Coral Snake - a High-Performance Distributed Striping HTTP Downloader

6.894 – Distributed Operating Systems – Final Project, December 2000

Anthony Johnson, Noshirwan Petigara, Daniel Vlasic

{anthonyj, nosh, drdaniel}@mit.edu

Abstract

In recent years, the emegence of broadband access to the internet has revealed several bottlenecks within the Internet. Due to these bottlenecks, broadband clients are unable to utilize their bandwidth effectively. In order to overcome these bottlenecks, we propose an architecture to stripe file downloads across multiple servers simultaneously over HTTP/1.1. By striping, we can raise the effective bandwidth that is utilized by the client. Striped downloading of replicated files fits well with existing Internet architecture as mirror sites for servers are readily available. We present Coral Snake, a HTTP/1.1 striped file downloader, developed in Java. We discuss our algorithm for efficiently striping files and document results showing that Coral Snake can greatly speed up file transfers in most situations.

1. Introduction

The development of broadband Internet access over the past couple of years has lead to a dramatic change in the needs of users. Until recently, the limitation on the speed of data transfer was the speed of a modem. However, with broadband connections, the user is now confronted with the actual bottlenecks within the Internet itself. These bottlenecks are often the servers themselves, which can be single Unix machines serving the entire community (e.g. www.rpmfind.com). This may lead to the server being overwhelmed by requests, or intermediary network hops being congested, which can bring the data transfer rate down considerably.

Data transfer rate, however, is not the only consideration. While the data rate achieved is the only heuristic that is important in a file download situation, there are many cases in which it is necessary that the data arrive in order as best as possible. Examples of this are such data as real-time video, in which the player must be able to start playing the video without having to wait for the entire file to download.

This paper first presents some previous work done in the field, followed by an explanation of the basic technique used. Once this has been shown, the algorithm is presented, followed by an explanation of the implementation used. Finally, we present the performance of the algorithm under real conditions, and conclude with possible future work.

1.1 File Striping

The technique that is used in our solution uses file striping. File striping effectively divides the full file into separate request, each of which is then brought in from separate servers. In theory, as the number of servers increase, it should become possible to saturate the client's bottleneck.

In a simple implementation of file striping, the client requests a file fragment of size (total file size)/(number of servers.) The main disadvantage of this approach is that the completion of the download will be entirely dependent on the slowest server. In order to optimize this, it becomes necessary to partition the file fragment requests in a manner that minimizes the total download time. This is dealt with in the algorithm section of this paper.

1.2 Previous work

Although it seems that little work has been done in this field with respect to simple over-the-Internet file downloading, a considerable amount has been done in order to lower download time for clients.

Akamai and Web Caches

The most prominent work in this field has been done by Akamai. In order to increase bandwidth to clients, they have replication servers located throughout the network topology, often with the ISPs themselves. This gives them a high-bandwidth to the backbone that the client is connected to.

This is a very simple and useful system for medium and large companies who can afford to access these sort of resource. It is unlikely to be used, however, by such things as open-source projects, where there are large files to download, but not the financial backing to afford Akamai.

Download Managers

Download managers increase download transfers by two main methods: multiple connections to single servers and optimizing of server selection. They often make multiple connections to the same server in order to saturate the bandwidth available to the client. This probably works as a result of routers attempting to share bandwidth equally among connections.

The optimizing of server selection is usually limited to using the ping time as a measure of the data rate. While at times this is an accurate measure, we also found through manual testing that this is not ideal, since servers can sometimes have low ping times, even though they are overloaded and therefore having a low bandwidth.

The choice of mirrors selected is often based on a database of common sites, and their replicas. This manager therefore works best for often used sites such as http://www.download.com and the such. They are, however, not able to get information directly from the server, which means that non-listed servers are not dealt with well.

Striping Download Managers

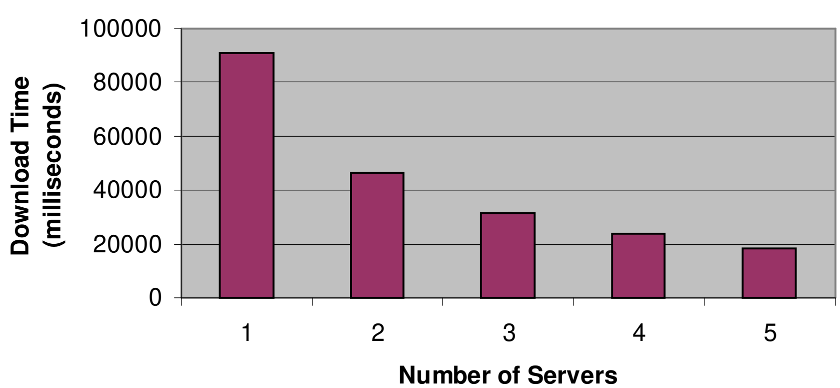
The newest generation of download managers combine these features, and perform a form of striping. They connect to multiple serves and download parts of the file from these servers, and recombine them at the end. The main limitation of this approach is that, despite the increase in speed, the file is not assured to come in order, which makes it inappropriate for anything other than simple downloading, since streaming-video must come in order.

2. File Striping with HTTP

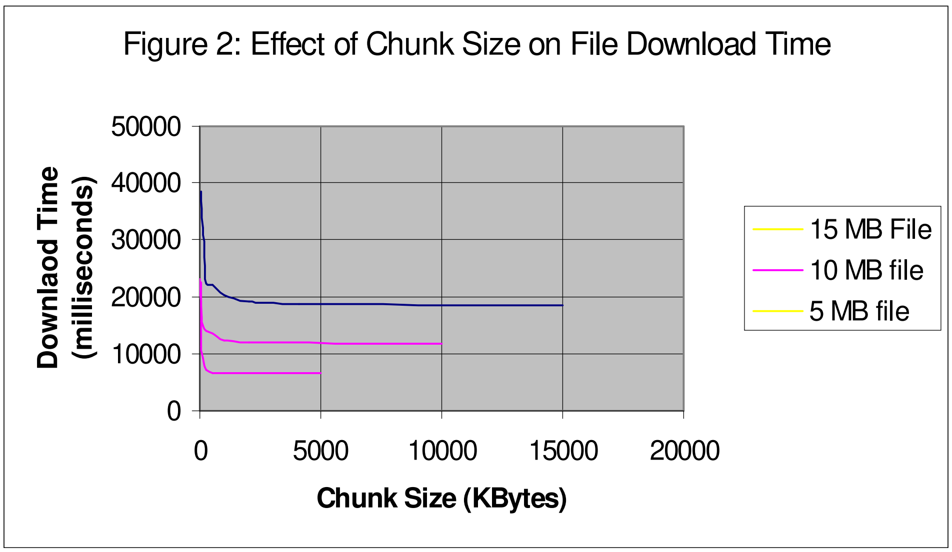
In a file-striping algorithm, a higher effective bandwidth is achieved by requesting different portions of a file simultaneously from different servers. In the simplest file-striping algorithm, the file is divided equally between multiple servers and each part is retrieved simultaneously from the different servers. This algorithm performs optimally when the bandwidth and latency between each server and the client is equal and constant. In this optimal situation, the effective bandwidth that can be achieved can be calculated as $\beta \times N$, where β is the constant bandwidth between each server and the client, and N is the number of servers. To see whether we could achieve this theoretical maximum file transfer speed with N servers, we conducted the following experiments. Web servers were setup on different machines on a local area network, to bring network delays to a minimum. The bandwidth of each server was kept equal and constant. We then divided the file between these servers and requested these chunks simultaneously. The results for this can be seen in Figure 1. It can be seen that the performance achieved is extremely close to the ideal of 1/(number of servers).

As we were designing Coral Snake to stripe files over the Internet using the HTTP/1.1 protocol, we had to adapt our algorithm to the network and protocol characteristics. The most important

Figure 1: File striping in on a controlled network



consideration is that all servers on the Internet are not created equal. There are substantial differences in bandwidth, processing power, and server load between different servers and even with the same server at different times. If the simple algorithm described earlier is used to stripe files, the download time for the striped file would be constrained by the slowest server. In order to work around this limitation, the file should be divided between servers according to their speed. However, server speed is not constant and hence this calculated file distribution between servers can change over the course of the download. To account for this instability, each server should be assigned a small chunk proportional to its speed. Once this chunk is downloaded, it can be assigned a new chunk proportional to its new speed, calculated over the time of the download. As downloads occur over HTTP, each request for a chunk of the file has an overhead that must be factored into the algorithm. A request has to be generated by the client, which travels across the Internet to the server. This request has to be processed by the server, which sends back the requested chunk along with a header.



Consequently, retrieving a file in chunks using HTTP can had adverse affects on download times unless requests are issued sparingly. To investigate these effects, we performed the following test. We ran a web server and web client on different machines on a high-speed local area network. The setup allowed us to ignore bandwidth and round-trip times in order to isolate the effects of the HTTP requests and headers on file download times. For each test, we started with a small chunk size and issued consecutive HTTP requests for sequential ranges of this size until the whole file was received. On each subsequent run, we increased the chunk size incrementally until the chunk size was equal to the file size. The results for 5MB, 10MB, and 15MB files can be seen in figure 2. As predicted, small chunk sizes gave higher download times because more requests are issued to the server. However, we observed that this effect tapered off with a chunk size of about 10% of the file size. This gives a good indication of the minimum chunk size that could be used to download files without observing deterioration in download time.

We also had to ensure that we did not stripe files that were so small that the HTTP headers accounted for a substantial portion of the data transmitted. We averaged the request and response headers from a number of servers and found that their added size was around 400-500 bytes. From this we decided that the minimum chunk that could be requested from any server would be 8 Kbytes (multiply the average header by 20 to get more than an order of magnitude larger). This ensured that any chunk that was requested from a server would have high ratio of data to headers.

3. Algorithm

3.1 Assumptions

The files downloaded using our striped downloading utility are replicated on a number of servers. For each requested file, we assume that the list of N best servers replicating that file is available. If the means were available, this list could be obtained from the server mentioned in the client's request by a specialized protocol, or from some known (centralized or distributed) database. Ideally, this protocol or database would also provide the file size, but we do not assume that.

3.2 Overview

As explained earlier, file striping over the Internet poses difficult problems. To start with, the N best servers each exhibit different performance charecteristics. Therefore, the percentage of the file downloaded from each server should be proportional to server's speed. Furthermore, the performance of each server is likely to vary throughout the download. Thus, the amount of data requested from each server should be dynamic as well. Finally, a server could slow down to a crawl or even stop sending data (due to either server failure or network failure). To counter that, the algorithm needs to detect this condition and recover from it.

Our high-level algorithm is as follows:

```
repeat
   for each server S
          assign a certain range (e.g. bytes 0-8191)
          send an HTTP range request to S
          save the reply content
          update speeds
          assign a new range
```

```
7. until the whole file is retrieved
```

3.3 Getting the File Size

Prior to striping, the algorithm has to obtain the file size. The file size is retrieved from the header of the first response. One server sends a request for the initial range of length *initialLength* (which we calculated earlier to be 8 Kbytes) and extracts the file size as soon as it parses the response header. If the file size is greater than this initial length, the algorithm can then assign ranges to other servers. The initial range is chosen such that short files not worth striping are downloaded from only one server with only one request.

3.4 Assigning Ranges

The ranges for each server are taken sequentially from the portion of the file still to be downloaded. This way our utility can receives the file chunks roughly in order, and we can sequentially write them from memory to disk or forward them to a client. This makes Coral Snake suitable for streaming media applications. Initially, as we do not know how "good" each server is, we assign each server ranges of the same length (referred to as *genericLength*). Later on, however, the range length reflects the server's performance.

In our implementation, we define three heuristics for server performance; we call them *globalSpeed*, *localSpeed*, and *betweenSpeed*. We update these every 0.5 seconds. The *globalSpeed* of the download at time t is calculated by dividing the total bytes read from the server by the time from the beginning of download. The *localSpeed* is equal to the number of bytes read since the previous update divided by the time since last update. The *betweenSpeed* is defined as the average of previous *betweenSpeed* and *localSpeed*, and it can be thought of as in between the global speed and local speed -- not quite as dynamic as the local speed, but more sensitive than the global speed.

$$globalSpeed(t) = \frac{TotalBytes\mathrm{Re}\,ad}{currentTime - initialTime}$$

$$localSpeed(t) = \frac{bytes\,\mathrm{Re}\,adSinceLastUpdate}{currentTime - lastUpdateTime}$$

$$\mathrm{between}Speed(t) = \frac{globalSpeed(t) + localSpeed(t)}{2}$$

Using these performance heuristics, we calculate the range length for a particular server proportional to *betweenSpeed* of that server and the *genericLength*. The *genericLength* is, in turn, proportional to *fileSize*. Hence, the range length is proportional to $\mathrm{betweenSpeed} \times \mathrm{fileSize}$.

$$\mathrm{normalizedSpeed} = \frac{\mathrm{betweenSpeed}}{\sqrt{\mathrm{betweenSpeed}_1{}^2 + \ldots. + \mathrm{betweenSpeed}_n{}^2}}$$

$$\mathrm{rangeLength} = \mathrm{normalizedSpeed} \times \mathrm{genericLength}$$

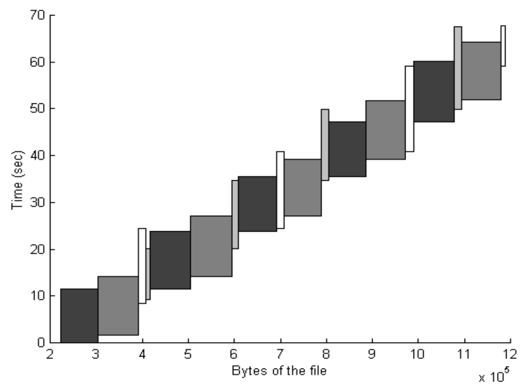


Figure 3a: Coral Snake Algorithm – 4 Servers

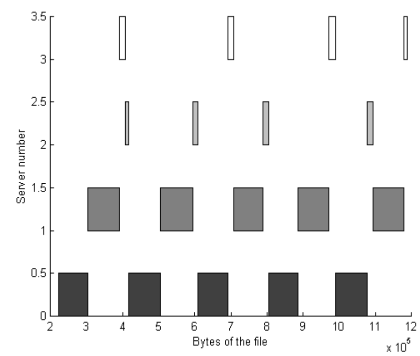


Figure 3b: distribution of requests to servers in figure 3a

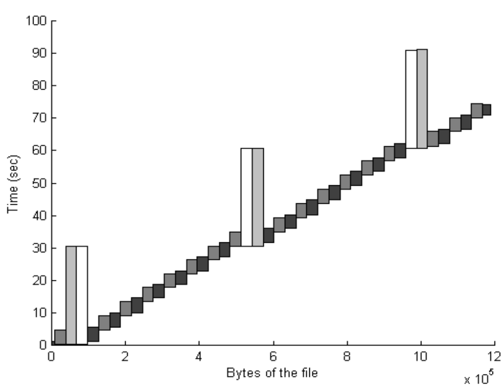


Figure 3c: Simple Algorithm - 4 Servers

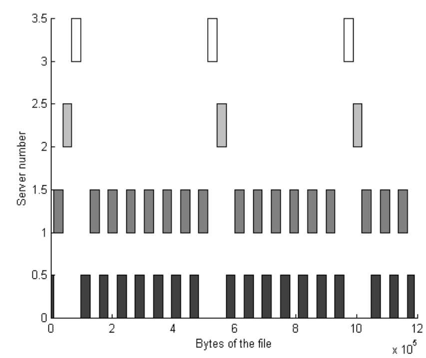


Figure 1d: distribution of requests made by servers in figure 3c

The overall effect that this algorithm has on the distribution of range requests can be seen in Figure 3. In Figure 3a, we can see that the manner in which file chunks are requested are requested and received are sequential, while in Figure 3b, we can see the actual distribution of requests between servers over the whole length of the file. This distribution of chunks with the Coral Snake algorithm can be compared to the distribution of chunks by the much simpler algorithm in figures 3c and 3d. In this algorithm, all requests to striping servers are of the same size. As we can see, the slowest server dominates the download time and breaks the sequential ordering of bytes received. All of these tests were done with four different servers in three different countries, to get differing bandwidths from each of the servers.

3.5 Abandoning Ranges

Since Internet is unstable and unreliable, the performance of a server can drop severely or even go to zero. This can be caused network congestion, network partition, or server failure. Due to this, it is possible that a large chunk is assigned to a server, whose performance then suddenly drops to a crawl midway through the transfer. In order to prevent this from affecting download time or sequential ordering, the algorithm "abandons" the current range. Specifically, it saves the part of the range that has already been downloaded, puts the remaining part in the set of abandoned ranges, resets the connection to that server, and assigns it a new range based on the newly calculated speed. As a result, the algorithm draws new ranges from the abandoned set (if it is nonempty) before it looks at the unassigned range. The connection has to be reset because we are dealing with HTTP messages, and there is no way of halting the request while keeping the connection alive.

The crawling behavior is detected upon updating server speeds. A server is labeled as crawling if the following three conditions are true:

1. its current speed is much slower than the average speed,
2. its current speed is much slower than at chunk allocation,
3. the remaining part of the chunk is much larger than what the algorithm would assign this server with its current speed

4. Implementation

For the development of Coral Snake, we decided to use Java for a variety of reasons. First, we felt that Java's platform independence was in keeping with the nature of the WWW. More importantly, the authors all used multiple platforms for development. Secondly, we have observed that recently a lot of developers have successfully developed distributed applications using the Java programming language. Lastly, we felt that Java's programming paradigm and easy to use APIs would allow us to concentrate on the development on the algorithm. We used Sun's JDK 1.3 for development.

We used the standard network packages to setup our connections to the multiple servers. Each connection to a server runs on a separate thread. As we have very few shared data structures, co-ordination was not a problem. We used an open source regular expression package from Apache to do the parsing of the HTTP headers. We designed the code base for Coral Snake such that it would be easy to add or change striping algorithms. This was very important during our algorithm development stage, when we were constantly modifying and switching algorithms. We also found the *Timer* classes in Java very useful for updating server speeds at a constant rate.

In general, we were pleased with our development experience with Java. The performance was satisfactory and we experienced no perceptible slow downs in code. However, we did observe that the code did take quite long to load on Solaris. However, we observed no such problems on Windows machines.

One problem that our implementation addresses is that there is a maximum number of requests that are allowed per server. Within a single "Keep-Alive" connection, the server can choose to terminate the connection if the actual number of request exceeds some limit, usually 100 or 200 requests. We resolved this by simply reconnecting the server, even though this can impact performance.

5. Results and Performance

Our goal was simple – we wanted Coral Snake to achieve the performance over the Internet that we had observed on a controlled network. In no possible situation did we want a Coral Snake download to be slower than a download from a single server. Our results show that Coral Snake comes close to achieving this theoretical maximum speed. To analyze the performance of Coral Snake, and insure that it met all of specifications, a number of situations had to be tested. We performed a variety of tests using different file sizes and different number of servers.

Figure 4 shows the performance of Coral Snake with different numbers of servers. Figure 4a graphs the download time of a 5MB file, while figure 4b show the bandwidth of the servers during this transfer. These tests were performed on a set of Mozilla mirrors scattered across the globe. Each test was performed ten times and averaged. As we can see, the effective bandwidth is close to the sum of the bandwidths of the individual servers. To understand how Coral Snake achieves this high level of performance, we have graphed the distribution of chunks between servers of different sizes.

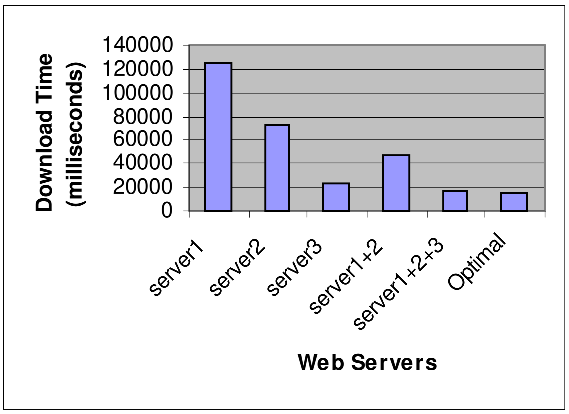


Figure 4a: Download time for a 5MB file transfer

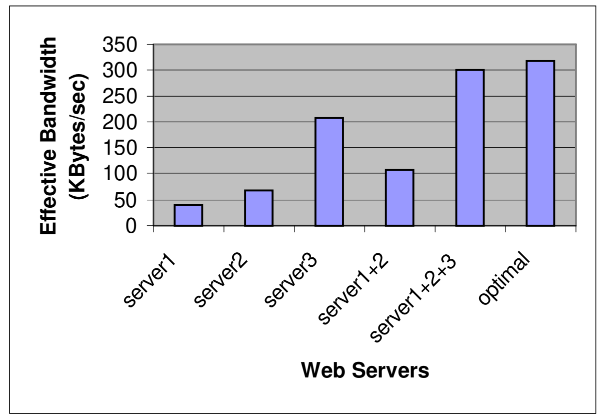


Figure 4b: Server Bandwidth during 5MB file transfer

Figure 5 shows the chunk distribution over in a similar download with three servers. In Figure 5a each different shade of gray represents one of the three servers used. The algorithm distributes the chunks between the different servers according to size. As expected, the wider chunks are also the slightly shorter ones. In addition, the chunks taken progress linearly through the file. This shows that our algorithm is actually able to provide the client with an uninterrupted stream of data.

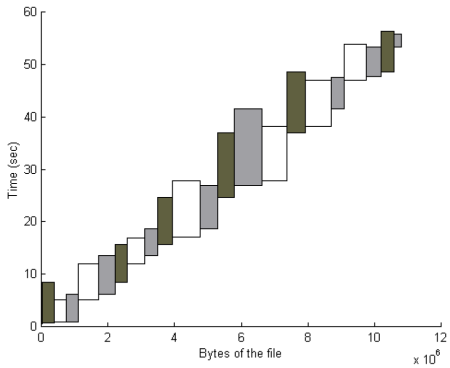


Figure 5a – request ranges vs. time for 3 servers (different shades indicate different servers)

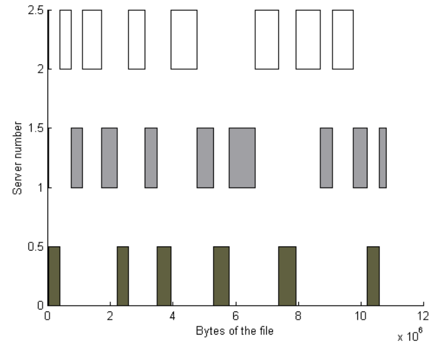
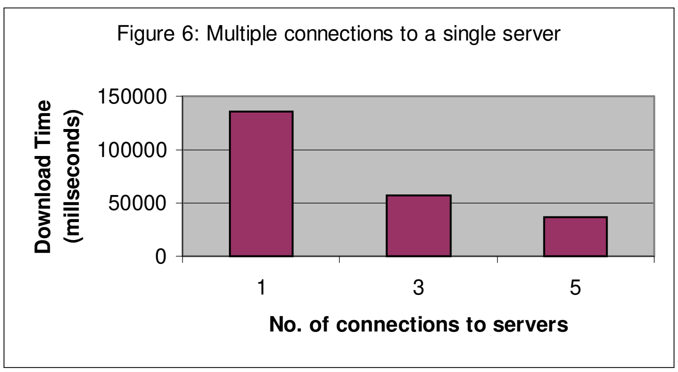


Figure 5b – request ranges vs. servers.

Figure 5b shows the actual distribution of chunks across servers – it is easy to observe the uneven distribution of the requests, with its preference for the faster servers.

In addition to the performance that can be gained from connections to multiple servers, we analyzed performance increases gained from the use of multiple connections to single servers. Figure 6 shows that the increase in bandwidth gained, which while not as good as from multiple servers, is considerable. We have surmised that this is due to an attempt by intermediary routers to share bandwidth equally among connections, rather than clients. We chose not to open multiple connections to a single server as results are inconsistent. However, in the case that no mirrors are available, multiple connections can be opened to the same server.



An interesting situation occurs when one server can saturate the client's bandwidth. In this case, the maximum file transfer speed can be achieved by downloading the file from this server-striping the file from multiple servers cannot decrease download time. Situations such as these can occur when the client's link is slow or the client link experiences high congestion. We wanted to ensure that that Coral Snake algorithm did not degrade in this situation. To simulate this, we ran Coral Snake and a web server on a local area network. The web server on the LAN was able to saturate the client's bandwidth. We added this server to Coral Snake's list of mirrors and proceeded to download the file. The Coral Snake algorithm was able to adapt to this situation and performed comparably to the download directly from the local web server.

6. Conclusion and Further work

Our results from Coral Snake have shown that downloads over the web can be sped up by close to a factor on N using our algorithm, where N is the number of servers used to download a file. In developing Coral Snake, we designed for simplicity. We allow the user to the number of servers to be used for striping. A useful addition to the algorithm would be to automatically detect how many servers to stripe a particular file from. In such an algorithm, striping would start with a small number of servers. Additional servers would be added incrementally until no further increase in bandwidth was detected. Theoretically, this could lead to a complete saturation of the client's bandwidth and faster downloads. In addition, we feel that the process for deciding when to abandon striping from a particular server could be improved. We currently use a static metric, which errs on the side of caution. With some fine-tuning, this could improved to only abandon connection when absolutely necessary.

Despite compelling results, there is a still a major obstacle that prevents deployment of Coral Snake or similar software. There is currently no good method to detect mirror sites for files across the web. Although mirror sites could be stored in some distributed database, we believe that the web servers, using an extension to the HTTP/1.1 protocol, could best deliver mirror information. We propose that a header be added to the HTTP reply such that web clients could be informed of available mirrors. A possible implementation of such a header could have the following format:

```
Mirrors:www.mirror1.com/path1,www.mirror2.com/path2,www.mirror3.c
om/path3
```

This would allow web clients to discover possible mirrors during the first HTTP transaction with a server. In addition, this would allow only the administrators of the web site to add approved mirror sites for striping.

In conclusion, we feel that Coral Snake is a viable solution to increase bandwidth utilization on the client side. With additions to existing WWW protocols, Coral Snake could be widely used.