TCP-LP: A Distributed Algorithm for Low Priority Data Transfer," in *Proceedings of IEEE INFOCOM 2003*, pp. 1691–1701, Apr. 2003.

[39] Kvilekval, K. and Singh, A., "SPREE: Object Prefetching for Mobile Computers," in *Distributed Objects and Applications (DOA)*, Oct. 2004.

[40] Lai, A. and Nieh, J., "Limits of Wide-Area Thin-Client Computing," in *Proceedings of the ACM SIGMETRICS 2002*, June 2002.

[41] Lara, E., Wallach, D. S., and Zwaenepoel, W., "Puppeteer: Component-based Adaptation for Mobile Computing," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2001.

[42] LBL, Xerox PARC, UCB, and USC/ISI, "The Network Simulator - ns-2." (Date Accessed: Mar. 2002).
`http://www.isi.edu/nsnam/ns`

[43] Lee, Y., Leung, K., and Satyanarayanan, M., "Operation Shipping for Mobile File Systems," *IEEE Transactions on Computers*, vol. 51, pp. 1410–1422, Dec. 2002.

[44] Loftus, J., "Enterprise Linux News: Desktop apps ripe turf for open source," Oct. 2004. (Date Accessed: Jan. 2007).
`http://searchenterpriselinux.techtarget.com/originalContent/0,289142,sid39_gci1011227,00.html`

[45] Mathis, M., Semke, J., Mahdavi, J., and Ott, T., "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *ACM Computer Communication Review (CCR)*, vol. 27, no. 3, pp. 67–82, 1997.

[46] Microsoft Corporation, "About MSHTML." (Date Accessed: Jan. 2007).
`http://msdn2.microsoft.com/en-us/library/bb508515.aspx`

[47] Microsoft Corporation, "Microsoft Internet Explorer." (Date Accessed: Dec. 2002).
`http://www.microsoft.com/windows/products/winfamily/ie/`

[48] Microsoft Corporation., "Microsoft Terminal Services." (Date Accessed: Feb. 2003).
`http://www.microsoft.com/windows2000/technologies/terminal`

[49] Microsoft Corporation, "The Official Microsoft ASP.NET 2.0 Site." (Date Accessed: Feb. 2006).
`http://www.asp.net`

[50] Microsoft Corporation, "Microsoft Word 97 Binary File Format," 1998. (Date Accessed: Mar. 2002).
`http://www.progsoc.uts.edu.au/ subtle/wword8.html`

[51] Miller, R. B., "Response Time in Man-Computer Conversational Transactions," in *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 33, pp. 267–277, 1968.

[52] Mogul, J., Krishnamurthy, B., Douglis, F., Feldmann, A., Goland, Y., van Hoff, A., and Hellerstein, D., "RFC 3229 - Delta Encoding in HTTP," Jan. 2002. (Date Accessed: Nov. 2004).
`http://www.ietf.org/rfc/rfc3229.txt`

[53] Mogul, J. C., "The Case for Persistent-Connection HTTP," *Computer Communication Review*, vol. 25, pp. 299–313, Oct. 1995.

[54] Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B., "Potential Benefits of Delta Encoding and Data Compression for HTTP," in *Proceedings of the ACM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pp. 181–194, 1997.

[55] Molina, M., Castelli, P., and Foddis, G., "Web Traffice Modeling Exploiting TCP Connections' Temporal Clustering through HTML-REDUCE," *IEEE Network*, pp. 46–55, May 2000.

[56] Mozilla Foundation, "Firefox - Rediscover the Web." (Date Accessed: Feb. 2006).
`http://www.mozilla.com/firefox`

[57] Mozilla Foundation, "Mozilla Layout Engine: Gecko Engine." (Date Accessed: Aug. 2006).
`http://www.mozilla.org/newlayout`

[58] Muthitacharoen, A., Chen, B., and Mazieres, D., "A Low-bandwidth Network File System," in *Proceedings of the 18th Symposium on Operating Systems Principles*, Oct. 2001.

[59] Nah, F. F., "A Study on Tolerable Waiting Time: How Long are Web Users Willing to Wait?," in *Proceedings of the American Conference on Information Systems (AMCIS)*, pp. 2212–2222, 2003.

[60] Nandagopal, T., Lee, K.-W., Li, J.-R., and Bharghavan, V., "Scalable Service Differentiation Using Purely End-to-End Mechanisms: Features and Limitations," *Elsevier Computer Networks*, vol. 44, pp. 813–833, Apr. 2004.

[61] Netscape Communications, "Netscape Navigator Web Browser." (Date Accessed: Feb. 2006).
`http://browser.netscape.com`

[62] Network Appliance, Inc., "NetCache." (Date Accessed Mar. 2006).
`http://www.netapp.com/products/netcache`

[63] Nielsen, J., "Top Ten New Mistakes of Web Design." (Date Accessed Jul. 2004).
`http://www.useit.com/alertbox/990530.html`

[64] Ninan, A., Kulkarni, P., Shenoy, P., Ramamritham, K., and Tewari, R., "Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks," in *Proceedings of the 11th international conference on World Wide Web*, (New York, NY, USA), pp. 1–12, ACM Press, 2002.

[65] Noble, B. D. and Satyanarayanan, M., "Experience with Adaptive Mobile Applications in Odyssey," *Mobile Networks and Applications*, vol. 4, no. 4, pp. 245–254, 1999.

[66] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R., "Agile Application-Aware Adaptation for Mobility," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[67] Open Mobile Alliance Inc., "WAP FORUM." (Date Accessed: Oct. 2006).
`http://www.wapforum.org`

[68] Opera Software, "Opera Browser." (Date Accessed: Aug. 2006).
`http://www.opera.com`

[69] Oracle Inc., "Oracal Portal-To-Go: Any Service to Any Device," Oct. 1999.

[70] Padmanabhan, V. N. and Mogul, J. C., "Using Predictive Prefetching to Improve World-Wide Web Latency," *ACM SIGCOMM Computer Communication Review*, pp. 22–36, July 1996.

[71] Pahdye, J. and Floyd, S., "On Inferring TCP Behavior," in *Proceedings of ACM SIGCOMM 2001*, pp. 287–298, 2001.

[72] Percival, C., "Binary diff/patch utility." (Date Accessed: Jun. 2002).
`http://www.daemonology.net/bsdiff`

[73] Pocket Soft Inc., "RTPatch." (Date Accessed: Mar. 2003).
`http://www.pocketsoft.com/rtpproducts.html`

[74] Privoxy Developers, "Privoxy Web Proxy." (Date Accessed: Jan. 2006).
`http://www.privoxy.org`

[75] Qbik New Zealand Limited, "WinGate." (Date Accessed: Jan. 2007).
`http://www.wingate.com/product-wingate.php`

[76] Qualcomm Inc., "Binary Runtime Environment for Wireless (BREW)." (Date Accessed: Oct. 2006).
`http://brew.qualcomm.com`

[77] Reed, I. S. and Solomon, G., "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, June 1960.

[78] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A., "Virtual Network Computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33–38, 1998.

[79] Rivest, R., "MD5 Message-Digest Algorithm," 1992. (Date Accessed: Feb.2002).
`http://www.faqs.org/rfcs/rfc1321`

[80] Rusay, C., "User-Centered Design for Large Government Portals," Jan. 2003. (Date Accessed: Jun. 2006).
`http://www.digital-web.com/articles/user_centered_design_for_large_government_portals`

[81] Satyanarayanan, M., "Accessing Information on Demand at Any Location: Mobile Information Access," *IEEE Personal Communications*, vol. 3, pp. 26–33, Feb. 1996.

[82] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C., "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–359, 1990.

[83] Shen, H., Kumar, M., Das, S. K., and Wang, Z., "Energy-Efficient Data Caching and Prefetching for Mobile Devices Based on Utility," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 475–486, 2005.

[84] Shi, W., Collins, E., and Karamcheti, V., "Modeling Object Characteristics of Dynamic Web Content," *Elsevier Journal of Parallel and Distributed Computing*, vol. 63, no. 10, pp. 963–980, 2003.

[85] Shneiderman, B., "Response Time and Display Rate in Human Performance with Computers," *Computing Surveys*, vol. 16, pp. 265–285, 1984.

[86] Smith, J., Mohan, R., and Li, C., "Transcoding Internet Content for Heterogeneous Client Devices," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 599–602, 1998.

[87] Stone, J. and Partridge, C., "When the CRC and TCP Checksum Disagree," in *ACM SIGCOMM 2000*, Sept. 2000.

[88] Sun Microsystems, Inc., "JavaServer Pages (JSP) Technology." (Date Accessed: Jan. 2007).
`http://java.sun.com/products/jsp`

[89] Sun Microsystems, Inc., "Sun Java System Web Server." (Date Accessed: Mar. 2007).
`http://www.sun.com/webserver`

[90] Team Squid, "Squid Web Proxy Cache." (Date Accessed: Feb. 2002).
`http://www.squid-cache.org`

[91] Terry, D. B., Theimer, M. M., Petersen, K., and Demers, A. J., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[92] Wang, Z. and Cao, P., "Persistent Connection Behavior of Popular Browsers," Dec. 1998. (Date Accessed: Oct. 2007).
`http://pages.cs.wisc.edu/ cao/papers/persistent-connection.html`

[93] Warp Solutions Inc., "SpiderCache." (Date Accessed: Mar. 2005).
`http://www.warpsolutions.com/Products/ProductsSpiderCache.php`

[94] Web Developers Notes Inc., "List of Browser Plugins." (Date Accessed: Jun. 2005).
`http://www.webdevelopersnotes.com/design/list_of_browser_plugins.php3`

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

[96] Woodruff, A., Aoki, P. M., Brewer, E., Gauthier, P., and Rowe, L. A., "An Investigation of Documents from the World Wide Web," in *Proceedings of the Fifth International World Wide Web Conference on Computer Networks and ISDN Systems*, pp. 963–980, May 1996.

[97] World Wide Web Consortium, "W3C Document Object Model," Jan. 2005. (Date Accessed Jan. 2006).
`http://www.w3.org/DOM`

[98] Xu, Z., Hu, Y., and Bhuyan, L., "Exploiting Client Cache: A Scalable and Efficient Approach to Build Large Web Cche," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 55–64, Apr. 2004.

[99] Yang, S. J., J. Nieh, M. S., and Tiwari, N., "The Performance of Remote Display Mechnisms for Thin-Client Computing," in *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[100] Yang, S. J., Nieh, J., Krishnappa, S., Mohla, A., and Sajjadpour, M., "Web Browsing Performance of Wireless Thin-Client Computing," in *Proceedings of the 12th International World Wide Web Conference*, May 2003.

[101] Yang, S. J., Nieh, J., and Novik, N., "Measuring Thin-Client Performance Using Slow-Motion Benchmarking," in *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001.

[102] Yin, L., Cao, G., Das, C., and Ashraf, A., "Power-Aware Prefetch in Mobile Environments," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[103] Zope Corporation, "Zope." (Date Accessed: Jan. 2007).
`http://www.zope.org`

# VITA

Tae-Young Chang was born in Seoul, Korea. He received his BE degree in Elecrtonic Engineering from Korea University in 1999, and his ME degree in Telecommunication System Technology from the same university in 2001. After that, he joined the PhD program of School of Electrical and Computer Engineering at Georgia Institute of Technology in 2001, and has worked with Prof. Raghupathy Sivakumar as a graduate research assistant in GNAN Research Group.