End-to-End Connectivity Across Firewalls

by

Vinod K. Rangarajan

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1998

© 1998 M.I.T. All rights reserved.
End-to-End Connectivity Across Firewalls

by

Vinod K. Rangarajan

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 1998

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As more and more sensitive data is transferred across the Internet, there is a growing need to protect this information. Currently, firewalls help to address this situation. However, many firewalls operate in an inefficient manner, due to the repeated opening and closing of underlying TCP connections. A better design would be to maintain a connection and reuse it as needed.

This paper proposes an efficient proxy-based firewall implementation that exploits persistent connections as specified by HTTP 1.1. It starts by giving relevant background information necessary to understand the mechanisms used in this project. It then introduces the design of a proxy which offers end-to-end connectivity. Finally, it examines the results of performance testing with the new proxy and proposes further work.

Thesis Supervisor: Dr. Karen R. Sollins
Title: Research Scientist, Laboratory for Computer Science
Acknowledgments

I would like to thank my parents and brother for their love and encouragement of my endeavors throughout my academic and personal life. Thanks to all of my friends for their helpful (most of the time) advice while I was working to finish my thesis. Thanks to Lewis Girod for his advice and for the many times he was able to help when my computer was not cooperating. Thanks to the Information Mesh group for valuable insights offered during our weekly meetings. Finally, I would like to thank my advisor, Dr. Karen Sollins for her guidance and assistance throughout this project.
# Table of Contents

1 Introduction ........................................... 7

2 Background ............................................ 11
   2.1 Motivation ........................................ 11
   2.2 Networking Models ................................ 12
      2.1.1 ISO Model ................................... 12
      2.1.2 Internet Model ............................... 13
   2.1 Firewalls .......................................... 16
      2.3.1 Gateway Example .............................. 19
   2.4 HTTP 1.1 ........................................... 20
      2.4.1 Overview ..................................... 20
      2.4.2 Requests ...................................... 21
      2.4.3 Responses ..................................... 21
      2.4.4 HTTP 1.0 vs 1.1 .............................. 22
   2.5 Related Work ...................................... 23

3 Design .................................................. 26
   3.1 Standard Proxy ..................................... 26
   3.2 "End-to-end" Proxy ................................ 28
   3.3 Code Design ....................................... 31
      3.3.1 Class Main ................................... 32
      3.3.2 Class Manager ................................. 32
3.3.3 Class Connection .................................. 32
3.3.4 Class Client ...................................... 33
3.3.5 Class Proxy ...................................... 33
3.3.6 Class ProxyServer ................................. 33
3.3.7 Class Message .................................... 34
3.3.8 Class Request .................................... 34
3.3.9 Class Reply ...................................... 35
3.3.10 Class RequestHandler ............................. 35

4 Performance 37

4.1 Performance Metrics ................................. 37
4.2 Test Setup and Execution ............................ 38
  4.2.1 Local Server .................................. 39
  4.2.2 Remote Server ................................ 40
  4.2.3 International Server ............................. 40
4.3 Discussion of Results ................................ 41

5 Conclusion 44

5.1 Summary ............................................ 44
5.2 Further Research ................................... 45
5.3 Concluding Remarks ................................. 46

Appendix A – Code Specifications 47

References 53
List of Figures and Tables

2-1  ISO OSI Protocol Layering Model ........................................ 13
2-2  Protocol Stack for Internet Architecture .................................. 14
2-3  Example of packet headers ...................................................... 16
2-4  Schematics of an isolation network .......................................... 18
2-5  Schematic of a firewall using a bastion host ............................... 19
3-1  Schematics of server socket operation ...................................... 27
4-1  Results of Latency Measurements for a Local Server ................. 39
4-2  Results of Latency Measurements for a Remote Server ............. 40
4-3  Results of Latency Measurements for an International Server .... 41
Chapter 1

Introduction

The tremendous growth of the Internet and Intranets has led to a corresponding growth in the amount of information travelling across networks. Much of this information can be classified as sensitive in nature, and as such, network security has become as important an issue as network connectivity. The need to protect information is important for a wide variety of reasons and for a wide range of people. Individuals do not want their private information accessed by others. Corporations do not want outsiders snooping around on their networks and looking at proprietary information.

Of course, one foolproof method to achieve this security is to remove computers from networks entirely. This way, there is no chance for anyone to infiltrate the machine short of physically stealing it, for which there is no good protection other than a security guard. However, removing it from the network renders the computer essentially useless, as it is no longer able to use the network’s resources. Therefore, this is not a viable means of achieving the desired security. As long as a machine is on a network, there will be the chance for a breach of security. The problem is thus to maximize the security of the machine without minimizing its functionality.

Currently, firewalls help to address this problem. In most cases, a firewall is implemented by simply having a machine act as a gateway to a network. This machine
intercepts all incoming requests and attempts to authenticate the external host that is making the request. If that host is allowed access to the internal network, then the firewall passes the request on to the internal host, waits for a reply, and then passes that reply back to the external host. Each time a request is made, the firewall opens a connection on the inside in order to process the request. Once the reply is received, it then closes the connection. A problem with this scheme is that if many requests are made to the same internal server, there will be correspondingly many connections opened and closed. Clearly, this is not the most efficient way to do this. A secondary problem is that the existing framework does not sufficiently support mobile hosts. As travel is becoming more prevalent in people’s lives, the need for mobile machines, and therefore, mobile security, is increasing as well. The following examples help to illustrate these problems more clearly.

An easily visualized situation is one involving a businesswoman – a consultant, for example – who travels often. While at her office, she has the security of the company’s firewall, so she does not need to worry about outsiders accessing information on her computer. However, when on the road, she takes her laptop with her and plugs into her client’s network. This creates several interesting situations. First, people on the client’s network can access data on her machine if the proper security is not in place. Second, people back on her company’s network may need to access data from her machine through the client’s firewall. Finally, she poses a threat to the client’s confidential information since she is an outsider and is now part of their network.

Assuming the proper security framework is in place (i.e. people can’t access what they’re not supposed to access), requests can be made to the consultant’s computer. As
described above, an inefficiency arises when multiple requests are made because several
connections are opened and closed. If it were possible to maintain a persistent, end-to-
end connection across a firewall, then this inefficiency can be eliminated.

This next example shows how this type of system could be useful from an Intranet
point of view. Most companies, especially large ones, have several divisions or groups,
each with its own network that is also part of the company’s overall network. For
example, suppose there is a company called XYZ Corporation and its network is named
xyz.com. Each division within XYZ would have its own domain; for example, there
could be sales.xyz.com, marketing.xyz.com, personnel.xyz.com, etc. Often, each group
has a firewall around its network, thus preventing access between internal networks. If
an employee wants to access information from a host that is not within his group, then he
will have to traverse a firewall to do so. The same inefficiency problem as in the
previous example is encountered in this example.

Specifically then, the problem is to develop a system such that a secure, end-to-
end connection can be maintained with a server residing behind a security perimeter. It
would also be nice to have support for mobile hosts. This project proposes such a system
based on a generic proxy server and extending it to support persistent connections under
the scheme presented in HTTP 1.1.

The remainder of this thesis will follow the following outline. Chapter 2 gives
background information relevant to this project, as well as discusses related works and
the motivation for this project. Chapter 3 describes the underlying design principles of
the system developed, and also gives a brief overview of the Java classes that were
written. Chapter 4 discusses the results of the performance analysis, and Chapter 5 draws some conclusions and proposes further work.
Chapter 2

Background

Before delving into the design of the system, it is important to first understand some of the background behind this project. In this chapter, I will talk about the motivation for this project, as well as describe relevant background information. This includes a discussion on networking models, firewalls, and protocols.

2.1 Motivation

The motivation for this project comes from previous graduate work by Matthew Condell [3]. In his work, Condell developed a security model for the Information Mesh project. The Information Mesh is involved in developing an information infrastructure architecture that supports longevity, mobility, and evolvability of information. Over time, information is prone to move from location to location. Currently, URLs (Uniform Resource Locators) are used as identifiers and they are adequate for data that just stays in one place. But as more and more long-lived information is placed on the World Wide Web, a new scheme for identification will need to be put in place. This is one of the primary goals of the Information Mesh.

Condell devised a security model for the Mesh that consists of several different kinds of servers, each of which plays a crucial role in the authentication of requests for
Web documents. His model uses a cascaded authentication protocol for servers to authenticate requests. A configuration of application servers (for example, web servers) is known as a domain, and each domain controls its own security model. To do this, each domain has associated with it a server that enforces that domain’s security policy. A domain server works with one or more authentication servers to decide whether or not to accept a request. A third type of server, known as a path server, is responsible for directing requests to the appropriate domain. Requests often have to pass through several of these servers, and necessarily, many connections are opened and closed. One can see the utility of having end-to-end connectivity in a situation like this. In addition, end-to-end connectivity can be used in situations such as the ones described in Chapter 1.

2.2 Networking Models

It is instructive to look at the structure of data networks to better understand the problem and how to approach solving it.

2.2.1 ISO Model

The International Standards Organization (ISO) open systems interconnection (OSI) model gives the most general form of network architecture. It calls for a seven-layer protocol stack at each end, starting at the bottom with the physical interface and ending at the top with the application layer (Figure 2-1). Under this kind of model, each layer does not need to know about how the other layers work. It only needs to provide an interface for communication with the layer directly above and the layer directly below it. Essentially, each layer extends the abstraction from the layer below it. For example, the
transport layer sends data to the network layer under the guise that the data will be sent directly to a destination transport layer. In reality, of course, the data flows down through the data link layer and across an actual physical link (or several links) and arrives at its destination. Once at its destination, the data is passed up the destination node’s protocol, eventually ending up at the transport layer. In practice, the ISO OSI model is not implemented exactly as specified, but for the most part it captures the essence of most network architectures and provides a good basis for understanding them.

![ISO OSI Protocol Layering Model](image)

**Figure 2-1 – ISO OSI Protocol Layering Model**

### 2.2.2 Internet Model

In widespread practice, a four-layer model for the Internet architecture is preferred to the seven-layer ISO model. The concepts behind the two models are similar, but obviously, their implementations are quite different. Figure 2-2 shows the protocol stack for the Internet. The four layers are the link layer, the network layer, the transport layer, and the application layer.

The link layer is often a combination of hardware and software. Its purpose is to provide networking capabilities to computers. Examples of protocols at this layer include
Ethernet or FDDI. The network layer protocol allows for different networking technologies to be interconnected. The protocol that is used at this layer is called IP (Internet Protocol). The network layer performs most of the routing functions. It figures out where each packet is going, what the best route is for each packet, and then sends it on its way via the lower level layers. It also can control the amount of traffic flowing over a connection so that delays are kept to a minimum, but most of this flow control is usually done in the transport layer.

![Protocol Stack for Internet Architecture](image)

**Figure 2-2 – Protocol Stack for Internet Architecture**

The transport layer performs several functions. These include breaking messages up into smaller sized packets; multiplexing many low-rate sessions into one session; splitting one high-rate session into several sessions; achieving reliable end-to-end communication; and performing end-to-end flow control. Depending on the network, the transport layer can perform either some or all of these functions. Some of these functions
can also be done at the network layer; again, this depends on the way the network was designed. In the Internet model, there are two popular transport protocols: TCP (Transport Control Protocol) and UDP (User Datagram Protocol). For this project, the relevant one is TCP. It provides a communication channel between machines through the use of reliable byte streams.

The application layer is, simply, the layer that takes care of application specific tasks. The other layers take care of establishing connections between hosts and the applications make use of these connections to achieve their goals. For example, when you want to transfer files from one place to another, there are certain things that need to be done specifically for this task that you cannot embed in other layers. The File Transfer Protocol (FTP) takes care of these things. The application with which we will be concerned is the Hypertext Transfer Protocol (HTTP) which deals with the transfer of hypertext documents, such as web pages.

In the Internet architecture, when one wants to move data from one node to another, HTTP represents the data in hypertext and passes it down to TCP. TCP then wraps the data up in packet form, in a manner determined by its specifications, attaches a header field, and passes it down to IP. IP then attaches its own header field to the packet it has received from TCP and sends the packet along to the receiving node. At the receiving node, the data is passed up a corresponding protocol stack, IP to TCP to HTTP (Figure 2-3).
2.3 Firewalls

When two machines are connected and communicating with each other, as described in the previous section, it is desirable to have an end-to-end connection. That is, you want the machines to talk directly between themselves. However, when one of the machines is protected behind a firewall, it is not possible to achieve this type of connection with the current protocols. A firewall is essentially a barrier between a network and the outside world. It acts as an intermediary between the two servers. It breaks the connection from the original server and then opens one with the internal server. It then passes messages on to the inside machine on behalf of the original server. Unfortunately, each of these connections is used only once and then closed. This results in inferior performance compared to an end-to-end connection. In terms of the model above, what we want to do is de-couple the HTTP layer from the TCP/IP stack. At the application layer, it should
appear as if there is an end-to-end connection, when in reality, there are multiple TCP connections.

As mentioned earlier, currently there exist frameworks by which a domain is secure from outsiders through the use of firewalls. The term firewall can be used to describe any of the several security schemes that are actually in use. The following describes three of the more common schemes in use today. They are router-based filters, isolation networks, and host computer gateways (or bastions).

Perhaps the simplest method of providing a firewall is the use of router-based filters. Basically, the router controls traffic at the IP level by examining the source and destination addresses in the packet header information. Based on this information, the router determines whether or not to allow the packet to pass through or to block it. The system administrator programs the router and tells it what to keep out. While packet filtering is a simple scheme, and supports end-to-end connectivity in the sense that control occurs at the IP level, it does not provide a fine enough granularity of security. It makes decisions based on the source and destination addresses, whereas we want to make decisions on a user by user basis.

A second scheme is the use of isolation networks. In this method, a completely separate subnetwork is created and placed between the external and internal networks. Both the internal network and the Internet can access this isolated subnet, but traffic across it is strictly prohibited (Figure 2-4). Usually, the isolation is achieved through router-based filtering between the subnet and the other networks.
A third method, which is similar to the second one, is to place a secure computer as a gateway between the external and internal networks. This gateway computer is known as a bastion. Unlike the filtering router, the bastion’s control is done at the application level. The applications allow data to pass through to the internal network only after verifying that it fits within the specified restrictions (Figure 2-5). The advantage of isolation networks and gateways is that they provide the proper level of security, however the connectivity is not adequate. In these schemes, the control is at the application level, and the connection is actually broken at the intermediate point. Ideally, we would like a system that provides the desired level of security, as well as maintains an end-to-end connection.
2.3.1 Gateway Example

Gateway machines execute software that are stripped down versions of common applications such as FTP, telnet, or HTTP. These “proxy” versions are written so that access can be controlled and they operate under a forwarding principle. They receive data from some external client, perform some authentication, and then forward the data to some internal server.

Going back to the earlier example of XYZ Corporation, if a host on the Internet tried to connect to a host on xyz.com, the following scenario would most likely take place. The proxy for xyz.com, as the single entry point for the firewall, receives the request from the source host, breaks the connection with the source, and then forwards the request on to the internal host. Upon receiving the reply from within the firewall, the proxy then re-establishes a connection with the source host and forwards the reply on to it. Evidently, this requires a lot of unnecessary openings and closings of connections, whereas maintaining one connection is the right thing to do. In this project, we will take an HTTP proxy and try to extend it to support end-to-end connectivity.
2.4 HTTP 1.1

Since the basis for this project lies in the exploitation of a property of HTTP 1.1, it is important to first understand the protocol before looking at the design of the system that was developed. In the following section, I will describe how the Hypertext Transfer Protocol works, and then examine the changes between versions 1.0 and 1.1 of the protocol.

2.4.1 Overview

HTTP is a protocol developed for data transfer across the Internet. It operates in a messaging fashion. A client sends a request message over a connection to a specified server. The server looks at the request, processes it, and replies to the client with a response message. This message passing usually takes place over a TCP connection. There can be zero to any number of intermediaries between the client and server. In other words, the client and server could be directly connected, or there could be proxies, gateways, and/or tunnels in between.

A message in HTTP consists of a start-line, zero or more header fields, and an optional message body. The format is as follows:

```
Message = start-line
message-headers
CRLF
message-body
```

The CRLF (carriage return/line feed), or empty line, denotes the end of the list of header fields. Note that the start-line and each of the headers appear one to a line. The start-line
will vary in format depending on whether it is for a request or response, as will the header fields. The general form for a message header is:

\[
\text{message-header} = \text{field-name} \ : \ : \ \text{field-value}
\]

### 2.4.2 Requests

A request message takes the form of the generic message illustrated above, with the start-line given by:

\[
\text{start-line} = \text{Method} \ \text{SP} \ \text{Request-URI} \ \text{SP} \ \text{HTTP-Version}
\]

where SP indicates a space. The Method token tells the server (identified by Request-URI) what kind of a request it is. Some of the more common methods include GET and POST, which are used to retrieve Web documents and to submit data to a form, respectively. HTTP-Version indicates which version of the protocol the client is using.

The start-line is followed by zero or more header fields that the client may use to pass additional information to the server. This information can include things such as the type of response it is expecting, the name of the requesting host, or even usernames and passwords. If the client is sending any data to the server, such as form inputs, it is placed in the message-body of the request.

### 2.4.3 Responses

A response message also follows the generic message format described in Section 2.4.1. Its start-line is given by the following:
start-line = HTTP-Version SP Status-Code SP Reason-Phrase

HTTP-Version in this case indicates the version of the protocol that the server will issue its response in. This is to ensure that both the client and server are using the same protocol. For instance, if the client makes a request in HTTP 1.1, and the server is not configured to handle that version of HTTP, then the server will respond in HTTP 1.0. All implementations of the protocol, whether they be client-side or server-side, must be backwards compliant. Status-Code is a three-digit code which indicates the result of the server trying to process the request. For example, if the request is understood and satisfied, then the server will set the code to 200. Reason-Phrase is just a plain text description of the status code.

The header fields are used to pass additional information from the server to the client. This information could includes things such as the name of the server, or how long to wait before retrying the request if it could not be processed immediately. The server can also use the header fields to request the client to authenticate itself to the server before the request is processed. Any data that the server needs to pass on to the client (i.e. HTML data) is sent in the message body of the response.

2.4.4 HTTP 1.0 vs HTTP 1.1

There were changes made between versions 1.0 and 1.1 of HTTP in several areas. These areas include caches, virtual hosts, and connections. This project tries to take advantage of the enhancements to connection management provided by the new version of the protocol.
Under HTTP 1.0, each request requires its own TCP connection. Once a response is received to the request, the connection is closed; if another request is made immediately afterwards, it has to open a brand new TCP connection. Inefficiency due to the cost of opening and closing these connections arises if several requests are made to the same server. This inefficiency is magnified when dealing with proxy-based firewalls. In this situation, there are already double the number of connections (one from the client to the proxy, and one from the proxy to the server), so when several requests are made, the overhead in opening and closing connections is larger than in the directly connected case.

In HTTP version 1.1, the default behavior is to maintain a persistent connection between two hosts. This allows multiple requests to be sent along the same connection. While similar functionality was provided for in version 1.0, it was not the default behavior and an explicit "Connection: Keep-Alive" header had to be sent along with each request in order to leave the connection open. Most servers were also not configured to handle such persistency. So, each time an HTTP request was sent, a new connection had to be opened, even if the request was being made to the same site. By developing a proxy that exploits this new feature of HTTP 1.1, one can establish the appearance of an end-to-end connection, at least at the application level, even when one host is behind a firewall.

2.5 Related Work

Several studies have been done examining the performance of HTTP. In this section, I will discuss three such studies, their implications, and their relevance to this project.
In 1994, before the release of HTTP 1.1, Spero [13] performed an analysis on the problems with HTTP 1.0. He looked at a TCP trace of a typical HTTP transaction and found that more time was spent idling than was spent actually transferring data. These delays can be directly related to the cost of opening and closing TCP connections. For this reason, and because only one request is made per connection, he concluded that HTTP performs badly with TCP. To help alleviate this problem, he suggested the use of "long-lived" connections.

Also in 1994, Padmanabhan and Mogul [10] conducted a similar study. They identified the same performance problems, but took it one step further and actually implemented a system with "long-lived" connections. They also suggested and implemented the ability to pipeline requests. That is, multiple requests can be sent along a connection without having to wait for responses from earlier ones. Their results indicated that performance improved significantly with persistent connections, and even more so with the addition of pipelining. The authors also thought that these improvements would reduce the load on the server, since it wouldn't have to create as many processes to handle requests.

In 1997, after the release of HTTP 1.1, Nielsen et al. [9] implemented client and server systems that were compliant with persistent connections and pipelining as described by the new specifications, and ran performance test with them. They found that four non-persistent connections running in parallel (as is the default behavior in Netscape’s Navigator) actually outperforms a single persistent connection. When the persistent connection is coupled with pipelining, however, it performs much better than
the HTTP 1.0 system. The authors believe that HTTP 1.1 with pipelining will "significantly change the character of traffic on the Internet."

These three studies all indicate that performance is enhanced with the use of persistent connections. However, none of them examined its effect on proxy-based firewalls. In the following chapters, I will describe my implementation of an HTTP 1.1 compliant proxy, and how it performs in comparison to a standard proxy.
Chapter 3

Design

This chapter details the design of a system that implements an end-to-end connection across a firewall as desired. First, we look at the design of a standard proxy server to understand how it works. Then, we extend the design to provide the functionality necessary to support persistent connections as specified by HTTP 1.1. With this added functionality, we are able to construct a system that supports connectivity through firewalls.

3.1 Standard Proxy Server

A standard proxy server must behave as previously described in Chapter 2. To summarize, the proxy acts as a middleman for client-server interactions. A client makes a request to the proxy, which then processes that request on its behalf. Once the proxy receives a reply from the destination server, it passes the reply back to the client. The proxy can also implement an authentication scheme, and thus act as a firewall as well. Under such a scheme, if an unauthorized client tries to make a request, it will be denied access by the proxy itself, and the request never has to reach the server. In the following
section, the technical details of the proxy are examined further in the context of the Java programming language.

When the proxy is first started, it opens a server socket on the local machine. This server socket listens on some port for client requests (Figure 3-1(a)). The client must know ahead of time on which port the server socket is listening for requests, so that it can send its requests to the appropriate port. Once a request is detected, and the server socket accepts it, it creates another socket that connects the proxy machine with the client machine (Figure 3-1(b)). While this new socket connection is handling the communication between the client and proxy, the server socket continues listening for other requests from clients. Each time a new request is accepted, a new socket is created in the manner just described. Obviously, this proxy is multi-threaded in nature; a new thread handles each new request.

![Schematics of server socket operation.](image)

**Figure 3 - 1 – Schematics of server socket operation.**
Each thread is responsible for handling one request. First, the thread must parse the startline of the request and determine the server from which it is requesting data. If the proxy is configured to act as a firewall, then the authentication is done at this stage as well. The proxy makes sure that the client is allowed access to the server before continuing the processing of the request. The authentication information may be on the same machine as the proxy, or it may be on a separate authentication server. Regardless, the proxy must make sure the client is granted permission before continuing. If the client is denied access, it is notified without any further processing, and the connection is closed.

Assuming the client does have access to the server, the thread then opens a second socket connection; this one connects the proxy machine with the server machine. The proxy passes the client’s request on to the server through this second socket. The server uses this same connection to send its reply back to the proxy. Once the proxy has received the server’s reply, it sends this reply back to the client via the first socket connection. After the reply has been sent to the client, the thread closes both of the sockets. The request has been processed and the thread’s job is done. One can recognize the inefficiency that arises when multiple requests occur from the client to the server. For each request, two socket connections are opened, used, and closed. Clearly, it would be advantageous to leave these sockets open until they are no longer needed.

3.2 “End-to-end” Proxy

Using the standard proxy server described in section 3.1 as a framework, we can extend the design to provide end-to-end connectivity through firewalls. Conceptually, the
behavior of the proxy is the same; it is merely processing requests on behalf of a client. However, the details of the implementation are different. These details are discussed in this section.

It is instructive to refer back to the networking model described in Chapter 2 to better understand the design of this enhanced proxy. The sockets described in the previous section correspond to the transport layer in the Internet architecture, specifically TCP. The proxy also works at the application level (HTTP, in this case) while processing requests. In terms of Figure 2.1, when a request arrives at the proxy from the client, it goes all the way up the protocol stack, before going all the way down it on the way to the server. Obviously, if we want to have an end-to-end connection, we want the request to go all the way through to the server without being dismantled at the proxy.

Technically speaking, it is not possible to have a strict end-to-end connection across a firewall because the firewall by definition breaks the connection. However, we can design the proxy in such a manner as to appear to have the desired connectivity at the HTTP layer, even though the underlying TCP implementation is still disjoint. As mentioned in Chapter 2, we will exploit the persistent connection feature of HTTP 1.1 to achieve this apparent end-to-end connectivity.

The proxy starts by opening up a server socket and waiting for requests. When a request is received, a new socket connecting the proxy to the client is opened, and control of this connection is given to a thread (handling thread). This handling thread looks at the request, determines the server to which the request is directed, and authenticates the client. So far, all of this has been exactly the same as the generic proxy, but with the next step the two implementations start to diverge.
Now, assuming the client is authenticated, the thread needs to provide a connection between the proxy and the server. Note that this connection does not need to be controlled by a separate thread, thus making the implementation easier. The handling thread consults a table of connections, and if it finds an existing persistent connection, it uses it. If not, the thread creates a new connection and inserts it into the table so that future requests can use it. The request is now sent from the proxy to the server over this connection. Once the server's reply is received by the proxy, it passes it back to the client. Neither the client-to-proxy nor the proxy-to-server connection is closed unless an explicit "Connection: close" token is received in either the request or reply.

There are now two active connections: one controlling the client to proxy connection (client connection) and one controlling the proxy to server connection (server connection). Since the client connection does not close after processing the first request, if multiple requests are made, the proxy just passes each one on to the server connection. If one of the subsequent requests were directed towards a different server, then the proxy would open a socket to this second server and handle it. The first server connection would remain open and be available for future requests. If a connection remains idle for some predefined amount of time, it will be closed and removed from the table of connections. A "manager" thread performs this and other cleanup services.

One can see now that if multiple requests are made to the same server, there is a greater level of efficiency because just one client-side and one server-side connection are opened; all the requests use these two connections. Another benefit of this scheme occurs when the server socket accepts a request from a second client. Suppose this client's request is directed to the server that is currently connected to by the server
connection. There is no need to open a new connection to this server; the server can thus process requests for multiple clients.

An alternative design that was considered was one in which the client connection and server connection would each be controlled by separate threads. In this way, the two connections would be truly "modular." The design that was ultimately decided upon was chosen for two reasons. First, implementation of the second design is more difficult because of the increased complexity in thread management. Second, the overhead involved in continually switching back and forth from the client thread to the server thread might actually lead to a degradation in performance.

In terms of the layering model for the Internet, we see a different picture than what we saw for the generic proxy server. Before, each request was essentially sent all the way up the proxy protocol stack, dismantled, reassembled, and sent back down the protocol stack before finally reaching the server. Now, we can view the request as going up the proxy's protocol stack to the transport (TCP) layer, before going back down the stack. In this sense, at the application (HTTP) level, we have managed to establish an end-to-end connection!

3.3 Code Design

In the following sections, the important classes developed for the proxy will be examined. The main purpose of each class will be discussed and where necessary, examples will be shown. For in-depth code specifications, see the Appendix.
3.3.1 Class Main

The Main class performs four major tasks. Most obviously, it starts the system by invoking the `main()` method. This method sets the port on which the proxy will listen for requests and calls the constructor for this class. The constructor method performs the remaining three tasks. First, it creates the graphical user interface. Second, it starts the Manager thread. Finally, it creates a new ProxyServer object and calls its run method.

3.3.2 Class Manager

The Manager class implements a thread object which "cleans up" after the rest of the system. Every thirty seconds, it gains control of the processor. While it has control, the manager checks the table of proxy-to-server connections for connections that have been idle too long, and closes them. This is done by calling the `clean()` method of Proxy. The manager also checks to see if there are any idling threads which need to be closed by calling the `clean()` method of ProxyServer. Since the proxy creates a new thread for each new client, this is effectively the same as closing idle client-to-proxy connections.

3.3.3 Class Connection

The Connection class is the basis for all connections in this system, whether they are from the client to the proxy, or from the proxy to the server. Under the TCP model, data is transferred over reliable streams. To this end, a Connection object has three variables: a socket, an input stream, and an output stream. This class also provides all the necessary functions to open and close connections.
3.3.4 Class Client

The Client class lays the framework for the connection between the client and the proxy. It inherits off of the Connection class. This class provides a constructor which sets up the socket and data streams necessary by calling its superclass’ constructor. It also provides read and write methods which transfer data on its input and output streams, respectively. Each time a request is received by the proxy server’s ServerSocket, this class is instantiated to connect to the requesting client.

3.3.5 Class Proxy

The Proxy class is the basis for all proxy-to-server connections. It also inherits off of the Connection class. In addition to providing a constructor which initializes the required socket and data streams, this class also establishes a hashtable of persistent connections. The first time a Proxy object is instantiated, a static table associated with this class is created. All future objects can use this table to insert or remove connections as needed. This class provides all the necessary methods to manipulate this hashtable, as well as methods to read and write requests and replies, and open and close connections.

3.3.6 Class ProxyServer

The ProxyServer class is the one that is responsible for listening on some pre-specified port on the proxy host for incoming requests. Its constructor creates a new ServerSocket (a class provided by the Java networking package) to do this, and also creates a new ThreadGroup which will contain all the threads that the proxy may start in the future. The ServerSocket blocks until a request is received, at which point it returns a socket that
the client is connected to. This new socket is then passed on to a new RequestHandler object which handles the request.

3.3.7 Class Message

The Message class is an abstract class (i.e. it cannot be directly instantiated) which is the basis for all requests and replies. The Request class and Reply class both are subclasses of Message. Message provides a readLine() method which reads one line of data from an input stream. It has a readHeaders() method that parses data from the input stream into header fields and tokens, and stores this data in a hashtable. This class also contains several methods to manipulate the table of headers.

3.3.8 Class Request

The Request class provides the functionality needed for HTTP request messages; it inherits from the Message class. A Request object knows which client it comes from, as well as the URL it is trying to access. It has a read() method that reads the first line from the input stream, which, if it is a valid request, will be of the form "Method SP Request-URI SP HTTP-Version CRLF." Read() then parses and stores the method, URI, and version. Finally, it reads in all the header fields and tokens by calling Message's readHeaders() method. Request also provides a method to write to the output stream, as well as several methods to assist in processing the request.
3.3.9 Class Reply

The Reply class provides the necessary functions for HTTP response messages, and it also inherits from the Message class. It has a `read()` method that reads the first line from the input stream, which, if it is a valid reply, will be of the form "HTTP-Version SP Status-Code SP Reason-Phrase CRLF." It then reads and stores all of the reply headers and tokens by calling the `readHeaders()` method of its superclass. Response also has a method to write to the output stream, as well as helper methods to aid in the processing of replies.

3.3.10 Class RequestHandler

The RequestHandler class is the heart of the proxy server. It implements a thread which is responsible for handling all requests from a client. When it is started, the first thing it does is creates a new Client object that connects to the requesting machine. Then it enters a loop in which it reads the request from the client. This request is processed by an internal method called `handleRequest()`. `handleRequest()` first determines whether the client wants a persistent connection and creates a proxy-to-server connection accordingly. Then, it sends the request to the server, reads the response, and writes the response to the client. If the server for some reason does not maintain the persistent connection, then `handleRequest()` will close the proxy-to-server connection. Finally, it returns to the main loop with a boolean value indicating whether persistency is or is not in effect. The loop goes back to the beginning and repeats so long as persistency is true. In this manner, the client can send multiple requests without having to open a new connection to
the proxy each time. ThreadHandler also provides methods to close client-side and server-side connections, as well as to copy data from an input stream to an output stream.
Chapter 4

Performance

In order to determine the effectiveness of this system, it was necessary to run some tests. These tests attempted to measure the performance of the new "end-to-end" proxy in relation to a standard proxy server. This chapter will discuss some of the issues involved in doing these tests, the setup and execution of the tests, and the data that resulted from them. Finally, there will be a discussion of some of the limitations of the testing.

4.1 Performance Metrics

There are two commonly used metrics for network performance, bandwidth and latency. Bandwidth can be thought of as the throughput of the link, or the number of bits of data that can travel across the link per second. In practice, throughput is often used to indicate the actual number of bits per second that are transmitted over the link, and is usually less than the theoretical bandwidth. This measurement is a characteristic of the physical network and not of the clients and servers, so we will not be explicitly measuring bandwidth as a performance indicator.

The second kind of measurement is known as latency, or delay. It tells you how long it takes for data to travel from one end of the link to the other, and is measured in units of time. Latency is often thought of being composed of three elements. First, there
is a propagation delay which is constrained by the fact that electrons can not travel faster than the speed of light. The speed of light has been measured to be $2.3 \times 10^8$ meters per second in a cable and $2.0 \times 10^8$ meters per second in a fiber. Second, there is a delay associated with transmitting the packets through the network. This depends not only on network bandwidth (or throughput), but also on the number of packets transmitted. Implicit in this delay is the cost arising from the opening and closing of TCP connections. Finally, there is a delay that results from the queuing of packets in the network switches. Obviously, by using fewer TCP connections, the area which we are trying to impact is the transmittance delay. To this end, we will measure the total latency in the retrieval of Web pages.

4.2 Test Setup and Execution

It is very important to try and achieve as much uniformity as possible when running tests over a network. Due to continuously evolving states of traffic and congestion, it is almost impossible to perform tests under identical, reproducible network conditions, short of directly linking the client with the server. Three different scenarios were tested, so that the effects of delays due to transmittance could be monitored under different settings. In all three cases, the client and the proxy were run off of the same machine so client-to-proxy delays were minimized, and proxy-to-server delays made the greatest contribution to latency. The client and proxy were both run on a Pentium Pro 200MHz machine with 64MB of RAM and Windows NT as its operating system. Note that in all cases, caching was turned off so that the client did not merely fetch a locally cached copy of the requested document.
4.2.1 Local Server

The first test that was performed was one in which the machine serving the web pages was located within the same building, approximately 50 meters away. Using an approximation of $2.0 \times 10^8$ meters per second as the speed of light, the propagation delay is 0.00025 ms in one direction, or 0.00050 ms for a round trip. Obviously, since the server was physically so close to the client, the propagation delay had a negligible effect on latency. The web page that was retrieved was one which contained 49 inline images, so a total of 50 HTTP requests were made.

Using the Java system function, `System.currentTimeMillis()`, the time at which the first request was made was recorded. The same function was called once the last reply was written back to the client to record the ending time. Table 4-1 shows the results of ten trial runs, normalized such that the start time equals 0 ms. These tests were done at approximately 10pm on a Friday night, when traffic on the local network was almost non-existent.

<table>
<thead>
<tr>
<th>Standard Proxy</th>
<th>Start Time (ms)</th>
<th>End Time (ms)</th>
<th>Average (ms)</th>
<th>End-to-End Proxy</th>
<th>Start Time (ms)</th>
<th>End Time (ms)</th>
<th>Average (ms)</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>5568</td>
<td>6028</td>
<td>0</td>
<td>3074</td>
<td>3242</td>
<td></td>
<td>46.21%</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5848</td>
<td>0</td>
<td>0</td>
<td>4026</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5668</td>
<td>0</td>
<td>0</td>
<td>3015</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5538</td>
<td>0</td>
<td>0</td>
<td>2915</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5728</td>
<td>0</td>
<td>0</td>
<td>2934</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>6810</td>
<td>0</td>
<td>0</td>
<td>4016</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5868</td>
<td>0</td>
<td>0</td>
<td>3024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5878</td>
<td>0</td>
<td>0</td>
<td>3215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5828</td>
<td>0</td>
<td>0</td>
<td>3175</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>7541</td>
<td>0</td>
<td>0</td>
<td>3025</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-1 – Results of Latency Measurements for a Local Server
4.2.2 Remote Server

The second test that was performed was one in which the serving machine was Sun Microsystems’s Java web site at java.sun.com, located in California, about 3000 miles away. The speed of light in miles per second is approximated at 125,000. This means that the latency due to propagation is at least 24 ms one way, or 48 ms round trip. The Java homepage contains 35 images, so a total of 36 HTTP requests are necessary. Table 4-2 shows the results of this trial run. These tests were performed at about 2am on a Saturday morning (corresponding to 11pm Friday night in California), when Internet traffic is relatively low.

<table>
<thead>
<tr>
<th>Start Time (ms)</th>
<th>End Time (ms)</th>
<th>Average (ms)</th>
<th>Start Time (ms)</th>
<th>End Time (ms)</th>
<th>Average (ms)</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5528</td>
<td>5815</td>
<td>0</td>
<td>2864</td>
<td>3573</td>
<td>38.56%</td>
</tr>
<tr>
<td>0</td>
<td>5097</td>
<td>0</td>
<td>0</td>
<td>3856</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5047</td>
<td>0</td>
<td>0</td>
<td>3415</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6119</td>
<td>0</td>
<td>0</td>
<td>3575</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4867</td>
<td>0</td>
<td>0</td>
<td>3615</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6950</td>
<td>0</td>
<td>0</td>
<td>3715</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6820</td>
<td>0</td>
<td>0</td>
<td>3706</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6029</td>
<td>0</td>
<td>0</td>
<td>3495</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6139</td>
<td>0</td>
<td>0</td>
<td>3585</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5558</td>
<td>0</td>
<td>0</td>
<td>3906</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-2 – Results of Latency Measurements for a Remote Server

4.2.3 International Server

The third test that was performed was one in which the homepage for France’s National Institute for Research in Computer Science and Control (INRIA) at www.inria.fr was requested. The physical distance between the client and server in this case is close to 4000 miles. This corresponds to a one way propagation delay of at least 32 ms, or 64 ms round trip. The INRIA homepage has 10 inline images, so there are a total of 11 HTTP
requests necessary to fetch this page. Table 4-3 shows the results of this trial. These trials were performed at 10am on a Saturday morning (or mid-afternoon in France) when Internet congestion is minimal.

<table>
<thead>
<tr>
<th>Start Time (ms)</th>
<th>End Time (ms)</th>
<th>Average (ms)</th>
<th>Start Time (ms)</th>
<th>End Time (ms)</th>
<th>Average (ms)</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4717</td>
<td>4709</td>
<td>0</td>
<td>2994</td>
<td>2993</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4777</td>
<td>4709</td>
<td>0</td>
<td>2704</td>
<td>2703</td>
<td>46.21%</td>
</tr>
<tr>
<td>0</td>
<td>4987</td>
<td>4709</td>
<td>0</td>
<td>3926</td>
<td>3925</td>
<td>38.56%</td>
</tr>
<tr>
<td>0</td>
<td>4537</td>
<td>4709</td>
<td>0</td>
<td>2764</td>
<td>2763</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4576</td>
<td>4709</td>
<td>0</td>
<td>2844</td>
<td>2843</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4847</td>
<td>4709</td>
<td>0</td>
<td>2574</td>
<td>2573</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4706</td>
<td>4709</td>
<td>0</td>
<td>4096</td>
<td>4095</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4527</td>
<td>4709</td>
<td>0</td>
<td>2764</td>
<td>2763</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4918</td>
<td>4709</td>
<td>0</td>
<td>2574</td>
<td>2573</td>
<td>36.43%</td>
</tr>
<tr>
<td>0</td>
<td>4497</td>
<td>4709</td>
<td>0</td>
<td>2694</td>
<td>2693</td>
<td>36.43%</td>
</tr>
</tbody>
</table>

Table 4-3 – Results of Latency Measurements for an International Server

4.3 Discussion of Results

It is clear from the preceding tables that the end-to-end proxy server with persistent connections achieves superior performance than a standard proxy server. In the case of the local server, a 46.21% improvement in performance occurred. For the remote server in California, a 38.56% improvement was observed, and for the remote server in France, a 36.43% improvement was observed.

Since the local server is subject to far less traffic than the two remote servers, it is expected that its performance should be somewhat better. The results show this to be the case. In practice, we can expect performance improvement in the range of what was observed for the remote servers because these measurements were taken along routes that traversed the Internet, rather than just a local area network. Also, we should expect
improvement to be lower than the 36-38% range during normal business hours since the Internet is subject to the most traffic in this time.

The biggest problem that was run into throughout the testing is that most of the world is not yet using the HTTP 1.1 protocol. This problem arose on both the client side and the server side. However, this was to be expected as HTTP 1.1 is still in the development stages and has not yet reached a stable state. In the coming months and years, as the protocol becomes more standardized, one can expect both clients and servers to become more compliant with it.

On the client side, there were two main problems. One was that the browser (Netscape Communicator 4.04) would open four connections to the proxy when requesting a page. The rationale for this is to have multiple connections working in parallel rather than opening and closing one connection several times. This is a good idea under HTTP 1.0, but with the new protocol a single persistent connection is preferred. As shown in the study by Nielsen et al., four non-persistent connections working in parallel should outperform a single persistent one. However, since both the standard proxy and the end-to-end proxy were using four connections with the client, there was no advantage gained by having the parallel connections. Most of the gain in performance can thus be attributed to server-side behavior. With the standard proxy, the connection from the proxy to server was not a persistent one, whereas with the end-to-end proxy it was. This explains why the end-to-end proxy outperformed the standard one.

The second problem is a consequence of the first one. Under HTTP 1.1, multiple requests should be pipelined; in this way, the client does not have to wait for a response before sending its next request. With a persistent connection, these requests can be sent
one directly after the other on the same link, but instead the requests were distributed over the four parallel connections.

There were also a couple of problems on the server side. Not too many servers on the Web are HTTP 1.1 compliant, so the first hurdle to clear was to find such servers. Once an HTTP 1.1 server was found, another problem cropped up. Under the current specifications for HTTP 1.1, there is no rule for how long a persistent connection should remain open when idle, so each server can handle this in its own manner. It was often noticed that persistent connections would time out on the server side within a few seconds after processing the last request. This meant that if the same web server was sent a request just moments afterwards, an entirely new connection had to be opened. Hopefully, these problems will be easier to deal with in the coming months, as more HTTP 1.1 compliant systems are introduced.
Chapter 5

Conclusion

In this chapter, I will provide a brief summary of the work presented in this thesis. Also, I will propose some areas where further work should be done before a proxy-based firewall with end-to-end connectivity can be fully implemented.

5.1 Summary

First, we discussed the derivation of this project from Condell’s thesis. He developed a security model for the Information Mesh project which comprised several servers and thus, used many connections. It was noted that end-to-end connectivity across these servers would be best.

Next, we discussed the layering model used by the Internet. We saw that HTTP runs on top of TCP, which in turn runs on top of IP. Also, we gave a synopsis of the three most commonly used kinds of firewalls. We saw that ideally, it is desirable to have a firewall which provides the security of a gateway, while providing an end-to-end connection. For this reason, we chose to extend an HTTP proxy to provide this end-to-end connectivity.

We also talked about the details of HTTP, and the changes between versions 1.0 and 1.1 of the protocol. We saw that persistent connections offer a chance to develop the
desired system. Also, we saw from previous work that persistent connections do offer better performance than non-persistent connections.

Next, we looked at the design of the end-to-end proxy. The design detailed how we can use persistent connections to achieve the end-to-end connectivity that is desired. Finally, we examined the results of performance tests using the new system. We saw that the end-to-end proxy performed significantly better than the standard proxy.

5.2 Further Research

There are three areas where further work should be done before a firewall using the end-to-end proxy can be used. They involve client-side persistent connections, pipelining, and authentication.

One of the problems we saw in Chapter 4 was that the client (Netscape’s Communicator) used four connections to the proxy instead of a single persistent one. Further tests should be run when future browsers support HTTP 1.1, or a client could be developed to provide the desired functionality. One possibility could be to modify the Libwww API put out by the World Wide Web Consortium (W3C). Developing a client can also help provide the means to fully take advantage of the benefits of pipelining, as discussed in Chapter 4.

The third area for further work is in authentication. In this project, we were not concerned with the type of authentication used by the proxy. Before any useful firewall can be built, however, research should be done into what the best kind of authentication is to use with the proxy, both in terms of security as well as efficiency. Also, the effects, if any, of the authentication on the speed of the proxy should be quantified.
5.3 Concluding Remarks

The end-to-end proxy described in this paper has much potential to be used in efficient implementations of firewalls. Its use of persistent connections eliminates much of the wasted time spent on opening and closing TCP connections. Once better client-side software is developed and authentication issues are resolved, a fully functional firewall can be built. This firewall will offer a high-level of security in addition to the high level of speed that is required for Internet applications today.
Appendix A

Code Specifications

A.1 Class Main
Starts the proxy. Sets the listen port and creates GUI.

Main()
Creates a proxy and runs it

gui()
Creates the GUI

actionPerformed(ActionEvent)
Handles events from GUI

main(String[])
Reads command line arguments; creates a Main object

A.2 Class Manager
Thread which cleans up after other threads

run()
Thread sleeps for 30 seconds and cleans up after other threads when it wakes up.

A.3 Class Connection
Provides socket and streams necessary for TCP connections

close()
Closes the connection

getInputStream()
Returns the input stream of the connection
getOutputStream()

Returns the output stream of the connection

A.4 Class Client
Provides implementation for client-to-proxy connections

Client(Socket)

Constructor - takes a socket as argument

read()

Reads from input stream and returns a request

write(Reply)

Writes a reply to output stream

A.5 Class Proxy
Provides implementation for proxy-to-server connection. Keeps track of persistent connections.

Proxy(String, int)

Constructor - takes hostname and port as arguments

clean()

Removes stale connections from table of persistent connections

close()

Closes the connection

open(String, int)

If connection to host, port already exists, returns it.

readReply(Request)

Reads reply from the server

writeRequest(Request)

Writes the request to the server
A.6 Class ProxyServer
Creates server socket which listens for incoming client requests

ProxyServer(int)
   Constructor - creates a server socket listening on port

clean()
   Closes stale client connections

run()
   Blocks until request received.

A.7 Class Message
Provides framework for HTTP messages

containsHeaderField(String)
   Returns true if message contains header field

getHeaderField(String)
   Returns token for header field

getHeaderValueCount(String)
   Returns number of tokens for header field

getStartline()
   Returns the start-line of an HTTP message

readHeaders(InputStream)
   Reads the header fields from an HTTP message

readLine(InputStream)
   Reads a line from the input stream

removeHeaderField(String)
   Removes header field

setHeaderField(String, String)
   Stores header field and its token
to<br>String()<br>Returns message in string format

A.8 Class Request
Implements an HTTP request

Request(Client)<br>Constructor - takes client as argument

getCommand()<br>Returns method of request

getHost()<br>Returns destination host of request

getPath()<br>Returns path of request URL

getPort()<br>Returns port number of destination of request

getProtocol()<br>Returns HTTP version of request

getRequest()<br>Returns start-line of request

getUrl()<br>Returns URL of request

read(InputStream)<br>Reads request from input stream

toString()<br>Returns request in string format

write(OutputStream)<br>Writes request to output stream
A.9 Class Reply
Implements an HTTP response

**Reply(InputStream)**
   Constructor - creates a reply that comes from input stream

**getChunkedFooter(InputStream)**
   Removes junk at the end of a chunk

**getChunkSize(InputStream)**
   Returns the chunk size of reply

**getInputStream()**
   Returns input stream that reply comes from

**getProtocol()**
   Returns HTTP version of reply

**getStatusCode()**
   Returns status code of reply

**hasContent()**
   Returns true if the reply has content

**read()**
   Reads the reply from input stream

**read(InputStream)**
   Reads reply from specified input stream

**write(OutputStream)**
   Writes reply to out

A.10 Class RequestHandler
Thread which processes requests

**RequestHandler(ThreadGroup, Runnable)**
   Constructor - creates thread and adds it to threadgroup

**close()**
Closes client-to-proxy and proxy-to-server connections

copy(InputStream, OutputStream, int)
Copies data from input stream to output stream

createProxy()
Creates proxy-to-server connection

go(Socket)
Starts the thread

handleRequest()
Processes a request

run()
Connects to client; reads request; processes request; loops while persistent
References


