CORE
Metadata, citation and similar papers at core.ac.uk
Provided by DigitalCommons@CalPoly

HTTP 1.2: DISTRIBUTED HTTP FOR LOAD BALANCING SERVER SYSTEMS

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Graham Michael O'Daniel
June 2010

© 2010
Graham Michael O'Daniel
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	HTTP 1.2: DISTRIBUTED HTTP FOR LOAD BALANCING SERVER SYSTEMS
AUTHOR:	Graham Michael O'Daniel
DATE SUBMITTED:	June 2010
COMMITTEE CHAIR:	Michael Haungs, Dr.
COMMITTEE MEMBER:	Aaron Keen, Dr.
COMMITTEE MEMBER:	Franz Kurfess, Dr.

ABSTRACT
HTTP 1.2: DISTRIBUTED HTTP FOR LOAD BALANCING SERVER SYSTEMS
Graham Michael O'Daniel

Content hosted on the Internet must appear robust and reliable to clients relying on such content. As more clients come to rely on content from a source, that source can be subjected to high levels of load. There are a number of solutions, collectively called load balancers, which try to solve the load problem through various means. All of these solutions are workarounds for dealing with problems inherent in the medium by which content is served thereby limiting their effectiveness. HTTP, or Hypertext Transport Protocol, is the dominant mechanism behind hosting content on the Internet through websites. The entirety of the Internet has changed drastically over its history, with the invention of new protocols, distribution methods, and technological improvements. However, HTTP has undergone only three versions since its inception in 1991, and all three versions serve content as a text stream that cannot be interrupted to allow for load balancing decisions. We propose a solution that takes existing portions of HTTP, augments them, and includes some new features in order to increase usability and management of serving content over the Internet by allowing redirection of content in-stream. This in-stream redirection introduces a new step into the client-server connection where servers can make decisions while continuing to serve content to the client. Load balancing methods can then use the new version of HTTP to make better decisions when applied to multi-server systems making load balancing more robust, with more control over the client-server interaction.

ACKNOWLEDGEMENTS

I'd like to acknowledge all the individuals in my life that stuck by me throughout the course of the thesis. So to my mother, Lee Anna, and father, Don, for supporting my educational efforts, to my wife, Julie, for lending me emotional support, to Dr. Haungs, for his continuing support as my committee chair, and to Dr. Keen and Dr. Kurfess, for giving me the opportunity to present my work, thank you all.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

1 INTRODUCTION

Internet sites like CNN[5] often provide content that is in very high demand. High demand content introduces loads on servers that provide the content as the number of clients wanting the content increases. An increased load can lead to problems encountered at clients' sides including lost connections, slow downloads, and slow response times, all of which reflect negatively on content providers.

It did not take long to realize that something had to be done in order to reduce the problems introduced by such scenarios, which led to the idea of load balancing. The basic idea behind load balancing is to provide multiple servers with identical content and distributing clients to servers in a fashion that avoids requiring a server to deal with too many clients at one time. Load balancing of Internet sites has become a critical field of study[17]. As the Internet hosts more and more large files such as video, software, and scientific data sets it is crucial that load balancing mechanisms account for long, persistent downloads.

There are a few commonly used load balancing methods, which have been implemented by various content providers over the years. Some rely on DNS (Domain Name Service) to provide clients with the IP of an unloaded server from a pool of servers defined as the multi-server system[6]. Another common method involves rewriting URLs (Unique Resource Locators) to point to unloaded servers from the multi-server system[2]. Yet another common method involves clusters of servers with one or many points of entry to serve as dispatchers to back-end content servers[17].

The three common methods of load balancing provide a substantial level of beneficial load balancing, but each comes with disadvantages. Relying on DNS-based methods means very coarse grained control with potential delays in response times due to the nature of DNS. Using a URL-rewriting method means creating and modifying links throughout a webpage, which provides coarse grained control and links that are not humanly decipherable. Clusters imply locality and can only deal with servers defined in the cluster, which leads to problems with scalability and the occlusion of WAN-based solutions. Finally, none of the above methods can specifically manage data transfers once they have begun.

We propose work that addresses the disadvantages experienced above and introduces additional levels of control to administrators of content servers by changing HTTP (Hypertext Transportation Protocol). Since 1991, HTTP has governed how most content is requested by clients and delivered by servers[22] over the Internet. Applications like Web browsers and Web servers all implement algorithms to handle HTTP, which means they can work together no matter who provided the application and no matter what operating system is used. By suggesting a change to HTTP, we hope to keep with the goals of cross-platform, cross-application portability, while still achieving a way of introducing a load balancing method.

Our work moves to modify HTTP to packetize the stream of content served through HTTP, include a set of control messages for insertion between packets, and finally define in detail a control message for use in in-stream redirection of clients.

The major feature of our HTTP, or HTTP 1.2, as it will be referred to from this point forth, is the support of in-stream redirection. In-stream redirection means a client can be redirected to another server after starting the download of content from a server. This ability leads to a particularly fine level of control not experienced by other load balancing methods. Our work makes use of current HTTP features to organize data into packets to intersperse redirection control messages between packets of content. TCP uses packets to achieve a similar level of control[8].

The significant areas of contribution included in HTTP 1.2 consists of the following:

- Stream-controlling primitives, i.e. control messages.
- Packetizing of streamed HTTP content.
- In-stream redirection of content between multiple servers.

This work lays out a path by which further developers can follow in order to fully develop the architectures and features supported in HTTP 1.2. The work done in this solution is to introduce the idea of adding redirection control messages. The solution is presented as a framework, not an end-all conclusion.

We discuss some other load balancing methods along with some of their advantages and disadvantages. Almost all the other methods can be combined with HTTP 1.2 to offer a multi-layered approach to load balancing. Such a combination of tools could lead to a level of control not yet provided by any single mean.

Later we will discuss the history behind HTTP, and why we chose HTTP as a starting point for creating the HTTP 1.2. We also provide the details about HTTP 1.2, along with a reflection back on the current HTTP. An experimentation section provides experimental results that indicate the usability and feasibility of using HTTP 1.2. Finally we discuss our findings and further work that can be done to advance HTTP 1.2.

2 BACKGROUND

Before we begin to describe how and what HTTP 1.2 does, we should first provide a brief overview of where HTTP 1.2 comes from, namely HTTP and the World Wide Web client-server model. By discussing the basic client-server model we hope to highlight key areas of potential improvement.

The World Wide Web is founded on a client-server model: servers host content, and clients retrieve content by requesting content from servers. All information about the content must be exposed by the servers in a manner clients can understand. Such information includes how long the content is, what type of file the content represents, and who should have access to content. The server acts as a subordinate to the client, and must take commands from the client.

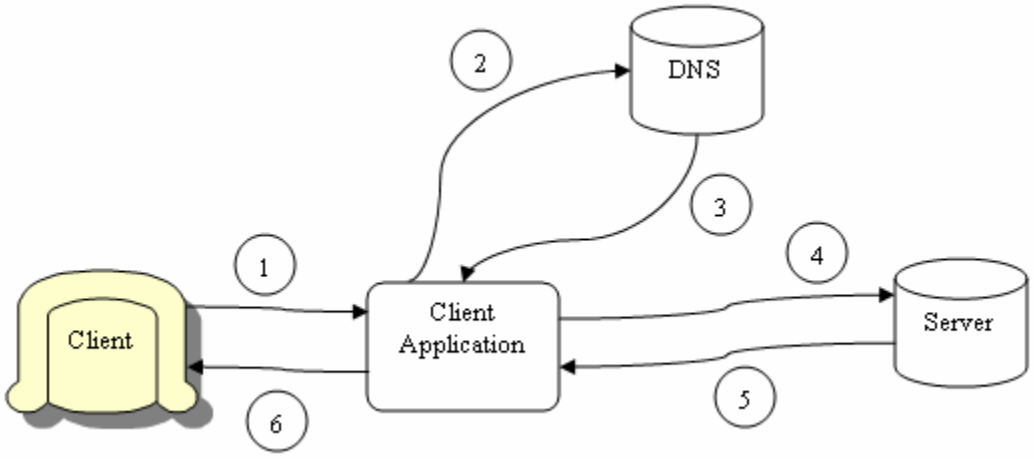

Figure 1: Basic client-server process

The client-server process is illustrated in Figure 1. Stage 1 – the client enters the website address and the application converts it into a web request. Stage 2 – the client application looks up the IP address of the server with its DNS. Stage 3 – the DNS responds to the

client application with the proper IP address. Stage 4 – the client application connects to the server issuing the request. Stage 5 – the server responds to the request. Stage 6 – the client sees their content.

Protocols are provided so clients and servers can communicate to accomplish a transfer of content. There are many different protocols used in conjunction to form the World Wide Web including HTTP, FTP, and TCP/IP just to name a few, but we will concentrate on HTTP.

HTTP is currently in version 1.1 coming from two predecessor versions dating to roughly 1990. It started as a simple protocol meant to share research information around various educational establishments but was soon recognized as a powerful tool for sharing other content as well.

The early assumption dictating the direction and construction of the early versions of HTTP is that content shared across networks is mainly text. The key concept related to the transmission of text is that it can be transmitted across the network much like reading a sentence letter by letter. This method of transmission is called a stream, and in this case, a text stream.

In stream-based transmission, the content is often unusable until it has been fully delivered. Imagine if you will living in a house as it is being built; the house is only usable once construction is complete. This poses a problem for load balancing methods,

because content that requires a long time to transmit will only be usable after all of the content is received. In a content-stream environment during the time the content is transmitted the load balancing method is quite limited in its decisions on what to do in loaded situations: continue serving the content but at a degraded speed, or disconnect some clients to speed up others. This is a fact because once a stream is started the client expects and should receive content from the stream until the stream is stopped.

The only load balancing method included in the definition of HTTP is the redirection status code that servers can issue when a client first connects. This redirection method is quite useful when new clients connect to a server that is overloaded because it provides the server to prevent serving any content to the new clients. However, we want to concentrate on events occurring after the client has connected to the server. The redirection facility in the HTTP specification is therefore inadequate for our needs. During the time content is transferred to a client, a server may incur extremely high load situations especially if the content is extremely large or the client has a slow connection. Therefore there is a lot of time in which making load balancing decisions would be beneficial to the server, but is simply wasted time in the current version of HTTP.

Fortunately the HTTP specification includes an area for transfer encodings that allow messages to be modified for transferring purposes[22], section 3.6. The specification remains quite vague in regards to any transfer encodings other than chunking.

Chunking was introduced as a transfer encoding in HTTP 1.1 in order to facilitate situations when not all the data is available when a client request is issued[22], section 3.6.1. Here chunks are served to the client one at a time until the server indicates the data is all served. The logic used here is a client will continue to receive chunks until either the connection is terminated or the data is all received. This is certainly a step in the right direction towards allowing finer control for load balancing because now the data is served in chunks or packets rather than a stream. We will in fact use chunking as a model to construct our own transfer encoding. The major problem with chunking is that the client must be expecting chunks and nothing more. In fact, the chunk header, information that appears before a chunk, is simply the length of the chunk in hexadecimal form. There is no specification for other chunk headers besides some tail headers that appear when all the chunks have been served.

Another avenue of HTTP that we explored but found little use towards load balancing, yet still found use as a model is in byteranges[22], section 19.2. Byteranges are just like they sound, ranges of bytes serving as partial content for requested files. Byteranges are useful because a client can request parts of some content rather than the entire content. For example, a smart client could request byteranges from multiple servers asking for different ranges from each server to speed up a download from slower servers. However, if a client requested multiple byetranges from a single server the server will respond with the byteranges in a text stream with no separation between byteranges. Just as in chunking, byteranges serve to split the content into packets, which again is a step towards

what we want. This still suffers from the problems inherent with text streams as the byteranges are delivered as one message rather than parts of a message.

This area of HTTP has not been visited by many solutions but the tools are nearly available to make a soft transition. Byteranges and chunking are both supported by most of the more recent Web browsers such as Microsoft's Internet Explorer and Mozilla Firefox. As far as server software goes, most servers support byteranges and chunking as defined in the HTTP 1.1 specifications.

According to the official World Wide Web Consortium HTTP page, no further development is being made on HTTP. This means the features present in HTTP are considered locked for all intents and purposes. With no further work being done on HTTP, we feel that there is a missed avenue of very important features to improve the workings of the Internet.

Solutions in the Related Work section below that dealt with a networked environment heavily based on HTTP made attempts to circumvent problems inherent in HTTP. By circumventing the problems found in HTTP the solutions suffered by introducing extra overhead and complicated network paradigms. Rather than work towards yet another solution circumventing HTTP, we hope to go right to the source of the problem by modifying HTTP.

3 RELATED WORK

Many solutions have been proposed in conferences, papers, and technical talks that approach solving the problem with distributing load across multiple servers with identical content. Throughout each of the solutions is a manipulation of the environment governed by the basic client-server process demonstrated in Figure 1, repeated below as Figure 2, which introduces similar disadvantages across the spectrum of solutions.

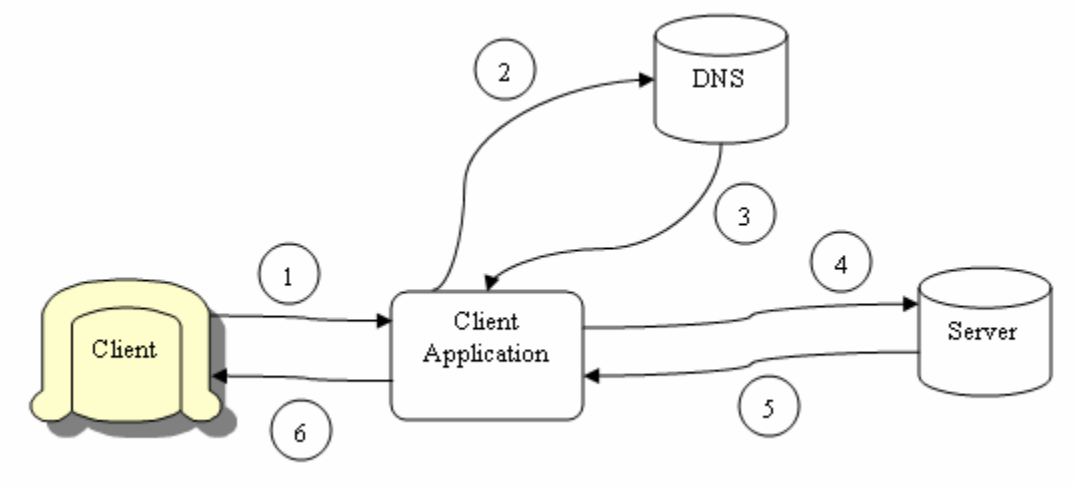

Figure 2: Basic client-server process (repeated)

The basic client-server process can be used to help categorize the solutions around where in the process the solutions manipulate the environment. The solutions are categorized into four main categories based on Figure 2:

- Solutions that make decisions at the client level (Stage 1)
- Solutions that make decisions at the DNS level (Stage 2)
- Solutions that make decisions when a client first connects to a server (Stage 4)
- Solutions that make decisions after a client has already received part of the file (Stage 5)

Here is a simplified table of related works placed in categories:

Table 1: Table of solutions

Client Level	DNS Level	First connection	In-stream
• Mirrored content • Smart clients	• Round robin • Variable TTLs	• URL rewriting • HTTP redirection • Dispatchers	• Redirectable sockets • Web clusters

3.1 Client level

In client level load balancing the entirety of deciding which server to connect to is placed at the client with no input from the server-side. There are two general methods provided: mirrored content and smart clients.

3.1.1 Mirrored content

Mirroring content is indicated as a poor solution in [6] because it is not "user-transparent" and does not provide server-side control. In [17] the authors point out that keeping track of clients is extremely difficult and expensive (p 1).

In mirrored solutions, clients choose the server from which they will receive content. There is no guarantee that clients will choose different servers, and so no guarantee load is spread across servers. Most clients lack the knowledge of current server load, network traffic, and locality that would make deciding upon a server an educated decision. Better yet, most clients do not care so much for a server, but are interested only in their connection with a server. Instead of using mirrors, a better solution would be "a distributed architecture that can route incoming requests transparently among several server nodes" ([6], p 28). We will visit upon this statement in much more detail later.

Mirroring is one of the simplest approaches to providing multiple servers to clients. Content is distributed to each mirror, a list of the mirrors and the location of the content on each mirror is compiled and distributed to each mirror, and a web page is displayed to the clients with links to the content on each of the mirrors. Maintenance consists of repeating these steps. Mirrored sites can be loosely or strongly linked. Mirrors can be added and removed very easily in loose-linked settings: a new mirror simply gets a list from a current mirror, hosts the content, and displays the list. Strongly linked settings would require only a few extra steps including having the mirrored servers communicate every so often to update their mirror links. There are a number of sites that use the mirrored-content approach to handle load balancing. Sourceforge.net [20] is an open source community hosting content in a variety of locations including educational institutions and open source companies. Another website that uses mirroring is GameSpot [9]. Patches and demos for popular computer games are served up throughout North America. Users connecting to GameSpot to download a game or patch are confronted with a list of servers throughout North America. However, unlike Sourceforge.net, users are also given current usage statistics for each server. Users are given a bit of extra information in order to make an educated selection, which is meant to benefit them and balance the load on the servers.

The standards for mirrored content, if such standards exist, are not used by all. There is no way for users to become accustomed to mirrored architectures. Most function similarly, but, as pointed out in the case above, some solutions are better than others in directing users to beneficial avenues of delivery.

The ease of adding and removing servers in a mirrored multi-server network is a desirable trait we would like to incorporate into our solution. However, we want to avoid giving full control to the client in making load balancing decisions as the servers can collect and use metrics much more effectively than a client.

3.1.2 Smart clients

In [6], the authors supply details about client software that makes decisions when connecting to servers. An older version of Netscape Communicator would connect to a server using a random number between 1 and the number of servers in the domain name, such as www2.cnn.com.

The number is generated right when the client first connects to a server. The authors indicate this solution would fit corporate intranets better than the Internet. The fact that this solution uses random numbers to determine which server to connect to does not make it an optimal solution. The nature of random numbers, which are actually pseudo random numbers in computers, is that there is no control whether a number comes up over and over again.

Client software solutions "lack general applicability because the client must be aware that the Web site is distributed" ([6], p 36). Clients must be aware of the layout of servers within a distributed system. This characteristic is not desirable because it exposes the network architecture to clients who can then take advantage of their gained knowledge by circumventing the random server method by choosing their own number. By

circumventing this method, the clients are also circumventing load balancing, which in actuality works against them.

This solution only works if all clients have equivalent software applications because *all* load balancing is handled by clients. Servers have no control over how load balancing is handled except to incorporate more servers into the system, and this is luckily an easy task. Adding a server means assigning the server the next available number, exposing the server to the Internet through a DNS, and updating the count of servers. Removing a server is easy as it is just reversing the steps.

In our solution we want to avoid leaving all load balancing decisions up to the client, because clients make guesses, not choices. Servers have much better awareness of situations within the multi-server system than any client.

3.2 DNS level

A simple DNS-based approach is described in [6]. The DNS, or Domain Name System, is the first step a client takes to connecting to a server and receiving their content. It is from the DNS that a client gets the information needed to connect to the server in the first place. Rather than require users to know the IP of a server, they instead can ask a DNS for the IP of a server with a given domain name. For instance, www.mywebsite.com is a domain name, which could be translated by a DNS to the IP 64.33.155.253 or any other IP for that matter. Whenever the user requests www.mywebsite.com they are requesting the IP from their DNS. For more information about how the DNS works see [1].

Some solutions make use of the DNS process by trading out the IP stored by the DNS when a server becomes overloaded with the IP of another server within the cluster. No two servers within the cluster are exposed to the Internet at one time. Instead, the servers must generate and use metrics to determine which server is the exposed server. "This process allows the cluster DNS to implement many policies to select the appropriate server and spread client requests" ([6], p 29).

DNS tables are updated based on a TTL value (Time-to-live), typical values of which are multiple days according to [1], so that clients are directed to several different servers over time through one domain name. When an amount of time passes equal to the TTL value, then the DNS updates the IP-Domain pairing to point to another server. The modification of TTL values as described by the authors of [6] is to either have constant TTL values, or create a situation in which TTL values are adaptive in the face of load across servers. "Constant TTL algorithms cannot adequately address client request skew" ([6], p 31). Client request skew refers to a burst of client requests in a short amount of time. An incorrectly assigned constant TTL could lead to overloaded servers. In their adaptive TTL algorithms, the authors use both server and client state information in order to pick the best server. Popular sites are given low TTL values so the DNS points to different servers more often. The DNS server records all the load metrics, so servers and clients are not requested to do any extra work.

Using small TTL values may at first seem like a good approach to add a particularly substantive amount of control. The authors of [1] indicate small TTL values will lead to

clients having to request IPs from the DNS more often, thereby increasing the network latency. Scalability can be negatively affected by small TTL values as well because more requests are transmitted through the network, implying that an increase in network performance would have to occur as TTL values decrease. Using TTL values of zero forces clients to request name translation every time they request pages, rather than relying on their own locally cached IP-Domain pairings. Such statements point out the added overhead introduced into the network, thereby reducing the abilities of such a network to handle increased loads. "The increase in network traffic due to additional UDP DNS packets is not insignificant" ([1], p 3).

To be effective, DNS-based solutions are restricted to using TTL values much larger than zero. The DNS was established with the idea that TTL values would be 1 day as minimums and 4 days for larger domains[1]. Such high TTL values provide an extremely low level of control in which changes cannot happen rapidly enough to handle bursts of clients or rapidly changing networks. "The [DNS-based] policies are ineffective because with address caching, each address mapping can cause a burst of future requests to the selected server and quickly obsolete the current load information" ([6], p 32). Unless the managers of the servers are also managers of the DNS, there is no guarantee on response times for updates to the DNS lookup tables. This is especially the case when one considers that there are many different DNS to which users connect to and receive information. Each DNS must propagate its changes to each other DNS in turn. This propagation certainly introduces a lag as the changes are passed along throughout the network of DNS. At any point in this chain of propagation a large number of clients may

be connecting to DNS servers with outdated IP-Domain pairings, thereby directing clients to potentially overloaded servers.

Another major drawback of this solution is that the DNS acts as a router to the various servers in this scheme, and therefore becomes a single-point-of-failure. However, DNS are often set up to deal with many different users, and only respond with small packets of information, so they do not fit the characteristics of a bottleneck.

In [17] the authors present a solution that involves clustering clients and directing clients in clusters to most beneficial servers through a DNS front end. Measurements are made through passive means by examining TCP information to determine the distance of a client to the servers. The DNS front end connects to a process they refer to as the Webmapper. When the clients ask the DNS for a IP-Domain pairing, they would be connecting to a Webmapper enabled DNS. Part of the Webmapper's duty is to cluster clients based on a pattern in their IP addresses so that future clients that fit the pattern can be handled without delay. Clients in a cluster, according to the measurements, are all located an equal distance from servers. Measurements are made and updated periodically after some fixed interval of time has passed. Clients are mapped to servers with a probability so that overloading the servers does not occur. The DNS is updated by assigning a dynamic TTL with each DNS response. Servers transmit load information to the Webmapper component through small 20 byte additions to served content. The Webmapper uses the distance measurements and server load measurements in order to modify the DNS entries for IP-Domain pairings.

Their experimentation shows that their algorithm for calculating the distance to the client is sometimes wrong. They indicate this is a problem due to the nature of DNS-based solutions. In many cases determining a best server through DNS methods is very difficult. Hence the reason their solution was wrong. One key point they bring up is that DNS-based solutions cannot deal well with bursts of clients. There just isn't enough time for the DNS to update the IP-Domain pairings of servers which are being saturated by client requests.

With the apparent drawbacks of DNS-based solutions we want to avoid using the DNS approach as a primary means of load balancing. DNS-based solutions may be a viable option in avoiding load under light situations but does not appear to be robust enough to handle high load situations.

3.3 First connection

There are a number of approaches that allow for redirection when the client first connects. These approaches are quite useful when a server is already under load but present no load balancing for clients already connected.

3.3.1 Dispatchers

The authors in [6] supply details for a dispatcher-based approach in which a client connects to a dispatcher, which then redirects the client to any of a number of back-end servers. This approach differs from that of DNS-based solutions because the dispatcher "has a single, virtual IP address" ([6], p 32). Back-end servers are accessed by the dispatcher through private IPs as they are on an internal network. The dispatcher is an

integral part of the Web-server collection rather than an external entity to which updates are made as that in the DNS-based approach. This places a much greater level of control in the hands of the Web-server collection administrators.

The central dispatcher uses one of three routing mechanisms to direct clients to servers. Dispatchers use “simple algorithms to select the Web server to manage incoming requests, as simple algorithms help minimize request processing” ([6], p 32). A Client connects to a dispatcher. The dispatcher then determines a suitable back-end server and directs the client requests to the server by one of three methods: packet rewriting, packet forwarding, or HTTP redirection [6]. In packet rewriting, the dispatcher examines all requests replacing its IP address with the pre-chosen back-end server's IP address, directing the packet to the server. The IP in the response packet is normally the IP of the server from which the content is being delivered. In rewriting, this IP must be replaced by the dispatcher's IP address, either by the back-end server or the dispatcher itself. In packet forwarding, packets are routed based on MAC addresses rather than IPs. No IP replacing is required in this solution. HTTP redirection, covered below, also does not require IP address modification.

In [14] the authors introduce a solution that incorporates the use of a cluster with what they refer to as a request distribution node, basically a dispatcher. Clients connect to the cluster through one IP, which initially points to the distribution node. The distribution node makes decisions about queueing requests for back-end servers. Requests are queued at the distribution node in an ordered list so that they can introduce Quality of

Service. Servers signal the distribution node when they are ready to handle a request. The back-end servers handle the request and respond to the client directly rather than going through the distribution node. Back-end servers are assumed to share a storage system to guarantee all content is equal across them.

They have introduced a method they call TCP splicing in which the distribution node makes the initial connection to a client in order to receive a request, then determines which back-end server to direct the client to, makes a connection to the server, then "splices" the two connections together to make a single communication channel between the back-end server and the client. The back-end server must replace IP addresses in all responses to fool the client into thinking they are communicating with the distribution node. This entails a lot of added process overhead for nearly everyone but the client involved in order to maintain the transparency desired by most load balancers, leading to a poorly scalable solution.

Information about the load on back-end servers is gathered by the distribution node every 10 msec. This information includes CPU usage, disk access time, and network bandwidth. With such a low cycle, the level of control is quite high, which is certainly a plus for load balancing. When clients connect to the distribution node, the distribution node examines the request and, using the most recent load information, chooses a back-end server that would best be able to handle the client's request. Having the back-end servers transfer information every 10 msec could get quite costly when there is a large number of back-end clients sharing one distribution node, so what the authors provide is

a method by which back-end server information is gathered in a staggered way so that only a few servers report the information at one time.

[15] presents another solution with a dispatcher inserted between clients and back-end servers. However in this solution they made attempts to be able to handle static, dynamic, and session-based connections. The dispatcher relays all client requests to servers, and server responses to clients, examining server status with heartbeat messages every so often. Should a server become overloaded or go down for whatever reason, the dispatcher has enough information to continue serving the response from another back-end server. To avoid delays, the solution uses pre-forked connections to multiple back-end servers. This entails a connection overhead within the system as a trade-off to supplying the client with low latency responses. Their inclusion of handling session-based connections is an interesting side note. The dispatcher keeps the state of how far along in a session a client is when dealing with the server. It records information every time a client changes the state of the session. An administrator dictates to the system what the different states are. Servers handle states of the session until they have completed it in full, at which point they respond to the dispatcher with an acknowledgement that they are ready to handle the next state. If a server becomes overloaded while handling one of the states, the dispatcher can migrate the connection and session over to another back-end server, handing the new server whatever session information it has stored from the last state.

The authors of [15] went on to implement the system using Java in [16]. Here a daemon runs on each back-end node to collect load information and report it to a dispatcher node. The dispatcher node also keeps track of which back-end nodes can handle particular content through a URL table. Administrators can access the dispatcher node to add/remove servers and view load statistics in the system. The authors have ensured cross-platform compatibility by writing in Java. So whether content is hosted on Linux machines or Windows machines the solution will be able to run without too many modifications. This would certainly assist in the adoption of the method by multiple content providers.

These solutions face the problem of single-points-of-failure and bottlenecking. All routing decisions are made by the dispatcher. Should the dispatcher go out of action for any reason, whether that is maintenance or catastrophe, the back-end servers will never be reached. Clients will be left without content until the dispatcher is put back into its place. All clients connect to the dispatcher to gain access to content from any of the back-end servers. If a large amount of clients connect to the dispatcher at the same time, then this solution may introduce an increased amount of response times as the dispatcher deals with the queue of backlogged client requests. In packet rewriting dispatchers must examine all IP headers and replace IP addresses of servers and clients. As the number of IP messages passed through the dispatchers increases the response times also increase. Such characteristics are not desirable in a system made to adjust load rather than cause load.

[6] offers no analysis of the fact that bottlenecking can occur. The authors of [14] indicate a possible solution to bottlenecks is to create a two-tier system in which some dispatchers handle routing and others make decisions about which servers are to be used. However, this will still suffer from bottlenecking as some point as it is an unfortunate side effect of the design. In [15] the authors suggest using multiple dispatchers to service requests, but this would lead to less transparency.

One other problem with this set up is that the dispatcher and back-end servers must be on the same network if the algorithm involves packet rewriting or packet forwarding, implying they must all be located in the same geographic area [6]. Any attempts to incorporate servers from outside the internal network of the dispatcher would increase the amount of security threats inherent in the system. What was an internal network, more secure than networks linked through significantly different geographic locations, is exposed across the vastness of the Internet.

The privacy of the backend servers is a nice feature, but the bottlenecking attributed with the dispatcher nodes is a drawback. We want to avoid bottlenecking if at all possible in our solution, so we cannot have just one entry into the multi-server system. This means that we will not be able to hide backend servers from clients.

3.3.2 URL rewriting

In URL rewriting, HTTP reference links are updated every so often to point to different servers. It could be the case that URLs are rewritten for every client who visits the page. Such solutions require dynamically changing pages rather than static ones introducing a

small overhead on the server machine. Clients may not even visit the links which have been rewritten.

One company that offers a lot of research and technology behind URL rewriting is Akamai. They assist Internet companies delivering content to users through supplying URLs that change depending on server load. URLs are assigned to all objects which the Akamai servers handle, which represent each object uniquely. When a client visits a page, URLs pointing to various servers are used for content available to the client. If a client clicks on a link, then they are served by a predetermined server. According to Akamai's website, “Akamai routinely handles up to 15% of total Internet traffic-more than one billion hits every day” ([2]). With so much of the traffic handled by this solution alone, it would seem this solution is a successful solution. Akamai delivers everything from advertisements to media content using this solution, including some very high demand content for clients such as CNN.com. The metrics collected by Akamai, however, are proprietary so we can supply no discussion about the actual logic behind their URL rewriting.

If you are to visit the CNN Website [5], the only way you know content is being served by Akamai is by watching the status bar in the bottom of your web browser. This solution is quite transparent to clients, requiring no decisions to be made by clients like that in mirrored content solutions. You may notice the page loads a little slower than other websites, but it has a consistent load time even when top news stories hit the front page.

In [23] the authors present a solution that uses URL rewriting to direct specific users to different servers. The client IP address, the client Web browser information, and cookies can all be used in determining how to rewrite the URLs. This method is meant to be used in a Quality of Service arrangement in which some clients may be of higher status than others, but it has the potential for load balancing as well. By handling only clients of higher status, and redirecting clients of lower status to other servers, the solution inadvertently sheds load from a server.

One problem with URL rewriting as pointed out in [23] is the assumptions made that URLs used in rerouting clients are assumed to have not changed in the DNS. Unless a tight coupling between the URL rewriting server and the DNS is established, then rewriting URLs could direct users to undesirable content or error pages. Whatever tables the rewriter is using need to be checked and updated on a regular basis to ensure nothing of the sort happens.

Another solution that uses URL rewriting is presented in [24]. In this solution, a server takes metric information periodically to determine its load. If it sees that it is about to be overloaded it enters into a pre-overload mode in which it begins rewriting URLs of images and media content in all its pages. Up until this time, the server acts as a normal server; it responds to clients with their requested content.

One problem we see with URL rewriting is possible problems with users bookmarking websites. URLs are generated on-the-fly and are not guaranteed to point to the same location at future dates. We instead use static server locations in our solution, with every server in the multi-server system storing content in the same place.

URL rewriting also does nothing to help transferring large files because the clients are still left to download the files without possibility of redirection while they do so. Such a solution does not present a flexible process for large file load balancing.

3.3.3 HTTP redirection

Redirection is supported as of HTTP 1.1 [22]. When a client requests content from a server, the server can redirect the client to another server by responding with a redirection message rather than the content. The protocol dictates a set of codes, which are found in the headers of HTTP responses. Clients and servers using programs that conform to HTTP 1.1 will understand the codes for redirection, and therefore not require any more software. According to the authors of [6], HTTP redirection "duplicates the number of necessary TCP connections" (p 35), and "HTTP redirection's main drawback is increased response time, since each redirected request requires a new client-server connection" (p 36). The authors of [12] point out that "HTTP redirection is scalable, but not application independent" (p 2), which we will take to mean the required conformance to HTTP 1.1.

In [21] the authors use HTTP redirection to ensure a particular Quality of Service. Administrators define some quality of service values, such as no more than 20% of network traffic should be devoted to ftp, on a quality of service server. Servers

periodically update a table of metric information, which they in turn report to the quality of service server. The table is updated after a server finishes any response. When a client connects to a server, the server checks with the quality of service server to make sure the quality of service values are not violated. The server asks the quality of service server whether it should handle the request, deny the request, or redirect it using HTTP redirection, to which the quality of service server responds with its selection.

HTTP redirection is used by some dispatchers. In dispatcher settings content servers must periodically report load information to the dispatcher. Based on the load information, the dispatcher redirects clients through the HTTP 1.1 method of sending back a redirection response with an address to the new server. The information may be up to date, but it may also be the case that servers haven't reported their information for some time. If the latter case happens to be true, then the dispatcher is redirecting clients on old information that could potentially lead to overloaded servers.

When a server is in overloaded mode in [24], it uses HTTP redirection to take an ignorant stance with all new clients, while continuing to handle the requests of older clients. Servers in this solution continue to use HTTP redirection until they return to a normal mode of operations, progressing through the pre-overloaded mode again.

HTTP redirection does not require any of the servers to have direct links to each other, although it would be necessary to have such links in order to properly gather load information. Due to this fact, the server distribution is not grounded in locality, IP

domains, or underlying software. All decisions are made by individual servers, rather than some dispatcher, thereby avoiding a single-point-of-failure.

HTTP redirection is visible to clients in the form of longer response times. This negative characteristic stems from the fact that a client sends a request to an initial server, which then sends a redirection response to the client. The client must then connect to the new server and issue the request again. Without tightly coupled servers, the client could potentially be bounced back and forth indefinitely.

In our solution, we want to be able to decide on redirecting traffic after a client has already connected to a server. HTTP 1.1's redirection capabilities do not include in-stream redirection, so it does not pose as a viable solution.

3.4 In-stream

There are some approaches to load balancing that offer a method while the client is getting the content. These in-stream approaches allow for decisions to be made no matter when the client connected. Approaches that fall into this category afford administrators much more flexibility when deciding when to balance load.

3.4.1 Web clusters

In [7], the authors implemented a load balancer with an entry point into an internal network that provided external clients access to a cluster of servers. There are a number of nodes within the cluster: nodes with content and a node to gather and manage distribution within the cluster. Load in the network is reduced to routing packets to servers based on dynamic traffic information collected by the management nodes.

Content nodes gather load information and report this information to a master node. The management nodes then gather the information from master nodes in order to determine to which set of servers to direct packets. The solution involves the collection of five performance counters: available memory, processor utilization, total network traffic, number of connections, and number of client requests per second ([7], p 241). The number of client requests is determined by running an analyzer on client machines, which means modifications made to every client.

Cluster membership is handled by humans interacting with consoles on the management nodes. As content nodes are added, administrators assign them to master nodes. The new content servers report to master nodes, who in turn report new membership to management nodes. Therefore scalability is automatically built into the system. The paper does not go over many details about what sort of messages are transmitted over the network for membership, nor how often these messages are transmitted, so no analysis of the network overhead is indicated.

Just as in the dispatcher-based solutions, we want to avoid the problems with bottlenecking experienced in clusters. This means avoiding the use of management nodes to control load.

3.4.2 Redirectable sockets

In [12], the authors propose a change to sockets to implement a redirection method. The authors introduce a protocol at the session layer rather than the network, application, or

transport layers. They have gone with this approach because they feel the network layer is too low of a layer, the application layer changes are not transparent, and the transport layer is still not at a high enough level. They are chiefly concerned with the amount of decisions already made at the particular levels. Lower levels deal with a lot of extra information that has nothing to do with redirection. Also of concern is the visibility by clients. Clients should not be exposed to redirection methods, and must avoid being dealt with in the application layer.

To facilitate their redirectable sockets the authors created a protocol that dictates where redirection information is located and what the information looks like. What they do is wrap all transport layer data with a header that includes redirection information recognized by other modified sockets. To redirect a connection, a server simply adds the header to the response and the client connects to a new server.

The overhead for the solution is indicated as roughly 1.5% over normal sockets, which means an increase in 1.5% of transmission time. 1.5% increase in time equates to less than a second for a 60 second download, and less than a minute for a 60 minute download. Since most downloads fall between these two times, users will most likely not notice the overhead.

As long as clients and servers include these modified sockets, that's all that is needed. The redirection logic is actually handled by some other program. This solution presents the means by which such a redirection program could implement redirection across

multiple computers but does not provide a protocol. A protocol establishes a set of rules for which all future implementations of sockets are to follow. Those who adopt the protocol can implement it in any way they want so long as the end products generated by the implementation follow the protocol.

A protocol makes it easier for others to adopt the solution for their own use. Making changes in the operating system is not necessarily an easy thing to do. Initially, the sockets could be adopted by individual users, but if the protocol and sockets were to catch on the change needs to be made in all future operating systems.

In our solution we want to focus on designing a protocol that can be used across multiple computer architectures. The protocol will also avoid making changes at the operating system level.

4 HTTP 1.2 SOLUTION

Developing HTTP 1.2 involves looking at related works, identifying disadvantages, describing our intended goals, and building out a protocol to support those goals. We draw upon existing elements of HTTP 1.1 to facilitate the necessary components of HTTP 1.2 in-stream redirection.

4.1 Approaching a solution

To begin formulating a solution, we begin by examining the disadvantages experienced with related works as a means of avoiding potential pitfalls and undesirable traits. Then we create a list of features explaining how the solution should avoid the drawbacks. Next, the features are checked for conflicts where including one feature automatically leads to the failure of another feature. In cases of conflict, one feature is chosen over the other in order of importance to the overall load balancing goals. Finally, the list of features is condensed into a description of a desirable solution.

Here is a list of drawbacks experienced by the various related works in no particular order:

- There is a considerable lack of standards used across multiple content providers.
- Some of the solutions are proprietary with no exposure of internal workings.
- Single-point-of-failure or bottlenecking are common problems faced by many of the works.
- The network layout is exposed to clients.
- Clients may not be able to connect to the multi-server system the same way every time.

- Changes involved in the solution must happen at the operating system level rather than just applications.
- Some solutions do not allow in-stream redirection.

The next step is to take the list of drawbacks and rewrite them as features we want in our system to avoid the drawbacks. We want a solution that:

- Includes a standard that can be used across multiple content providers.
- Avoids proprietary internal workings.
- Avoids single-point-of-failure and bottlenecking.
- Does not expose the network layout to clients.
- Allows clients to connect to the multi-server system the same way each time.
- Does not require operating system changes.
- Allows for in-stream redirection.

Most of the features do not conflict with each other. However, we cannot safely avoid bottlenecking without exposing the network layout to clients. We ultimately choose avoiding bottlenecking as a much more desirable feature than exposing the network layout to clients because if all nodes have load balancing capabilities it will not matter how the client gains access to the multi-server system.

We can condense the features list further into a single sentence describing our intended solution. *We want a solution that provides a standard to use in-stream redirection at the*

application layer without hiding internal workings while allowing clients to connect to the multi-server system consistently without bottlenecking or single-points-of-failure.

Here a standard is a set of rules or regulations used to unify a potentially heterogeneous environment of clients and servers. If all providers follow the standard, then a group of solutions that follow the standard will be compatible with other solutions in that group. This allows us to avoid dictating what server applications and client applications must be used to provide load balancing. Furthermore, the Internet and applications follow standards in many situations in the form of protocols. Therefore, our solution will be presented as a protocol.

Existing protocols of the Internet can be examined and drawn upon to assess the abilities we wish to include in our protocol implementation. TCP/IP[8] works at such a low level that we want to avoid dealing with it for sake of keeping our method simple. HTTP proves to be a protocol in use that is more at the level we intend to concentrate on. However HTTP lacks the functionality for in-stream redirection. More details on HTTP are provided in the next section.

An augmented HTTP that allows for in-stream redirection addresses the application layer issue and also exposes internal workings because all decisions are exposed in text through the medium that is HTTP. Allowing clients to connect to the system consistently every time while avoiding bottlenecks and single-points-of-failure is still an important feature we do not want to overlook. We can achieve consistent connections by allowing

every server node in the system the ability to make load balancing decisions. This is more of a network layout issue that will not be discussed ad nauseam in this text. However, if each node has the ability to make decisions, then the power is left in the hands of those in charge of the nodes.

4.2 HTTP 1.2 Overview

Our desired requirement statement for HTTP 1.2 is repeated to examine on a high level before going into the details of HTTP 1.2. *We want a solution that provides a standard to use in-stream redirection at the application layer without hiding internal workings while allowing clients to connect to the multi-server system consistently without bottlenecking or single-points-of-failure.*

HTTP 1.2 is not an entirely new protocol, but builds off of pre-existing elements of HTTP in such a way that adoption of HTTP 1.2 for servers and clients would require a short development cycle. HTTP already operates at the application layer, so by extending HTTP we already support an application layer level of redirection.

Clients should be able to connect to any server and be redirected by that server to another server. This only requires the server to be aware of mirrored servers that make up part of the same "cluster." A cluster consists of servers with identical content from which clients can request. By allowing clients to connect to any server we avoid the problems of bottlenecking and single-points-of-failure.

In-stream redirection will consist of the following set of steps: 1) a client connects to a server and requests content, 2) the server responds to the client by beginning to serve the content, 3) at some point during the transfer, the server becomes overloaded and informs the client to seek the rest of the content from another server, and 4) the client connects to the new server and finishes receiving the content going through steps 2 through 4 should the new server become overloaded.

The benefits gained by this approach of in-stream redirection are: a) servers remain in control of load balancing, b) the granularity of control is determined by the server, and c) the cluster of servers does not have a predetermined size. Even more for the last benefit, every server in a cluster does not need to know about every other server. The more servers known by a server means a better load balancing because there is more potential redirection targets, but it is not required.

Also, leaving the in-stream redirection at the HTTP level means if a client can request content from a server, the server is potentially part of a cluster. Servers no longer need to be housed in the same server farm, or exist on the same Local Area Network. The servers could potentially belong to different companies or organizations and managed completely separately.

One last benefit gained is that HTTP 1.2 is not operating system or architecture dependent. Servers running Linux with Apache as the HTTP server can interoperate with servers running Microsoft Windows with Microsoft IIS without problems.

4.3 HTTP 1.2 Implementation

HTTP presents a unifying structure to the Internet because applications that deal with serving or downloading content use it as the standard means of communication. A lot of work has already been done in adding some of the ideas behind multi-server systems into HTTP, such as byteranges and chunks, but it has not yet achieved the features we listed above as desirable features for a multi-server system. We will go over the details that make up HTTP followed by details for HTTP 1.2.

4.3.1 HTTP 1.1 details

HTTP 1.1 is a text based protocol, meaning all communications are handled as a text stream from server to client and vice versa. The messages transmitted in HTTP have two parts: information and data. The information tells clients and servers what they are getting, or if any errors occurred, such as bad requests, and the data is the actual content associated with the information.

Byteranges and chunks already show that the data can be split up into little text packets and communicated as such. We intend to take this one step further and transmit not only these data packets but also some information packets with redirection information. A client receiving data packets may receive an information packet with redirection information indicating to the client to get the rest of the data packets from another server. This accomplishes our goal of adding in-stream redirection to HTTP.

4.3.2 HTTP 1.2 details

HTTP 1.2 is proposed as HTTP version 1.2 rather than its own protocol because it is meant to augment HTTP version 1.1 and so includes all the functionality of HTTP version 1.1 while introducing the desired new elements to support in-stream redirection.

Introducing HTTP 1.2 as the new HTTP encourages future clients and servers to support HTTP 1.2 rather than include it as a possibility. The transition to HTTP 1.2 can be incremental, as servers can recognize when clients are connecting with HTTP 1.1 or HTTP 1.2 and process the response as necessary. Furthermore, this lends well to a formal adoption process similar to when HTTP 1.1 replaced HTTP 1.0, thereby allowing for a well described transition.

Also, introducing the ideas behind HTTP 1.2 as a part of the de facto standard encourages future development leading to improved usage and support. HTTP 1.2 as proposed in this document not complete, although some future ideas are presented in the Summaries section.

One future concept that should be called out is the introduction of HTTP-based peer-to-peer networks. Packets, in-stream control messages, and subscription components of HTTP 1.2 can be combined to natively support distributed content.

With a framework for in-stream control messages, people can continue to propose new control messages introducing entirely new concepts and abilities.

4.3.3 Packets

Aside from generating the headers of the response to the client, we are also concerned with generating efficient packets. We want to avoid including so much information at the beginning of each packet that it incurs a large overhead. Instead we must keep the information brief and to the point. A session ID is provided that can be used at the beginning of each packet to identify to which message the packet belongs. The length of each packet and the total number of packets the client is to receive are also defined at the beginning of the entire message, so each packet does not need to include the length unless the packet is the last packet. If the packet is a last packet, then it can have any number of bytes less than the packet-length defined in the initial headers, which must be defined somewhere in the packet headers. Table 2: Packet header fields, shown in Table 2, provides a summary of the headers associated with each packet.

Table 2: Packet header fields

Header field	**Description**
PACKET	Message type.
Session-ID	The session ID that appears in the initial response headers. All packets related to a response will have identical Session-ID values.
Packet-number	The number of the packet if aligned in offset order with the other packets.
Packet-length	The length of the packet if it is the last packet. All other packets will have a length equal to that specified in the initial headers.

Control messages as defined below include a keyword as the first entry of the message. The keyword PACKET is used in the front of every packet. This makes it easier for the client to identify the packets from the control messages.

The PACKET format is demonstrated in the following example.

```
PACKET CRLF
Session: <Session-ID> CRLF
Packet-number: <Packet-number> CRLF
[Packet-length: <Packet-length> CRLF]
CRLF
```

4.3.4 In-stream control messages

In-stream control messages are the way in which a server can issue commands to clients between packets. We define two control messages below. We have provided a message for redirecting a client to another server and a message for suspending a client for a specified amount of time. Both messages are meant to assist in load balancing on the server by providing the means of informing clients to stop using the current server. We believe the SUSPEND message is a novel approach to load balancing that involves having clients wait instead of being redirected to another server. In our Analysis section we only explore the REDIRECT message, and leave the SUSPEND message for future work, although it is described below.

4.3.4.1 REDIRECT message

If a server becomes overloaded or predicts it will become overloaded, then it needs to shed load. In HTTP 1.2, the server can do this by issuing clients REDIRECT control messages. The important factors in deciding what makes up a REDIRECT control message are the functions that it needs to serve:

- The control message must provide other servers from which the client can RESUME their download, see section 4.3.4.3.

- The control message must allow servers to pass messages to each other through the client. Such information may be state information as in the case of an online store, or even more sensitive information. This information is defined as the *payload* of a control message.
- The control message must provide a time limit after which the client can return to the current server should the client not be able to connect to other providers.

A REDIRECT message may have a payload attached to it if indicated in the headers. A payload is information that the server requires the client to pass on to the next server. This could include state information to help the next server decide how to handle the newly redirected client. For example, if a client is downloading a file that is subject to change frequently, the payload could consist of file information such as a MD5 hash, last modified date, path, etc., so the new server can easily identify if the file it has is really the one sought after by the client.

The details regarding the payload are left to be determined by individual implementers and therefore will not appear in this work. One suggestion for any parties considering payloads is to have the payloads encrypted with a method known to both servers. The payloads could contain vital information that should not be allowed to be intercepted by unwanted third parties.

Table 3: REDIRECT header fields

Header field	Description
REDIRECT	Message type.
Server-list	A colon delimited list of servers from which the client can issue RESUME methods to finish the download.
Payload-length	The length of the payload. This value will be 0 (zero) if there is no payload.
Time-limit	A length of time in seconds that the client must wait before returning to the current server should it not be able to connect to other servers. This is like the case of the SUSPEND control message as defined in the next section.
Payload	The payload data as defined above.

To better illustrate the REDIRECT message format we've provided an example below.

```
REDIRECT CRLF

Servers: <server-list> CRLF

Packet-number: <packet-number> CRLF

Payload-length: <payload-length> CRLF

Time-limit: <time-limit> CRLF

[<Payload> CRLF]

CRLF
```

4.3.4.2 SUSPEND message

When a server becomes overloaded or predicts it will become overloaded, the server may choose to have clients suspend downloads and return at a future time to finish. This is useful in scenarios where there may not be other servers with identical content or if the server needs to have clients reconnect directly to the same server. Some scenarios are as follows:

- There is too much state information to encapsulate as a payload without incurring large network loads.
- There are no other available servers to handle the client's request.
- Clients have been guaranteed to be served by one server only.
- The server has determined all other available servers are already overloaded. This may require tight coupling between servers or period testing from server-to-server.

A SUSPEND message may have a payload attached to it just as in the REDIRECT message shown above. However, in this case because the client is going to reconnect to the same server, the payload would most likely be abbreviated with most of the information still stored on the server. One example would be for the server to pass on a session ID to the client through a payload so the server knows where to look up stored data when the client returns.

Table 4: SUSPEND header fields

Header field	**Description**
SUSPEND	Message type.
Time-limit	A length of time in seconds that the client must wait before returning to the current server.
Payload-length	The length of the payload. This value will be 0 (zero) if there is no payload.
Payload	The payload data as defined above.

To better illustrate the SUSPEND message format we've provided an example below.

```
SUSPEND CRLF
Time-limit: <time-limit> CRLF
Payload-length: <payload-length> CRLF
```

```
[<Payload> CRLF]

CRLF
```

4.3.4.3 RESUME method

A client who has been redirected by a server will contact a new server using the RESUME method rather than the standard GET method.

A client will also use the RESUME command if previously issued a SUSPEND message as noted later. The procedure is handled as though the client was redirected, except the redirecting server and the redirection server are the same server. This is useful especially in quality of service cases where some clients are allowed to continue downloading while others must wait.

Lastly, a client can use the RESUME method if they have already received part of some content, but a previous connection was terminated voluntarily or involuntarily. For example, a modem user attempting to download a driver file receives half of the driver file and is disconnected. Rather than start the entire file download over, the client can issue a RESUME method to the server and receive the rest of the content. This process is useful for both fault tolerance and large files.

The RESUME method differs mainly from the GET method by allowing content following the headers. Normally a GET method includes only headers. However, as is provided in details under the REDIRECT message, an in-stream redirection may include a payload, which needs to be handed off to the next server. Using the standard GET

method would not allow the payload to be handed off by the client because the new server is not expecting anything passed the GET method headers.

Creating the RESUME method rather than modifying the GET method is ideal because the GET method will be used many more times than the RESUME method. Since the RESUME method will require extra headers this implies extra overhead over the traditional GET method. We avoid the extra overhead by defining the RESUME method based on GET but with some extra required headers.

4.3.4.3.1 GET headers

Rather than define a whole new set of headers, all the headers normally found in a GET method call are available in the RESUME method call. For further information regarding the required and optional headers see [22].

One header field we are chiefly concerned with is the **Host** header field. The Host field "specifies the Internet host and port number of the resource being requested, as obtained from the original URI given by the user or referring resource" ([22], section 14.23). This fits our intention perfectly because we want the new server to know whence the client was redirected.

4.3.4.3.2 New header fields

In order to facilitate the RESUME method we need to add some more header fields to the list offered by the GET method. There are a few items that need to be addressed by appropriate header fields:

- The new server must be made aware that a payload exists or does not exist.
- If a payload exists, then the new server needs to know the length of the payload.
- The new server needs to know the list of servers for which the client has already attempted RESUME methods. This will avoid the server sending a client to a server the client has already attempted with failure. Such a scenario could lead to an endless loop of redirection should two servers be too overloaded.

Table 5: RESUME method fields

Field	Description
RESUME	Method type.
Payload-length	The length of the payload attached to the RESUME file in hexadecimal notation. The value of this field is 0 (zero) if there is no payload.
Servers	A colon-delimited list of servers for which the client has already issued RESUME methods. The colon character is not allowed in URLs, so we can use it as the delimiter in our list.

4.3.5 HTTP 1.2 Header Fields

New headers are needed in the response to the client in order to inform the client of particular redirection parameters. The headers are an addition to those included in the HTTP 1.1 responses but will only appear when HTTP 1.2 is used by the client and server.

Table 6: Header fields and acceptable values

Header field	Description
Packet-length	An integer value greater than 0 to indicate the byte length of the packet.
Number-of-packets	An integer value greater than 0 indicating the total number of packets in the transmission.
Session	10-digit hexadecimal value shared by all packets in the transmission.

4.3.5.1 Previous header fields

All header fields included in HTTP version 1.1 are included in version 1.2 for backwards compatibility. For more information regarding previous header fields refer to [22].

4.3.5.2 New transfer encoding header value

Splitting the content into packets serves as an encoding of the HTTP message in order to transfer it "safely" as defined in [22], section 3.6. This means that we must specify a transfer encoding in the transfer-encoding header field. The following line will appear as a header field in the response message.

```
transfer-encoding = "packets"
```

Just as in chunking, the packets encoding must be the last encoding applied to the message. Chunking and creating packets are mutually exclusive and cannot both be applied to a message.

4.3.5.3 Packet-length

The server splits content up into packets of a fixed and constant length. The length is indicated in the "packet-length" header by an integer value greater than zero. The packet length cannot change over the course of serving an entire piece of content.

4.3.5.4 Number-of-packets

A server can determine the number of total packets that will be served to a client using the packet-length discussed above. This will allow clients to keep track of missing packets based on packet number rather than total number of received bytes.

4.3.5.5 Session

If the content is split into packets, then a unique session ID is created and attached both to the initial response to the client as well as each packet. The session ID is a 10-digit hexadecimal value. A session ID is required to mark all packets belonging to one transmission. This is important if a connection between a client and server is used to transmit multiple files concurrently to avoid mixing packets.

4.4 Example

An example will help illustrate what content might look like with control messages.

1. The client issues a request.

GET /file.mp4 HTTP/1.2

2. The server finds the file and responds with the packetized file.

```
200 HTTP Ok
[HTTP headers]
Transfer-encoding: packets
Number-of-packets: 202
Packet-length: 2000
Session: 149AC2BB80

PACKET
Session: 149AC2BB80
Packet-number: 1

[2000 bytes of packet data]

PACKET
Session: 149AC2BB80
Packet-number: 2
```

```
[2000 bytes of packet data]
```

3. The server discovers it is becoming overloaded and issues a REDIRECT control message.

```
REDIRECT
Session: 149AC2BB80
Servers: server1:server2:server3
Packet-number: 3
Time-limit: 10000
Payload-length: 0
```

5 HTTP 1.2 CLIENT/SERVER APPLICATION IMPLEMENTATION

The main contributions provided here are the protocol modifications that make in-stream redirection possible. The implementation is focused on a proof of concept for dealing with in-stream redirection control messages without developing the SUSPEND message. We are presenting the policy by which programmers of Web-based software will adhere and implement in their own ways.

First we will begin by approaching the implementation by providing pseudo-code, see Pseudo-code 1 and Pseudo-code 2, which details the steps involved in the programming of HTTP 1.2. This will serve as a guide to those who will need to modify or create software projects to use HTTP 1.2. There are two pieces to the pseudo-code because both clients and servers must be modified to use HTTP 1.2.

Pseudo-code 2: Client

```
Submit a GET request to a server
IF server response is HTTP 1.2 AND is marked for packets THEN
    WHILE there is data ready to read from the socket
        READ from the socket
        IF data IS packet THEN append to file
        IF data IS control message THEN
            IF control message IS redirect THEN
                Connect to another server using the RESUME method
            END IF
        END IF
    END WHILE
END IF
```

Pseudo-code 3: Server

```
IF receive GET request from client AND request is http 1.2 THEN
    IF content length < threshold THEN serve as in 1.1
    ELSE
        WHILE there is still data to send
            IF load state IS overloaded THEN send redirect message
            ELSE send packet
            END IF
        END WHILE
    END IF
END IF
```

Line 2 of Pseudo-code 3 includes the keyword *threshold*, which requires a bit of explanation. The threshold is a number indicating the amount of bytes that can safely be served by a server by means of a standard HTTP 1.1 transmission rather than breaking the content into packets. When servers have virtually zero load, it is wasted effort to break up content. Further discussion is provided about developing an accurate and effective threshold, see section 8.

We chose to make a proof of concept for both a server application and a client application. The intension is not to make fully distributable applications with highly polished algorithms and full documentation. Instead we want to showcase that HTTP 1.2 can be effectively used to create a load balancing method.

5.1 Apache

Rather than create our own server application we chose to use Apache's HTTP Server[4]. Apache's HTTP Server is available as open source code and easily modified. The architecture of the code helps us add in our HTTP 1.2 without having to overhaul the entire code.

As of version 2.0, Apache introduced what they call *Bucket brigades* and the *Filter chain*. Basically bucket brigades are data containers used to break content up into more manageable segments. This fits perfectly with our desire to send content as packets. Furthermore, the filter chain allows filters to modify the content as it makes its way to the client.

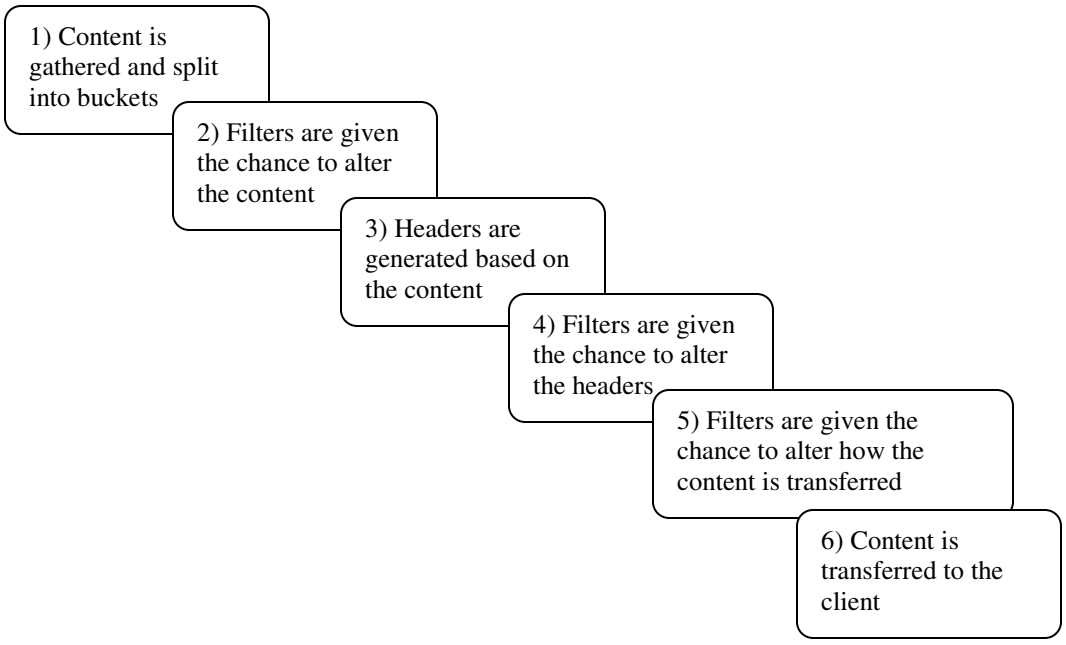

Figure 3: Flow through Apache

Filters can be written separately from the main source code and enabled through some configuration files. This makes our chore of adding support for HTTP 1.2 quite easy with Apache's API. We created a filter at Step 4 to generate our new HTTP 1.2 headers, see Appendix A. We also altered how the content is transferred to the client so we created a filter at Step 5, see Appendix B.

Our header filter generates the required new headers to facilitate the packet transfer encoding including the number of packets and the packet size. The packet size is saved as an *environment variable*, which is Apache's form for global variables. We have the luxury of being able to change environment variables outside the code through configuration files read in when the application is started. This means changing the packet size is quite easy.

Our transfer filter creates packet headers and sends packets to the client. In between sending packets, the filter also does some rudimentary metrics to determine if the client needs to be redirected. The metric is based on how many clients are concurrently receiving content and how big is the content. The more clients and bigger the content, the more likely the filter is to redirect clients. The actual numbers that dictates how many clients and size of content that constitute enough load to redirect is dependent on the computer system running the server application. Developing optimal methods for developing suitable load characteristics is outside the scope of this work.

5.2 HTTPerf

For our proof of concept we also want to have a client that can do some load measuring, metric collection, and reporting. HTTPerf[18] is an application that is meant to generate high traffic loads and measure the performance of the HTTP server. The code is originally designed to work with HTTP 1.1, so there were a number of modifications needed to bring it in line with HTTP 1.2 proposed in this paper. These modifications consisted of coding the pseudo-code, provided in Pseudo-code 2: Client above. HTTPerf

was designed to be somewhat modular so we were able to modify the request generator code for the standard HTTP request process, see Appendix C.

6 ANALYSIS

We needed to establish in which cases HTTP 1.2 makes sense to use and what sort of performance gain or loss was encountered through the use of HTTP 1.2. We ran a series of tests using both unmodified and modified versions of both Apache and HTTPerf. The results are then compared against each other with conclusions drawn about the usefulness of HTTP 1.2 in particular situations. We go into the analysis knowing HTTP 1.2 will not perform well in some situations as there is more for both the server and client to do. However, it is our goal to determine the areas in which HTTP 1.2 will prove useful.

6.1 Overhead

The first experiment measures the amount of overhead introduced both in size of the content and the amount of time for transfer of content that is served in packets but never redirected. In many cases a server may break content into packets without ever having to redirect clients, so this is an important measure of the affect when the server is in normal operations. Overhead of all underlying transport methods, including TCP/IP encapsulation, socket layer overhead, OS overhead, and HTTP 1.1 standard overhead, are included in these results but not separated out for individual analysis.

6.1.1 Configuration

The computers in use for this experiment are Compaq 2.4 GHz machines running Linux RedHat 9, kernel version 2.4.20 attached to a 100 Mbps LAN. One computer is the server, and the other computer is the client.

6.1.2 Results

The test was run against content that was not broken up into packets, a packet size of 1150 bytes, 2650 bytes, 4150 bytes, and 5650 bytes. These values were considered due

to the Maximum Transmission Unit(MTU) as defined in TCP as 1500 bytes. The idea being, content plus header information will be spread across MTUs as little as possible. Table 7 shows how long it took to download files of particular sizes across the LAN as well as % of the download that was overhead.

Table 7: Overhead test results

Packet size	8 KB file	600 KB file	6 MB file	% Overhead
Content not in packets	.001 s	.052 s	.504 s	0
1150 bytes	.002 s	.057 s	.793 s	8.5
2650 bytes	.002 s	.054 s	.525 s	3.9
4150 bytes	.002 s	.053 s	.517 s	2.5
5650 bytes	.002 s	.053 s	.514 s	1.9

What we see from the results is that the larger the packet size, the less time it takes to download the entire content and the less overhead incurred for packetizing. We expected the packet size of 1150 bytes, which should not span across more than one MTU, would be the fastest, but Apache may include some sort of buffering code later on down the line of command. However, the smaller the packet size the more times the system has to decide if redirection is required. This is a classic trade-off between granularity of control and overhead. It is our intention to leave the decision ultimately up to the server administrators as to what packet size to use. We will see in later results that the underlying buffering done by Apache negatively impacts our implementation of redirection.

6.2 Requests per second

HTTPerf reports the maximum requests per second as a metric. We can use this as a measurement of how capable a server is of handling client load. A high request per second ratio is ideal, because that means the server is able to fulfill client requests without dropping them.

6.2.1 Configuration

This test was run using four HP Kayak Pentium II 450 MHz computers with 192 MB RAM running Fedora Core 2. The computers were connected through a 10/100 Mbps switch with static IPs. Two of the computers were used as HTTP 1.2 servers, one was used as a HTTP 1.1 server for control purposes, and one was used as the client.

6.2.2 Results

The test was run for both HTTP 1.1 and HTTP 1.2 with different file sizes of 2 KB, 20 KB, 200 KB, and 2 MB and different amounts of client connections including 2, 20, 200, and 2000.

Clients were redirected at receiving 50% of the content regardless of actual server load. This is a very rudimentary load balancing method meant to illustrate a fixed test to compare repeatable results.

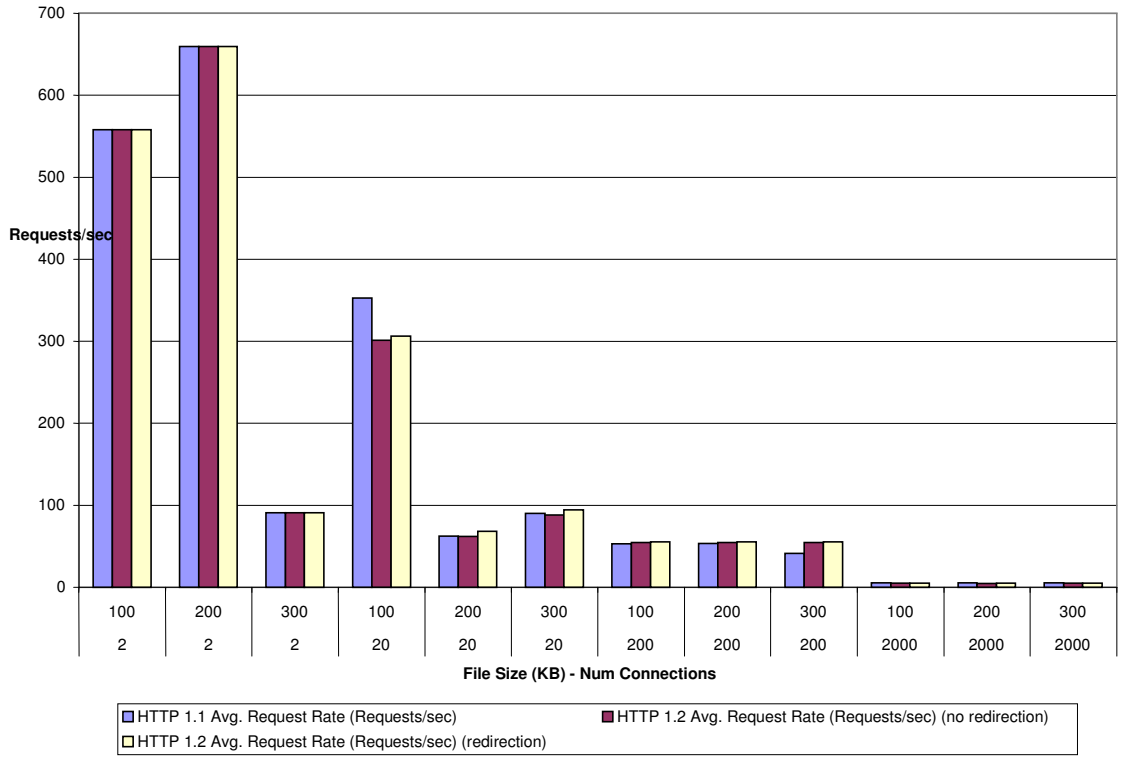

Figure 4: Average requests per second

What we can see from Figure 4 is the regular implementation of HTTP 1.1 with no redirection does very well with a size of 20 KB and 100 client requests but that HTTP 1.2 with redirection edges slightly ahead on further sizes and increased number of client requests. There is a significant drop off from 100 client requests to 200 client requests and then an increase in responsiveness when the client requests increases to 300. This pattern, although odd, was repeatable in subsequent experimentation. Apache may be using some sort of caching mechanism in the increased number of client requests.

Of significant importance is that HTTP 1.2 with redirection handles an average 3% more requests per second than HTTP 1.1 with the biggest gain of 34% occurring at 300 client requests with a file size of 200 KB. We actually expected an even bigger improvement

using HTTP 1.2 because there are two servers in use rather than just one used in HTTP 1.1.

6.3 Net I/O usage

Since our new method requires clients to open new connections to the redirection server, this added connection introduces a new overhead. We need to test to see what kind of overhead is associated with creating these new connections.

6.3.1 Configuration

The same hardware configuration was used in this test as in the previous test. Refer back to section 6.1.1 for details.

The test was run for both HTTP 1.1 and HTTP 1.2 with different file sizes of 2 KB, 20 KB, 200 KB, and 2 MB and different amounts of client connections. As in the test for replies per second, the metrics for the 20 MB files and above were not helpful and are not reproduced here.

6.3.2 Results

The backbone of the network is a 10/100 Mbps switch, which means throughput rates are quite higher than what most people have at home. With such high throughput rates, the time associated with new connections is actually more of an overhead than that for lower throughput rates. This is because setting up a new connection is relatively constant no matter what connection speed a client has.

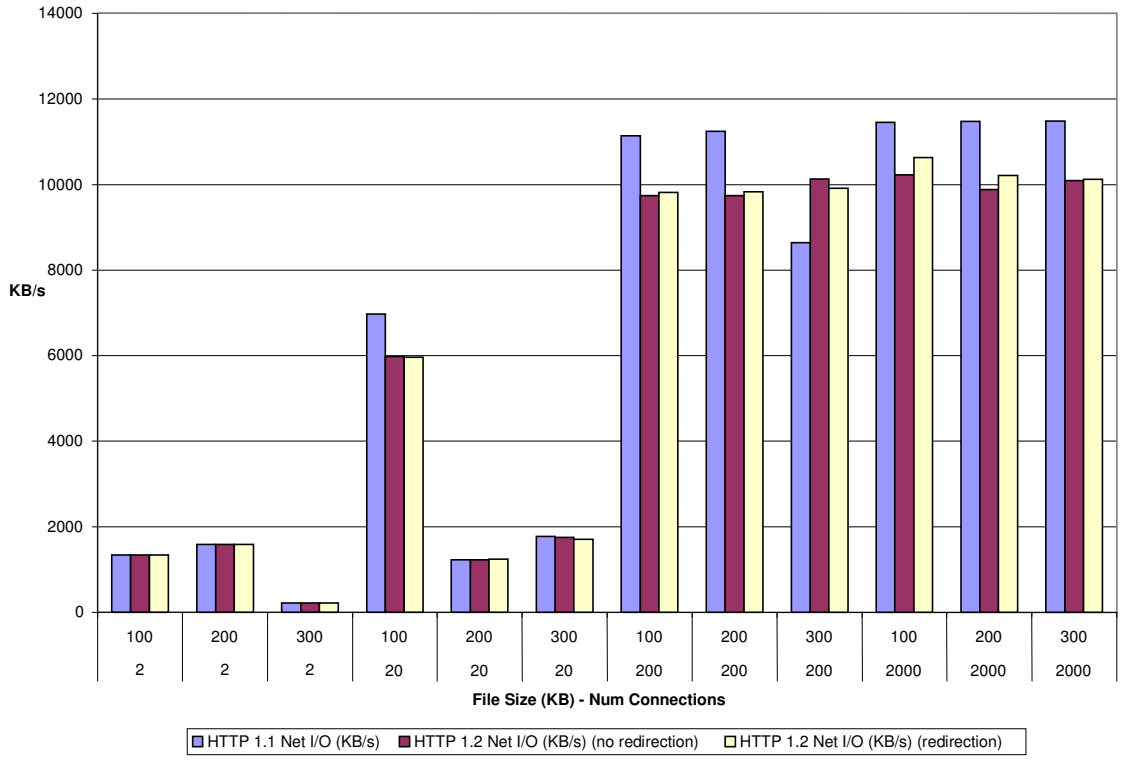

Figure 5: Net I/O usage

The results in Figure 5 indicate the implementation of HTTP 1.1 has a higher throughput rate than HTTP 1.2. We would expect this as the act of redirecting takes away from actual data transfer enough to show in the throughput rate. However, we do see that HTTP 1.2 with redirection has a better overall throughput rate than HTTP 1.2 without redirection.

The theoretical throughput limit for a 100 Mbps connection in KB per second is 12,500 KB per second. Although none of experimental results indicate rates reaching this high, the HTTP 1.1 implementation gets the closest with a top rate of 11,478 KB per second. Given a 1 Gbps connection, we may see HTTP 1.1 increase even further ahead of HTTP 1.2.

6.4 Testing in the Cloud

There is something to be said for testing in an environment as close to a real-world example as possible. In an attempt to match real-world performance we used Amazon Web Services Elastic Compute Cloud to stage virtual HTTP servers to host both the original HTTP daemon as well as our redirecting HTTP daemon.

6.4.1 Configuration

Servers were staged in the Elastic Compute Cloud as small instances with a single core processor, 1.7 GB of RAM, and a 10 Mbps connection to the Internet running Ubuntu 8.10. The client was staged as a virtual machine on a computer running VMware Server with a single core processor, 512 MB of RAM, and a 1.5 Mbps connection to the Internet running Ubuntu 8.10. In the case of testing redirection two servers were staged in the Cloud: one to operate as the primary server and one to operate as the secondary server to which clients are redirected.

The client made 1000 sequential requests to the server for a 300 KB file. The file was chosen prior to testing at random out of a set of large files. The results include the average time it took to fulfill a request. We let HTTPerf issue the requests and handle responses at the rate it deemed appropriate.

We wanted to see how the performance changed when the servers were operating at normal conditions versus operating under loaded conditions. Redirection is meant to help when the server is under extreme load so we wanted to see if this was the case. Load was generated against the primary server by running HTTPerf on the server running local

requests with the hog parameter, meaning it would use as much local resources as possible to fulfill the requests.

Just as in the previous tests we also wanted to see the impact introduced by packetizing at different sizes and redirection points. Packet sizes were tested using 1000, 2000, 3000, and 4000 byte packets. Redirection points, the percentage at which a client is redirected, were tested using 10%, 30%, 50%, 70%, and 90%.

6.4.2 Non-loaded Results

Figure 6: Non-loaded Server: Redirection vs. Time/Request

The results in Figure 6 indicate redirection negatively impacts performance when a server is not loaded. This conclusion comes as no surprise based on our previous tests that show packetizing when no redirection is intended means extra work. What is surprising is how poorly increasing the packet size affected performance. The 1000 byte packet method

remained relatively stable while all others show a significant decrease in performance as the redirection percentage increased. This matches how we originally predicted keeping packet sizes below the MTU would perform, so in this case it worked well packetizing in smaller sizes. As the packets grew the data was spread across multiple TCP/IP packets. On a LAN, such as our previous tests, this presents a lot less of a problem than when making multiple hops over the Internet going through multiple switches and routers that each deal with the TCP/IP packet sizes.

6.4.3 Loaded Results

Figure 7: Loaded Server: Redirection vs. Time/Request

The results in Figure 7 indicate redirection is beneficial for a loaded server. The amount of effort to produce the content in packets is the driving factor here as the results indicate using larger packets, thereby requiring less packetizing at the HTTP daemon level, leads to better overall performance. The benefits of packetizing are diminished as the primary

server takes on more of the load, redirecting to the secondary server later in the lifetime of requests.

These results more closely match our previous LAN-based tests indicating the larger packets led to better system performance due to overhead of smaller packet sizes and internal Apache mechanisms.

6.4.4 Random URI Usage

We recognized the OS may be doing some level of caching when fetching the same file over and over again. We wanted to test using a random URI from a list of URIs in order to realize if caching in the OS was skewing any results. Through some simple modifications to HTTPerf we were able to get it requesting a random URI from a list of 50 files of equal size. The files were in fact copies of the 300 KB file used in the normal testing.

After running all the same tests we did not find the caching done by the OS led to any noticeable gains or losses in performance.

6.4.5 Response Data Processing Overhead

HTTP 1.2 processing has an inherent overhead over HTTP 1.1 due the need of inspecting the data in HTTP 1.2 for in-stream control messages. HTTPerf does not inspect HTTP 1.1 data. We wanted to identify what sort of impact this had on the processing of requests. HTTPerf collects samples of response processing times and includes those times in its standard set of statistics. We inspected those statistics to discover the overhead of HTTP 1.2 data processing.

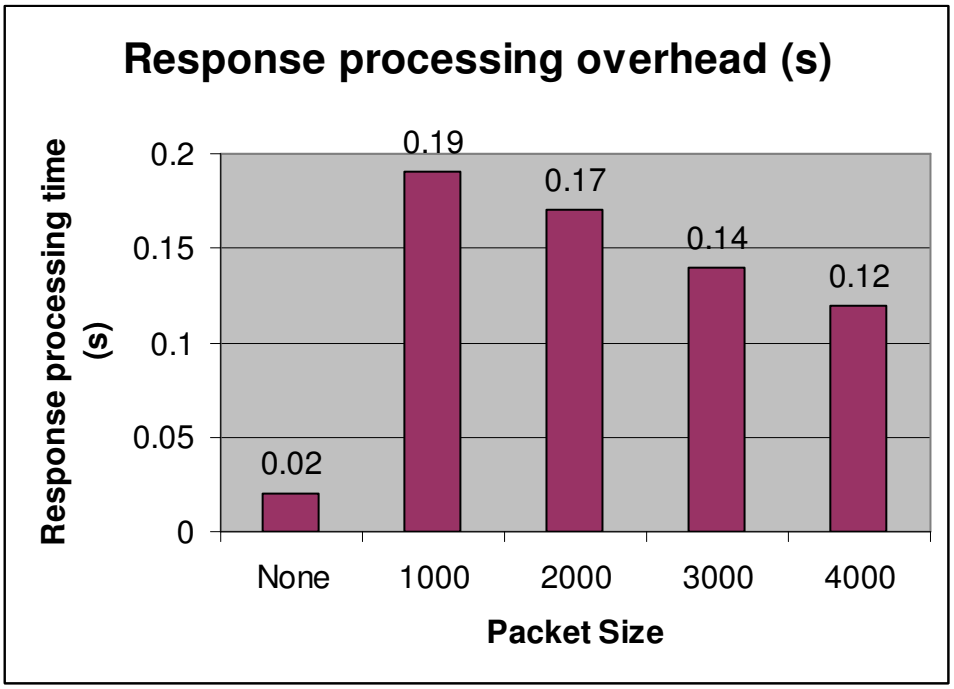

Figure 8: Response processing overhead (s)

The results illustrated in Figure 8 show that there is a significant difference in processing the data in HTTP 1.1 responses and HTTP 1.2 responses. In the case of 1000 byte packets the difference is .17 seconds per response. In our testing we are using only one client to process all the requests and responses for 1000 responses. We believe this would be less in the real environment where the responses are spread across a number of clients.

6.5 Summary

In all the tests where HTTP 1.2 without redirection was tested against HTTP 1.2 with redirection, redirecting proved to be beneficial. This indicates that having a secondary server to deal with requests when the primary server is saturated can present beneficial performance gains. However, there are definitely indicators that HTTP 1.2 could be detrimental to performance even with the benefits of redirection.

All client load in our testing was generated on a single computer, whereas in the real environment multiple computers would be acting as clients. It is quite possible the single client in our testing is creating a bottleneck, which we cannot avoid. It is certainly not a trivial activity to arrange hundreds of client computers for testing and so was not considered in any of our analysis.

The key indicator to use when deciding if redirection will be beneficial is based on the network speeds. On LAN-based speeds there is virtually no reason to use HTTP 1.2. The overhead and hassle of running two servers outweighs the performance gains by using HTTP 1.2. However, HTTP 1.2 can be beneficial when clients connect to servers over the Internet because the work done to packetize data is less than the benefits of load balancing across the two servers.

It might go without saying, but another major factor in deciding to use HTTP 1.2 is the possibility for high-demand content. As HTTP 1.2 performs weaker than HTTP 1.1 in day-to-day tasks due to overhead even when not redirecting, an environment where content is not in high-demand from multiple clients at one time, or in which the servers are significant enough to not incur saturation, would not benefit at all from HTTP 1.2. In these cases HTTP 1.1 is the winner and HTTP 1.2 should not be used.

The conclusion that must be drawn is that HTTP 1.2 is most useful only when clients connect to the servers over the Internet and when content on the servers is in high-demand most of the time or gaining access to the content when in high-demand is crucial.

HTTP 1.2 could also be useful when trying to guarantee QOS to a subset of clients. In this scenario special treatment could be given to some clients by redirecting them to secondary server when load situations arise on the primary server.

7 CONCLUSIONS

We set out to show that redirection with a modified version of HTTP would increase the performance when serving a file from a server subject to overloading. The results do indicate some performance gain in some areas, but it is not enough at this time to warrant widespread use of a new version of HTTP. However, the implementation done during this work was intended more as a proof of concept rather than a final solution, and as such has certainly served its purpose. A revisit into Apache's HTTP server or other HTTP server implementation may result in findings to increase the performance of HTTP 1.2 with redirection.

The changes proposed to HTTP are not overly complex, and introduce a new feature into an otherwise complete and "finalized" protocol. Internet resources increasingly become distributed across multiple servers, as is the case in many open source communities[20], and a need for dealing with distributed content in a controlled and described manner remains. A solution in the form of modifications to HTTP can help guarantee cross-platform and cross-company adoption. Standards and protocols exist to reduce confusion and expense on the part of clients and entities owning the servers.

Finally, the implementation of HTTP 1.2 provides server administrators with a level of control not afforded by other implementations. They can use a multitude of operating systems, data structures, and patterns of server distribution and network topography. Assuming a client uses a HTTP 1.2 enabled browser, the administrator is guaranteed the tools to implement a distributed content system, with no need to force clients to install

new software. Even more important is that HTTP 1.2 can be used in conjunction with most, if not all, the other methods for load balancing visited in this paper. Such a combination would introduce a level of load balancing far more powerful than what is currently available.

8 FUTURE WORK

Much like any new development, further development can introduce new ideas and concepts that never existed or were possible in the previous environment. Below you will find a few ideas we have for new features and improvements.

8.1 File size threshold

The idea behind defining a file size threshold is meant to avoid wasted efforts by a server when load is particularly low. Splitting files into packets takes more server effort than if the server just served content strait. The only reason to split content into packets is when a load is expected to degrade a server's ability to serve content to clients.

If a server is serving particularly small content, then the chance that load balancing decisions will be required during the time it takes to serve the content is small. So, we suggest setting a *threshold* value that defines the minimum size required of content in order for the server to split the content into packets. The threshold will most likely vary from server to server and will require some experimental research to establish a sufficient threshold value.

The threshold will be determined by two main questions. How fast is the server's connection to clients? What is the performance specifications of the server's hardware configuration? A faster hardware usually indicates the ability to handle more clients than that of a slower system. A faster Internet connection increases the capacity at which the server can respond to client requests and digest client requests. The threshold is therefore

directly proportional to the answers provided to these two questions. A faster hardware setup and faster connection will both allow for a higher threshold.

8.2 Web browser

The proof of concept for this work was done using a performance measuring tool, which is not fully featured like a popular web browser such as Mozilla's Firefox[19]. In the future we hope that the protocol will be implemented in web browsers to prove that it can be useful in viewing web content or in a download manager.

8.3 Concurrent connections

By splitting the content up into packets it is quite possible that a client can get packets from different servers at the same time. For instance, a client could connect to two servers getting the first half of the content from one server and the second half of the content from the other server at the same time. If both servers upload at 100 Kbps but the client can download at 200 Kbps, then the client will be able to download the content twice as fast as if they downloaded from one server alone. This technique is commonly referred to as *pipelining*.

8.4 Striped content

Clients are receiving packets from servers rather than the entire content all at once. We can make use of this fact by hosting only some packets by servers. This is referred to as striped files[10]. Striping files is a way of backing up files without storing the entire file in one place. This means that if one server gets corrupted only part of the file is lost, and if the part is hosted by another server then the file can be easily recovered.

Striping files can also be considered a security measure, especially when talking about Internet sites. A client can only get a file it they have access to all the servers that host the segments of the file. So even if a client were to manage to get passed other security measures and get a part of the file, they would have to overcome the security measures of all the servers in order to obtain the file. This can prevent illegal downloading of some content.

8.5 Extra methods

We concentrated our work on the RESUME method but there are a few messages that can be considered in achieving the peer-to-peer network functionality.

8.5.1 SUBSCRIBE method

HTTP 1.2 is meant to work hand-in-hand with multi-server systems hosting identical content. One feature that needs to exist is an easy way for servers to notify other servers of their intention to host identical content. The SUBSCRIBE method is introduced to facilitate just such a subscription service for Internet servers. A server intending to host identical content of another server will issue a SUBSCRIBE method to the host of the content. The content server can then add the subscribing server to its list of possible redirection servers if it decides the subscriber is a trustworthy source.

A simple pass-code authentication method has been added to the SUBSCRIBE method to avoid entry of malicious entities. The authentication portion of the SUBSCRIBE method is optional to avoid limiting server systems that require no authentication.

SUBSCRIBE method fields are described in Table 8 and include enough information for a sever to uniquely identify itself amongst other subscribers.

Table 8: SUBSCRIBE method fields

Header Field	Description
SUBSCRIBE <content-name>	Method name along with the URL minus the domain name of the content that was duplicated.
Server-domain	A domain name of the server subscribing to the content. i.e., www.cnn.com.
Port	The port number on the subscribing server associated with HTTP traffic.
Authentication	An authentication code known by both subscriber and subscribed. This field should be encrypted by an encryption method known by both parties. This field is optional. If subscribing to a public server system, then there may be no barrier to entry.

The SUBSCRIBE method follows this format:

```
SUBSCRIBE <content-name> CRLF

Server-domain: <domain-name> CRLF

Port: <http-port-number> CRLF

[Authentication: <encrypted-authentication-code> CRLF] CRLF
```

8.5.2 UNSUBSCRIBE method

Just as servers can subscribe to content, they must also be able to unsubscribe from the content. The UNSUBSCRIBE method allows servers that no longer wish to host content to advertise to other servers that they are no longer valid redirection servers. The header fields for an UNSUBSCRIBE request are presented in Table 9.

Table 9: UNSUBSCRIBE method fields

Header Field	Description
UNSUBSCRIBE <content-name>	Method name along with the URL minus the domain name of the content that was duplicated.
Server-domain	A domain name of the server subscribing to the content. i.e., www.cnn.com.
Port	The port number on the subscribing server associated with HTTP traffic.
Authentication	An authentication code known by both subscriber and subscribed. This field should be encrypted by an encryption method known by both parties. This field is optional. If subscribing to a public server system, then there may be no barrier to entry.

The UNSUBSCRIBE method follows this format:

```
UNSUBSCRIBE <content-name> CRLF

Server-domain: <domain-name> CRLF

Port: <http-port-number> CRLF

[Authentication: <encrypted-authentication-code> CRLF] CRLF
```

8.6 Load balancing policies

An entire second paper could be written on the subject of suggesting load balancing policies. We only implemented some rudimentary static load balancing in our analysis section. In addition to our rudimentary load balancing method we also only had one secondary server to which to redirect.

Given a handful of servers one could devise a load balancing method that involved the servers communicating amongst each other to actively direct traffic to under utilized

servers. Such a method would lead to a better overall spread of client load across a set of servers.

Load balancing methods could be driven by other factors such as a quality of service model in which some clients are considered above other clients in deciding which can remain downloading content and which must wait. Such a method would be useful for premium services in which customers pay to be included in a VIP group to ensure their content is always delivered as fast as possible. The load balancing method could either redirect VIP clients to an under utilized server or force non-VIP clients to wait with the SUSPEND message.

The areas of load balancing techniques is far too vast to try and cover within the confines of this paper, and so we leave it up to future individuals to explore, develop, and include their own load balancing suggestions.

9 BIBLIOGRAPHY

[1] A. Shaikh, R. Tewari, and M. Agrawal, "On the effectiveness of DNS-based server selection," in Proceedings of the IEEE INFOCOM, Anchorage, AK, April 2001.

[2] Akamai Technologies. “Akamai: World's largest distributed computing platform for your e-business needs.” http://www.akamai.com/html/technology/index.html. Last visited April 18, 2010.

[3] Amazon. “Amazon Elastic Compute Cloud (Amazon EC2).” http://aws.amazon.com/ec2/. Last visited April 18, 2010.

[4] Apache Software Foundation, The. “The Apache HTTP Server Project.” http://httpd.apache.org. Last visited April 18, 2010.

[5] Cable News Network LP, LLLP. “CNN.com”. http://www.cnn.com. Last visited April 18, 2010.

[6] Cardellini, V.; Colajanni, M.; Yu, P.S., “Dynamic Load Balancing On Web-Server Systems.” IEEE Internet Computing, v 3, n 3, June 1999, p 28-39.

[7] Choi, Eunmi, “Performance test and analysis for an adaptive load balancing mechanism on distributed server cluster systems.” Future Generation Computer Systems, v 20, n 2, Feb 16, 2004, p 237-247.

[8] Cisco Systems. “TCP/IP.” http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a008014f8a9.shtml. Last visited April 18, 2010.

[9] CNET Networks. “GameSpot: for your PC, Playstation 2, Xbox, GameCube, GBA, and video game needs.” http://www.gamespot.com. Last visited April 18, 2010.

[10] Corry Publishing, Inc. “Business Solutions – Segment those files.” http://www.businesssolutionsmag.com/Articles/2001_05_15/01051508.htm. Last visited May 20, 2005.

[11] Garber, Lee. “Denial-of-Service Attacks Rip the Internet.” Computer [Magazine], April 2000, p 12-17.

[12] Haungs, Michael; Barnes, Fritz; Barr, Earl; Pandey, Raju, “A Fast Connection-Time Redirection Mechanism for Internet Application Scalability.” International Conference on High Performance Computing (HiPC 2002), December 18-21 2002, Bangalore, India.

[13] Jackson, Michael. Software Requirements & Specifications. Addison-Wesley, London 1995.

[14] Li, Chang; Peng, Gang; Gopalan, Kartik; Chiueh, Tzi-Cker. “Performance guarantees for cluster-based internet services.” Proceedings - International Conference on Distribute Computing Systems, 2003, p 378-385.

[15] Luo, Mon-Yen and Yang, Chu-Sing. “Constructing Zero-Loss Web Services.” INFOCOM 2001.

[16] Luo, Mon-Yen and Yang, Chu-Sing, “System Support for Scalable, Reliable and Highly Manageable Web Hosting Service.” USITS 2001.

[17] M. Andrews et al., “Clustering and Server Selection using Passive Monitoring.” INFOCOM 2002.

[18] Mosberger, David and Jin, Tai. “httperf---A Tool for Measuring Web Server Performance.” http://www.hpl.hp.com/research/linux/httperf/index.php. Last modified January 30, 2009. Last visited April 18, 2010.

[19] Mozilla Organization, The. “Firefox web browser | Faster, more secure, & customizable.” http://www.mozilla.com/en-US/firefox/firefox.html. Last visited April 18, 2010.

[20] Open Source Technology Group. “SourceForge.net: Find and Develop Open Source Software.” http://sourceforge.net. Last visited April 18, 2010.

[21] Pandey, Raju and Barnes, J. Fritz. “Supporting Quality of Service in HTTP Servers.” In Proceedings of the Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC). (Puerto Vallarta, Mexico, June 1998).

[22] R. Fielding et al. “Hypertext Transfer Protocol - HTTP/1.1.” http://www.w3.org/Protocols/rfc2616/rfc2616.html. Last visited April 18, 2010.

[23] S. Rangarajan (Bell-Labs), S. Mukherjee (Bell-Labs), P. Rodriguez. "A Technique for User Specific Request Redirection in a Content Delivery Network". Workshop on Web Content Caching and Distribution, New York. Sep, 2003.

[24] T. Stading, P. Maniatis, and M. Baker. “Peer-to-peer caching schemes to address flash crowds.” In 1st International Workshop on Peer-to-Peer Systems (IPTPS).

[25] Wesley, Tony. “Tony Wesley’s Home Page.” http://tonywesley.com/pipetop.html. Last visited April 18, 2010.

APPENDIX A: APACHE HEADER FILTER

```
/* Add our headers to the content */
ap_set_content_type(r, apr_pstrcat(r->pool, "multipart",
     use_range_x(r) ? "/x-" : "/",
     "byteranges; Session-ID=",
     ctx->boundary,
     NULL));
```

APPENDIX B: APACHE TRANSFER FILTER

```
AP_CORE_DECLARE_NONSTD(apr_status_t) ap_byterange_filter(ap_filter_t
*f, apr_bucket_brigade *bb)
{
...
    while (current_packet <= ctx->num_ranges && !redirect ) {
        apr_bucket *e2;
        apr_bucket *ec;

        /* these calls to apr_brigade_partition() should theoretically
         * never fail because of the above call to
apr_brigade_length(),
         * but what the heck, we'll check for an error anyway */
        if ((rv = apr_brigade_partition(bb, range_start, &ec)) !=
APR_SUCCESS) {
            ap_log_rerror(APLOG_MARK, APLOG_ERR, rv, r,
                          PARTITION_ERR_FMT, range_start, clength);
            continue;
        }
        if ((rv = apr_brigade_partition(bb, range_end+1, &e2)) !=
APR_SUCCESS) {
            ap_log_rerror(APLOG_MARK, APLOG_ERR, rv, r,
                          PARTITION_ERR_FMT, range_end+1, clength);
            continue;
        }

        redirect = (current_packet >= redirectat &&
!has_starting_packet);

        found = 1;

        /* For single range requests, we must produce Content-Range
header.
         * Otherwise, we need to produce the multipart boundaries.
         */
        if (ctx->num_ranges == 1) {
            apr_table_setn(r->headers_out, "Content-Range",
                           apr_psprintf(r->pool, "bytes "
BYTERANGE_FMT,
                                        range_start, range_end,
clength));
        } else if (redirect) {
            ctx->bound_head = apr_pstrcat(r->pool,
                                          CRLF "REDIRECT",
                                          CRLF "Server: ", "localhost",
                                          CRLF CRLF, NULL );
            ap_xlate_proto_to_ascii(ctx->bound_head, strlen(ctx-
>bound_head));

            e = apr_bucket_pool_create(ctx->bound_head, strlen(ctx-
>bound_head),
                                       r->pool, c->bucket_alloc);
            APR_BRIGADE_INSERT_TAIL(bsend, e);
        } else {
```

```
            char *ts;

            ctx->bound_head = apr_pstrcat(r->pool,
                                  CRLF "PACKET",
                                  CRLF "Session-ID: ", ctx->boundary,
                                  CRLF,
                                  NULL);
            ap_xlate_proto_to_ascii(ctx->bound_head, strlen(ctx-
>bound_head));

            e = apr_bucket_pool_create(ctx->bound_head, strlen(ctx-
>bound_head),
                                       r->pool, c->bucket_alloc);
            APR_BRIGADE_INSERT_TAIL(bsend, e);

            if (current_packet != ctx->num_ranges) {
               ts = apr_psprintf(r->pool, "Packet-number: %u" CRLF
CRLF,
                                 current_packet++);
            } else {
               ts = apr_psprintf(r->pool, "Packet-number: %u" CRLF
"Packet-length: %u" CRLF CRLF,
                                 current_packet++, range_end -
range_start + 1);
            }
            ap_xlate_proto_to_ascii(ts, strlen(ts));
            e = apr_bucket_pool_create(ts, strlen(ts), r->pool,
                                       c->bucket_alloc);
            APR_BRIGADE_INSERT_TAIL(bsend, e);
        }

        if (!redirect) {
            do {
                apr_bucket *foo;
                const char *str;
                apr_size_t len;

                if (apr_bucket_copy(ec, &foo) != APR_SUCCESS) {
                    /* this shouldn't ever happen due to the call to
                     * apr_brigade_length() above which normalizes
                     * indeterminate-length buckets.  just to be sure,
                     * though, this takes care of uncopyable buckets
that
                     * do somehow manage to slip through.
                     */
                    /* XXX: check for failure? */
                    apr_bucket_read(ec, &str, &len, APR_BLOCK_READ);
                    apr_bucket_copy(ec, &foo);
                }
                APR_BRIGADE_INSERT_TAIL(bsend, foo);
                ec = APR_BUCKET_NEXT(ec);
            } while (ec != e2);
        }
      /* GMO: I added these two lines in the hopes that it will send
along
       * what it has so far so we can decide later if we need to
redirect.
```

```
       */
        ap_pass_brigade(f->next, bsend);
        bsend = apr_brigade_create(r->pool, c->bucket_alloc);
        range_start += packet_size;
        range_end += packet_size;
        if (range_end > clength) range_end = clength - 1;
    }

    if (found == 0) {
        ap_remove_output_filter(f);
        r->status = HTTP_OK;
        /* bsend is assumed to be empty if we get here. */
        e = ap_bucket_error_create(HTTP_RANGE_NOT_SATISFIABLE, NULL,
                                   r->pool, c->bucket_alloc);
        APR_BRIGADE_INSERT_TAIL(bsend, e);
        e = apr_bucket_eos_create(c->bucket_alloc);
        APR_BRIGADE_INSERT_TAIL(bsend, e);
        return ap_pass_brigade(f->next, bsend);
    }

    e = apr_bucket_eos_create(c->bucket_alloc);
    APR_BRIGADE_INSERT_TAIL(bsend, e);

    /* we're done with the original content - all of our data is in
bsend. */
    apr_brigade_destroy(bb);

    /* send our multipart output */
    return ap_pass_brigade(f->next, bsend);
}
```

APPENDIX C: HTTPERF CALL GENERATOR

```
static char redHost[50];

static void
call_destroyed (Event_Type et, Call *call)
{
  Conn_Private_Data *priv;
  Conn *conn;

  assert (et == EV_CALL_DESTROYED && object_is_call (call));

  conn = call->conn;
  priv = CONN_PRIVATE_DATA (conn);

  if (++priv->num_destroyed >= MIN (param.burst_len, param.num_calls))
    {
      if (priv->num_completed == priv->num_destroyed
        && priv->num_calls < param.num_calls)
      issue_calls (conn);
      else {
      core_close (conn);

        if (call->redirect.needed) {
          // Set up the redirect call and connect it so the timer
process will pick up the tab
          Conn *red = conn_new();

          uint n = sprintf (redHost, call->redirect.server);

          red->hostname = redHost;
          red->hostname_len = strlen (redHost);
          red->fqdname = red->hostname;
          red->fqdname_len = red->hostname_len;
          red->is_redirecting = 1;
          red->last_packet_received = call-
>redirect.last_packet_received;

          core_addr_intern (red->hostname, strlen (red->hostname),
param.port);

          core_connect(red);

          param.num_redirects++;
        }
      }
    }
}

static char packetHeader[50];

static void
call_send_start (Event_Type et, Call *call)
{
  if (call->conn->is_redirecting) {
```

```
     uint n = sprintf ( packetHeader, "Starting-packet:%u\n\0", call-
>conn->last_packet_received + 1 );
     call_append_request_header(call, packetHeader, n);
  }
}
```