Improving the Performance of the World Wide Web over Wireless Networks

by

Todd B. Fleming

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

November 4, 1996
Blacksburg, Virginia

Keywords: wireless networks, WWW, MHSP, proxy, threads, protocols

Improving the Performance of the World Wide Web over Wireless Networks

Todd B. Fleming

(ABSTRACT)

The World Wide Web (WWW) has become the largest source of Internet traffic, but it was not designed for wireless networks. Documents with large inline images take a long time to fetch over low-bandwidth wireless networks. Radio signal dropouts cause file transfers to abort; users have to restart file transfers from the beginning. Dropouts also prevent access to documents that have not yet been visited by the user. All of these problems create user frustration and limit the utility of the WWW and wireless networks.

In this work, a new Wireless World Wide Web (WWWW) proxy server and protocol were developed that address these problems. A client based on NCSA Mosaic connects to the proxy server using the new protocol, Multiple Hypertext Stream Protocol (MHSP). The proxy prefetches documents to the client, including inline images. The proxy also reduces the resolution of large bitmaps to improve performance over slow links. MHSP provides the ability to resume file transfers when the link has been broken then reestablished.

The WWWW system was tested and evaluated by running script-controlled clients on different emulated network environments. This new system decreased document load time an average of 32 to 37 percent, depending on network configuration.

Acknowledgments

I thank my advisor, Dr. Midkiff, for recommending me to this project and for guiding me while I worked on the project and while I wrote this thesis. I also thank the rest of my committee for their efforts in the same. I thank the Center for Wireless Telecommunications and the Virginia Center for Innovative Technology for sponsoring this project and the CWT staff for their assistance. I thank Barry Mullins for his advice on testing issues. I thank the Software Productivity Consortium for supporting me before this project and the National Center for Supercomputing Applications for providing the source to Mosaic.

I thank my parents, Paul and Linda Fleming, and my brother, Brett, for their support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Problem

The World Wide Web (WWW) has become the largest source of Internet traffic [1]. It is used for research, education, entertainment, and commerce. It provides an easy-to-use interface and is built around several standards, including the Hypertext Markup Language (HTML) [3], which is used to format documents, and the Hypertext Transfer Protocol (HTTP) [2], which is used to transmit documents. However, it is not designed for wireless networks.

Several problems emerge when the Web is used over wireless networks. Documents with large inline images take a long time to retrieve over slow wireless links. Wireless links also fail due to radio signal dropouts. This can cause file transfers to abort; users have to restart file transfers from the beginning. Dropouts also prevent access to documents that have not yet been visited by the user. All of these problems create user frustration and limit the utility of the WWW and wireless networks.

1.2 Approach

In this work, a new Wireless World Wide Web (WWWW) proxy server and protocol were developed that address these problems. A client based on Mosaic [11] connects to the proxy server using the new protocol, Multiple Hypertext Stream Protocol (MHSP). The proxy prefetches documents to the client, including inline images. This improves data availability when the wireless link is broken. The proxy also reduces the resolution of large bitmaps to improve performance over slow links. MHSP provides the ability to resume file transfers when the link has been broken then reestablished. MHSP also improves on other HTTP shortcomings described in Chapter 2. In a set of experiments, this new system decreased document load time an average of 32 to 37 percent, depending on network configuration.

1.3 Thesis Organization

Chapter 2 describes the problem and discusses related work. Chapter 3 describes the functionality and architecture of the WWWW proxy. Chapter 4 describes the additions made to Mosaic. Chapter 5 specifies MHSP. Chapter 6 describes the testing procedure and results. Chapter 7 summarizes the work and includes suggestions for further research.

Chapter 2 - Background

2.1 Problem

The Hyper-Text Transfer Protocol (HTTP) is the standard protocol of the World Wide Web (WWW) [2]. It is designed to transfer hypertext documents that include inline images over the Internet. Its simple architecture allows it to be implemented quickly on both clients and servers, which facilitates the rapid growth of the Web. Its simplicity, however, comes with a price. HTTP transfers a single binary file per connection; a complex document with many inline images requires several HTTP connections. Also, it is a stateless protocol; Web servers are unable to maintain state information about clients which would allow certain optimizations such as prefetch caching and automatic resolution reduction. Likewise, lack of state information prevents HTTP from being able to resume a file transfer when a connection is broken and reestablished. Although these problems are bearable in a wired environment, they can become quite noticeable in a wireless system.

Wireless networking allows portable computers to communicate with each other and with desktop computers without tethering them. It also allows desktop computers to communicate with each other when network cabling is impractical. Unfortunately, wireless networking has its disadvantages. Users have to endure lower network bandwidth and frequent communication dropouts caused by radio cancellation and interference. This can create difficulties for many protocols, such as HTTP, which were not designed for the wireless environment.

Several problems emerge when HTTP is used over a wireless link. Many documents contain large inline images, which take a long time to transfer over low-bandwidth wireless networks. Dropouts caused by clients moving out of the network area can cause file transfers to abort. When a client returns to the network area, the file transfer has to be restarted from the beginning. Long-term dropouts also reduce data availability; users are usually restricted to documents that have been visited in the recent past. All of these problems affect when documents are available to users.

2.2 Prior Work

Researchers have taken several approaches to improve the performance of the Web in general and over wireless links specifically. These include protocol modifications, client modifications, and proxies. Some approaches are optimized for Personal Digital Assistants (PDAs).

Mogul and Padmanabhan discuss HTTP latency and throughput problems in [4]. They show that each document and inline image requires a minimum of two round trip times (RTT). This occurs because a new Transmission Control Protocol (TCP) connection is required for each file. In addition, there is processing overhead in TCP implementations for each connection. They also show that throughput is significantly reduced by TCP's slow-start mechanism. A 2-kilobyte file has a throughput of less than 10 percent of the best case. They propose additions to HTTP which would allow it to send several files over a single connection, thus avoiding these obstacles. Their proposal also utilizes pipelining; a client may request the next file before the current file has completed. Thus, there would be no gaps between consecutive files. They report that their changes reduce latency by more than half when transferring 2544-byte inline images.

Weerasinghe and Joshi describe an ongoing project in [5]. Mowser is a Web browser designed for mobile hosts. It addresses data rate limits by limiting the size of files it can retrieve. It also formats images for small screens by cropping, scaling, and dithering them. They have not published results yet.

Kaashoek, Pinckney, and Tauber describe two approaches that improve the performance of mobile Web browsers in [6]. First, they propose using dynamic documents - programs that generate content at the client. Second, they propose a caching and prefetch mechanism that relies on dynamic documents to form strategies. Unfortunately, their mechanism only prefetched an average of 0.5 objects for every object requested by the user with normal Web pages.

Bartlett describes a PDA-based wireless browser in [7]. It uses a proxy to fetch, parse, and separate Hypertext Markup Language (HTML) documents into multiple screens. The proxy sends the preformatted screens to the client using a new protocol. The client automatically prefetches the next screen while the user is viewing the current one. The client also caches the screens as it receives them. Although the system does not support images, they could be added in the future.

Gessler and Kotulla describe their PDA-based browser in [8]. It also uses a proxy to fetch, cache, and break down HTML documents. But unlike Bartlett's work, the proxy converts images into PDA-usable form (1-bit) and the client parses HTML. The client communicates with the proxy using a new protocol. To conserve bandwidth, images are not transferred until the user requests them.

Watson describes the Wit system in [9] and [10]. It is a mobile system that partitions applications and manages communication. It supports caching, prefetching, data reduction, and other performance-enhancing features. One of his example applications is the Wit WWW Browser (W*). It uses a proxy to prefetch documents ahead of the browser and reduce documents by outlining them. The client prefetches the first *n* links of the current document, but allows the user to specify which documents to prefetch instead. It also allows users to write scripts that search through the proxy's cache and send filtered data back to the client in outline form.

Mogul and Padmanabhan's proposal to fetch several files over a single connection using pipelining could be extended to fetch several files simultaneously if a new protocol is used. This would work well with Web clients that fetch several inline images at once. It would also prevent stalled files from holding up later files in the pipeline. Another useful extension would be the ability to resume file transfers when a connection has been broken. The approaches proposed by Weerasinghe and Joshi, by Bartlett, and by Gessler and Kotulla are geared towards PDAs. They do not prefetch whole documents and they format documents for small screens. The proposal by Kaashoek, *et al.*, to use dynamic documents to reduce bandwidth does not help with existing Web pages. A more opportunistic prefetch policy than they propose is needed to improve performance without using dynamic documents. Watson's proposal includes prefetching and content reduction by outlining. It does not use resolution reduction and it is not clear whether inline images for prefetched documents are sent.

Chapter 3 - WWWW Proxy

3.1 Introduction

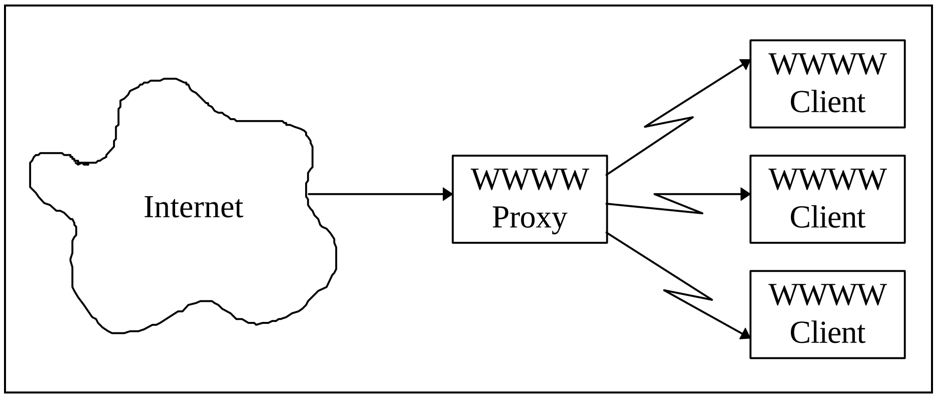

Figure 3.1 WWWW System

The WWWW proxy connects wireless clients to the World Wide Web in a way that is designed to improve performance (Figure 3.1). The proxy has the following features.

- It prefetches documents to clients to improve responsiveness and increase document availability when the wireless link is broken.
- It communicates with clients using MHSP (Chapter 5), which is designed for wireless networks.
- It maintains client state information, which allows it to make prefetch decisions based on client cache content.
- It reduces the resolution of large bitmaps when the wireless link is slow in order to speed up document fetches.
- It has a multithreaded design which prevents the resolution reduction operation from hindering other proxy functions from executing in a timely manner.

3.2 Overview

The proxy architecture is shown in Figure 3.2. It consists of an HTTP thread, which communicates with Web servers; a cache thread, which stores and parses files; a resolution reduction thread; and MHSP threads, which communicate with clients. It also has a graphical monitor for testing and demonstration purposes.

Clients connect to the proxy using MHSP. The proxy loads client state information for clients which have connected before. Whenever a client switches to a new document, its dedicated MHSP thread automatically sends the necessary files. It retrieves the files from the cache, which in turn retrieves files it has not yet received from the HTTP thread. The cache parses HTML files and sends dependency information to the appropriate MHSP threads, which use it to make prefetch decisions. The MHSP threads also decide when to send reduced images to clients. They request reduced files from the cache, which invokes the resolution reducer for files that have not yet been reduced.

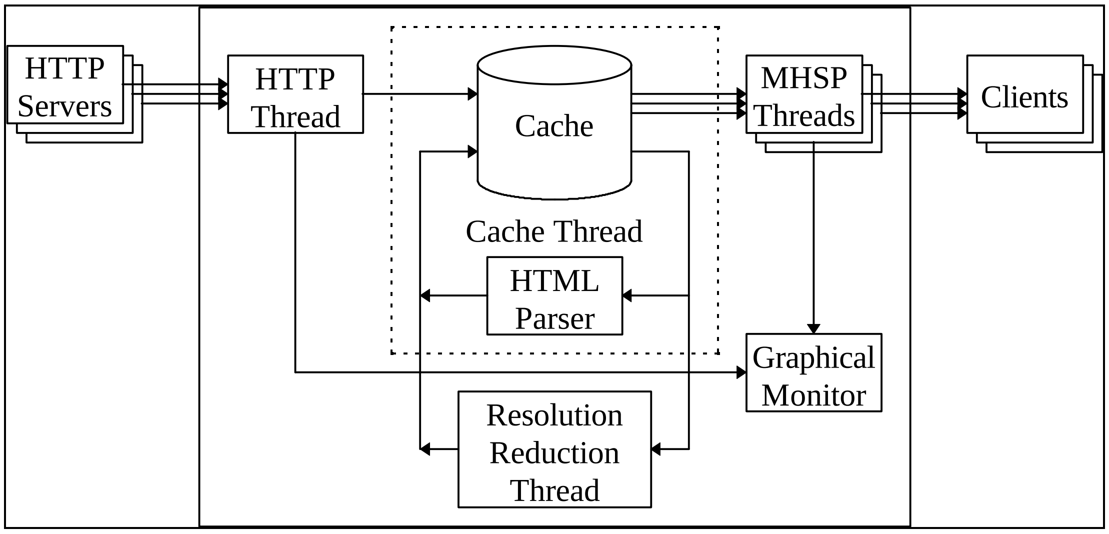

Figure 3.2 Proxy Architecture

3.3 Architecture

3.3.1 Job Management

The job management classes form the foundation of both the proxy and the client. They allow each thread to perform a set of operations in an event-driven manner. They also provide flexible inter-thread communication which is built on by the stream classes.

Most threads in the system are represented by a descendant of the JobManager class, which contains a set of objects derived from the Job class. Each job has a set of events that it acts on; the job manager calls on the appropriate job whenever an event is signaled. This allows a single thread to manage several asynchronous operations, such as file input/output (I/O). It also allows threads to communicate by signaling each other’s jobs.

Jobs are dynamically created and destroyed as needed. They can be created by any thread and posted to any other thread without deadlock. The job management and stream management classes, discussed in the next section, handle most inter-thread communication and synchronization, allowing the higher-level classes to easily communicate with each other.

3.3.2 Stream Management

The stream management classes provide low-level support for inter-thread and inter-process communication. Their descendants provide most of the functionality of the system.

The StreamJob class (derived from Job) manages a single file or socket. Its use of overlapped I/O allows it to simultaneously read and write several buffers to the same file handle. It also includes optimizations that transfer data directly between buffers, preventing unnecessary reads. This feature is used by the cache, which may have one or more clients retrieving a file that is being fetched by HTTP.

The StreamConnection class transfers data between two StreamJobs, which are usually in different threads. It uses a multiple-buffer scheme which allows one side to send data while the other side is concurrently processing data. The overlapped I/O operations in StreamJob operate directly on StreamConnection buffers. Classes derived from StreamConnection manipulate buffers when they move between the two sides of a connection. This is used to parse and transform data.

The StreamManager class (derived from JobManager) manages several StreamJob classes and helps StreamConnection objects connect to StreamJob objects. Classes derived from StreamManager perform high-level management functions, such as cache management and HTTP connection management.

3.3.3 HTTP Interface

The HTTP classes allow the proxy to make several simultaneous connections to HTTP servers. They currently support HTTP 1.0 [2]. HTTPJob (derived from StreamJob) establishes and maintains a single HTTP connection. It also sends a file request when a connection is made. HTTPParse (derived from StreamConnection) separates and parses the header from the received data and transfers the result to the cache. HTTPManager (derived from StreamManager) manages several active connections from a single thread. No classes are derived from these.

3.3.4 Cache

The cache is the centerpiece of the proxy. In addition to storing retrieved data, it connects the other parts of the system together and is involved in prefetch decision making. Part of it is also used by the client. To facilitate reuse, the cache is broken up into two sets of classes. The base set provides file caching. The derived set provides proxy-specific interface and control.

3.3.4.1 Base Cache

The cache as a whole is managed by the Cache class (derived from StreamManager) within a single thread. It maintains the cache state and discards least recently used files to make space when needed. It saves the state when the program shuts down and reloads it when the program is restarted. This provides cache persistence for both the proxy and client.

Individual cache files are managed by instances of CacheJob (derived from StreamJob). It opens a cache file and provides overlapped I/O operations on the file for StreamConnection objects. It automatically creates a file if it does not exist and stores information about the file, such as size, attributes returned from the HTTP parser, and whether the file has been completely received. Derived classes automatically open connections to a source when the file is either new or has not been retrieved completely.

3.3.4.2 Proxy Cache

The proxy's cache is managed by the ProxyCache class, which is derived from Cache. It provides additional support functions for ProxyCacheJob.

ProxyCacheJob, derived from CacheJob, interfaces to the rest of the system. If its file is not present and reduction is not required, ProxyCacheJob automatically establishes a stream to HTTP to retrieve it. If its file is to be reduced, but is not present (reduction is requested by MHSP threads), it creates a stream to the resolution reduction thread, which requests the unreduced file from the cache. Both reduced and unreduced versions of files are stored in the cache.

ProxyCacheJob also adds prefetch support. Whenever it requests a new file from HTTP, it also launches the HTML parser within the cache thread. The parser collects dependency and prefetch information if the file is an HTML document. ProxyCacheJob stores the information in its cache state and signals the appropriate MHSP threads to send the files to their clients. The MHSP classes make the final decision on which files to send. MHSP may also may request dependency information without using a stream. This occurs when a client switches to a document that is already present in the cache.

3.3.5 MHSP Interface

The MHSP interface connects the proxy to one or more clients using the protocol defined in Chapter 5. It also decides which files to send to clients for prefetching and whether to reduce the resolution of bitmap images. Like the cache, these classes are also used by the client and are partitioned to support reuse. The base classes implement the MHSP protocol and the derived classes provide proxy-specific behavior.

3.3.5.1 Base MHSP

The MHSPJob class (derived from StreamJob) maintains a socket connection, processes inter-thread streams, and manages MHSPSender and MHSPReceiver objects. It determines when to send messages based on stream state and priority, but does not assign priorities. It is not responsible for establishing the socket connection and it is unable to process streams initiated by the other side of the connection.

MHSPSender (derived from StreamConnection) sends MHSP messages to the other side of the socket. It works with MHSPJob to maintain stream states.

MHSPReceiver (derived from StreamConnection) receives MHSP messages. It also works with MHSPJob to maintain stream states.

3.3.5.2 Proxy MHSP

MHSPServerJob extends MHSPJob to create internal connections to the cache for outgoing MHSP streams. It also notifies MHSPServerManager of state changes.

MHSPServerManager (derived from StreamManager) represents a proxy thread that manages a single client connection, represented by an MHSPServerJob object. It actively gathers state information from the client's cache and dependency information from the proxy's cache in order to make prioritized prefetch decisions. It maintains a list of active streams and terminates them if it determines that they are no longer needed. It also decides when to reduce the resolution of bitmaps because of limited network throughput to the client. MHSPServerManager saves the client's state information when the connection is broken and reloads it when the client identifies itself in a new connection.

The acceptConnections function, which runs in a dedicated thread, waits for clients to connect to the proxy. When a client connects, it creates a new socket, a new thread, and a new MHSPServerManager object to service the client.

3.3.5.3 Priority System

The MHSP classes use a priority system (Table 3.1) to control which files to send to clients. Candidate files are assigned different priorities for each client. A priority value is composed of four signed integers, action, class, I, and J, with a greater value corresponding to a higher priority. Action indicates which user action triggered the file and is the most significant. Each time the user switches documents, the current action is incremented by 1 and the new document gets the new action number. Class indicates the importance of the file relative to the current document. I and J are position values; J is the least-significant integer.

Table 3.1 Priority System

Value (A=action #)	**Description**
A.5.0.0	Background image for document A
A.4.0.0	Document A
A.3.I.0	Inline images • I≤0, I = - position within document A
A.2.I.0	Prefetch documents referenced by A • I≤0, I = - link position within document A
A.1.I.J	Prefetch inline images • I≤0, I identifies prefetch document • J=0: Background image • J<0: J = - 1 - position within prefetch document

3.3.6 Resolution Reducer

The resolution reducer reduces the resolution of Graphics Interchange Format (GIF, version 89a [12] or earlier) files to be sent to the client. It reduces images to half of their original size in each direction. Since GIF files are often interlaced, arbitrary reduction ratios would require that the entire file be retrieved before reduction could begin. This would increase both latency and memory requirements. Images with a width smaller than 100 pixels or a height smaller than 50 pixels are not reduced.

The reduction is performed by several classes. GIFManager (derived from StreamManager), which runs in a single thread, creates a GIFJob for each image. GIFJob (derived from StreamJob) manages the interaction between the decoder and encoder. GIFDecoder (derived from StreamConnection) retrieves the original file from the cache, decompresses it, and sends the reduced bitmap to the encoder. GIFEncoder (derived from StreamConnection) compresses the reduced bitmap and sends it back to the cache for storage. The reduction process can take place at the same time the original file is being received and the reduced file is being transmitted.

3.3.7 Graphical Monitor

The graphical monitor tracks and regulates certain proxy performance parameters. It monitors aggregate throughput of incoming data from HTTP connections and outgoing data through MHSP connections. It limits the total outgoing data capacity of MHSP connections in order to emulate a wireless environment. The user may either set a fixed data rate or select two data rates and enter probabilities that the channel is at either of the two rates or at 0 kilobits per second. The monitor then randomly selects the data rate each second to emulate a dynamic channel. MHSPSender objects work with the monitor to limit the data rate.

3.4 Operation

3.4.1 Fetch and Prefetch

Document fetch and prefetch are controlled by MHSPServerManager with the help of the cache. The system is designed to automatically fetch and send all the files needed to display a document and prefetch linked documents to one level deep. This process increases document availability when the user is disconnected.

Whenever a client switches to a different document, indicated by an MHSP status message, MHSPServerManager queries the cache for all the document’s dependencies and assigns priorities. Whenever a file qualifies for more than one priority, such as when an inline image is in several prefetch documents, the file gets the greater of the priorities. If the dependencies are not yet available, the cache automatically sends the dependencies as it receives files.

MHSPServerManager attempts to send the top five unsent files for the current action. Whenever the number of open streams for the current action drops below five, MHSP starts sending enough of the highest-priority unsent files to bring the number back up to five. The number of files may grow above five if the client requests a file that the proxy is not currently sending. The requested file is assigned priority A.4.0.0. Files may have identical priorities. When all of the files for the current action have been sent, MHSPServerManager starts sending files from the previous action using the same rules. MHSPServerManager cancels outgoing streams for files that are two actions old. It does not send any files when the transfers are complete for both the current and previous actions.

When two or more open streams have data available to send, MHSPServerJob sends data for the file with the highest priority. This prevents the prefetch mechanism from delaying files that the client needs first. This also allows the prefetch mechanism to take advantage of available bandwidth to the client if a high-priority file is arriving at the proxy at a slow rate.

3.4.2 Resolution Reduction

Resolution reduction is controlled by MHSPServerManager. Whenever the estimated utilization of available bandwidth to the client for the past 2 seconds is 80 percent or greater, MHSPServerManager requests the reduced version of bitmaps for new streams from the cache. This allows the proxy to more effectively utilize the link. Since reducing compressed images is a CPU-intensive operation, the utilization may fall, causing new streams to use unreduced bitmaps. The process is self-regulating and is affected by the presence of other clients since both the network and the server processor are shared.

MHSPServerManager estimates utilization of available bandwidth by tracking the percentage of time MHSPSender is writing data to its socket. This measurement accounts for both available bandwidth, which is affected by wireless properties and utilization by other computers, and data availability. It does not account for the buffering provided by the operating system.

The cache stores both original and reduced versions of images. Clients benefit when they request images on a slow link that have already been reduced for previous clients.

3.5 Summary

The WWWW proxy connects wireless clients to the World Wide Web in a way that is designed to improve performance. It communicates with clients using MHSP, maintains client state information, prefetches documents, and reduces the resolution of large bitmaps. It also resumes interrupted file transfers. These features require both a new proxy design and a modified client design.

Chapter 4 - WWWW Client

4.1 Introduction

The WWWW client software is a modification of NCSA Mosaic 2.0 [11]. It reuses several components of the proxy to speed development. There is also a baseline version for performance comparison. The client has the following features.

- It uses MHSP to communicate with the proxy. The baseline version uses HTTP to contact Web servers directly.
- It fetches several files at once from the proxy. The baseline version fetches one file at a time.
- It sends cache state information to the proxy. The baseline version does not.
- It allows the proxy to prefetch documents to it. The baseline version has no prefetch support.
- It maintains a connection to the proxy over long periods of time. Since it uses HTTP, the baseline version connects to Web servers for every file retrieved.

4.2 Overview

The client architecture is shown in Figure 4.1 and the baseline is shown in Figure 4.2. Both are based on Mosaic, with the addition of several components. The fetch thread, MHSP or HTTP, fetches files requested by the cache. The MHSP thread also receives files from transfers that the proxy initiates. The cache is able to receive multiple files simultaneously from the MHSP thread and supports proxy-controlled prefetch caching. The client API module provides a transition between the multithreaded architecture and Mosaic's single thread. The client plug replaces Mosaic's HTTP module. The plug and API make the cache appear to Mosaic as a set of external servers.

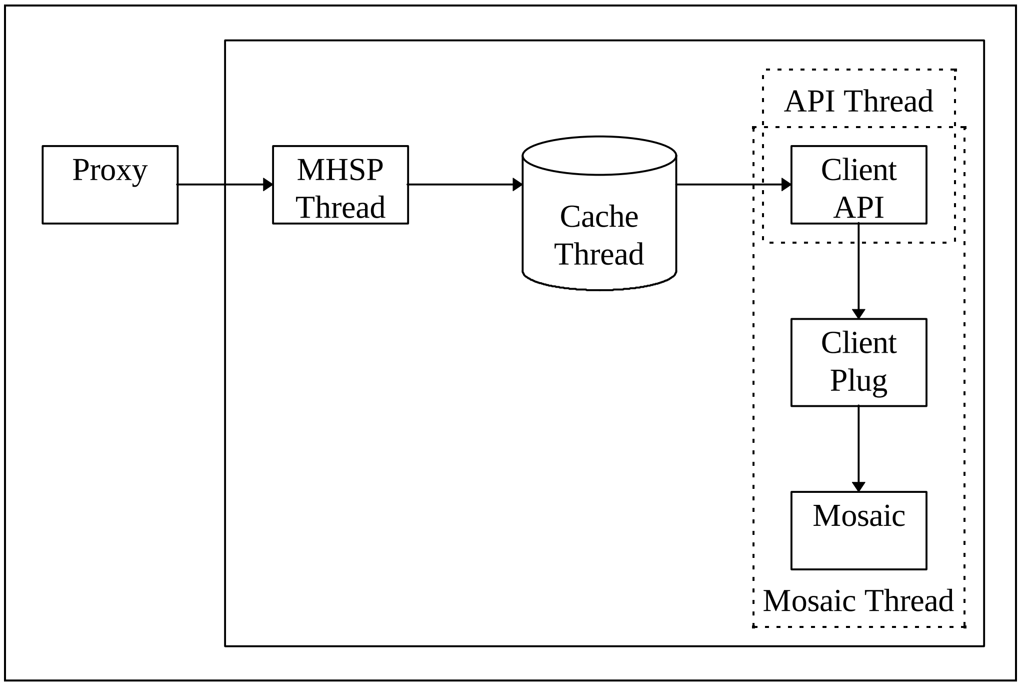

Figure 4.1 Client Architecture

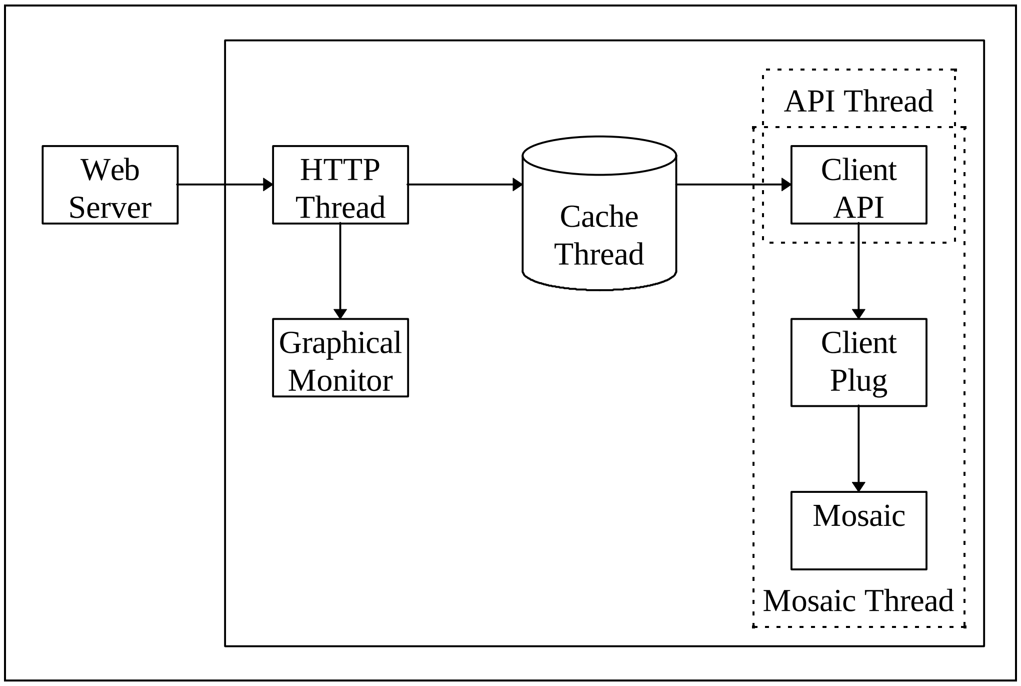

Figure 4.2 Baseline Client Architecture

4.3 Architecture

4.3.1 Cache

The client's cache is managed by the ClientCache class, which is derived from Cache (Section 3.3.4.1). It sends status messages to the proxy to aid prefetch caching and it keeps the client's unique identifier, which is assigned by the proxy.

ClientCacheJob, derived from CacheJob, manages a single cache file. It automatically requests its file from the fetch thread if the file is not present or incomplete.

4.3.2 Fetch Thread

MHSPClientManager (derived from StreamManager) and MHSPClientJob (derived from MHSPJob) interface the client to the proxy. They work with the cache to identify the client and send state information to the proxy. They also accept prefetch files sent by the proxy. The connectionThreadMain function continually tries to reconnect to the proxy whenever the link is broken.

The baseline client uses the same classes as the proxy for HTTP access. They are unable to resume file transfers when the link is broken and have no prefetch support.

4.3.3 Client API

The client API classes provide a bridge between the multithreaded environment and Mosaic's single threaded architecture by emulating non-blocking socket operations, which are used by Mosaic to keep the user interface alive during file transfers. The ClientStreamManager class (derived from StreamManager) has a single thread which communicates with the cache. It creates a single ClientStream (derived from StreamJob) object to fetch a file from the cache. Both classes operate on a dummy socket handle provided by Mosaic. When the Mosaic thread requests data that is not immediately available, its message loop is reentered. The API indicates completion by posting a message to the Windows message queue. This architecture allowed minimal changes to be made to Mosaic's source code.

4.3.4 Client Plug

The function HTLoadMHSP is a modification of Mosaic's HTLoadHTTP function. It utilizes the client API to retrieve data from the cache then reformats it. Since neither the new HTTP or MHSP modules support the post operation, it passes these requests back to HTLoadHTTP. This allows the client to handle post-method forms.

4.3.5 Graphical Monitor (Baseline Client Only)

The graphical monitor is the same as the proxy’s, except it operates in a different mode. Instead of regulating MHSP, it regulates HTTP. This allows the baseline client to emulate operation in a wireless environment. The full client does not include the monitor since the proxy’s monitor regulates the connection.

4.4 Summary

Both the WWWW client and the baseline client are modifications of NCSA Mosaic. They use the same source code to cache files and interface to Mosaic’s user interface. They use different modules to interface to the Web. The WWWW client uses MHSP to connect to the proxy and the baseline uses HTTP to connect directly to Web servers. The WWWW client retrieves several files simultaneously and supports prefetching. The baseline client does not do either. The WWWW client supports long term connections, unlike the baseline client, which connects to a server for every file retrieved.

Chapter 5 - Multiple Hypertext Stream Protocol (MHSP)

5.1 Introduction

MHSP is a full-duplex, symmetric protocol that the proxy uses to communicate with wireless clients. It is designed to reduce latency while allowing the client to take advantage of available bandwidth. MHSP has the following features.

- It maintains a connection over long periods of time, reducing the number of TCP round trip times required to initiate and tear down file transfers.
- It transfers more than one file simultaneously over a single connection, which also reduces setup and tear-down time.
- It transfers state information between the proxy and clients, allowing the proxy to make prefetch decisions.
- It allows both clients and proxy to initiate file transfers. Clients start transfers based on user requests and the proxy starts transfers for prefetching.
- It allows both clients and proxy to stop file transfers, for the reasons stated above.
- It tracks the progress of file transfers, allowing stopped transfers to be resumed at a later time.
- It uniquely identifies a client when a connection is made, allowing the proxy to store long-term state information.
- It maintains compatibility with existing networks by running on top of TCP.

5.2 Overview

When a connection is made between client and server, both sides identify the protocol being used and negotiate a unique identification number for the client. Once the connection is established, either side may initiate a file transfer by sending a message. The message indicates the file to be transferred, the direction of the transfer, and the position to start the transfer if the request is made by the destination. If the request is from the source, it also contains a unique file identification number and may contain file attributes and file content. Subsequent source messages usually carry file content. Either side may terminate the file transfer with a cancel or end-of-file message. The protocol allows several file transfers to be active simultaneously.

Both sides also send status messages. These messages may indicate that a user switched documents or that the cache state has changed. Messages may also be used to query the client on its cache state.

5.3 Format

All integer values in this protocol are unsigned little-endian. Fields are byte-aligned.

5.3.1 Session Header

The session header (Table 5.1) identifies the protocol and version at connection setup and is used to negotiate a session identifier (ID) to identify the client. The session header is followed by zero or more messages. This session identifier is 64 bits in order to minimize the probability that one client can pick another client's ID.

Table 5.1 Session Header Format

Field	Size (bits)	Description
Protocol ID	32	Protocol identification: 'MHSP'
Minor version	8	Currently 9
Major version	8	Currently 0
Session ID	64	Negotiated session identifier

5.3.2 Message Header

Each message begins with a message header (Table 5.2). It contains the stream identifier, a status code, and optionally a file identifier or position.

Table 5.2 Message Header Format

Field	Size (bits)	Description
Stream ID	16	Stream identification
Status	8	Status of the stream if stream ID $\neq 0$
		Message type if stream ID $= 0$
File ID	32	File identifier
File Position	32	Offset to start file transfer

The stream identifier uniquely identifies which stream the message is a part of, or indicates a status message when it is zero. It is 16 bits so it does not impose a practical limit on the number of files being transferred. The high bit of the stream identifier indicates which side of the connection initiated the transfer (0: server, 1: client) for stream messages. Its purpose is to guarantee that both sides do not initiate transfers with the same identifier. The low bit of the stream identifier identifies whether the stream is initiated by the source (0) or sink (1). It is the responsibility of both sides to store the direction of transfer after the initial message is sent; both sides will use the same identifier when sending messages for the stream.

The status code (Table 5.3) indicates the stream is continuing, marks the end of the file, or cancels the stream if the message is a stream message. It is only 8 bits because few codes are defined. If the message is a status message, then the status code indicates the message type (Table 5.4). The file identifier field is present in status messages when the high bit of the status code is 1.

The file identifier field is present in the first message sent by the source in the stream and in some status messages. It is assigned by the source as a way to identify files in status messages. It is 32 bits so it does not place a practical limit on the number of files in the system. There are two file identifier spaces since files can travel in either direction. The space can be inferred by the message's use.

The file position field is present in the first message sent by the sink in the stream if the message initiates a transfer. It indicates the position in bytes within the file that the source should start sending. It is 32 bits to allow files larger than 64 kilobytes.

The header is followed by one or more message blocks.

Table 5.3 Stream Status Codes

Code	Value	Description
OK	0	Stream is continuing
EOF	1	Message marks end of file
Terminate	2	Message terminates stream
Duplicate	3	Message terminates a duplicate stream

Table 5.4 Status Message Codes

Code	Value	To	Description
Switch document	0	Proxy	Client is switching to document identified by URL block
Query file	80h	Sink	Query status of cached file
Not present	81h	Source	File is not present in cache
Received EOF	82h	Source	File received
Received partial	83h	Source	File partially received. Parameter block contains a 32-bit integer specifying length received
Remove file	84h	Sink	Delete cache file
Removed file	85h	Source	File was deleted

5.3.3 Message Block

Each block begins with a block header (Table 5.5). It identifies the block type (Table 5.6) and length (up to 64 kilobytes, including header bytes). Except for the file content block and end block, blocks do not have to be in any specific order. Each block type may appear only once in a message.

Table 5.5 Block Header Format

Field	Size (bits)	Description
Type	8	Block type
Length	16	Length of block including header

Table 5.6 Block Type Codes

Type	Value	Description
URL	0	Universal Resource Locator
Attributes	1	File attributes
Content	2	File content
Parameter	3	Message parameters
End	15	End of message

5.3.3.1 URL Block

The URL block identifies the file being transferred. It is sent once by the side that initiated the transfer. Its contents contain the file’s URL (not terminated) in complete form, including protocol (case-insensitive, normally HTTP), server, port number (usually 80), and file (at least “/”). The URL block is also used in the “switch document” message.

5.3.3.2 Attributes Block

This block contains file attributes defined by the HTTP protocol, but in a different format. It is sent at most once by the stream source. A message that contains this block may contain a content block, but no messages in the same stream with content blocks may precede a message with the HTTP header block.

The block header is extended (Table 5.7) to contain HTTP server information for the file. The result code is large enough for the codes defined in [2]. The block also contains 0 or more name-value pairs after the header. Each name and value is null-terminated.

Table 5.7 Attributes Block Header Format

Field	**Size (bits)**	**Description**
Type	8	Block type
Length	16	Length of block including header
Minor version	8	Minor version of HTTP server that filled request
Major version	8	Major version of HTTP server that filled request
Result code	16	Result code returned by HTTP server

5.3.3.3 File Content Block

This block contains file content sent by the source. This block may be present only if the message status code is OK or EOF and may only appear just before the end block. The block header is extended (Table 5.8) to contain the position within the file that the contents fill.

Table 5.8 File Content Block Header Format

Field	**Size (bits)**	**Description**
Type	8	Block type
Length	16	Length of block including header
Position	32	Position of content within file

5.3.3.4 Parameter Block

This block contains parameters for status messages. It is currently only used for the “received partial” message.

5.3.3.5 End Block

The end block is the last block of a message. It contains no data.

5.4 Operation

5.4.1 Establishing a Connection

The client initiates a connection to TCP port 12345 on the proxy (eventually, this port number could be changed to be in the system services range). Both ends send the protocol identifier and protocol version (see Table 5.1). Next, they verify that the received protocol identifier is correct and adapt to the lowest protocol version for backward compatibility. The client then sends its session identifier, or 0 if it is connecting for the first time. If the identifier is valid, the proxy sends it back. Otherwise, the proxy sends a new identifier and the client clears its cache and stores the identifier for future connections.

5.4.2 Updating Cache Status

After a connection is established, the client should notify the proxy of any files it deleted from its cache since the last connection. The proxy may also query the client on the status of files for which it has not received acknowledgments from the previous connection. This may be necessary when a connection was abnormally terminated because of a radio signal dropout. The proxy will not initiate any transfers of files that it has queried about that the client has not responded to yet. The proxy will, however, send files that the client requested before receiving a client status message.

While the connection is established, the client should notify the proxy whenever it deletes a cache file with a “removed file” message. The client should also send a cache status message whenever a stream is terminated by either side.

5.4.3 Tracking User Actions

The client sends a “switch document” message whenever the user moves to a document, regardless of whether the document is in its cache. This allows the proxy to make prefetch decisions. The proxy automatically sends all files needed to display the document that are not already in the client’s cache. If the client does not receive a needed file automatically, it initiates the proper file transfer. This can occur if the client fails to notify the proxy of a deleted file or the proxy fails to detect a dependency.

5.4.4 Initiating File Transfer

The side initiating the transfer must send a message with a unique stream identifier and the file's URL. If the transfer is initiated by the source, the message header also contains the file’s identifier. If it is initiated by the sink, the header contains the offset for the transfer to begin. A source-initiated message may also contain file attributes and file content. The message status code for new transfers must be either OK (source or sink) or EOF (source only). The other end recognizes that the stream is new by the stream identifier.

5.4.5 Sending a File

The source keeps sending messages with file content while the transfer is not finished. The source controls priority when several files are being sent.

5.4.6 Terminating a Stream

The source may terminate a stream by sending a message with a status code of EOF**,** terminate**,** or duplicate. The sink may terminate a stream by sending message with a terminate status code. The end receiving the message must also send a termination message if it has not already. The stream is considered active until both sides have sent termination messages. After either end sends a termination message, the sink will automatically send a status message for the file. A stream identifier can be reused after the stream has been terminated.

5.4.7 Handling Duplicate Streams

If both sides initiate streams with the same URL in the same direction, the source is expected to terminate the stream initiated by the sink with a duplicate message code.

5.4.8 Terminating a Connection

Either side may terminate the connection by closing the socket. The connection may also be terminated by a failure at the TCP level. Incomplete messages should be ignored. All streams are considered terminated at this point.

5.5 Implementation

The client and proxy use the same source code for most of the MHSP protocol. This is possible because of the protocol's symmetry. They use different source code to establish connections and handle the transferred data. There is no current support to transfer files from client to proxy. The client uses HTTP to handle post-method forms. The client does not keep track of files that were deleted from its cache when it is not connected to the server. Instead, it automatically requests files it needs that the proxy does not deliver automatically. Neither the proxy or server implement the "remove file" message.

5.6 Summary

The WWWW proxy uses MHSP to communicate with wireless clients in order to reduce latency and to take advantage of available bandwidth. MHSP allows the proxy and client to maintain a connection over long periods of time, transfer more than one file simultaneously, transfer state information, and resume partially sent files.

Chapter 6 - Testing Procedure and Results

6.1 Overview

The WWWW system was tested and evaluated by running script-controlled clients on different emulated network environments. The scripts were generated by observing users complete given tasks. The graphical monitor was used to emulate wireless environments. Both the enhanced client and baseline client were tested and compared. The enhanced client loaded documents in 32 to 37 percent less time on average than the baseline, depending on the network configuration.

6.2 Testing Procedure

Scripts were derived from user actions. Four Center for Wireless Communications graduate students were asked to complete a different task (Table 6.1), starting from the text version of the Virginia Tech (VT) home page. They were asked to only use pages fetched by HTTP, since the system does not yet support other Web protocols. They were also asked to not use image maps because they do not work properly with the current client. They were not permitted to use search tools since that would make the tasks too short for good test scripts. The links followed by each user were recorded and later compiled into a list of URLs. The scripts (Table 6.2) were truncated to 18 documents, including duplicates, so testing could proceed at a reasonable pace. Appendix A contains a list of the URLs visited for each script.

Table 6.1 Tasks Assigned to Users

1	Find the Hybrid Electric Vehicle Team overview
2	Find the list of campus computer lab hours
3	Find the Video Broadcast Service's home page
4	Find the email address of VT's Chief of Police

Table 6.2 Script Properties

Script	# Documents	Truncated
1	18	Yes
2	7	No
3	18	No
4	18	Yes

Two machines were used in testing. A 90 MHz Pentium with 32 MB of memory running Windows NT 3.51 was used for the proxy server. A 100 MHz Pentium with 16 MB of memory running Windows 95 was used for the client. Both machines had two Ethernet cards. One card in each machine connected to Virginia Tech’s campus network. The other card in each machine connected to an internal laboratory network. The enhanced client software was set to communicate with the proxy through the laboratory network. This emulates how clients would communicate with the proxy over a wireless link. The proxy used the campus network to access Web servers. The baseline client also used the campus network.

The graphical monitor was used to emulate different operating environments. It limited incoming HTTP data to the baseline client and outgoing MHSP data from the proxy to 1000 kilobits per second (kbps), 100 kbps, 20 kbps, or a variable rate. The variable rate emulates a dynamic wireless channel. The monitor randomly chose from 100 kbps, 20 kbps, or 0 kbps each second with probabilities of 0.14, 0.30, and 0.56, respectively. This provides an average capacity of 20 kbps, for comparison with the constant 20 kbps rate.

Both clients had separate cache directories with a size limit of 5 MB. The proxy had a cache size limit of 32 MB. Only one client was run at a time during testing to prevent network interaction.

Both clients executed scripts to conduct testing. They loaded each page in the script in order. After a page was loaded, the running client waited ten seconds before starting the next page load. Upon completion, the client reported the total time spent loading pages. Wait time was not included in the total.

Each script was run through each network configuration and client in the order shown in Table 6.3. The proxy and client caches were cleared between each pair of runs. This was permissible since the clients had separate cache directories and the baseline client did not access the proxy's cache. A pretest run was completed before testing began on each script to make sure all servers were operational. The results of the pretest were not recorded so the first recorded run of each script would have the same advantage as the other runs. Since some Web servers cache their content, this first run may be at a disadvantage. If a document timed out during a script run, the caches were cleared and the run repeated. This occurred when Web servers did not accept an HTTP connection.

Table 6.3 Test Execution Order

	1000 kbps		100 kbs		20 kbps		Variable	
Script	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP
1	1	2	3	4	5	6	7	8
2	9	10	11	12	13	14	15	16
3	17	18	19	20	21	22	23	24
4	25	26	27	28	29	30	31	32

6.3 Results

The total load times for each run are shown in Table 6.4. The average total load time, standard deviation, and 90 percent confidence intervals are shown in Table 6.5, Table 6.6, and Table 6.7, respectively. The average load time per page is shown in Table 6.8. The results from the different scripts are combined in Table 6.9, which shows the average improvement of the new system for each of the configurations.

Table 6.4 Total Load Time (seconds)

		1000 kbps		100 kbs		20 kbps		Variable	
Run	Script	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP
1	1	57.8	34.7	55.2	40.0	151.1	106.1	217.2	147.3
	2	44.9	18.2	47.7	24.3	93.8	56.0	147.1	72.6
	3	34.0	22.8	40.0	19.7	103.6	73.0	130.4	116.3
	4	21.1	14.6	24.0	16.6	77.6	51.5	70.2	78.4
2	1	49.9	33.6	72.1	37.7	160.2	102.8	230.6	130.7
	2	41.7	18.8	44.1	35.9	91.8	42.2	125.0	81.2
	3	34.5	20.9	35.2	17.6	106.3	75.1	136.2	116.1
	4	29.9	14.6	26.2	18.4	76.6	51.1	97.4	95.9
3	1	49.7	46.2	57.0	42.3	152.7	101.8	251.0	184.6
	2	22.9	13.7	27.7	21.2	73.8	39.1	84.9	59.1
	3	49.1	65.3	52.9	34.6	109.5	91.7	112.0	97.4
	4	20.5	12.8	23.7	18.2	75.6	57.8	92.8	89.6
4	1	52.7	37.0	62.5	42.2	153.7	119.4	178.4	157.5
	2	22.5	14.1	26.9	20.5	75.5	39.6	133.5	70.5
	3	70.8	39.9	61.4	32.5	107.1	81.2	159.9	109.1
	4	23.4	12.9	24.7	13.7	85.6	53.0	90.9	86.1
5	1	81.8	56.1	119.9	70.6	154.6	107.1	169.0	120.5
	2	22.6	14.5	25.0	21.7	73.7	39.9	128.9	54.9
	3	31.8	27.6	40.4	20.5	97.1	89.7	135.2	80.8
	4	29.1	13.3	23.1	22.3	74.4	54.1	101.3	63.8

Table 6.5 Average Total LoadTime (seconds)

	1000 kbps		100 kbs		20 kbps		Variable	
Script	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP
1	58.4	41.5	73.3	46.6	154.5	107.4	209.2	148.1
2	30.9	15.9	34.3	24.7	81.7	43.4	123.9	67.7
3	44.0	35.3	46.0	25.0	104.7	82.1	134.7	103.9
4	24.8	13.6	24.3	17.8	78.0	53.5	90.5	82.8

Table 6.6 Total Load Time Sample Standard Deviation (seconds)

	1000 kbps		100 kbs		20 kbps		Variable	
Script	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP
1	13.5	9.5	26.8	13.6	3.5	7.0	34.8	24.9
2	11.4	2.4	10.7	6.4	10.2	7.2	23.3	10.6
3	16.5	18.3	10.8	7.9	4.8	8.4	17.1	15.0
4	4.4	0.9	1.2	3.1	4.4	2.7	12.1	12.3

Table 6.7 Total Load Time 90% Confidence Interval (seconds)

	1000 kbps		100 kbs		20 kbps		Variable	
Script	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP
1	± 9.9	± 7.0	± 19.7	± 10.0	± 2.5	± 5.2	± 25.6	± 18.3
2	± 8.4	± 1.8	± 7.9	± 4.7	± 7.5	± 5.3	± 17.2	± 7.8
3	± 12.1	± 13.5	± 8.0	± 5.8	± 3.5	± 6.2	± 12.6	± 11.1
4	± 3.3	± 0.7	± 0.9	± 2.3	± 3.3	± 2.0	± 8.9	± 9.1

Table 6.8 Average Load Time Per Page (seconds)

	1000 kbps		100 kbs		20 kbps		Variable	
Script	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP	HTTP	MHSP
1	3.2	2.3	4.1	2.6	8.6	6.0	11.6	8.2
2	4.4	2.3	4.9	3.5	11.7	6.2	17.7	9.7
3	2.4	2.0	2.6	1.4	5.8	4.6	7.5	5.8
4	1.4	0.8	1.4	1.0	4.3	3.0	5.0	4.6

Table 6.9 Aggregate Average Load Time Per Page (seconds)

	1000 kbps	100 kbs	20 kbps	Variable
HTTP	2.9	3.2	7.6	10.5
MHSP	1.8	2.1	4.9	7.1
Improvement	37%	34%	35%	32%

6.4 Discussion of Results

The new system performed better on average than HTTP for all tested configurations. The 1000 kbps data rate shows the largest improvement because the client has a higher probability of receiving pages through prefetch caching before they are needed. The 20 kbps and variable data rates also show significant improvement because of image resolution reduction. MHSP's long-term connection and simultaneous file transfer mechanisms aid performance for all of these configurations. This data demonstrates that the system performs well on both wired (1000 kbps) and wireless (100kbps, 20kbps, and variable data rate) networks.

6.5 Summary

The system was tested by running script-controlled clients through different emulated network environments. Both the enhanced client and baseline client were tested and compared. The enhanced client loaded documents in 32 to 37 percent less time on average than the baseline, showing improvement on both wired and wireless networks.

Chapter 7 - Summary and Conclusions

A system was developed that improves the performance of the World Wide Web over wireless links. It consists of a proxy server and a modified client and uses a new protocol, MHSP. The client maintains a long-term connection to the proxy and transfers several files simultaneously to overcome limitations with TCP. The proxy maintains state information on the client, so it can make prefetch decisions and resume partially completed file transfers. The proxy prefetches documents to the client, which improves performance over high bandwidth links, and improves document availability when the connection is broken due to radio signal dropouts. The system also resumes partially completed file transfers when the network link has been broken and reestablished. The proxy reduces the resolution of large bitmaps, which improves performance over low bandwidth wireless links. This system reduces document load time by 32 to 37 percent when compared to the use of HTTP.

Several options may be explored to further improve the performance of the WWW over wireless links. Text and HTML files could be compressed by the proxy to reduce transmission time. The image resolution reduction criteria, which includes the estimated network utilization trigger point and the minimum bitmap size, could be chosen using empirical data. The GIF encoder in the resolution reduction code could be replaced with an encoder which uses a format with a higher compression ratio. TCP could be replaced with another transport protocol that does not invoke exponential backoff procedures when packets are lost due to errors on the wireless channel. The transport layer could also be modified to provide performance indicators to the proxy, which would use them to decide when to reduce bitmaps. These suggestions, when combined with this work, should make the Web run well on computers connected with wireless networks.

References

1) “NSFNET Backbone Traffic Distribution by Service,” Merit Network Information Center Services, April 1995, http://www.cc.gatech.edu/gvu/stats/NSF/9504.html.

2) T. Berners-Lee, R. Fielding, and H. Frystyk, “Hypertext Transfer Protocol - HTTP/1.0,” RFC 1945, Network Working Group, May 1996.

3) T. Berners-Lee and D. Connolly, “Hypertext Markup Language - 2.0,” RFC 1866, Network Working Group, November 1995.

4) J. C. Mogul and V. N. Padmanabhan, “Improving HTTP Latency,” *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, October 1994, http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html.

5) R. Weerasinghe and A. Joshi., “Mowser - A Web Browser for Mobile Platforms,” July 1996, http://www.cs.purdue.edu/research/cse/mobile/mowser.html.

6) M. F. Kaashoek, T. Pinckney, and J. A. Tauber, “Dynamic Documents: Mobile Wireless Access to the WWW,” *Workshop on Mobile Computing Systems and Applications,* December 1994, pp. 179-184, http://snapple.cs.washington.edu/library/mcsa94/kaashoek.ps.

7) J. F. Bartlett, “W4 - the Wireless World Wide Web,” *Workshop on Mobile Computing Systems and Applications,* December 1994, pp. 176-178, http://snapple.cs.washington.edu/library/mcsa94/bartlett.ps.

8) S. Gessler and A. Kotulla, “PDAs as Mobile WWW Browsers,” *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, October 1994, http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/gessler/www_pda.html.

9) T. Watson, “Application Design for Wireless Computing,” *Workshop on Mobile Computing Systems and Applications,* December 1994, pp. 91-94, http://snapple.cs.washington.edu/library/mcsa94/watson.ps.

10) T. Watson, “Effective Wireless Communication through Application Partitioning,” *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995, pp. 24-27, http://snapple.cs.washington.edu/papers/hot_os.ps.

11) *NCSA Mosaic 2.0*, National Center for Supercomputing Applications at the University of Illinois in Urbana-Champaign, September 1995, http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/

12) “Graphics Interchange Format, Version 89a,” CompuServe, 1990.

13) “LZW and GIF explained,” public file available on many Internet sites, S. Blackstock, http://www-iapr-ic.dimi.uniud.it/Udine/WebRes/ImageCoding/formats/gif/lzwgif.txt.

Appendix A - Testing Scripts

Script 1 Contents

http://www.vt.edu/vt.text.html
http://www.vt.edu/student.html
http://milieu.grads.vt.edu/rgs.html
http://milieu.grads.vt.edu/ResearchCenters.html
http://oting.ee.vt.edu/vpec.html
http://oting.ee.vt.edu/VPEC/directory.html
http://oting.ee.vt.edu/vpec.html
http://oting.ee.vt.edu/VPEC/intro.html
http://oting.ee.vt.edu/vpec.html
http://www.ee.vt.edu/
http://www.ee.vt.edu/servers.html
http://www.visc.vt.edu/index.html
http://www.ee.vt.edu/servers.html
http://www.vpec.vt.edu/cgi-bin/home.html
http://oting.ee.vt.edu/VSPG/
http://www.vpec.vt.edu/cgi-bin/home.html
http://www.ieee.org/
http://www.vpec.vt.edu/cgi-bin/home.html

Script 2 Contents

http://www.vt.edu/vt.text.html
http://www.vt.edu/academic.html
http://www.vt.edu/CollegesDepts.html
http://www.vt.edu/academic.html
http://www.vt.edu/InfoServices.html
http://www.cclab.vt.edu/
http://www.cclab.vt.edu/labhours.html

Script 3 Contents

http://www.vt.edu/vt.text.html
http://www.vt.edu/VTWeb-Index.html
http://www.vt.edu/vt.text.html
http://www.bev.net/
http://www.bev.net/project/research/
http://www.bev.net/
http://www.vt.edu/vt.text.html
http://www.vt.edu/student.html
http://www.vt.edu/publishing.html
http://www.vt.edu/publish/using.html
http://www.utoronto.ca/webdocs/HTMLdocs/intro_tools.html
http://www.utoronto.ca/webdocs/HTMLdocs/online_tools.html
http://www.utoronto.ca/webdocs/HTMLdocs/intro_tools.html
http://www.utoronto.ca/webdocs/HTMLdocs/pc_tools.html
http://www.utoronto.ca/webdocs/HTMLdocs/intro_tools.html
http://www.vt.edu/publish/using.html
http://www.vt.edu/InfoServices.html
http://www.ms.vt.edu/ms/

Script 4 Contents

http://www.vt.edu/vt.text.html
http://www.vt.edu/directories.html
http://www.vt.edu/vt.text.html
http://www.vt.edu/student.html
http://www.vt.edu/vt.text.html
http://www.vt.edu/admininfo.html
http://www.vt.edu/vt.text.html
http://www.vt.edu/directories.html
http://www.unirel.vt.edu/dir/index
http://www.unirel.vt.edu/dir/deptsp-q.html
http://www.unirel.vt.edu/dir/index
http://www.vt.edu/directories.html
http://www.vt.edu/vt.text.html
http://www.vt.edu/aboutvt.html
http://www.vt.edu/vt.text.html
http://www.vt.edu/academic.html
http://www.vt.edu/student.html
http://www.vt.edu/academic.html

Vita

Todd Fleming was born in Lakeland, Florida on September 26, 1972. He received an Associate of Arts in Engineering in 1991 from Polk Community College in Winter Haven, Florida. He then received a Bachelor of Science in Computer Engineering, Summa Cum Laude, in 1994 from Virginia Polytechnic Institute and State University in Blacksburg, Virginia. He is finishing up this theses for his Masters degree at Virginia Tech while taking courses for his Doctorate.

Todd created a cross-platform hypertext multimedia system which was used to produce two commercial CDs. He has also produced custom database applications and worked as a consultant. He spent one and a half years working as a Graduate Research Assistant in the EE Department and is now a Bradley Fellow. Todd is interested in both hardware and software development.