Performance Analysis of a Dynamic Parallel Downloading Scheme from Mirror Sites Throughout the Internet

Allen Miu and Eugene Shih
Laboratory of Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
{aklmiu, eugene}@wind.lcs.mit.edu

Abstract

In this paper, we describe *paraloading*, a novel approach to retrieving files from the Internet by establishing parallel connections with multiple mirror sites. This method contrasts with the conventional download method where the client retrieves data from a single source. The advantages of paraloading over single-connection downloading include improved performance gained from aggregating the bandwidths of the parallel connections, increased efficiency gained from load balancing download requests among the parallel connections. and increased resilience against congestion or failures on any one path, In order for paraloading to work, servers must be mirrored throughout the Internet. However, as mirror sites become more widespread and as end users upgrade to higher connection speeds, we believe that paraloading—if implemented properly—will offer significant performance gain over the traditional single-source file accesses.

Paraloading is a subject that has not been extensively studied by the research community. This paper examines the performance and the design of a paraloading scheme proposed in [21]. We have developed a paraloader application in Java that uses HTTP 1.1 for its range-request and persistent connection features. We have conducted a series of experiments using our paraloader at MIT and at UC Berkeley, and we have found that the performance gains of paraloading are not as good as those claimed in [21]. This suggests that paraloading may not fair well in different network environments. Nevertheless, we believe that there are a number of enhancements that can be made to the paraloader to improve its performance in different network environments. We will outline some of these enhancement techniques and discuss some open research issues on paraloading.

Keywords: Parallel downloading, paraloading, mirror servers, download performance, Internet measurements.

1 Introduction

The conventional method for downloading a file from the Internet is to open one or more connections between the client and a single server. The *download performance*, the time to download a file, is directly influenced by the load of the server, the bandwidth of the bottleneck link, and any traffic fluctuations that may intersect the route between the client and the server. To help balance load and bring the content physically closer to the client in the hope of improving download performance, various organizations have deployed mirror servers in different network domains. Thus, whenever the users wish to download from the mirror servers, they can select a mirror site that contains the fastest available path.

Presently, users are seldom given accurate performance metrics to help them select the fastest server. Most often, clients use *ad hoc* mirror selection techniques that yield poor download performance. Even when the fastest server has been chosen, throughput can still fluctuate because the traffic patterns may change during a download session. For example, intermediate routers can become congested during a download session. In this case, the transient congestion forces the server to decrease its offered load, which then lowers the client's observed throughput.

Thus, rather than trying to identify and connect to the fastest available server, clients could improve download performance more simply and effectively by connecting to two or more servers containing an exact copy of the document. Instead of downloading the entire document from one server, clients would download *unique* parts of the same document from each of the mirror servers in parallel. Once all the parts of the documents have been received, the client can recover the original document by reassembling the pieces. This parallel file access scheme was first proposed in [21]. In the rest of this paper, we will refer to this scheme as "paraloading".

There are several advantages to employing a paraloading scheme. First, because a paraloading scheme opens multiple connections to different servers, paraloading is inherently more resilient to route or link failures and traffic fluctuations than the traditional scheme of downloading from a single source. Second, the mirror selection process can be eliminated by a paraloading scheme that opens a connection to all of the available mirrors. In this case, the fastest path will be among the set of opened parallel connections. Third, aggregating the bandwidth of the individual connections can potentially increase overall throughput to the client. Ideally, the total bandwidth to the client is equivalent to the sum of the bandwidths from each individual server. Fourth, a paraloading scheme allows the client to employ a variety of scheduling algorithms for assigning download requests. This flexibility enables an intelligent paraloader to perform dynamic load balancing. For example, a paraloader can choose to download a bigger proportion of a file from a faster connection or drop connections that have become heavily congested during a download session.

1.1 The Basic Paraloading Scheme

The basic paraloading scheme is conceptually simple. When a client wishes to obtain a document from a particular site, the paraloader will first obtain a list of mirrors servers assoicated with that site. We assume this information can be obtained easily by querying a directory service. Once the servers are known, the paraloader will query one of the mirrors to obtain the length of the target document. The file length is needed for partitioning the file into multiple block units and for ensuring that all the blocks have been successfully received by the client. After the file has been partitioned into the proper block sizes, the paraloader will begin issuing requests for different blocks to each of the mirror servers. When a connection finishes downloading a data block, the client immediately issues a request for the next block to that same connection. This scheme continues until all the different blocks of the document have been received. As the paraloading procedure progresses, it is likely that the client will receive the data blocks out of order. Therefore, the final step for the paraloader is to recover the original document by reordering and reassembling all the blocks it has received.

Notice that this simple algorithm automatically achieves load balancing: it is not difficult to see that a faster connection will end up transporting more block requests from the paraloader. Hence, a faster connection will download a larger proportion of the file than a slower connection. However, a drawback to this algorithm is that connections become idle when they wait for a block to arrive after issueing a request. This idle time typically spans one round-trip time (RTT), during which no useful data is transmitted and bandwidth is wasted.

Conceivably, a paraloader introduces at least three new kinds of overhead not found in the traditional downloading schemes. These are the overhead associated with scheduling the block requests, the overhead associated with introducing request messages into the network, and the overhead associated with maintaining buffers to receive incoming data blocks. Hence, a parallel downloading scheme should not be widely deployed unless it can be proven that paraloading achieves a measurable performance improvement in today's network environment.

To this end, we developed a simple Java paraloader application and used it to conduct a series of preliminary experiments at MIT and UC Berkeley. Our experiments are similar to those conducted by Rodriguez *et al.* except we have performed our experiments in two different networks, using three different sets of mirror servers. In addition, we have varied different parameters such as the *degree of parallelism*, the number of open parallel connections, and the data block sizes in order to examine how each of these parameters impact paraloading performance.

Our results have shown that paraloading achieves good download performance in general. This is similar to the results recently published by Rodriguez *et al.* However, we have found that the performance gains of paraloading in our network environment are not as good as those claimed by Rodriguez *et al.* This suggests that paraloading may not fair well in all network environments. Nevertheless, we believe that there are a number of enhancements that can be made to the paraloader to improve its performance. We will attempt to outline three enhancement techniques for improving paraloading performance: fast file length retrieval, pipelined block requests, and last block download preemption. Finally we will discuss some open research issues pertaining to fairness and efficiency.

2 Related Work

Prior to our research and that conducted by the authors in [21], there has been a number of techniques developed to improve download performance[1]. One simple technique used by web browsers [18] and by some FTP clients [1] involves the opening of multiple connections to a single server. In this technique, download time is decreased because the client consumes network bandwidth more aggressively [3] when opening parallel connections to a single server. While this scheme is simple to implement, the aggressiveness can cause an unfair allocation of network resources[2]. Moreover, parallel connections add extra overhead to the server by requiring the server to maintain multiple connections. Because the server can only open a limited number of simultaneous connections, the number of distinct clients that the server can handle will decrease by a factor proportional to the average number of parallel connections opened per client.

Another technique to reduce download time relies on the existence of mirror servers that contain exact replicas of the data that a client wishes to retrieve. Given a set of mirror servers, a client could minimize download time by accurately selecting the mirror server with the shortest response time and highest bandwidth. Different methods of selecting the best server have been studied. In [6], server performance is measured dynamically by sending probe packets from the

[1]While the paraloading scheme is first formalized by [21], the earliest implementation of a paraloading client we found is a Perl script that is part of the CPAN library called the Parallel User Agent written by Marc Langheinrich. The PUA is a simple script that allows users to download *multiple files* in parallel from different sources. However, unlike the paraloading scheme proposed by Rodriguez, the PUA does not support parallel access of an individual file from multiple servers as does the paraloading scheme.

[2]Balakrishnan *et al.* [4] have proposed a technique to solve this unfair allocation problem by introducing a "Congestion Management" layer between the application and the transport layer to coordinate parallel flows connected to a single server.

client to each server. In [8], data about servers is maintained in a resolver that clients can query to obtain the identity of the best server. Specifically, servers push information about their performance to resolvers used in application-layer anycasting. In addition, probe agents send periodic queries to servers to determine performance. Another method to select the best server is based on examining statistical record-keeping [22].

These methods of selecting the "best" server from a set of mirror servers have many disadvantages. Using historical measurements can often lead to gross inaccuracies because traffic patterns vary over time. On the other hand, while using probe packets to measure the network conditions can be more accurate, periodic probing can introduce undesirable overhead. Moreover, this file access scheme does not take advantage of aggregating bandwidth from multiple connections. Thus, download performance is still limited by the bottleneck bandwidth of the optimal path.

Rather than trying to tackle the problem of selecting the optimal server, other researchers have examined the idea of parallel-access for information dispersal. Rabin's research on information dispersal explored the idea of disseminating a document to a receiver by breaking it into several pieces and delivering the pieces along multiple network paths [20]. However, the idea of using multiple mirror servers in parallel to improve download performance is fairly recent.

One novel parallel-access approach to minimize download time was proposed by Byers *et al.* In [5], the authors describe a "feedback-free Tornado solution" to facilitate the delivery of data in parallel. In their scheme, a file of size F is encoded on each mirror server with redundant information. The encoded file consists of $n = k + l$ different blocks of size p, where $np > kp > F$. When a receiver makes a request for the file, each mirror site delivers the n packets continuously to the receiver[3]. To minimize the number of duplicated packets received at the client, each sender delivers the packets in a random order. As soon as the receiver collects k distinct packets from the senders, the original file can be immediately decoded and reconstructed. This scheme is "feedback-free" because once the transmission begins, the receiver is not required to send any explicit requests to ask for a new packet nor is the receiver required to send any acknowledgements back to the sender to confirm the reception of a packet.

While the Tornado code solution greatly simplifies the mechanism for data packet delivery (and provides an elegant solution to the multicast implosion problem), there are two main drawbacks that make it unsuitable for wide-area paraloading. First, there is an overhead associated with encoding and decoding the file, allocating storage space for storing the redundant information in the encoded file, and transmitting the extra $kp - F$ bytes of data used to reconstruct the original file. Second, the feedback-free scheme can potentially cause congestion collapse due to *undeliverable packets* [10]. Consider when the receiver has received k distinct packets and wants to terminate all the parallel connections. When the receiver breaks the connections, all the packets currently in transit will be dropped and bandwidth is wasted. This problem is aggravated when the number of parallel connections is large or when the bandwidth-delay product of a connection is large.

As mentioned previously, the authors of [21] propose a dynamic parallel-access scheme where clients and servers connect via unicast TCP. Application-level negotiations are used to request different parts of a document from mirror servers. The results presented by Rodriguez *et al.* have shown good speedup over single connection downloading.

2.1 Assumptions

For a paraloading scheme to be beneficial, there are a few assumptions that must hold. First, we assume that the underlying protocol (in our case, HTTP 1.1) transports data reliably and implements range requests correctly. Otherwise, the paraloader needs to perform a final checksum to verify that the reassembled file is correct.

Second, we assume that the data being fetched is *static*, meaning that the file undergoes no changes on any of the mirror servers during a paraloading session.

Third, we assume that the paraloader can quickly and readily obtain the locations of the available mirror servers without incurring a significant overhead in the system. In the version of the paraloader that we have implemented, the mirror locations have been hard coded. A better approach is to have the paraloader obtain this information automatically from widely deployed network services. For example, we can enhance the functionality of the Domain Name System (DNS) server by adding some of the changes proposed in [14] to provide a list of mirror servers containing the desired document. Alternatively, a directory or search engine could be queried to provide the mirror list.

Finally, we assume that the paths to the mirror servers are *bottleneck-disjoint*. In other words, the different paths of the parallel connections from the client to the servers do not intersect at a bottleneck[4]. Sharing bottlenecks is undesirable in the following two situations. In the first situation, if, at the bottleneck link, there are no connections other than the ones originating from the paraloader, the parallel connections may "cannibalize" each other's bandwidth and cancel any gain in aggregate throughput. In the second situation, if two or more paraloader connections share the bottleneck with many other connections in the network, then the paraloader connections will become overly aggressive [3] and start to dominate other TCP-friendly traffic in the network [10].

[3]In [5], Byers *et al.* assume that the sending rate is TCP-friendly and does not introduce aggressive behavior in the network.

[4]Two or more connections may intersect each other as long as the point of intersection is not a bottleneck link.

3 Theoretical background of paraloading

The basic idea behind parallel downloading is that clients open connections to multiple servers. In doing so, clients could experience a bandwidth equivalent to the sum of the individual bandwidths. Theoretically, this decreases the overall time to download the file.

Let the set of all servers containing a common document be M. Define the *serial bandwidth* to server i to be $\mu_{s,i}$. This is the connection bandwidth that a client experiences when only opening a single connection. Therefore, the set of serial bandwidths of all mirror servers can be defined as:

$$U = \{\mu_{s,1}, \mu_{s,2}, \mu_{s,3}, \ldots, \mu_{s,|M|}\}.$$

We define the *ideal bandwidth* to be equal to the sum of the individual serial bandwidths:

$$\mu_{ideal} = \sum_{i=1}^{|M|} \mu_{s,i}. \quad (1)$$

As more mirror servers are introduced, it will be possible to open more connections. Theoretically, if bottleneck-disjoint paths are used and the client's capacity is infinite, opening more connections will increase the ideal bandwidth and result in a noticeable decrease in download time.

Given S, the size of the desired file, we can calculate the time to download the file serially from server i as

$$t_{s,i} = \frac{S}{\mu_{s,i}} + t_{s,i}^{overhead} + t_{s,i}^{conn}. \quad (2)$$

The first term in equation (2) represents the transmission time. In the serial download case, the overhead time $t_{s,i}^{overhead}$ is basically $t_{s,i}^{response}$, which is the time between making a block request and receiving the first byte of data. The $t_{s,i}^{response}$ essentially equals the round-trip time RTT_i to server i. Finally, $t_{s,i}^{conn}$, is the time to open a serial connection[5] to server i.

To determine the time to download a file using parallel-access, we must consider more parameters. When paraloading, not all the servers need to be used. Let M' be the set of mirror servers actually used for paraloading. In other words, $M' \subseteq M$. $|M'|$ represents the degree of parallelism.

Another parameter that we can vary is the block size. Before paraloading, the desired file is broken into many blocks. If F is the size of a block and B is the number of blocks, then $B = \lceil \frac{S}{F} \rceil$. In order to fully utilize all the servers in the set M', F should be chosen such that $B \geq |M'|$. In other words, a minimum of $|M'|$ blocks should be downloaded simultaneously. If B_i is the number of blocks downloaded by server i such that $B = \sum_{i=1}^{|M'|} B_i$, the time for server i to download B_i blocks is:

$$t_{p,i} = B_i \left(\frac{F}{\mu_{p,i}} + t_{p,i}^{overhead} \right) + t_{p,i}^{conn}. \quad (3)$$

In this equation, the bandwidth to each server is denoted by $\mu_{p,i}$, which represents the realized bandwidth achieved from server i when all $|M'|$ connections are active. Thus, $\mu_{p,i} \leq \mu_{s,i}$. In the paraloading case, the overhead time $t_{p,i}^{overhead}$ is the sum of $t_{p,i}^{response}$ and $t_{p}^{process}$, where $t_{p,i}^{response} \approx RTT_i$ and $t_{p}^{process}$ is the processing overhead for scheduling a request to connection i. In the ideal case, the processing time is negligible.

Assuming that mirror server discovery is fast, and that only a single server performs the initial file length request, given equation (3), the time to download a file from multiple servers using paraloading is:

$$t_{p,total} = t_{get\_length} + \max\{t_{p,i}\} \forall i \in M' \quad (4)$$

where $t_{get\_length} \approx RTT_i$ is the time required to get the file length from some server i.

Now, consider the case where $M' = M$. Summing the individual bandwidths $\mu_{p,i}$ to each server during paraloading yields the total bandwidth during paraloading[6]:

$$\mu_{optimal} = \sum_{i=1}^{|M|} \mu_{p,i} \quad (5)$$

In the forthcoming version of this paper, we will show how equation (4) can be made equal to (5) when we assume the overhead is negligible. The result of this will produce a set of mathematical constraints that an ideal paraloader must satisfy for achieving optimal performance.

Notice that $\mu_{optimal}$ will not necessarily be equal to μ_{ideal}. Assuming that processing time is negligible and that all paths are bottleneck-disjoint, the upper bound on the optimal bandwidth is $\min(\mu_{ideal}, \mu_{client})$, where μ_{client} is the local client bandwidth.

As we have shown in the above equations, parallel downloading allows clients to experience a gain in total throughput by aggregating the individual bandwidths. However, the penalty paid for paraloading is the increased overhead. In theory, if the overhead can be made small, the performance of paraloading should be no worse than serial downloading. In particular, if the fastest mirror server k is included in the set of M', then $\mu_{s,k}$ is a component of $\mu_{optimal}$, which implies that $\mu_{optimal} \geq \mu_{s,k}$.

[5]If the underlying protocol uses TCP, this is equivalent to the time it takes to perform the SYN packet exchange.

[6]This is equivalent to the definition of optimum bandwidth as given by Rodriguez *et al.*. In their paper, the optimum bandwidth is determined based on the optimum transmission time. "The optimum transmission time is the transmission time achieved by a parallel-access scheme where all servers send useful information until the document is fully received and there are no idle times between reception of two consecutive blocks."

4 Paraloading Parameters

Some of the inefficiencies of paraloading can be reduced by merely increasing the degree of parallelism and the block size.

Intuitively, as the client increases the degree of parallelism, the total download time decreases because each added server contributes some bandwidth to the optimal bandwidth $\mu_{optimal}$. This intuition is really the consequence of 5; as $|M'|$ increases, $\mu_{optimal}$ increases as well.

Hence, in theory, setting $M' = M$ will always result in the best possible performance. However $\mu_{optimal}$ is bounded by μ_{client} when $\mu_{client} < \mu_{ideal}$. In this situation, the client's link is completely saturated by the parallel connections so there is little gain in adding another connection. Furthermore, practical consequences such as increased amounts of overhead and adverse interactions among parallel connections at the client's link will likely decrease a paraloader's performance. Therefore in practice, one should choose $M' \subset M$ to achieve maximum performance gain without wasting network resources.

There are other subtle issues that explain why one would choose a degree of parallelism less than $|M|$. We will discuss these issues in the Discussion section.

While retrieving a file using paraloading, bandwidth is wasted during the idle time between the request for a block and the arrival of the requested data block. Typically, the idle time is at least one RTT.

One way to reduce this inefficiency is to increase block sizes and thus decrease the number of blocks and the number of idle times. Unfortunately, making blocks too large will reduce the total number of blocks. This in turn reduces the effectiveness of load balancing the block requests among the active parallel connections.

In theory, the optimal way to size the blocks is to assign a different block size for each of the connections so that each connection downloads one block and all downloads finish at the same time. In particular, the size of the block F_i for server i should be:

$$F_i = \frac{\mu_{p,i}}{\mu_{optimal}} S. \tag{6}$$

Assigning the blocks this way ensures that each connection finishes at exactly the same time and only downloads one block. Clearly, this minimizes the number of requests. In practice, however, it is difficult to determine both $\mu_{p,i}$ and $\mu_{optimal}$ as bandwidths can fluctuate considerably over time.

5 Optimization Techniques

In order to achieve maximum performance, we want to fully utilize all of the parallel connections in a paraloader. This implies that an ideal scheduler for a degree $|M'|$ paraloader will schedule block requests such that a) the first $|M'|$ block requests going to each of the mirror servers are made at the earliest possible time, b) no wasted idle time between successive block requests going to each of the mirror servers exists, and c) none of the connections become idle before the last block has been fully received. To meet each of the corresponding requirements listed above, an ideal paraloader must strive to minimize the initialization delay, the idle time between requests, and any idle time that arise as a result of a poorly scheduled request to retrieve the last block. We will describe how the paraloader can be optimize to meet each of these goals.

5.1 Minimizing the Initialization Delay

The initialization process must take place before a paraloader can begin sending requests. This process affects the initialization delay, which involves retrieving the mirror list and file length information and establishing a connection to each of the mirror servers. While the latter delay depends entirely on the network characteristics and the underlying protocol, the former delay can be minimized by caching the required information in the local system.

We recognize that maintaining an accurate cache containing information about the characteristics of the network is not always possible; hence, we assume that such information must be fetched from one of the mirror servers. In this situation, the initialization delay can be minimized by piggybacking a data block request onto the mirror-list/file-length query that is sent to one of the mirror servers. In this scenario, we assume that the location of at least one server is given at the start of the paraloader[7].

5.2 Minimizing the Idle Time Between Requests

For each request sent to a particular server, a simple paraloader must wait at least one RTT before it starts to receive a data block. During this *idle time*, the link is not utilized and bandwidth is wasted. In [21], the use of pipelining the requests is proposed as a solution to this problem. In a typical pipelining scheme, the paraloader will initially send $n > 1$ requests for different blocks to each server. When a connection downloads the first byte of data from a data block, another request is sent to that connection for the next unrequested block. There will always be one pending data block download at the client and $n - 1$ pending data block transmission "in the pipeline" for each connection. It is intuitive to see that n should be set to 2. By setting n to 2, we minimize the number of pending requests that the server must keep track of. However, this requires that the block size be at least the bandwidth-delay product of the particular connection that the request is being sent to.

[7]If all the mirror locations are known ahead of time, then one can further optimize by sending a unique block and file length query for the first $|M'|$ different blocks to each of the $|M'|$ servers. The block size can be arbitrarily set and any error responses returned as a result of requesting out-of-range data blocks can be ignored.

By introducing pipelining, the time due to requesting a block ($t_{p,i}^{response}$) is eliminated. We believe that this is an effective optimization especially in situations where the block size is relatively small compared to the bandwidth-delay product of the individual connections. Our simulated experimental results agrees with this claim.

5.3 Minimizing the Idle Time in Downloading the Last Block

The amount of time spent downloading the last data block can be significant. This is especially true when the block size is large and the last block download has been scheduled to a slow connection. As this connection downloads the last block, other connections may become idle. One way to minimize this waste bandwidth is to set smaller block sizes so that the wait time for the last block is reduced. However, small block sizes may be impractical because they will require the paraloader to send many more request messages to the network. Moreover, small block sizes will increase the total number of idle gaps when pipelining is not implemented.

Another approach to minimize the last block delay is to dynamically adjust the block sizes according to (6) so that the last $|M'|$ data blocks finish downloading at roughly the same time. However, this will require accurate bandwidth measurements for each connection at runtime, which may be difficult to obtain.

Alternatively, the paraloader can send requests to one or more of the other $|M'|-1$ connections that have become idle to download the remaining portions of the last block from the different mirror servers. However, in this case, an idle time period still exists before the paraloader can identify the last block. For example, imagine two slow connections that start to paraload the final two blocks of a file at the same time. In this situation, the last block cannot be identified until one of the two blocks finishes downloading. Because this wait can still be significant, an ideal optimization technique would assign range requests for the unreceived portions of each of the last $|M'|-1$ blocks to the connections that have become idle.

Clearly, the block assignment policy can be varied to tradeoff performance and the amount of redundant data transferred. In maximizing performance, it is possible that redundant data will be received. In both of the previous cases, redundant data may be received since the paraloader typically cannot revoke a block request that has already been sent.

6 Dynamic Paraloading Experiment

The main objective of our experiment was to verify that paraloading decreases download time relative to downloading from a single connection. At the same time, we had several other objectives:

1. To determine whether changing the block size or the degree of parallelism could affect download performance.
2. To collect an extensive set of data to show how well paraloading performs under different network environments. Toward this goal, we ran experiments at two different client sites paraloading from three different sets of mirror servers. We have plans to try paraloading from additional sites.
3. To compare our results with those obtained in [21] and to verify that paraloading indeed produces significant speedup over the single connection case.
4. To determine the impact of paraloading performance by applying some of the optimizations discussed in the previous section.

We first designed and implemented a paraloader. In ourimplementation, we wrote a Java application called `jphttp`, which stands for Java Parallel HTTP. The program's underlying protocol is HTTP 1.1. We chose this protocol primarily because it is widely deployed and because it supports range requests and persistent connections[8]. The range request feature is used for fetching a document block with arbitrary start and finish offsets from a range-request-enabled mirror server. The persistent connection feature is used to enhance the efficiency of the paraloader[9].

In our paraloader, only two of the optimizations were implemented. We save one round trip time by issuing a GET message to a random server in order to request for the file length and the first data block. We also also attempt reduce the last block delay by selecting the fastest connection[10] to download the unreceived portion of the last block. Requests are not pipelined in our implementation.

We have ensured that the receiver socket buffer is set large enough so that it does not create a bottleneck at the client by advertising a smaller-than-optimal TCP receiver window value. In our experiments, the receiver socket buffer is set to 32 KB.

[8]Although HTTP 1.1 was chosen in our implementation, the choice of an underlying protocol is orthogonal to the idea of paraloading. For example, one can still implement a paraloading scheme without using range requests by pre-partitioning the file at the mirror servers and naming the files with the appropriate block numbers.

[9]Rather than wasting possibly many round-trip times for re-negotiating a new connection every time the paraloader sends a request to a mirror server, the persistent connection feature allows the paraloader to send requests by using the same connection as the one first established between the mirror server and the paraloader.

[10]We determine the fastest connection to be the connection that downloaded the most blocks at the time when the paraloader identifies the last block.

6.1 Experiment Setup

In our experiment, we downloaded a single file from three different sets of seven mirror servers ($M = 7$). In particular, data from the mirrors of the sites `http://www.kernel.org` (Set 1), `http://mars.jpl.nasa.gov/mgs` (Set 2), and `http://www.tucows.com` (Set 3) were downloaded. In addition to paraloading from different sets of mirrors, we also paraloaded at different client locations; we paraloaded to hosts at MIT and UC Berkeley.

To examine how the degree of parallelism would affect the download time, we set the degree of parallelism to one, three, five, and seven servers and downloaded the same file for each value of $|M'|$. To examine how performance varies for different file sizes S, we downloaded a 1 MB file and a 300 KB file. We fixed the block size to 32 KB.

Because traffic over the Internet can vary over a day and throughout an entire week, we conducted our experiment for 24 hours over a period of seven days. On day i, at the beginning of every hour, a 1 MB file and a 300 KB file was downloaded from server i. To increase the accuracy of our samples, we repeated each download five times. Once the file was downloaded serially, the same two files were downloaded from a randomly chosen set of three servers, a randomly chosen set of five servers, and finally from all seven servers. Again, each of these downloads was performed fives times. This procedure was repeated for each of the three sets of mirror servers listed above.

As the downloads proceeded, we tracked various statistics about each connection such as the response time, download time, and the scheduling overhead for each block. The response time is the time between sending a block request and receiving the first byte of the data block. Typically, this is the sum of the round trip time and the server processing overhead. The download time is the time between sending a block request and receiving the last byte of the data block. The scheduling overhead is the amount of time that the connection remain idle while waiting for the scheduler to assign the next block request.

6.2 Results and Analysis

In this section, only the results for the MIT/MARS paraloading experiments are presented. We are still in the process of gathering and analyzing the results of the other experiments mentioned in the previous section. We will present these results in the forthcoming version of this paper.

Before analyzing the data, we first averaged the five trials per degree of parallelism at each hour. This average formed a paraload sample per set for that hour. For each degree of parallelism, we then averaged each paraload sample for each hour across seven days. This average forms a data point on the graph for the specific number of servers used.

In figure 1, we graph the time to download a 1 MB file from each of the servers throughout a single day. In addition, we determined the average time to download the file from one server, if the server were chosen randomly from M. This is done by summing the single-server download times at each hour and dividing by the total number of servers. For comparison, we also graph the ideal download time that would result if a bandwidth of μ_{ideal} as defined in Equation 1 was used. The value of μ_{ideal} is calculated by summing the *experimental* values of $\mu_{s,i}$ at each hour.

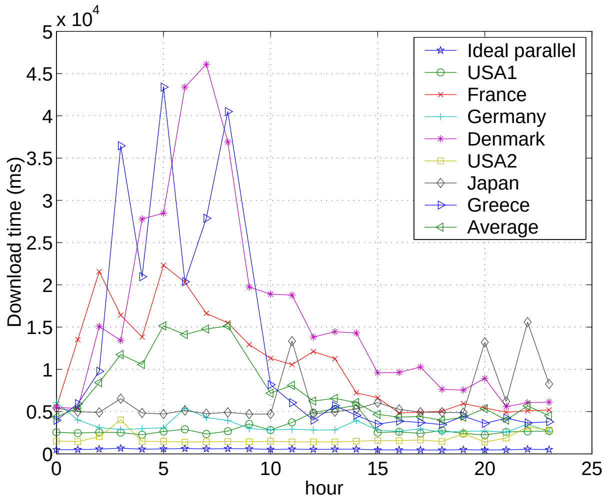

Figure 1: Average time to download a 1 MB file serially from several MARS web mirrors throughout a single day ($S = 1$ MB, $F = 32$ KB, $|M'| = 1$).

As shown, the download time for the file varies considerably to some of the servers used throughout the day. For example, the slowest server (Denmark) takes between 4 and 130 seconds to deliver the file as the day progresses. However, the download time for the fast servers seems to be fairly stable throughout an entire day. For example, the USA2 server downloads the file consistently in 2 to 4 seconds.

In figure 2, we graph the time to download a 1 MB file using various degrees of parallelism. The servers examined are again MARS mirrors. The single server case represents the average of the individual download times of each server. In addition to graphing the download time for different degrees of parallelism, we also graphed the download time derived from μ_{ideal} and $\mu_{optimal}$. The value of μ_{ideal} is calculated as before and the value of $\mu_{optimal}$ is obtained by summing the *simulated* values of $\mu_{p,i}$ for all i. The value of $\mu_{p,i}$ is simulated by subtracting the measured scheduling overhead times and the response times from the total download time for all blocks in connection i. This number was then divided into B_iF_i, the total number of bytes downloaded from connection i. As shown in figure 2, the times to download from parallel connections is clearly less than the time to download from an average server. This means that on average, paraloading performs better than downloading from a single server. Moreover, the download time decreases as more mirror servers are added. For example, the average time reduc-

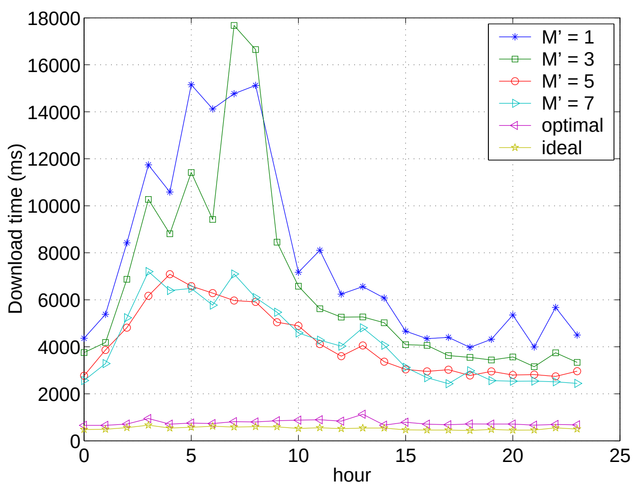

Figure 2: The download time for a 1 MB file is compared among different degrees of parallelism (S = 1 MB, F = 32 KB, $|M'| = 1, 3, 5, 7$)

tion between the one and three servers case is about 7.93 s. The average difference between the one and five server cases is about 12.3 s. Finally, the average difference between the one and seven servers case is 13.5 s.

One interesting feature to note is that as $|M'|$ increases from five to seven, the performance gain is less dramatic. A possible reason for this behavior is that two slow servers were added. The servers probably did not contribute in further reducing the download time.

Another interesting feature of this graph is that the download time using the optimal bandwidth is near the ideal bandwidth. However, this does not provide any knowledge about the bottleneck-disjointness at the client. *We will explain this in detail in a forthcoming paper.*

In figure 3, we again graph the time to download a 1 MB file using various degrees of parallelism. This time, however, the $|M'| = 1$ case represents the time to download the file from the fastest server (USA2). Again, we also graphed the download time derived from μ_{ideal} and $\mu_{optimal}$. The most surprising result shown by these graphs is that the download times when paraloading is actually worse when compared to the fastest server. This result seems to contradict the results obtained in [21], where the authors found consistent performance gain in all their dynamic paraloading experiments. We will attempt to resolve these inconsistencies in the next section.

In figures 4 and 5, we show the times to download a 300 KB file with varying degrees of parallelism and a fixed block size of 32 KB (same as before). In figure 4, the $|M'| = 1$ case represents the average of the individual download times from each server. In figure 5, the $|M'| = 1$ case represents the time to download from the fastest server (USA2). In general, the performance gained over the average case from paraloading the small file is less than the performance gained

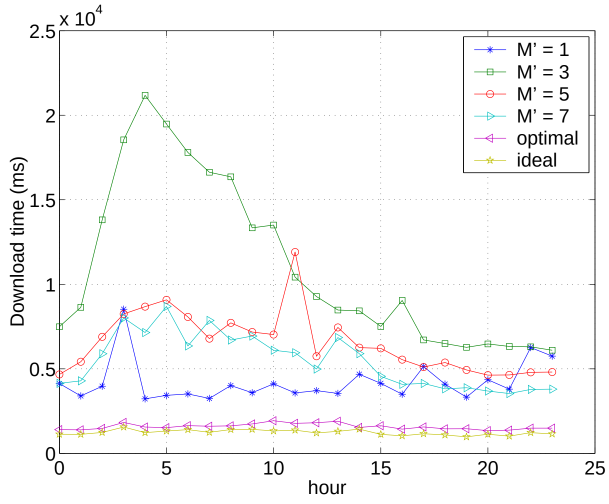

Figure 3: The download time for a 1 MB file is compared among different degrees of parallelism where degree 1 is represented by the fastest server (S = 1 MB, F = 32 KB, $|M'| = 1, 3, 5, 7$)

from paraloading the large file. In some instances (e.g. when $|M'| = 3$), the paraloading performance is worse than the average serial download case. These results are expected because the effectiveness of load balancing the block assignments has decreased due to the reduced total number of blocks. Hence, we can conclude that paraloading becomes less effective when it is used for downloading a small file [11].

6.3 Comparison with Rodriguez *et al.* results

Our experimental results have shown that paraloading does not provide significant performance gain over from the best single server case. This is in contrast to the results reported in [21]. In this section, we will compare and contrast our experiment setup against that report by Rodriguez *et al.* in an attempt to explain why our results differ.

In our experiment, the file sizes and block sizes used were roughly the same size as those in [21]. In addition, the degree of parallelism used was also approximately the same; in most of their experiments, they use $|M'| = 4$, while we use $|M'| = 3, 5$. However, our experiments differ in the following major ways:

1. The servers used in their experiment were roughly one order of magnitude (7 times) slower than ours. The average single source throughput that was reported ranged

[11]However, we do not rule out the possibility for achieving high performance gain by using a paraloader to retrieve *many* small files in parallel. Such a scheme will most definitely be useful for web browsing applications. Although Rodriguez has suggested that small files, such as web objects, be combined into a larger file in order to increase paraloading effectiveness, combining the small files together will cause excessive delays because the larger file might have to be entirely retrieved before the application can access any one of the small files.

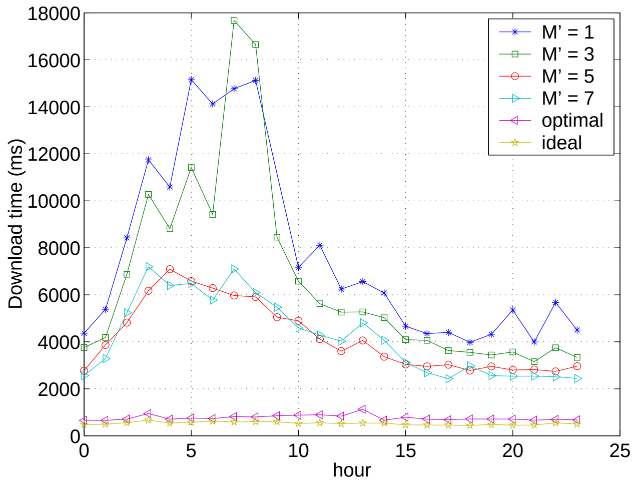

Figure 4: The download time for a 300 KB file is compared using different degrees of parallelism ($S = 300$ KB, $F = 32$ KB, $|M'| = 1, 3, 5, 7$)

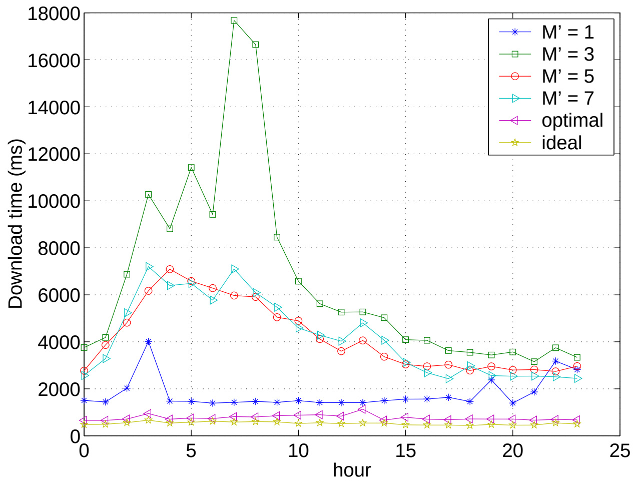

Figure 5: The download time for a 300 KB file is compared using different degrees of parallelism where degree 1 is represented by the fastest server ($S = 300$ KB, $F = 32$ KB, $|M'| = 1, 3, 5, 7$)

from 40 to 120 Kbps. In our experiment, the single source throughput ranged from 210 to 890 Kbps.

2. From their paper, it is likely that their paraloader employed a different strategy for obtaining the file length. More specifically, they did not implement the optimization that combines the first block request with the file length query.

3. The paraloader used by Rodriguez *et al.* employs a different strategy for minimizing the last block delay. Specifically, their strategy does not require the idle connections to wait for the paraloader to identify the last block, but downloads redundant data for the last $|M'| - 1$ blocks.

We now show that these differences can affect the performance of the paraloader in the following ways:

1. High speed networks will increase the ratio between request idle time and data block transfer time. Hence, if we assume that the round trip times are similar for all connections in both experiments, our experiment will suffer from a higher ratio of wasted bandwidth.

2. In analyzing the file length query optimization, we found a bug in our paraloader. Ideally, full scale paraloading should begin as soon as the file length information is obtained, i.e. when the HTTP header arrives at the paraloader. However, our paraloader does not begin paraloading until *after* the entire first block has been received. Thus, the bug adds a total delay of the time required to download a data block, which can be significant if a slow connection were chosen for the file length query. To test the effect of this bug, we have simulated new results by subtracting the data transfer time for the first block of data from the original results. The simulated results for paraloading a 1 MB file from the MARS mirror servers to MIT are shown in Table 1. Fortunately for us, there is no significant performance difference between the simulated and original results.

Table 1: The performance impact of a bug introduced in our paraloader is shown. The time for paraloading a 1 MB file is the average of the 24 hour averages. Hence the times in the first column corresponds to the averages of the 24 data points for each $|M'| > 1$) in figure 2. The times in the second column corresponds to the same set of averages subtract the averaged first block data transfer time.

$\|M'\|$	Original paraloading time, averaged (s)	Simulated paraloading time, averaged (s)	Speedup
3	11.025	10.466	5.74%
5	6.548	6.114	7.29%
7	5.462	5.134	6.87%

3. It turns out that the difference in the last block optimization (LBO) policy has an enormous impact on paraloading performance. We have simulated new results by subtracting the fastest connection's wait time from the total time. The fastest connection's wait time is the difference between the time when the fastest connection becomes idle due to an empty block request queue and the time when the last block request is assigned to it. In Table 2, we show the gain realized by simulating this optimization. While we recognize the simulated results can report an overly optimistic paraloading perfor-

mance, the significant time difference between the two results still gives a good idea of how much bandwidth was wasted during the fastest connection's wait period.

Table 2: Compares the performance of a paraloader with LBO. The same averaging method used in Table 1 was also used here to construct the table values.

$\|M'\|$	Average time without LBO (s)	Average time with LBO (s)	Speedup
3	11.025	10.785	2.08%
5	6.548	5.269	23.05%
7	5.462	3.671	48.13%

We believe that the combination of the three key differences outlined above have reduced the realized performance gain in our experiments. Hence, we believe that after resolving the implementation differences, the performance gain should match those reported in [21].

One important lesson to be learned is that the implementation differences among paraloaders can greatly impact the performance of the paraloader. Because paraloading has such a large parameter space, the number of possible optimizing designs are numerous.

6.4 Pipelining Simulation

We have also simulated new results for examining the performance gained by pipelining block requests. Our paraloader has kept a record of the response time between sending a request and receiving the first byte of data for each block requested. To simulate the pipelining results, we summed the total download time and then subtracted the response time for the individual blocks. This gives us a set of new values indicating the amount of time each connection had spent downloading blocks. We then take the maximum over this set of values to be the new total time for paraloading the entire file by pipelining requests. Notice this simulation gives a conservative estimate of the theoretical pipelining case as there may be a better request assignment among the connections.

Table 3 shows the simulated results for the different degrees of $|M'|$. The average improvement of pipelining over the non-pipelined case is about 38%. As shown, the pipelined simulation shows a significant performance improvement in our experiment. Also, we note that the improvements we have obtained in the pipeline simulation is much greater than the improvements due to pipelining obtained in [21]. We believe that this is caused by the higher average bandwidths of the network we used. As explained above, higher bandwidths lead to a higher ratio between idle time and data transfer time. Since the pipelining optimization is designed to eliminate this waste, it is not surprising to see a greater performance improvement in our simulated experiment.

Table 3: Compares the performance of a paraloader with the pipelined request optimization. The same averaging method used in Table 1 was also used here to construct the table values.

$\|M'\|$	Average time without pipelining (s)	Average time with pipelining (s)	Speedup
3	11.025	8.320	36.15%
5	6.548	4.787	38.70%
7	5.462	3.948	42.09%

7 Discussion

Up to this point, the paper has focused on performance issues. In this section, we attempt to discuss other issues that may become important for deploying a complete, efficient paraloading system.

7.1 Cost of paraloading

While paraloading may significantly improve download performance, it does not come without cost. A good paraloading system should consider the cost of its design. Here, we outline three general types of costs that would appear in any paraloading scheme:

1. First, as we have seen, there is a processing overhead at the receiver for scheduling block assignments among the parallel connections.

2. Second, there is a memory overhead associated with creating an application buffer large enough to hold data blocks that arrive out of order. In our current implementation, the memory cost is $|M'| \cdot F$ bytes. In addition, there is also a memory cost associated with opening a new receiver socket for each parallel connection.

3. Third, there is the cost for the block request messages to generate extra traffic in the network. The magnitude of this cost depends greatly on the block size and on the paraloader's underlying protocol.

4. Fourth, there is an increase in server resource consumption as paraloaders open multiple connections. Consequently, the maximum number of clients per set of mirror servers will decrease. We will use the following algebra to illustrate what we mean. The consequence of this cost is that the total resources at all the mirror servers must increase by a factor A (defined below) if those servers wish to provide service to the same number of clients as the serial downloading case.

Define M to be number of mirror servers and N be the maximum number of simultaneous connections per server. Let A be the average number of parallel connections per client where each connection connects to a different server and C be the average number of parallel connections per client where each connection connects to a single server. Let V be the total number of clients served.

In the serial downloading case, $V_{serial} = NM$, in the paraloading case, $V_{paraload} = (NM)/A$, and in the cases where parallel connections are opened to a single server, $V_{parallel} = (NM)/C$.

7.2 When To Use Paraloading

There are situations where we do not want to start paraloading. An intelligent paraloader must devise heuristics to determine when it should start paraloading or when to revert to the serial download scheme. We outline three such situations below.

- As already shown in our results, paraloading does not provide much of a performance gain when it is used to retrieve a single small file. When the performance gain is too small, a paraloader should revert to serial downloading to conserve network resources.

- In the case where there are a few *outlying* connections with relatively high bandwidths, that is, when all other connections have very low bandwidths relative to the outliers, almost all block requests will be assigned to the outlying connections. In this case, a paraloader may wish to drop the slow connections to conserve server resources without significantly affecting performance. In the case where there is only one outlying connection, an intelligent paraloader should revert to a serial download from the single outlier in order to save the block request overheads. Note that this effect can be emulated by increasing the block size dynamically.

- Finally, as we already mentioned in the Assumptions section, we note that paraloading may become overly aggressive in a congested network and start to dominate other TCP-friendly flows. An intelligent paraloader should be conservative and drop connections that are sharing a bottleneck. However, detecting which set of connections share a bottleneck is a very difficult problem[12].

7.3 Open Research Issues

Here is a list of open research issues relevant to the design of a successful paraloading system.

[12]We believe that the paraloading scheme will provide many interesting ways to attack this or a similar problem. For example, the paraloader can treat block request messages as "probe" packets to measure the response time of the connection. Also, it is not necessary to drop bottleneck-sharing connections. The paraloader can interleave block requests among the bottleneck-sharing connections to counter their aggressive behavior.

- Detecting shared bottlenecks and minimize the potential aggressiveness when sharing a bottleneck with existing connections.

- Developing a common API that offers paraloading services to a variety of different applications (e.g. ftp, http).

- Determining whether paraloading should be implemented at the application level or at the network level.

- Designing a directory service that reports the locations of all the mirror servers containing the same document.

- Determining whether network scope affects throughput of mirror servers. If we can find out how network scope affects throughput, then a paraloader can open parallel connections to a specific set of mirror servers without relying on explicit network metric information.

- Designing a process for a cheap, easy, fast, safe, and secure mirror update and replication that can handle ongoing paraloading sessions. When there is an ongoing paraloading session, a file cannot be updated easily across all servers. Also, updates should be done cheaply and easily. A paraloader cannot be widely deployed unless there is a robust mirroring infrastructure available. Finally, we need a mechanism to verify the integrity of a file that has its various parts downloaded from different mirror servers.

- Determining and developing solutions to the security issues related to paraloading.

- Determining the implications of the wide-spread use of paraloading. More specifically, could paraloading shift network congestion points closer to the edge (the clients) of the network? If so, what impact would it have on the existing network environment?

8 Future Research

Our immediate research goal is to conduct more paraloading experiments on different networks using different sets of mirror servers. We will attempt to implement as many optimizations as we can to see how consistent and how large the performance gains are. A couple of the optimizations that we are especially interested in include pipelining and dynamic block size adjustment.

Since the Internet is driven mainly by web applications today, we will be especially interested in examining how paraloading can be optimized to provide performance gain for retrieving web objects from different mirror servers. Initially, we plan to conduct a series of experiments with the non-optimized paraloader to help us design the web-optimized paraloader.

We also want to explore whether we can incorporate the *application level framing (ALF)* [7] idea into determining the block size for retrieving web objects. Doing so may give the paraloader the flexibility to perform out-of-order delivery for web applications.

9 Conclusion

In this paper, we have described a simple method of decreasing download time. By paraloading, opening parallel connections to multiple mirrors, download time can be decreased. Moreover, because opening multiple connections provides an aggregation of the individual bandwidths, the improvement can be significant.

In an attempt to explore whether paraloading would perform better than traditional downloading, we performed some paraloading experiments to various mirror servers. During the course of our experiment, we determined that the number of servers used (the degree of parallelism) and the size of blocks a file is broken into can have great impact on the download time. Overall, our results show that paraloading consistently downloads files faster than traditional downloading on the average case. In addition, we simulated several different optimizations using the data collected as the starting foundation. We discovered that download time can be further decreased by adding a number of enhancements. In particular, pipelining requests for blocks and employing an intelligent policy for downloading the last block can have tremendous impact on download time.

In conclusion, based on our results, we believe that paraloading can be beneficial in any network environment provided the implementation of the paraloader application is robust.

10 Acknowledgements

We would like to thank Dave Anderson for providing us with some scripts to help with analysis of the data, Dorothy Curtis for helping us install some of the key applications for our experiments, and all those who provided insightful comments during our class presentation. In addition, we would like to thank Wendi Heinzelman for reviewing our paper. Last, but not least, we would like to thank Hari Balakrishnan for guiding us throughout this project.

References

[1] Agiletp. http://members.xoom.com/_XMCM/2Cities/software.htm, December 1999.

[2] Apache server. http://www.apache.org, November 1999.

[3] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. Katz. Tcp behavior or a busy internet server: Analysis and improvements. In *Proceedings of IEEE INFOCOM 1998*, pages 252–262, March 1998.

[4] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. In *Proceedings of ACM SIGCOMM 1999*, pages 175–187, September 1999.

[5] John Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *Proceedings of IEEE INFOCOM 1999*, pages 21–25, April 1999.

[6] Robert Carter and Mark Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proceedings of IEEE INFOCOM 1997*, pages 1014–1021, April 1997.

[7] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, pages 200–208, September 1990.

[8] Zongming Fei, X. Bhattacharjee, Ellen W. Zegura, and Mostafa H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE INFOCOM 1998*, pages 783–791, March 1998.

[9] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. In *RFC 2616*, June 1999.

[10] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control. *IEEE/ACM Trans. on Networking*, 7(4):458–472, August 1999.

[11] Go!zilla. http://www.gozilla.com, December 1999.

[12] Van Jacobsen. Pathchar. ftp://ftp.ee.lbl.gov/pathchar/, April 1997.

[13] Van Jacobsen, Tim Seaver, Ken Adelman, C. Philip Wood, Dan Nydick, Jamshid Mahdavi, Jon Boone, and Ehud Gavron. traceroute (with modifications).

[14] J. Kangasharju, K.W. Ross, and J.W. Roberts. Locating copies of objects using the domain name system. In *Proceedings of the 4th International Caching Workshop*, March 1999.

[15] Marc Langheinrich. Parallel user agent. http://bauhaus.cs.washington.edu/homes/marclang/ParallelUA, November 1999.

[16] Steve Lewontin and Elizabeth Martin. Client side load balancing for the web. http://decweb.ethz.ch/WWW6/Poster/707/LoadBal.HTM, October 1999.

[17] Andy Myers, Peter Dinda, and Hui Zhang. Performance characteristics of mirror servers on the internet. In *Proceedings of IEEE INFOCOM 1999*, pages 304–312, March 1999.

[18] Netscape navigator. http://www.netscape.com, December 1999.

[19] V. N. Padmanabhan and J. Mogul. Improving http latency. In *Second World Wide Web Conference '94: Mosaic and the Web*, pages 995–1005, October 1994.

[20] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.

[21] Pablo Rodriguez, Andreas Kirpal, and Ernst W. Biersack. Parallel-access for mirror sites in the internet. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[22] S. Seshan, M. Stemm, and R. Katz. Spand: Shared passive network performance discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[23] Ronald Tschalar. Httpclient. http://www.innovation.ch/java/HTTPClient/, December 1999.

Appendix A

Some Peculiarities of HTTP 1.1 implementations

During the course of our experiment, we discovered several peculiarities with the HTTP 1.1 specification that can cause problems for our paraloader. While the HTTP 1.1 specification is designed to have more stringent requirements than HTTP 1.0, implementors of the specification still have a great deal of leeway. As a result, server behavior on certain types of requests can be unpredictable. The following is a list of server behavior that we had to pay attention to during analysis:

- Servers are not required to honor every `RANGE REQUEST` command. The only requirement for servers is that they must respond with some data that covers the requested range. This means that the server can send data exactly in the requested range, some data that includes and exceeds the requested range, or even the entire file itself. Clearly, if a server chooses to send more data than that requested, there will be added overhead for the paraloader. Furthermore, the efficiency of paraloading can be dramatically decreased. In the worst case, if every connection chooses to send the entire file, there would be no gain in download performance.

- Servers may choose *not* to honor range requests for certain types of files. We discovered that servers may not allow range requests for HTML files such as `index.html`. However, range requests was allowed for larger binary files such as JPEG and GIF files. Hence, any implementation of the paraloader must not assume that range requests are honored for every file format.

- The HTTP 1.1 specification does not require all servers to support persistent connections. Furthermore, for those servers that do support persistent connections have the option to close it after any request transactions. Hence, any implementation of the paraloader must check for this condition carefully and avoid using HTTP1.1 servers that do not support "persistent" persistent connnections. Otherwise, reopening a connection on every block request would incur too much overhead, thus degrading the overall download performance.