Reverse HTTP Tunneling for Firewall Traversal

by

Eric N. Sit

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

© Eric N. Sit, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis and to
grant others the right to do so.

Author ...........................................................
Department of Electrical Engineering and Computer Science
February 2, 2000

Certified by ..........................................................
Hari Balakrishnan
Assistant Professor
Thesis Supervisor

Accepted by ....................................................
Arthur C. Smith
Chairman, Department Committee on Graduate Students
Reverse HTTP Tunneling for Firewall Traversal

by

Eric N. Sit

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes a reverse hypertext tunneling protocol that allows a web browser from outside a firewall to access a web server inside a firewall.

Reverse HTTP utilizes pre-existing proxy services to create a transparent and scalable method to tunnel through a firewall. Reverse HTTP provides an excellent solution to the problem of providing controlled access to firewall protected resources. This is because unlike traditional firewall traversal methods, reverse HTTP is completely software based and requires no changes to the existing firewall configuration. Reverse HTTP works because it tunnels traffic in a form which a web proxy server on the firewall allows to pass.

A Java software implementation has been developed and tested to demonstrate the usefulness and efficacy of the reverse HTTP protocol. This experimental implementation proves that a portable, scalable solution can be developed with comparable performance characteristics to a normal web proxy. Although the implementation can be used to view protected web sites, it does not compromise the overall network security.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor
Acknowledgments

Without the assistance and support of many people, this research would not have been possible.

I'd like to thank my thesis supervisor, Hari Balakrishnan, for his invaluable guidance and support. His ideas, technical assistance, inspiration, motivation, and kindness have seen this project through from beginning to end.

James Clough and Dean Nelson, both of Hewlett-Packard, have been invaluable in providing technical assistance for the mass of code. Their willingness to answer questions and assist in debugging my code was enormously helpful.

My manager, Alan Raney, guided my stay at HP and made it an enjoyable place to work.

Finally, I'd like to thank my family for their love and patience. They always took the initiative to keep in contact with me to make sure that I was alright. Their support will always be extremely important to me.
## Contents

1 Introduction 8
   1.1 The problem of firewall traversal 8
   1.2 Previous solutions 9
      1.2.1 Poking holes in firewalls 9
      1.2.2 Proxies on the firewall 9
      1.2.3 Virtual private networking 10
   1.3 Reverse HTTP 10
   1.4 Outline 11

2 Design of reverse HTTP protocol 13
   2.1 Design criteria 13
   2.2 Conceptual design 14
   2.3 Design Issues 15
      2.3.1 Why bother encoding and decoding? 15

3 Design of Agent-Proxy Architecture of reverse HTTP protocol 17
   3.1 Design Criteria 17
      3.1.1 Transparency 17
      3.1.2 Scalability 18
      3.1.3 Operability 18
   3.2 Agent-Proxy architecture 19
      3.2.1 Agent 19
      3.2.2 Proxy 20
3.3 Reverse HTTP protocol session example .......................... 22
  3.3.1 Control initialization ........................................ 22
  3.3.2 Client initialization ......................................... 24
  3.3.3 Control request ............................................. 25
  3.3.4 Resource retrieval .......................................... 26
  3.3.5 Data transmission ........................................... 26
  3.3.6 Control termination ......................................... 27

3.4 Design issues .................................................... 28
  3.4.1 Separation of control and data connections ............... 28
  3.4.2 Error handling ............................................. 29

4 Implementation ..................................................... 31
  4.1 Implementation components .................................... 31
    4.1.1 Agent .................................................. 31
    4.1.2 Proxy .................................................. 32
    4.1.3 Encoding and decoding of requests and responses ........ 32
    4.1.4 Peripheral classes ...................................... 33
  4.2 Implementation language ..................................... 33
  4.3 Evaluation .................................................. 33
    4.3.1 Environment ........................................... 33
    4.3.2 Methodology ............................................ 34

5 Applications and security ........................................ 37
  5.1 Applications ................................................ 37
    5.1.1 Publishing data to specific outsiders .................. 37
    5.1.2 Demonstrations to customers ............................. 38
    5.1.3 Providing customer support .............................. 38
  5.2 Security ................................................ 39
    5.2.1 Auditing and controlling access ......................... 39
    5.2.2 Encryption ............................................. 39
    5.2.3 Authentication ......................................... 40
5.2.4 Security loopholes ............................................. 40

5.3 Extensions ......................................................... 40
5.3.1 Double firewalls ............................................... 40
5.3.2 Other protocols ............................................... 41
5.3.3 Access lists ................................................... 41

6 Conclusions .......................................................... 42

A Source Code .......................................................... 43
A.1 RHA.java ........................................................... 43
A.2 RHP.java ........................................................... 51
A.3 RHPWeb.java ....................................................... 53
A.4 Proxy.java .......................................................... 56
A.5 ProxyFile.java ....................................................... 64
A.6 HTTP.java ........................................................... 76
A.7 HttpInputStream.java ............................................. 80
A.8 HttpOutputStream.java ......................................... 85
A.9 HttpProcessor.java ............................................... 87
A.10 HttpFile.java ...................................................... 87
A.11 RhttpClose.java .................................................. 89
A.12 RhttpControl.java ................................................ 91
A.13 RhttpData.java ................................................... 93
A.14 RhttpPac.java ..................................................... 96
A.15 RhttpServers.java ............................................... 99
A.16 SocketList.java ................................................... 101
A.17 HttpException.java .............................................. 102
A.18 RedirectException.java ........................................ 103
# List of Figures

1-1 The problem of firewall traversal ............................................. 11
2-1 Firewall with web proxy .......................................................... 14
2-2 Encoding and decoding requests and responses ............................ 15
3-1 Agent-Proxy architecture ......................................................... 19
3-2 Control connection ................................................................. 20
3-3 Data connection ................................................................. 21
3-4 Agent-Proxy architecture ......................................................... 23
4-1 Average roundtrip latencies ..................................................... 35
4-2 Overhead ............................................................................. 36
Chapter 1

Introduction

The reverse hypertext protocol (RHTTP) allows a web browser from outside a firewall to access a web server inside the firewall. This tunneling is done through the web proxy on the firewall.

Chapter one provides background on firewalls and firewall traversal techniques. It then describes the motivation for RHTTP.

Chapter two describes the design of RHTTP.

Chapter three describes the Agent-Proxy architecture which implements RHTTP.

Chapter four describes the implementation of the protocol. It also analyzes the performance of RHTTP.

Chapter five describes the applications for RHTTP, including the security implications on firewall design. New security considerations are examined too.

Chapter six draws some conclusions on RHTTP.

1.1 The problem of firewall traversal

Firewalls have been proven to be essential safeguards for protecting computing resources from unauthorized access from public networks such as the Internet. These firewalls restrict the exchange of information with outside computers. Placed between a local computer system and the Internet, the firewall blocks undesirable incoming requests and information. The term firewall can refer to any hardware, software,
or combination of the two that is used to protect the internal network from intrud-ers. Consequently, a local computer system that is protected by a firewall cannot be unconditionally accessed from a remote location.

Firewalls provide greater security in that they provide a centralized point of con-trol. Here, a network administrator can focus an intranet’s security efforts. The firewall also provides an excellent opportunity to do extensive auditing of traffic.

However, firewalls are usually not flexible enough to allow controlled outside access in a secure and cost-effective manner. It is undesirable to move the resource outside of the firewall and hence subject it to all kinds of intrusion attacks. Also, changing the firewall configuration to poke a hole forwarding IP packets of specific addresses usually raises security concerns, not to mention the added cost of administration.

1.2 Previous solutions

1.2.1 Poking holes in firewalls

If the network is using a "screened host" type of firewall, the firewall can be made to selectively pass requests for port 80 that are bound to or returning from the WWW server machine. This, in effect, pokes a small hole in the dike through which the rest of the world can send and receive requests to the WWW server machine.

This scheme has a number of disadvantages. This solution does not scale very well because the network administrator must personally make direct changes to the firewall. Furthermore, every time a hole is poked, it creates a new security hole. Very soon, the firewall configuration will become so complicated that it will become unmanageable and its security will be difficult to evaluate.

1.2.2 Proxies on the firewall

If the network is using a "dual homed gateway," then a proxy is needed on the firewall machine. The proxy can see both sides of the firewall. Requests for information from the web server are intercepted by the proxy, forwarded to the server machine, and
the response forwarded to the requester.

This is the solution that many corporations use to allow internal users to access outside resources. The most popular proxy on the firewall is the web proxy which allows users to browse external web sites.

Proxies are generally configured to allow connections that originate internally to pass externally. They can be configured to allow external connections, but care must be taken as an improperly configured proxy may compromise the security that a firewall affords. Furthermore, any changes done to the proxy must be done by the firewall administrator.

1.2.3 Virtual private networking

Currently, the most popular solution for commuter employees to access an internal corporate network is the virtual private network (VPN). Originally, VPN's have been developed to securely connect two or more subnets over the Internet. The security gateways of the subnets are linked by a cryptographically secured tunnel via a public network such as the Internet. This technique has been extended to securely connect computers to a corporate network.

VPN's leverage public networking infrastructure, but they have a number of disadvantages. They tend to be expensive to maintain. They require special configuration of the firewall. Also, they often require specialized software to be installed on the client machines. Installation and troubleshooting this software may prove overwhelming for casual non-technical users. As such, a VPN is overkill for a user who only requires occasional outside access to the network.

1.3 Reverse HTTP

In this thesis, we propose the reverse HTTP protocol that solves the deficiencies of current firewall traversal techniques. We present the Agent-Proxy architecture that encapsulates the reverse HTTP protocol and allows for easy installation into an existing firewall setup.
The reverse HTTP system creates a channel that allows multiple external browsers to interact with multiple internal servers transparently and efficiently. Since it is independent of firewall technology, it can be easily installed onto any existing firewall architecture as long as there is a pre-existing web proxy server. Reverse HTTP works because it tunnels traffic in a form which a web proxy server on the firewall allows to pass.

Installation nor usage of this protocol requires the intervention of network administrators. Thus, reverse HTTP provides an excellent solution to the problem of providing access restricted access to resources, especially for the occasional user. Furthermore, installation of the reverse HTTP system is scalable enough to handle all services on an Intranet. See figure 1-1.

![Diagram of firewall traversal](image)

Figure 1-1: The problem of firewall traversal

### 1.4 Outline

This thesis focuses on the design and implementation of the reverse HTTP protocol and the Agent-Proxy architecture, which allow browsers outside a firewall protected network to view specific resources inside.

First, we discuss our design criteria for RHTTP (Chapter 2). Then we present the conceptual design of the RHTTP. We then provide the design criteria for the Agent-
Proxy architecture which implements the RHTTP (Chapter 3). Next we explain the structure of our implementation of the RHTTP and the Agent-Proxy architecture (Chapter 4). To evaluate the performance of our implementation, we discuss some experiments and their results. Finally we describe some applications for the protocol and system, and we also describe some security implications (Chapter 5). We then conclude with a summary of our work (Chapter 6).
Chapter 2

Design of reverse HTTP protocol

We designed the Reverse HTTP Tunneling Protocol in order to provide a simple mechanism for a web browser to view web sites protected by a firewall. It allows the transmission of web traffic requests and responses in a form that the firewall and proxy will allow to pass.

In Section 2.1, we present our design criteria for the reverse HTTP Protocol. In Section 2.2, the conceptual design of RHTTP will be described. In Section 2.3, we talk about some of the design issues of RHTTP.

2.1 Design criteria

RHTTP should allow a web browser to browse an internal web server. The user should feel as if he or she were actually inside the firewall.

RHTTP should be compatible with almost any existing firewall implementation. The usage of the protocol should be relatively transparent to both browsers outside of the firewall and servers inside the firewall. They should be minimally aware of the presence of the RHTTP so that it can be used without necessitating changes to software and hardware of the existing environment.
2.2 Conceptual design

If the firewall was configured such that absolutely no traffic could pass between the internal network and external network, then a tunneling protocol would be impossible to design. However, many organizations install a web proxy server in order to allow internal browsers to access outside Internet resources.

Web proxy servers are generally configured to allow internal browsers to make HTTP requests to the outside. They also allow external servers on the outside to send responses to the internal network as long as they are on the same socket as the original request. They reject all other external requests. See figure 2-1.

![Firewall with web proxy](image)

**Figure 2-1: Firewall with web proxy**

In the design of the reverse HTTP protocol, we take advantage of the existence of this web proxy server on the firewall. In essence, we will attempt to funnel all the necessary traffic in our protocol by disguising it such that it looks like legitimate HTTP requests and responses.

Suppose an entity inside the firewall opens a normal HTTP request to an external server. There is now a socket open between the internal and external networks.

Now assume that there is an external browser which wishes to access an internal web site. Suppose it could send its HTTP requests through the existing open socket. These browser requests would be disguised to look like responses to the original HTTP request. With the disguised browser request safely tunneled through the firewall, it can be extracted as a request on the desired server.

To send back the response to the executed request, another HTTP request connection can be opened from inside to outside, this time with the data embedded in
it. Some entity on the outside would need to decode this HTTP request, extract the response data, and send it to the browser. Thus, the external browser has successfully viewed an internal web resource.

In summary, HTTP requests are disguised to look like HTTP responses, and HTTP responses are disguised to look like HTTP requests. This is done in order to trick the web proxy server on the firewall into allowing web traffic to pass. See figure 2-2.

![Diagram of encoding and decoding requests and responses](image)

**Figure 2-2: Encoding and decoding requests and responses**

### 2.3 Design Issues

#### 2.3.1 Why bother encoding and decoding?

From this design, the question arises whether it is really necessary to go through all the hassle of encoding requests to look like responses and responses to look like requests. Why not just open a socket internally to the outside, and then pipe all the raw traffic through the socket?

To answer this question, one needs to ask another question, "What exactly does the web proxy do?" It turns out that corporations like to use firewalls and web proxies
for two reasons. The first is to provide a centralized point of security where one can focus all their security efforts.

The other, however, is to monitor and log all traffic going in both directions. In order for that to happen, the web proxy expects the requests and responses to be in a certain format. That expected format is the standard HTTP request and response [3]. In particular, proxies like to log URLs and server error response codes.

If the data is not in a format where this information can be easily extracted, then the web proxy will raise a security flag and will not allow the traffic to pass. It is for this reason that we need to do the conversions from requests to responses and vice versa.
Chapter 3

Design of Agent-Proxy

Architecture of reverse HTTP protocol

In this chapter, we present an Agent-Proxy system that implements RHTTP. Section 3.1 describes the design criteria used in the design of the Agent-Proxy architecture. Section 3.2 presents the Agent-Proxy architecture and describes each of its components. Section 3.3 traces through a sample session using the Agent-Proxy architecture. Section 3.4 discusses some of the design decisions of the architecture.

3.1 Design Criteria

Our design criteria for the Agent-Proxy architecture can be divided into three categories: Transparency, Scalability, and Operability.

3.1.1 Transparency

1. RHTTP should be completely transparent to the firewall and web proxy. There should be no configuration changes to the firewall or web proxy in order to support RHTTP. The designed system should be able to be easily plugged into
any firewall environment with a web proxy. This allows installation of the system without involving network system administrators.

2. RHTTP should be completely transparent to the web browser. No code should be added to them to support reverse HTTP. Any HTTP 1.0 or better browser [3] should work with the system.

3. RHTTP should be completely transparent to the servers inside the firewall. No code should be added to them to support reverse HTTP. Any HTTP 1.0 or better server [3] should work with the system.

3.1.2 Scalability

1. The implementation should be scalable. This design criterion influenced us to a completely software based solution. No additional hardware installation should be necessary. Compared to hardware, software is also much easier and faster to distribute.

2. Protocol should scale to allow many browsers to access the same servers without additional configuration.

3. The system should have the ability to allow RHTTP to have multiple servers accessed by the same browser.

3.1.3 Operability

1. Entities inside the firewall should be able to give access to an entity outside the firewall.

2. Entities inside the firewall must have the ability to kill a reverse HTTP protocol session at any time. Once the connection is killed, the web browser outside the firewall must not be able to access the internal web server anymore.

3. The system should be multi-threaded for better performance. Requests and responses should be able to be handled as soon as the system receives them.
3.2 Agent-Proxy architecture

In the design of the reverse HTTP system, we attempt to encapsulate the protocol in two objects. The object that interfaces with the web server is the Agent. The object that interfaces with the browser is the Proxy. This modularity minimizes the impact of the protocol on the surrounding system. See figure 3-1.

![Agent-Proxy architecture](image)

**Figure 3-1: Agent-Proxy architecture**

3.2.1 Agent

The Agent lies internally within the protected network and is responsible for interfacing the web server with the firewall. Its first function is to initiate the connection. It does this by making a HTTP request through the firewall to the Proxy outside of the firewall. This connection is responsible for controlling the operation of the protocol, and thus it is called the control connection. This control connection is always kept open until the user closes the connection. See figure 3-2.

The second function of the Agent is to extract encoded browser requests coming in from the open control connection. It takes these browser requests and proxies them along to the appropriate web server. In other words, it makes the requests on behalf of the browser. It is for this reason that this object is called the Agent.

The last function of the Agent is to encode the web server’s responses as requests.
These requests contain the data from the web server. The Agent opens a new *data* connection to the Proxy. This data connection is always closed by the Proxy after the data has been received by it. See figure 3-3.

It is important to realize that from the perspective of both the web server and the firewall, the Agent looks like a generic browser.

### 3.2.2 Proxy

The Proxy lies outside of the protected firewall network and is responsible for interfacing the outside browser with the firewall. It is much more complex than the Agent because the Proxy must maintain the states of each connection. The two types of sockets that the Proxy must remember are control sockets and browser sockets.

The Proxy must remember which Agents have initiated connections with it so that it knows what servers are accessible. It does this by maintaining a relationship between a unique *agentID* and the Agent’s control socket. When the connection is terminated, this relationship disappears.

Similarly, the Proxy must remember which browsers have opened connections with it. This is so that when the data comes back from the server, it knows which...
browser the data should go to. This is done by maintaining a relationship between a unique browserID and the browser’s socket. When the Proxy has finished handling and returned the browser’s request, this relationship disappears.

The Proxy is designed to be accessed by the browser as a generic proxy server. When the Proxy receives a control initialization request from the Agent, it allocates a new port to talk to the Agent. This is the port through which browsers can communicate with the Agent. Thus, for a browser to use this port, it must change its proxy configuration to this new port.

The advantage of having the Proxy act as a generic proxy server is transparency. The user can type the actual URL of the desired web site into the browser. Relative URL’s on retrieved web pages will be properly interpreted. Interfacing the Proxy as a generic web proxy server provides a transparent, well-known, and easily understood interface.

The Proxy receives requests from the outside browser and converts them into responses to be sent over an existing control connection. Because the Proxy has already established a relationship between port to Agent, it knows which Agent to send the request to.
The Proxy also needs to be able to receive the requests containing server data coming in on the data connection. These need to be decoded into normal responses and then sent to the appropriate browser. The proper browser socket is determined by reading the browserID coming in on the data connection and matching it to the appropriate browser socket in the Proxy's internal database.

It is important to realize that from the perspective of the web browser, the Proxy looks like a generic web proxy server. From the perspective of the firewall, the Proxy looks like a web server.

### 3.3 Reverse HTTP protocol session example

A sample reverse HTTP protocol session can be divided into six phases: control initialization, client initialization, control request, resource retrieval, data transmission, and control termination. This is summarized in figure 3-4.

#### 3.3.1 Control initialization

When the Agent wishes to initialize the control connection, it sends to the firewall:

\[
\text{POST } \text{http://RHPhost/control/} \tag{3.1} \\
\text{Agent-ID: agentID} \tag{3.2} \\
<\text{CRLF}> \tag{3.3}
\]

Note that the complete URL needs to be specified in the POST request (3.1). This is a requirement of the HTTP standard for using proxies [3]. Also note that there is no need to specify a content-length for this POST since no data is being sent. It is simply an empty POST.

The Agent-ID header (3.2) simply contains a distinct nonce that will later be used to aid the Proxy in distinguishing among multiple Agents.

The web proxy on the firewall realizes that this is a HTTP request and proxies
Figure 3-4: Agent-Proxy architecture
the connection along to RHPhost. When RHPhost, or the Proxy, receives the HTTP request, it adds this Agent to its internal database. It does this by associating the agentID with the socket of the control connection. The Proxy also allocates a port through which browsers can access the Agent.

Since the transaction is successful, it then returns the following response:

\[ HTTP/1.0 200 \text{ Ok} \quad (3.4) \]

\[ < \text{CRLF} > \quad (3.5) \]

The web proxy on the firewall allows this to pass because it looks like a typical HTTP response.

It is very important to note that the Proxy does NOT close the control connection. In fact, the system will attempt to keep a persistent control connection until a user decides to close the reverse HTTP connection. Besides giving better performance [5], the use of a persistent control connection makes the system more stable. This is described in detail in section 3.4.1

Note how the content length of the response has NOT been defined. This is because the Proxy has no idea how many requests will be sent across the control connection. The HTTP protocol says that in these cases with no content length defined, the end of the response is determined by when the Proxy closes the connection [3].

When the Agent receives this response, it knows that the Proxy has acknowledged and accepted the control connection.

3.3.2 Client initialization

For our implementation, the user can get a list of protected resources available for browsing by requesting the Proxy’s root page at standard HTTP port 80. This is a normal HTTP transaction. The Proxy acts as a normal HTTP server and returns a page listing all the Agents and their corresponding servers that are available to be browsed.
In the listing of available servers, the user is instructed to change the proxy configuration of the browser. The proxy host will always simply be the address of the Proxy. However, the proxy port varies depending on which Agent the user wishes to browse. The reasons for this reconfiguration of proxies was described in 3.2.2.

For convenience, the Proxy will dynamically generate a PAC (Proxy Automatic Configuration) file for each server. This speeds up the chore of proxy configuration by the user by allowing the browser to automatically configure the proxy settings simply by accessing an URL site. Many browsers including Netscape and Internet Explorer support this feature.

3.3.3 Control request

Once the user has picked a server to browse and has appropriately reconfigured the browser's proxy settings, the user can begin to start browsing. By typing an URL or clicking on a link, the browser automatically sends standard HTTP requests to the Proxy at a the designated proxy port.

The Proxy takes the browser request (3.10), inserts a response headers (3.6-3.8), and sends the following through the open control connection:

\[
\text{HTTP/1.0 200 Ok} \\
\text{Browser - ID : browserID} \\
\text{Content - length : #} \\
< CRLF >
\]

\[
\text{GET filename HTTP/1.0} \\
< CRLF >
\]

The browserID (3.7) is a randomly generated nonce that is needed in order to distinguish between different browser requests. The Proxy must be able to associate a browserID with its corresponding browser socket. The browserID will be sent along
to the Agent. It is expected that when the Agent sends back the data, it will also return the same browserID so that the Proxy knows which browser request the data should be sent to.

The Proxy must also calculate a Content-Length (3.8) of the response. The reason for this is because we are not closing the control connection. The Agent, therefore, has no easy way to distinguish between one HTTP response and another. However, if the content length is passed along, then the Agent can simply read the appropriate number of bytes with the assurance that all the data for that response has been read.

Any subsequent browser requests will also be sent along the same control connection. Each will have their own unique browserID. Note that by not closing the control connection, it allows the Proxy to send multiple browser requests to the Agent. If the control connection had been closed, we would have had to wait for another control request before the Proxy could send another browser request.

### 3.3.4 Resource retrieval

Once the Agent receives the response from the Proxy, it extracts the encoded HTTP request. Now it is all a matter of executing the HTTP request to the appropriate server using a normal HTTP transaction.

### 3.3.5 Data transmission

Now that the Agent has retrieved the data (3.19) from the web server, it is ready to send it back to the Agent. First the Agent opens a new connection to the Proxy, called the data connection. The Agent will wrap the HTTP response from the web server (3.17-3.19) and encode it as a HTTP request (3.12-3.15). It sends the following to the Agent:

\[
\text{POST http://RHPhost/data/ HTTP/1.0} \tag{3.12}
\]

\[
\text{Browser-ID : browserID} \tag{3.13}
\]
The browserID (3.13) that is sent is the same browserID that was sent in the Proxy’s original request to the Agent. This is to ensure that the Proxy knows which browser socket should be the one receiving the data.

The agentID (3.14) is sent to aid the Proxy in distinguishing between multiple Agents.

The Content-Length (3.15) is sent for HTTP 1.0 compliance [3]. It lets the Proxy know how long the data portion of the request will be so that the Proxy knows when to close the data connection.

Upon receiving this information, the Proxy will decode the request to extract the data. It will then match the browserID to the appropriate browser socket and pass the data to it.

### 3.3.6 Control termination

Either party can close the connection at any time. To do this, all one needs to do is set their browser to the kill connection link on the server listing page. Doing so sends the following to the Proxy:

\[
\text{GET http://RHP\textit{host}/close/agentID} \\
< \text{CRLF} >
\]

Note how the agentID is passed along in the request (3.20) so that the Proxy
knows which connection to kill.

3.4 Design issues

3.4.1 Separation of control and data connections

In the original design for the Agent-Proxy architecture, we combined the control and data connections onto a single connection. In other words, we sent the data from the web server back on the original control connection.

The implementation did work and fulfilled the transparency criterion. However it suffered from a number of problems, including complexity and lack of robustness.

Combining the control and data connections made the system more complex. If errors never happened in the entire connection, then this was not really an issue. However, when an error occurred, things became very complicated. It would be difficult to determine what went wrong or how to correct it. One was never really sure what state the system was in. Separating the control and data connections provided a much simpler design and made the system more modular and easier to visualize and implement.

The other problem of the original design was that it simply was not robust. Much of this arose from the added complexity as described earlier. We depended too much on the connection always being available. Yet we added additional and unnecessary stress to the system by sending data on the control connection! This created more opportunity for failure of the entire system.

By separating the control and data connections, we modularized the system such that it does not matter as much that the transmission of web data connection is successful. If it fails, the overall connection integrity is still maintained. Error handling is described further in section 3.4.2.
3.4.2 Error handling

One of the major issues in designing network protocols is how to handle errors. Failures can be classified into either acceptable and catastrophic.

Clearly, the most catastrophic errors occur on the control connection. A breakage in the control connection implies that the Proxy would no longer be able to send requests to the Agent.

Many events could cause a connection to break. One example is if a router on the Internet were to break. But the more common situation would be when the connection times out. Sockets cannot be kept open indefinitely. Sockets going through firewall proxies are even more tightly monitored; for obvious security concerns, firewall administrator strictly monitor sockets and their connection times. Though we try in the implementation to keep the socket from ever timing out, there is always a chance of breakage.

The scheme for error handling in this implementation is to keep the control connection open as long as possible. Keeping the connection open not only provides better performance, but allows the Proxy and Agent to communicate keep-alive error messages more efficiently. If the control connection should break down, the Agent will detect this and attempt to reinitialize it by making a new control connection to the Proxy. The Proxy will know that the Agent is attempting to reinitialize an already dead connection because the agentID will be the same as the old one.

How about failures in the data connection? As discussed in section 3.4.1, the separation of the control and data connections allows the system to only depend on the control connection staying available. We have the luxury in this design to say that we do not care that the data was not transmitted successfully. If the data was really important, the user will simply request it from the server again. This scheme of retrying is the main error handling technique of standard HTTP [3].

Note how this solves a lot of data transmission problems. What if a user requests a page with ten graphics? If we had to guarantee the success of each request, then in the original combined control-data connection, if one graphic were to fail, it would
jeopardize the success of the other nine graphics.

Another example of an acceptable error on the data connection includes the situation where the user clicks on a different link on the browser before the first one is completed. By clicking on a new link, the browser closes the old browser socket to the Proxy.

The outcomes of this situation can be divided into two cases. The first is if the connection was closed fast enough such that the Proxy never had a chance to receive the request. This would mean that the Agent will not handle the request, so this is acceptable. But what if the request has already been received by the Proxy and thus sent to the Agent? Then when the Agent sends back the server data to the Proxy, we simply discard it.
Chapter 4

Implementation

In order to demonstrate the effectiveness of the reverse HTTP protocol, we developed a prototype implementation of the reverse HTTP system.

In Section 4.1 we describe the components of our implementation. Section 4.2 explains our choice of Java as an implementation language, and discusses the language features we use. Section 4.3 evaluates the performance of our implementation.

4.1 Implementation components

Our implementation of the Reverse HTTP System is made up of several components. In our Java prototype, these components can be divided into those that implement the Agent and those that implement the Proxy. A number of classes such as the encoders and decoders of requests and responses are shared by the Agent and Proxy. The actual source code can be found in Appendix A.

4.1.1 Agent

The Agent is made up by the class RHA. The class simply starts the connection with initialize() and then loops through two methods: addAgent(), and sendResponse(). AddAgent() will decode the incoming response to extract the browser's request. It will then send the request to the appropriate server. When the server responds with
the data, sendResponse() will send the data back to the Proxy. This continuously loops to handle subsequent requests until someone closes the control connection.

4.1.2 Proxy

The Proxy, being more complicated, is made up by the classes RHP and RHPWeb. These classes give the Proxy the functionality of a typical web server. Different incoming requests are resolved using the interface class HttpProcessor. Each request is then subclassed into one of the following classes: RHttpControl, RhttpData, RhttpClose, RHttpPac, RhttpServers, and HttpFile.

RhttpControl handles requests to initialize or reinitialize (in case of an error) a control connection. RhttpData handles a request from the Agent containing data from a server. RhttpClose handles a request to kill the entire connection. RhttpPac handles a request for a PAC file to automatically configure a browser’s proxy settings. RhttpServers handles a request for the list of available servers. HttpFile handles a request for the Proxy to act as a normal web server and return the appropriate file in normal HTTP.

Connection state in the Proxy is maintained primarily by two hashtables. The first maintains the relationship between every agentID with and its control socket. The other maintains the relationship between every browserID and its browser socket.

4.1.3 Encoding and decoding of requests and responses

The classes Proxy and ProxyFile do most of the real work of the protocol. They are responsible for the encoding and decoding of responses and requests. They are shared by both Agent and Proxy.

Proxy and ProxyFile both require an input socket and an output socket. Both classes will attempt to read bytes coming in on the input socket and write them to the output socket. However, Proxy handles the decoding of requests and responses, while ProxyFile handles the encoding of requests and responses.
4.1.4 Peripheral classes

HTTP, HttpInputStream, HttpOutputStream, HttpException, RedirectException, and SocketList are the useful classes that are shared by both the Agent and Proxy. They subclass existing Java classes to add functionality that facilitates in the implementation of the reverse HTTP protocol.

4.2 Implementation language

We made the decision to use Java\(^1\) as the language for our prototype implementation of the Reverse HTTP System. We chose Java over Perl because of its powerful object oriented abilities. We also chose Java over C++ due to Java’s extensive high-level networking libraries and its ease of implementation. Additionally, Java has excellent cross-platform portability.

For maximum compatibility, all implementation was done using the standard Java 1.1 API.

4.3 Evaluation

In order to evaluate the performance of the Agent-Proxy architecture, we ran some tests to characterize the overhead of the Agent-Proxy architecture.

4.3.1 Environment

The Agent and Proxy were each run on a Windows NT 4.0 machine\(^2\) with dual Intel Pentium III processors running at 450 MHz and 256 Mb RAM.

Squid\(^3\) was installed on a Red Hat Linux 6.0 machine\(^4\) with dual 200 MHz Pentium processors. Squid is a free, open-source, high-performance proxy caching server for

\(^1\)http://java.sun.com/
\(^2\)http://www.microsoft.com/ntworkstation/
\(^3\)http://squid.nlanr.net/
\(^4\)http://www.redhat.com/
web clients, supporting FTP, gopher, and HTTP data objects. In this evaluation, all caching was disabled on Squid to ensure accurate measurements between repeated tests.

An Apache web server running on a Red Hat Linux 6.0 box was used to hold the test pages.

The machines were interconnected using standard 10 base-T Ethernet. All addresses were entered in IP form instead of using hostnames to prevent delays in DNS resolution.

4.3.2 Methodology

A Java program named *HttpBench* was developed to test the system. The user could input a list of URL’s into the system. *HttpBench* would then measure the time it took to retrieve the resource. In other words, it would return the total roundtrip time of the request and response.

A series of different files were placed on the web server, each with different sizes. Intended to simulate the size of typical HTTP transactions, the sizes ranged from zero bytes to two megabytes.

*HttpBench* was executed three times, each using different methods to grab the files:

1. Directly using normal HTTP.

2. Using the Squid proxy.

3. Using the Agent-Proxy architecture with the Squid proxy as the firewall.

Each file on the web server was grabbed repeatedly ten times. The roundtrip latencies were then averaged. Figure 4-1 shows the averaged roundtrip latencies for each trial.

From 0K to 200K, the curve is relatively flat, suggesting that the network’s bandwidth is large enough that it can handle the additional messages generated by either proxy. However, when the size of the file reaches 200K, the throughput limitation of
the Ethernet connection becomes more evident. The time it takes to transmit the file increases linearly with the size of the file.

To calculate the overhead of the Squid proxy over a direct, we subtract the two graphs. We do a similar subtraction to calculate the overhead of the Proxy-Agent architecture over the Squid proxy.

As shown in 4-2, the overhead of reverse HTTP increases linearly with the size of the file. The overhead in general is very low. Some of the reasons why the overhead is so low are:

1. Encoding the requests and responses is inexpensive. This is because all that needs to be done is to insert a few extra lines for encoding.

2. Decoding the requests and response is inexpensive. This is because all that needs to be done is to skip over a few lines.

3. System is multi-threaded. This allows both the Agent and Proxy to handle requests and responses as soon as they get them.
The Agent-Proxy architecture of the reverse HTTP protocol compares favorably with the regular proxy in that it adds a relatively constant 0.6 second delay regardless of the transmission size. See figure 4-2.

In terms of performance scalability, adding more browsers to the system should not make any noticeable effect. Neither is adding more web services to the system. The main reason for this is that when directing network traffic, the cost to lookup the appropriate browser or server in the hashtable is very cheap.

![Overhead](image)

Figure 4-2: Overhead
Chapter 5

Applications and security

5.1 Applications

The ideal applications for RHTTP are situations in which one needs to access a firewall protected resource occasionally. Using RHTTP is far simpler than trying to set up a VPN. No configuration changes are needed to the firewall. There is no need to install software on the client machines. In this section, we will give some examples of occasional firewall tunneling.

5.1.1 Publishing data to specific outsiders

Suppose an external customer needs to access a valuable computing resource behind a company’s protected firewall. A solution would be to simply move the resource outside the firewall, but this is often not acceptable from a security perspective.

If VPN were used as a solution, then this would be a relatively large expenditure in time and money on the part of the corporation. Furthermore, there remains the problem of limiting the access of the customer to only that specific resource.

RHTTP offers the most attractive solution. RHTTP would allow the owners of the vital data to give controlled, occasional access to customers. It does this without the security risks of placing the resource outside the firewall and without the time and expense of setting up a VPN to the customer.
5.1.2 Demonstrations to customers

A company representative visits a customer and wants to present a demonstration. However, the demo requires access to resources behind his company firewall. The usual solution to this would be to set up a dialup connection from their computers to a computer behind the firewall. This, in itself, is a potential security violation. Furthermore, the limited bandwidth of a dialup connection may not be adequate.

It would be ideal to use the customer's existing high speed connection to the Internet to access the resource behind the firewall. Setting up a brand new VPN network just for the purposes of a demo would be time consuming and expensive. In any event, the customer probably would not be all that happy with an outsider reconfiguring the network. However, using RHTTP would provide a quick and easy solution, allowing the representative to access the required resource for the purpose of the demo. Once the demo is finished, access can be disabled.

5.1.3 Providing customer support

Another useful application of the reverse HTTP protocol would be in providing technical customer support across the Internet.

Suppose a corporation purchased a network management system to be used on a company intranet. This network management system has a web interface to allow an administrator to control and diagnose network devices using a web browser. Now suppose this corporation requires assistance of a technician from an outside supplier. The technician wants to access the network management system from an outside browser. This would allow that person to do the necessary diagnosis from outside the firewall. A remote customer support solution is far more cost effective than physically sending someone to the customer's location just so that he or she could be behind the firewall.

Clearly, putting the management web server outside the firewall is not feasible as one could potentially be jeopardizing every network device in the corporation. Also, the customer would probably not be very willing to go through the time and expense
to set up a VPN with the support agency. But by using RHTTP, the technician can access the management web server without requesting the network administrator to reconfigure the firewall. The customer can temporarily let the technician provide support and then close the connection.

5.2 Security

In this section we will outline some of the security considerations that RHTTP presents.

5.2.1 Auditing and controlling access

Note how in order to use this protocol, every browser must access the Proxy. Likewise, every web server must only interact with the Agent. The Agent tightly controls what resource is available for outside browser, and all outside interaction always goes through the Agent-Proxy combination. Thus, the RHTTP system provides the ability to audit traffic and control access.

5.2.2 Encryption

The implementation of the reverse HTTP protocol in this thesis does not provide for any type of encryption. However, if such security is required, it would be relatively easy to encrypt the traffic between the Agent and the Proxy, and it could be done depending on the security requirements of individual servers.

For example, Secure Sockets Layer (SSL) [2] could be used to transmit the data in the data connection. SSL is an open protocol for securing data communication across computer networks, and it provides a secure channel for data transmission through its encryption capabilities. It allows for the transfer of digitally signed certificates for authentication procedures and provides message integrity, ensuring that the data cannot be altered en route.
5.2.3 Authentication

This design does not provide for authentication of browsers to the Proxy. However, an application of the end-to-end argument [7] would argue that it would be better to leave the problem of authentication up to the application level. For example, an appropriate authentication scheme would be to use the existing primitive HTTP standard for authentication [4] which could easily be applied at the Proxy. Another solution would be to use certificates for authentication.

5.2.4 Security loopholes

Perhaps some would say that the reverse HTTP protocol is simply exploiting a security hole and is dangerous. However, we argue that the security holes that this protocol may seem to introduce is no worse than the security holes inherent to the proxies installed on the firewall. The protocol does not use any holes other than what the proxy does in normal operation. This makes a strong case that one cannot rely on a firewall for all of an intranet’s security needs. Point-to-point security (such as Kerberos) is still required for true security.

Of course, given all the potential beneficial applications stemming from RHTTP, the ability for abuse is great. RHTTP’s high portability makes it very easy to install on any network. If somehow unauthorized agents were to be installed behind the firewall, it would make for an incredibly nasty trojan horse. Installing one of these behind the firewall could potentially let an attacker access every machine behind the firewall. This reiterates the need to have other security measurements in addition to having the firewall as a perimeter defense.

5.3 Extensions

5.3.1 Double firewalls

For simplicity, we have only discussed the situation where there is only one firewall. However, the protocol can easily be extended to the case where the browsers are also
protected by a firewall. The easy way to do this is to place the Proxy outside the firewall. The change is totally transparent as the browsers interact with the Proxy using normal web proxy.

5.3.2 Other protocols

For this thesis, we have focused on tunneling the HTTP protocol. However, it would not be unreasonable to extend the reverse HTTP protocol to tunnel other protocols. All that would need to be done is encode these protocols into the request/response structure of reverse HTTP.

Some potential examples would be FTP (File Transfer Protocol) [6] which can be used to transfer files, or SNMP (Simple Network Management Protocol) [1] which can be used to manage and monitor all sorts of equipment including computers, routers, wiring hubs, toasters and jukeboxes.

5.3.3 Access lists

Precise control over authorization to access resources can be obtained by having both the Agent and Proxy maintain dynamic access lists. In other words, the Proxy can maintain a list of browsers authorized to use the system. The Agent can maintain a list of servers that are accessible.
Chapter 6

Conclusions

This thesis proposes a new protocol called the reverse HTTP protocol to address the need of providing controlled access of protected resources by outside browsers through unsecured public networks such as the Internet.

Current solutions usually involve different implementations of the Virtual Private Network concept and provide a means to leverage the public networking infrastructure to access resources within a protected intranet. Unfortunately, such solutions usually open up access to all the resources in the Intranet. Selective filtering techniques through the firewall or proxy servers can provide controlled access to specific resources, but they are difficult to administer on an ongoing basis.

The reverse HTTP protocol provides a cost-effective solution to this problem without necessitating changes to the browsers, the servers, or the firewall configuration. A Java software implementation has been developed and tested to demonstrate the usefulness and efficacy of the reverse HTTP protocol. This experimental implementation proves that a portable, scalable solution can be developed with comparable performance characteristics to a normal web proxy. This is because very little overhead is generated in the additional encoding and decoding of requests and responses.
Appendix A

Source Code

A.1 RHA.java

```java
package rhttp;
import java.io.*;
import java.net.*;
import java.util.Date;

/**
 * A class containing describing the Reverse HTTP Agent
 *
 * author <a href="mailto:esit@mit.edu">Eric Sit</a>
 * version %I%, %C%
 * (c) Eric Sit 1999
 */
public class RHA implements Runnable {

    Thread t;

    String wwwHost;
    int wwwPort;

    static String RHPHost;
    static int RHPPort;

    Socket control_socket = null;
    Socket data_socket = null;
    Socket server_socket = null;
```
InputStream from..control = null;
HttpInputStream httpFromControl;
OutputStream to..control = null;
HttpOutputStream httpToControl;

InputStream from..server = null;
HttpInputStream httpFromServer;
OutputStream to..server = null;
HttpOutputStream httpToServer;

Proxy agent;
ProxyFile responseProxy;

String bID;
String aID;

boolean init = true;

/**
 * Creates an reverse HTTP agent object.
 * The agent is for web host wwwHost, port wwwPort.
 * The Reverse HTTP proxy to be given access is RHPHost, port RHPPort.
 */
public RHA(String wwwHost, int wwwPort, String RHPHost, int RHPPort) {
    this.wwwHost = wwwHost;
    this.wwwPort = wwwPort;
    this.RHPHost = RHPHost;
    this.RHPPort = RHPPort;
    t = new Thread(this, "RHA Thread");
    System.out.println("New thread: "+t);
    t.start();
}

/**
 * Starts the Reverse HTTP Agent.
 * The agent is for web host wwwHost, port wwwPort.
 * The Reverse HTTP proxy to be given access is pHost, port pPort.
 * The firewall proxy is at proxyHost, port proxyPort.
 * Usage: java RHA <wwwHost> <wwwPort> <pHost> <pPort> <proxyHost> <proxyPort> "
 */
public static void main(String[] args) throws IOException {

    String wwwHost;
    int wwwPort;

}
String RHPHost;  
int RHPPort;

try {  
// check arguments
if (args.length != 6)
    throw new IllegalArgumentException("Wrong number of arguments.");

// Get command-line arguments.
wwwHost = args[0];
wwwPort = Integer.parseInt(args[1]);
RHPHost = args[2];
RHPPort = Integer.parseInt(args[3]);
System.getProperties().put("proxySet", "true");
System.getProperties().put("proxyHost", args[4]);
System.getProperties().put("proxyPort", args[5]);

System.out.println("Starting RHA for web server "+wwwHost+": "+wwwPort+", using Proxy "+ System.getProperties().getProperty("proxyHost") + ":" + System.getProperties().getProperty("proxyPort");

RHA thisRHA = new RHA(wwwHost, wwwPort, RHPHost, RHPPort);
} catch (Exception e) {
    System.err.println(e);
    System.err.println("Usage: java RHA <wwwHost> <wwwPort>
<proxyHost> <proxyPort> <proxyPort>”);
}

/**
 * Handle incoming responses and make the request to the web server
 */
public void run() {
    // figure out the localhost
    String localHost;
    String localIP;
    try {
        localHost = InetAddress.getLocalHost().getHostName();
        localIP = InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException ex) {
        localHost = "localhost";
localIP = "127.0.0.1";
}

System.out.println("This machine is called "+localHost+" with an IP of "+localIP);
initialize();

System.out.println("Control connection initialized.");

while (true) {

    // wait until data is available before opening client to WWW
    waitForData(httpFromControl);

    if (init) {
        // skip over initial Ok header
        try {
            httpFromControl.readHeaders();
            init = false;
        } catch (IOException e) {
            System.err.println("Couldn't read initial headers on response");
            e.printStackTrace();
            try {
                httpFromControl.close();
            } catch (IOException f) {
                System.err.println("Couldn't close control.");
                e.printStackTrace();
            }
            continue;
        }

        // we don't care for what we just read, so just dump it.
        httpFromControl.resetHeaders();
        waitForData(httpFromControl);
    }

    // add an agent to handle requests coming in to RHA from RHP
    addAgent();

    // send the response back to RHP as a request
    try {

46
agent.join();
} catch (InterruptedException e) {
    System.err.println("agent interrupted!");
    e.printStackTrace();
}

if (!agent.dataSent) {
    System.err.println("Agent Abort!\n//abort
    System.err.println("closing control");
    try {
        control_socket.close();
    } catch (IOException e) {
        System.err.println("Can't close control connection");
    }
    System.err.println("Attempting to reinitialize");
    initialize();
    init=true;
    // skip to end of loop...
    continue;
} else {
    System.out.println("Agent data sent!\n    System.out.println();
}

System.out.println("Trying to encode as request and send back to RHP.");
sendResponse();

try {
    responseProxy.join();
} catch (InterruptedException e) {
    System.err.println("response proxy interrupted!");
    e.printStackTrace();
}

if (!responseProxy.dataSent) {
    System.err.println("Response Abort!\n//abort
    // we don't care that the data didn't make it to RHP.

    //
they will rerequest it.

    continue;

} else {
    System.out.println("Response proxy data sent!");
    System.out.println(


test code

    httpFromControl = new LoggedInputStream(from_control, "RHAin.txt");
    */

to_control = control_socket.getOutputStream();
    httpToControl = new HttpOutputStream(to_control, httpFromControl);
}
}

} catch (UnknownHostException e) {
    System.err.println("Don't know about host: RHP.");
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for the connection to: RHP.");
    System.exit(1);
}

// generate key for agent
aID = wwwHost;

try {
    httpToControl.write(HTTP.METHOD_POST + " http://
    + RHPHost +":" + RHPPort +"/" + HTTP.CONTROL_BIN +" HTTP/1.0\r\n");
    httpToControl.write("Content-length: 0\r\n");
    httpToControl.write("Agent-ID: " + aID +"\r\n");
    httpToControl.write("\r\n");
    httpToControl.flush()
} catch (IOException e) {
    System.err.println("Could not write to client!");
    System.exit(1);
}

/**
 * Create an agent that makes requests on behalf of the client
 * and sends the response back to the client.
 */
public void addAgent() {

    try {
        server_socket = new Socket(wwwHost, wwwPort);
        from_server = server_socket.getInputStreamO;
        httpFromServer = new HttpInputStream(from_server);
        to_server = server_socket.getOutputStreamO;
        httpToServer = new HttpOutputStream(to_server, httpFromServer);
    } catch (UnknownHostException e) {
        System.err.println("Don't know about host: "+ wwwHost);
        System.exit(1);
    } catch (IOException e) {
        System.err.println("Couldn't get I/O for the connection to: "+ wwwHost +": "+ wwwPort);
        System.exit(1);
    }

    System.out.println("Starting agent");

    // make a proxy to web server. persistent is set to true because it is
    // the web server's responsibility to close the connection
    agent = new Proxy(control_socket, server_socket, true);
agent.setPreprocess("decodeFromResponse");
agent.start();
}

/**
 * Sends back the response to the RHP request in the form of a request.
 */
public void sendResponse() {
try {
    // First grab the browser ID from the agent and save it.
    bID = agent.bID;

    System.out.println("Attempting to setup connection to RHP for response.");

    // Create a brand new socket.
    data_socket = new Socket(System.getProperties().getProperty("proxyHost"),
                             Integer.parseInt(System.getProperties().getProperty("proxyPort")));
}
catch (UnknownHostException e) {
    System.err.println("Don't know about host: RHP.");
    System.exit(1);
}
catch (IOException e) {
    System.err.println("Couldn't get I/O for the connection to: RHP. RHP seems to have disappeared.");
    System.exit(1);
}
}

System.out.println("Starting response ProxyFile");

responseProxy = new ProxyFile(server_socket, data_socket, true);
responseProxy.setPreprocess("encodeAsRequest");
responseProxy.bID = bID;
responseProxy.aID = aID;
responseProxy.start();
}

/**
 * Waits until data is available in testIn.
 */
void waitForData(InputStream testIn) {

    int bytes_available = -1;

    while (true) {
        try {
            bytes_available = testIn.available();
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to: RHP. RHP seems to have disappeared.");
            System.exit(1);
        }
    }
}
} catch (IOException e) {
    e.printStackTrace();
}

if (bytes.available > 0) {
    System.out.println(bytes.available + " available to be read.");
    break;
} else {
    t.yield();
}
}

/**
 * Returns the port of the reverse HTTP Proxy that the agent is giving access to.
 */
public static int getRHPPort() {
    return RHPPort;
}

/**
 * Returns the hostname of the reverse HTTP Proxy that the agent is giving access to.
 */
public static String getRHPHost() {
    return RHPHost;
}

} // end RHA

A.2 RHP.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.Vector;

/**
 * A class containing describing the Reverse HTTP Proxy
 *
 * author <a href="mailto:esitemit.edu">Eric Sit</a>
 * version %I%, %G%
 * (c) Eric Sit 1999
 */
public class RHP implements Runnable {


boolean listening = true;

InputStreamReader from_client = null;
OutputStreamWriter to_client = null;
InputStreamReader from_server = null;
OutputStreamWriter to_server = null;

RHPWeb web;

/**
 * A vector containing all information about RHP hosts that class knows about.
 */
SocketList socketList = new SocketList();

SocketList browserHash = new SocketList();

/**
 * Creates a reverse HTTP object.
 */
public static void main(String[] args) throws IOException {

RHP thisRHP = new RHP();
thisRHP.run();

}

/**
 * Handle incoming responses and make the request to the web server
 */
public void run() {

    // figure out the localhost
    String localhost;
    String localIP;
    try {
        localhost = InetAddress.getLocalHost().getHostName();
        localIP = InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException ex) {
        localhost = "localhost";
        localIP = "127.0.0.1";
    }

    System.out.println("This machine is called "+localhost+" with an IP of "+localIP);
}
addWebServer();

while (true) {
    try {
        web.join();
    } catch (InterruptedException e) {
        System.err.println("web service interrupted!");
        e.printStackTrace();
    }
}

/**
 * Creates web server to list all servers available
 */
public void addWebServer() {
    ServerSocket webServerSocket = null;
    try {
        webServerSocket = new ServerSocket(HTTP.PORT);
    } catch (IOException e) {
        System.err.println("Could not start web server on port: " + HTTP.PORT);
        System.exit(-1);
    }

    web = new RHPWeb(webServerSocket, socketList, browserHash);
    web.start();
}

A.3 RHPWeb.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.*;

/**
 * A class describing the RHP web server.
 */
public class RHPWeb extends Thread {

    Thread t;

    private ServerSocket ss = null;
    private Socket client = null;

    SocketList socketList;
    SocketList browserHash;
    Hashtable controlHash;

    public RHPWeb(ServerSocket socket, SocketList socketList, SocketList browserHash) {
        this.socketList = socketList;
        this.browserHash = browserHash;
        t = new Thread(this, "RHPWeb Thread");
        System.out.println("Child thread: "+ t);
        this.ss = socket;

        controlHash = new Hashtable();
    }

    public void run() {
        try {
            for (;;) {
                // Wait for client to connect. The method will block, and when it
                // returns the socket will be already connected to the client
                client = ss.accept();

                InputStream in;
                HttpInputStream httpIn;
                HttpProcessor processor;
                OutputStream out;
                HttpOutputStream httpOut;
                boolean keepAlive = false;

                try {
                    // get the inputStream

in = client.getInputStream();
httpIn = new HttpInputStream(in);

// figure out the right processor to use for this stream
processor = getProcessor(httpIn);

if (processor instanceof RhttpControl) {
    RhttpControl subProcessor = (RhttpControl) processor;
    keepAlive = subProcessor.keepAlive;
}

// get the output stream
out = client.getOutputStream();
httpOut = new HttpOutputStream(out, httpIn);

// process it
processor.processRequest(httpOut);

httpOut.flush();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (!keepAlive) {
        try {
            client.close();
        } catch (IOException ignored) {}
    }
}

} // Loop again, waiting for next connection
} catch (IOException e) {
    e.printStackTrace();
}

protected HttpProcessor getProcessor(HttpInputStream httpIn) {
    try {
        httpIn.readRequest();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
if (httpIn.getPath ().startsWith (HTTP.SERVERSBIN))
    return new RhttpServers (httpIn, socketList);
else if (httpIn.getPath ().startsWith (HTTP.CONTROL_BIN))
    return new RhttpControl (httpIn, socketList, browserHash, controlHash, client);
else if (httpIn.getPath ().startsWith (HTTP.DATABIN))
    return new RhttpData (httpIn, socketList, browserHash, client);
else if (httpIn.getPath ().startsWith (HTTP.PACBIN))
    return new RhttpPac (httpIn, socketList);
else if (httpIn.getPath ().startsWith (HTTP.CLOSE_BIN))
    return new RhttpClose (httpIn, socketList, controlHash, client);
else
    return new HttpFile (httpIn);

} catch (HttpException ex) {
    return ex;
} catch (Exception ex) {
    StringWriter trace = new StringWriter ();
    ex.printStackTrace (new PrintWriter (trace, true));
    return new HttpException (HTTP.STATUS_INTERNAL_ERROR,
    "<PRE>" + trace + "</PRE>");
}

}

A.4 Proxy.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.*;

/**
 * A class describing a generic proxy server. The proxy takes the
 * incoming bytes from the client_socket, processes it, and then
 * passes it along to the server_socket
 *
 * @author <a href="mailto:esitmit.edu">Eric Sit</a>
 * @version %1%, %G%
 * (c) Eric Sit 1999
 *
 public class Proxy extends Thread {

Thread t;

InputStream from_client;
OutputStream to_client;
InputStream from_server;
OutputStream to_server;

HttpInputStream http_from_client;
HttpOutputStream http_to_client;
HttpInputStream http_from_server;
HttpOutputStream http_to_server;

Socket client_socket, server_socket;

SocketList browserHash;

/**
 * The stored browser ID.
 */
public String bID;

String preprocessType = "";

/**
 * The contentLength of the HTTP body.
 */
private int contentLength = -1;

/**
 * If true, data has been sent sucessfully, otherwise false.
 */
public boolean dataSent = false;

/**
 * If a serversocket was passed in during creation, isServerSocket is true, otherwise false.
 */
public boolean isServerSocket = false;
ServerSocket ss;

/**
 * if false, Proxy will close data to outgoing connection after sending data.
 */
boolean persistent = true;
public Proxy(Socket client_socket, Socket server_socket, boolean persistent) {
    t = new Thread(this, "Proxy Thread");
    System.out.println("Child thread: " + t);
    this.server_socket = server_socket;
    this.client_socket = client_socket;
    this.persistent = persistent;
}

public Proxy(ServerSocket ss, Socket server_socket, SocketList browserHash, boolean persistent) {
    t = new Thread(this, "Proxy Thread");
    System.out.println("Child thread: " + t);
    this.ss = ss;
    isServerSocket = true;
    this.persistent = persistent;

    this.server_socket = server_socket;
    this.browserHash = browserHash;
}

/**
 * Main execution of Proxy thread.
 */
public void run() {
    System.out.println(t + " of preprocessType: "+ preprocessType +" has started.");

    if (isServerSocket) {
        try {
            client_socket = ss.accept();
            System.out.println("Connection accepted in Proxy!");

            // save the client socket in hashtable. The hashtable will notify.
            bID = client_socket.toString();
        }
    }
}
if (browserHash != null)
    browserHash.put(bID, client_socket);

} catch (IOException e) {
    System.err.println("Could not accept client socket for proxy");
    if (!getPreprocess().equals("decodeFromRequest"))  // mostly normal
        e.printStackTrace();
    dataSent=false;
    System.out.println(t +" of preprocessType: "+ preprocessType +" has returned with exception.");
    return;
}

try {
    createStreams(client_socket, server_socket);
    process();
} catch (IOException e) {
    // problem. firewall sent error, so there's no Browser ID?
    if (!getPreprocess().equals("decodeFromRequest"))  // mostly normal
        e.printStackTrace();
    dataSent=false;
    System.out.println(t +" of preprocessType: "+ preprocessType +" has returned with exception.");
    return;
}

try {
    toSocket();
} catch (IOException e) {
    if (!getPreprocess().equals("decodeFromRequest"))  // mostly normal
        e.printStackTrace();
    dataSent=false;
    System.out.println(t +" of preprocessType: "+ preprocessType +" has returned with exception.");
    return;
}

dataSent= true;

System.out.println(t +" of preprocessType: "+ preprocessType +" has returned.");

}  

/**
 * Creates the necessary streams from the passed in sockets.
 */
void createStreams(Socket client_socket, Socket server_socket) throws IOException {
    // create the streams
    try {
        from_server = server_socket.getInputStream();
        http_from_server = new HttpInputStream(from_server);
        to_server = server_socket.getOutputStream();
        http_to_server = new HttpOutputStream(to_server, http_from_server);
    } catch (IOException e) {
        System.err.println("System could not connect to server.");
        e.printStackTrace();
        throw e;
    }

    try {
        from_client = client_socket.getInputStream();
        http_from_client = new HttpInputStream(from_client);
        // ENABLE the following in order to LOG socket data
        /*
        if (preprocessType.equals("decodeFromResponse"))
            http_from_client = new LoggedInputStream(from_client, HTTP.LOG.BIN+"RHAin.txt");
        else
            http_from_client = new LoggedInputStream(from_client, HTTP.LOG.BIN+"RHPin.txt");
        */

        to_client = client_socket.getOutputStream();
        http_to_client = new HttpOutputStream(to_client, http_from_client);
    } catch (IOException e) {
        System.err.println("System could not connect to client.");
        System.err.println(e.toString());
        throw e;
    }
}

/**
 * Processes bytes depending on processType before proxying the data along.
 */
void process() throws IOException {
    if (preprocessType.equals("decodeFromResponse")) {
        try {
            // TEST code
            //System.out.println(http_from_client.bytesAvailable() + " bytes available to be read.");
        } catch (IOException e) {
            System.err.println("Error processing bytes.");
            e.printStackTrace();
        }
    }
}
http_from_client.readHeaders();

// TEST code
//System.out.println(http_from_client.bytesAvailable() + " bytes available to be read.");

bID = http_from_client.getHeader("Browser- ID");
if (bID != null)
    System.out.println("The browser ID read from RHP is " + bID);
else {
    // there must have been an error! firewall is probably throwing error message
    throw new IOException("BID is null. Firewall sending error message?");
}

String clStr = http_from_client.getHeader("Content-length");
if (clStr != null)
    setContentLength(Integer.parseInt(clStr));
else
    setContentLength(-1);
}

} catch (IOException e) {
    System.err.println("Couldn’t read headers in preprocessing");
e.printStackTrace();
    throw e;
}

} else if (preprocessType.equals("decodeFromRequest")) {
    // do nothing
}

/**
 * Proxies the data from one socket to another.
 */
void toSocket() throws IOException {
    // start proxying
    int bytes_read;
    byte[] buffer;
    // check to see if we have a content-length defined. If we do, we want to
    // stop when we have proxied reached contentLength.
    if (getContentLength() != -1) {
        if (getContentLength() < 4096) {  

            ...
buffer = new byte[getLength()];
else
    buffer = new byte[4096];

int bytes_read_total = 0;

try {
    while((bytes_read = client.read(buffer)) != -1) {
        http_to_server.write(buffer, 0, bytes_read);
        http_to_server.flush();
        bytes_read_total += bytes_read;
        if (bytes_read_total >= getLength())
            break;
    }
} catch (IOException e) {
    System.out.println("IOException in proxying. Socket seems dead.");
    //e.printStackTrace();
    throw e;
} else {
    buffer = new byte[4096];

    try {
        while((bytes_read = client.read(buffer)) != -1) {
            to_server.write(buffer, 0, bytes_read);
            to_server.flush();
        }
    } catch (IOException e) {
        System.out.println("IOException in proxying. Socket to WWW server seems dead.");
        //e.printStackTrace();
        throw e;
    }

    if (!persistent) {
        try {
            to_server.close();
        } catch (IOException e) {
        }
    }

    // if a non-persistent proxy was requested, then we must
    // close the streams
    if (persistent) {
        try {
            to_server.close();
        } catch (IOException e) {
        }
    }
public void setPreprocess(String preprocessType) {
    this.preprocessType = preprocessType;
}

public String getPreprocess() {
    return preprocessType;
}

public void setContentLength(int size) {
    this.contentLength = size;
}

public int getContentLength() {
    return contentLength;
}

public String getBrowserID() {
    return bID;
}
A.5 ProxyFile.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;

/**  
 * A class describing a proxy server. The proxy takes the bytes from a socket, writes  
 * it to a file, and copies the bytes from the file to a socket.  
 *  
 * author <a href= "mailto:esit@mit.edu">Eric Sit</a>  
 * version %1%, %G%  
 * (c) Eric Sit 1999  
 */

public class ProxyFile extends Thread {
    Thread t;

    InputStream from;
    HttpInputStream http_from;

    InputStream from_server;
    HttpInputStream http_from_server;
    OutputStream to_server;
    HttpOutputStream http_to_server;

    SocketList browserHash;

    /**  
     * The stored browser ID.  
     */
    public String bID;

    /**  
     * The stored agent ID.  
     */
    public String aID = "";

    File file;

    /**  
     * The contentLength of the HTTP body.  
     */
int contentLength;
/**
 * If a serversocket was passed in during creation, isServerSocket is true, otherwise false.
 */
public boolean isServerSocket = false;
/**
 * If true, data has been sent sucessfully, otherwise false.
 */
public boolean dataSent = false;

/**
 * If true, the ProxyFile thread will return at the first available moment.
 * This usually right after the serversocket accepts.
 */
public boolean shutdown = false;

Socket s;
ServerSocket ss;

Socket outSocket;

String preprocessType ="";

/**
 * if false, Proxy will close data to outgoing connection after sending data.
 */
boolean persistent = false;

/**
 * Creates a proxy to copy data from the socket to outSocket.
 * If persistent, connection to the socket will remain open after proxying data.
 */
public ProxyFile(Socket socket, Socket outSocket, boolean persistent) {
    t = new Thread(this, "ProxyFile Thread");
    System.out.println("Child thread: " + t);

    this.s = socket;
    this.outSocket = outSocket;
    this.persistent = persistent;
}

/**
 * Creates a proxy to copy data from the socket accepted by ss to outSocket.
 * BrowserHash maintains a hash between the browserID and the accepted socket.
 * If persistent, connection to the socket will remain open after proxying data.
 */
public ProxyFile(ServerSocket ss, Socket outSocket, SocketList browserHash, boolean persistent) {
    t = new Thread(this, "ProxyFile Thread");
    System.out.println("Child thread: "+t);
    this.ss = ss;
    this.outSocket = outSocket;
    isServerSocket = true;
    this.persistent = persistent;
    this.browserHash = browserHash;
}

/**
 * Main execution of Proxy thread.
 */
public void run() {
    System.out.println(t + " of preprocessType: "+ preprocessType +" has started.");

    boolean loop = true;
    while (loop) {
        if (isServerSocket)
            try {
                getClient();
            } catch (IOException ignored) {
                continue;
            }
        
        if (shutdown)
            return;

        try {
            toFile();
        } catch (IOException ignored) {
            continue;
        }

        try {
            synchronized(outSocket) {
                createStreams();
                preProcess();
                toSocket();
                dataSent = true;
                System.out.println("Proxyfile has successfully proxied along the data!");
            }
        } catch (IOException ignored) {
            continue;
        }
    }
}
System.out.println();
}
} catch (IOException e) {
    e.printStackTrace();
dataSent = false;
    System.err.println("Proxy file failed!");

if (getPreprocess().equals("encodeAsResponse")) {
    System.err.println("The control connection must be down!!!");

    try {
        System.out.println("Trying to close control connection...");
        outSocket.close();
    } catch (IOException f) {
        System.out.println(" Couldn't close control!");
    }

dataSent= false;
// we should wait until another control connection is established.
}
} finally {
    boolean deleted = file.delete();
    if (!deleted)
        System.out.println(file + " not deleted!");
}

if (!dataSent)
    try {
        wait(1000);
    } catch (InterruptedException e) {
    }

if (!getPreprocess().equals("encodeAsResponse"))
    loop=false;
}

System.out.println(t + " of preprocessType: "+ preprocessType +" has returned.");
}

/**
 * Waits to accept a socket. The browserID is stored in browserHash.
 */
void getClient() throws IOException {
    // If its a server socket, we need to wait for client socket
    try {
        s = ss.accept();
        if (shutdown) {
            System.out.println("Attempting to shutdown control thread!");
            try {
                ss.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                outSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return;
        }
        System.out.println("Request received!");
        // generate socket key.
        bID = s.getInetAddress().getHostAddress() + "\"+String.valueOf(s.getPort());
        if (browserHash != null)
            browserHash.put(bID, s);
        System.out.println("The saved browser ID is: "+bID);
    } catch (IOException e) {
        System.err.println("Could not accept client socket for proxy");
        e.printStackTrace();
        throw e;
    }
}

/**
 * Copies the data from a socket to a file.
 */
void toFile() throws IOException {
    // write to a file
String filename = HTTP.TMP_BIN + new Integer(s.getLocalPort()).toString() +".tmp";
file = new File(filename);

FileOutputStream fout;
try {
    fout = new FileOutputStream(file);
} catch (IOException e) {
    System.err.println("I0 Exception in creating fileoutputstream for writing");
    fout = null;
    e.printStackTrace();
    throw e;
}

try {
    from = s.getInputStream();
} catch (IOException e) {
    System.err.println("could not open file inputstream");
    e.printStackTrace();
    throw e;
}

if (preprocessType.equals("encodeAsResponse")) {
    // first try to read the request
    http_from = new HttpInputStream(from);

    try {
        http_from.readRequest();
    } catch (IOException e) {
        System.err.println("Could not read request from stream.");
        e.printStackTrace();
        throw e;
    }

    // if its not a POST, we can just write what we know to disk
    String requestString = http_from.getMethod() +" "+ http_from.getPath() +" "+
    "HTTP/"+ String.valueOf(http_from.getVersion()) +"\r\n";
    try {
        fout.write(stringToBytes(requestString));
    } catch (IOException e) {
        //System.err.println("Could not write request to disk.");
        e.printStackTrace();
    }
}
throw e;
}

// now write the rest of headers
Enumeration headerNames = http_from.getHeaderNames();
while (headerNames.hasMoreElements()) {
    String header = (String) headerNames.nextElement();
    try {
        fout.write(stringToBytes(header + ":
" + http_from.getHeader(header) + "\r\n"));
    } catch (IOException e) {
        System.err.println("Could not write the rest of the headers to disk");
        e.printStackTrace();
        throw e;
    }
}
try {
    fout.write(stringToBytes("\n");
} catch (IOException e) {
    System.err.println("Could not write \n after headers");
    e.printStackTrace();
    throw e;
}

if (http_from.getMethod() == HTTP.METHOD_POST) {
    // if it is a POST we need to write the body also.
    byte[] buffer;

    // first figure out if the request has a content length.
    String clStr = http_from.getHeader("Content-length");
    if (clStr != null) {
        contentLength = Integer.parseInt(clStr);
        buffer = new byte[contentLength];
    } else {
        contentLength = -1;
        buffer = new byte[4096];
    }

    int bytes_read;
    int bytes_read_total = 0;

    try {
        while ((bytes_read = http_from.read(buffer)) != -1) {
            fout.write(buffer, 0, bytes_read);
        }
    }
fout.flush();

bytes_read_total += bytes_read;

if ((bytes_read_total >= contentLength) && (contentLength != -1))
    break;
}
} catch (IOException e) {
    e.printStackTrace();
    throw e;
}
} finally {
    try {
        fout.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
else { // any other request other than POST
    try {
        fout.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
else if (preprocessType.equals("encodeAsRequest")) {

// all we need to do is write to disk directly.
// it will stop when the web server closes connection
byte[] buffer = new byte[4096];
int bytes_read;

http_from = new HttpInputstream(from);

try {
    while((bytes_read = http_from.read(buffer)) != -1)
        if (fout != null)
            fout.write(buffer, 0, bytes_read);
} catch (IOException e) {
    System.err.println("IOException in write (part of encodeAsRequest)\n");
    e.printStackTrace();
    throw e;
}
} finally {
    if (fout != null)
        try {
            
        }
fout.close();
}
}

} catch (IOException e) {
    e.printStackTrace();
}
}

/**
 * Creates the necessary streams from the sockets.
 */
void createStreams() throws IOException {
try {
    from-server = outSocket.getInputStream();
    http.from-server = new HttpInputStream (from-server);
    to-server = outSocket.getOutputStream();
    http.to-server = new HttpOutputStream (to-server, http.from-server);

    // ENABLE the following in order to LOG socket data
    /*
    if (preprocessType.equals("encodeAsResponse"))
        http.to-server = new LoggedOutputStream(to-server, http.from-server, HTTP.LOG_BIN+"RHPout.txt");
    else
        http.to-server = new LoggedOutputStream(to-server, http.from-server, HTTP.LOG_BIN+"RHAout.txt");
    */
} catch (IOException e) {
    System.err.println("System could not connect to server.");
    e.printStackTrace();
    throw e;
}
}

/**
 * Processes bytes depending on processType before proxying the data along.
 */
void preProcess() throws IOException {
    if (preprocessType.equals("encodeAsResponse")) {
        contentLength = (int) file.length();
        // first send additional headers
        http.to-server.setHeader("Content-Length", String.valueOf(contentLength));
        http.to-server.setHeader("Browser-ID", bID);
        try {
            } catch (IOException e) {
                e.printStackTrace();
        }
    }
}
http_to_server.sendHeaders();
http_to_server.flush();
}
catch(IOException e) {
    System.err.println("Couldn't set header in preprocessing");
e.printStackTrace();
    throw e;
}
} 410
else if (preprocessType.equals("encodeAsRequest")) {
    contentLength = (int) file.length();

    // send
    try {
        http_to_server.write(HTTP.METHOD_POST+" http://"+
RHA.getRHPHost()+":"+RHA.getRHPPort()+""+HTTP.DATA_BIN+" HTTP/1.0"+"\r\n");
        http_to_server.write("Agent-ID: "+aID+"\r\n");

        if (bID != null) {
            http_to_server.write("Browser-ID: "+bID+"\r\n");
        } else {
            System.err.println("Browser ID not set!");
        }

        http_to_server.write("Content-length: "+String.valueOf(contentLength)+"\r\n");
        http_to_server.write("\n");
        http_to_server.flush();
    } catch (IOException e) {
        System.err.println("Couldn't write to RHP");
e.printStackTrace();
        throw e;
    }
}
}
*/

/**
 * Copies the data from the file to the outSocket.
 */
void toSocket() throws IOException {
    // start proxying
    int bytes_read;
    byte[] buffer;

    if (contentLength<4096)
        buffer = new byte[(int) contentLength];
    else
        buffer = new byte[4096];
FileInputStream from_file;

try {
    from_file = new FileInputStream(file);
} catch (FileNotFoundException e) {
    System.err.println("File wasn't saved so it couldn't be found.");
    throw new IOException(e.toString());
}

try {
    while ((bytes_read = from_file.read(buffer)) != -1) {
        http.to_server.write(buffer, 0, bytes_read);
        http.to_server.flush();
        if (bytes_read >= contentLength)
            break;
    }
} catch (IOException e) {
    System.err.println("We couldn't write to the socket going to server for some reason!");
    try {
        from_file.close();
    } catch (IOException ef) {
        System.err.println(" Couldn't close file!");
        ef.printStackTrace();
    }
    e.printStackTrace();
    throw new IOException(e.toString() + "Couldn't write to socket going to server for some reason!");
} finally {
    try {
        from_file.close();
    } catch (IOException e) {
        System.err.println(" Couldn't close file!");
        e.printStackTrace();
    }
}

if (!persistent) {
    try {
        to_server.close();
    } catch (IOException e) {
    }
}
/**
 * Sets the preprocess type which affects what kind of preprocessing needs to be done.
 */
public void setPreprocess(String preprocessType) {
    this.preprocessType = preprocessType;
}

/**
 * Gets the preprocess type which affects what kind of preprocessing needs to be done.
 */
public String getPreprocess() {
    return preprocessType;
}

/**
 * Changes the outSocket to a new socket. Useful when a control connection needs
 * to be changed to a new socket.
 */
public synchronized void setOutSocket(Socket s) throws IOException {
    outSocket = s;
    try {
        createStreams();
    } catch (IOException e) {
        throw e;
    }
}

/**
 * Returns the target socket where data is being proxied into.
 */
public Socket getOutSocket(Socket s) {
    return outSocket;
}

/**
 * Reads a line of text from an inputStream.
 */
public String readLine(BufferedInputStream in) throws IOException {
    StringBuffer line = new StringBuffer();
    int c;
    while (((c = in.read ()) != -1) & & (c != '\n') & & (c != '\r'))
        line.append ((char) c);
    //if ((c == '\r') & & ((c = in.read ()) != '\n') & & (c != -1))
    //  - in.pos;
    return ((c == -1) & & (line.length () == 0)) ? null : line.toString();
}
public byte[] stringToBytes (String msg) throws IOException {
    return msg.getBytes ("latin1");
}

A.6 HTTP.java

package rhttp;

import java.io.*;
import java.net.*;
import java.util.*;

HTTP contains useful constants and routines to implement the HTTP protocol.

* * *
* (c) Eric Sit 1999
* * *

public class HTTP {
    public static final String SERVER_INFO = "HTTP/1.0";
    public static final String CONTROL_BIN = "/control/";
    public static final String DATA_BIN = "/data/";
    public static final String PAC_BIN = "/pac/";
    public static final String CLOSE_BIN = "/close/";
    public static final String SERVERS_BIN = "/servers/";
    public static final String LOG_BIN = "log/";
    public static final String TMP_BIN = "tmp/";
    public static final File SERVER_LOCATION =
        new File (System.getProperty("user.dir"));
    public static final File HTML_ROOT =
        new File (SERVER_LOCATION, "html");
    public static final int PORT = 80;
    public static final String DEFAULT_INDEX = "index.html";
    public static final String METHOD_GET = "GET";
public static final String METHODPOST = "POST";
public static final String METHODHEAD = "HEAD";

//standard HTTP error codes
public static final int STATUS_OKAY = 200;
public static final int STATUS_NOCONTENT = 204;
public static final int STATUS_MOVED_PERMANENTLY = 301;
public static final int STATUS_MOVED_TEMPORARILY = 302;
public static final int STATUS_BAD_REQUEST = 400;
public static final int STATUS_FORBIDDEN = 403;
public static final int STATUS_NOT_FOUND = 404;
public static final int STATUS_NOT_ALLOWED = 405;
public static final int STATUS_INTERNAL_ERROR = 500;
public static final int STATUS_NOT_IMPLEMENTED = 501;

/**
 * Given a HTTP code, returns an appropriate string message
 */
public static String getCodeMessage (int code) {
    switch (code) {
        case STATUS_OKAY: return "OK";
        case STATUS_NOCONTENT: return "No Content";
        case STATUS_MOVED_PERMANENTLY: return "Moved Permanently";
        case STATUS_MOVED_TEMPORARILY: return "Moved Temporarily";
        case STATUS_BAD_REQUEST: return "Bad Request";
        case STATUS_FORBIDDEN: return "Forbidden";
        case STATUS_NOT_FOUND: return "Not Found";
        case STATUS_NOT_ALLOWED: return "Method Not Allowed";
        case STATUS_INTERNAL_ERROR: return "Internal Server Error";
        case STATUS_NOT_IMPLEMENTED: return "Not Implemented";
        default: return "Unknown Code (" + code + ")";
    }
}

/**
 * Vector containing server information
 */
protected static final Vector environment = new Vector();
static {
    environment.addElement ("SERVER_SOFTWARE= " + SERVER_INFO);
    environment.addElement ("GATEWAY_INTERFACE= " + "CGI/1.0");
    environment.addElement ("SERVER_PORT= " + PORT);
    environment.addElement ("DOCUMENT_ROOT= " + HTML_ROOT.getPath());
    try {
        environment.addElement
("SERVER_NAME=" + InetAddress.getLocalHost().getHostName());
)

} catch (UnknownHostException ex) {
    environment.addElement("SERVER_NAME=localhost");
}

/**
 * Strips all relative components from a path, including //, /., and ../.
 * returning the resulting properly canonicalized path.
 */
public static String canonicalizePath(String path) {
    char[] chars = path.toCharArray();
    int length = chars.length;
    int idx, odx = 0;
    while ((idx = indexOf(chars, length, '/', odx)) < length - 1) {
        int ndx = indexOf(chars, length, '/', idx + 1), kill = -1;
        if (ndx == idx + 1) {
            kill = 1;
        } else if ((ndx >= idx + 2) && (chars[idx + 1] == '.') {
            if (ndx == idx + 2) {
                kill = 2;
            } else if ((ndx == idx + 3) && (chars[idx + 2] == '.')) {
                kill = 3;
                while ((idx > 0) && (chars[--idx] != '/'))
                    ++kill;
            }
        }
    }
    if (kill == -1) {
        odx = ndx;
    } else if (idx + kill >= length) {
        length = odx = idx + 1;
    } else {
        length -= kill;
        System.arraycopy(chars, idx + 1 + kill,
                         chars, idx + 1, length - idx - 1);
        odx = idx;
    }
    return new String(chars, 0, length);
}

/**
 * Helper method returns index of the character chr in the length-character array
 * chars, starting from index from.
 */
protected static int indexOf (char[] chars, int length, char chr, int from) {
    while ((from < length) && (chars[from] != chr))
        ++ from;
    return from;
}

/**
 * Translates an HTTP filename into a local pathname.
 */
public static String translateFilename (String filename) {
    StringBuffer result = new StringBuffer();
    int idx, odx = 0;
    while ((idx = filename.indexOf ('/', odx)) != -1) {
        result.append (filename.substring (odx, idx)).append (File.separator);
        odx = idx + 1;
    }
    result.append (filename.substring (odx));
    return result.toString();
}

/**
 * Decodes a URL-encoded string, converting all + characters to spaces
 * and all %xy hex-encodings to their decoded values.
 *
 * Note: JDK 1.2 class URLDecoder provides a decode() method.
 */
public static String decodeString (String str) {
    String replaced = str.replace ('+', ' ');
    StringBuffer result = new StringBuffer();
    int idx, odx = 0;
    while ((idx = str.indexOf ('%', odx)) != -1) {
        result.append (replaced.substring (odx, idx));
        try {
            result.append ((char) Integer.parseInt
                (str.substring (idx + 1, idx + 3), 16));
        } catch (NumberFormatException ex) {
        }
        odx = idx + 3;
    }
    result.append (replaced.substring (odx));
    return result.toString();
}

/**
 * Table of known MIME types.

protected static final Hashtable mimeTypes = new Hashtable();
static {
    mimeTypes.put("gif", "image/gif");
    mimeTypes.put("jpeg", "image/jpeg");
    mimeTypes.put("jpg", "image/jpeg");
    mimeTypes.put("html", "text/html");
    mimeTypes.put("htm", "text/html");
}

/**
 * Guesses a MIME type based on extension of filename
 */
public static String guessMimeType (String fileName) {
    int i = fileName.lastIndexOf(".");
    String type = (String) mimeTypes.get(fileName.substring(i + 1).toLowerCase());
    return (type != null) ? type : "text/plain";
}

A.7 HttpInputStream.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.*;

/**
 * An InputStream with special routines for HTTP requests.
 *
 * author <a href="mailto:esit@mit.edu">Eric Sit</a>
 * version %I%, %G%
 * (c) Eric Sit 1999
 */
public class HttpInputStream extends BufferedInputStream {

    /**
     * Creates an HttpInputStream from an InputStream.
     */
    public HttpInputStream (InputStream in) {
        super (in);
    }
}
protected String method, path, queryString;
protected float version;
protected Hashtable headers = new Hashtable();

/**
 * Parses an HTTP request, storing the method, request, and version in the object.
 */
public void readRequest() throws IOException {
    String request = readLine();
    if (request == null)
        throw new HttpException(HTTP.STATUS_BAD_REQUEST, "Null query");
    StringTokenizer parts = new StringTokenizer(request);
    try {
        parseMethod(parts.nextToken());
        parseRequest(parts.nextToken());
    } catch (NoSuchElementException ex) {
        throw new HttpException(HTTP.STATUS_BAD_REQUEST, request);
    }
    if (parts.hasMoreTokens())
        parseVersion(parts.nextToken());
    else
        version = 0.9f;
    if ((version < 1.0f) && (method == HTTP.METHOD_HEAD))
        throw new HttpException(HTTP.STATUS_NOT_ALLOWED, method);
    if (version >= 1.0f)
        readHeaders();
}

/**
 * Parses an HTTP method portion of the HTTP request.
 */
protected void parseMethod(String method) throws HttpException {
    if (method.equals(HTTP.METHOD_GET))
        this.method = HTTP.METHOD_GET;
    else if (method.equals(HTTP.METHOD_POST))
        this.method = HTTP.METHOD_POST;
    else if (method.equals(HTTP.METHOD_HEAD))
        this.method = HTTP.METHOD_HEAD;
    else
        throw new HttpException(HTTP.STATUS_NOT_IMPLEMENTED, method);
}

/**
 * Parses the request portion of the HTTP request
 */
protected void parseRequest (String request) throws HttpException {

    System.out.println("Parsing request: " + request);

    if (request.startsWith("http://")) {
        // we want to remove the http://hostname part

        String relPath;  // result after we remove http://

        int pathIdx = request.indexOf('/', 7);  // the slash after the hostname
        if (pathIdx == request.length()-1)
            relPath = "/";
        else
            relPath = request.substring(pathIdx);

        // now do the normal stuff
        path = HTTP.canonicalizePath (relPath);
        queryString = "";
    }

    else if (request.startsWith("/")) {

        path = HTTP.canonicalizePath (request);
        queryString = "";
    }

    else
    throw new HttpException (HTTP.STATUS_BAD_REQUEST, request);
}

/**<n
* Parses the version part in the HTTP request.
*/

protected void parseVersion (String verStr) throws HttpException {

    if (!verStr.startsWith ("HTTP/"))
        throw new HttpException (HTTP.STATUS_BAD_REQUEST, verStr);
    try {
        version = Float.valueOf (verStr.substring (5)).floatValue();
    } catch (NumberFormatException ex) {
        throw new HttpException (HTTP.STATUS_BAD_REQUEST, verStr);
    }
}
/**
 * Reads through all the headers in the HTTP request.
 */
public void readHeaders () throws IOException {
    String header;
    while (((header = readLine ()) != null) && !header.equals ("")) {

        // TEST
        // System.out.println("readHeaders: "+header);

        int colonIdx = header.indexOf (':');
        if (colonIdx != -1) {
            String name = header.substring (0, colonIdx);
            String value = header.substring (colonIdx + 1);
            headers.put (name.toLowerCase (), value.trim ());
        }
    }
}

/**
 * Reads a line and returns a string.
 */
public String readLine () throws IOException {
    StringBuffer line = new StringBuffer ();
    int c;

    while (((c = read ()) != -1) && (c != '\n') && (c != '\r'))
        line.append ((char) c);

    if ((c == '\r') && ((c = read ()) != '\n') && (c == -1))
        -- pos;

    return ((c == -1) && (line.length () == 0) ? null : line.toString ());
}

/**
 * Returns the request's method.
 */
public String getMethod () {
    return method;
}

/**
 * Returns the request's path.
 */
public String getPath () {
    return path;
}

/**
 * Returns the request's query.
 */
public String getQueryString () {
    return queryString;
}

/**
 * Returns the request's version.
 */
public float getVersion () {
    return version;
}

/**
 * Returns the request's header.
 */
public String getHeader (String name) {
    return (String) headers.get (name.toLowerCase ());
}

/**
 * Returns an Enumeration of all the headers in the request.
 */
public Enumeration getHeaderNames () {
    return headers.keys ();
}

/**
 * Erases all the object's stored headers from the request.
 */
public void resetHeaders() {
    headers = new Hashtable ();
}

/**
 * Returns the number of bytes available in stream.
 */
public int bytesAvailable() {
    int bytes_available;
    try {
        bytes_available = this.available();
    } catch (IOException e) {
    }
}
public class HttpOutputStream extends BufferedOutputStream {
    protected int code;
    protected boolean sendHeaders, sendBody;
    protected Hashtable headers = new Hashtable();

    /**
     * Creates an HttpOutputStream from an OutputStream.
     */
    public HttpOutputStream (OutputStream out, HttpInputStream in) {
        super (out,1);
        code = HTTP.STATUS_OKAY;
        setHeader ("Server", HTTP.SERVER.INFO);
        setHeader ("Date", new Date ().toString ());
        //send headers depending on version?
        //sendHeaders = (in.getVersion () >= 1.0);
        sendHeaders = true;

        sendBody = !HTTP.METHOD_HEAD.equals (in.getMethod ());
    }
/**
 * Changes the response code.
 */
public void setCode (int code) {
    this.code = code;
}

/**
 * Adds a header.
 */
public void setHeader (String attr, String value) {
    headers.put (attr, value);
}

/**
 * Sends the HTTP headers that have been initialized with the setCode() and setHeader() methods and then returns whether the caller should send a subsequent body. If not, it will be because this is a head request and no body is required.
 */
public boolean sendHeaders () throws IOException {
    if (sendHeaders) {
        write ("HTTP/1.0 " + code + " " + HTTP.getCodeMessage (code) + "\r\n");
        Enumeration attrs = headers.keys();
        while (attrs.hasMoreElements()) {
            String attr = (String) attrs.nextElement();
            write (attr + ": " + headers.get (attr) + "\r\n");
        }
        write ("\n");
        return sendBody;
    }
}

/**
 * Writes a line of text. Coverts the String into an array of bytes in ISO Latin 1 encoding and writes these raw bytes to the attached stream.
 */
public void write (String msg) throws IOException {
    write (msg.getBytes ("latin1"));
public void write (InputStream in) throws IOException {
    int n, length = buf.length;
    while ((n = in.read (buf, count, length - count)) >= 0)
        if ((count += n) >= length)
            out.write (buf, count = 0, length);
}

A.9  HttpProcessor.java

package rhttp;
import java.io.*;

/**<n*
 * The common interface for all HttpProcessors. HttpProcessors handle HTTP requests.
 *<n*/
* author <a href="mailto:esit@mit.edu">Eric Sit</a>
* version %I%, %G%
* (c) Eric Sit 1999
/**<n*/
public interface HttpProcessor {

/**<n*/
 * Processes the request, writing any appropriate data to out.
 *<n*/
public void processRequest (HttpOutputStream out) throws IOException;
}

A.10  HttpFile.java

package rhttp;
import java.io.*;

/**<n*/
 * The HttpProcessor that handles simple requests for files.
 *<n*/
* author <a href="mailto:esit@mit.edu">Eric Sit</a>
* version %I%, %G%
* (c) Eric Sit 1999
/**<n*/
public class HttpFile implements HttpProcessor {
    protected File file;

    /**
     * Constructor for HttpFile.
     *
     * Client's file request for a file is passed into the constructor which throws any
     * exception that result from the client's request.
     *
     * The constructor fully parses the client's request. If the client attempts to post
     * to a file, we throw an appropriate HttpException.
     *
     * We translate the request into a file within the local HTML document directory, appending
     * index.html if the request ends in /.
     */

    public HttpFile (HttpInputStream in) throws IOException {
        if (in.getMethod () == HTTP.METHOD.POST)
            throw new HttpException (HTTP.STATUS_NOT_ALLOWED,
                                "<TT>" + in.getMethod () + " " + in.getPath () + "</TT>");
        file = new File (HTTP.HTML_ROOT, HTTP.translateFilename (in.getPath ())); 30
        if (in.getPath ().endsWith ("/"))
            file = new File (file, HTTP.DEFAULT_INDEX);
        if (!file.exists ())
            throw new HttpException (HTTP.STATUS_NOT_FOUND,
                                "File <TT>" + in.getPath () + "</TT> not found.");
        if (file.isDirectory ())
            throw new RedirectException (HTTP.STATUS_MOVED_PERMANENTLY,
                                in.getPath () + "/");
        if (file.isFile () || !file.canRead ())
            throw new HttpException (HTTP.STATUS_FORBIDDEN, in.getPath ());
    }

    /**
     * When it is time to send a response, this method is called.
     * We set Content-type and Content-length headers, then call sendHeaders() to send
     * the headers; and finally, transmit the body of the file if necessary.
     */

    public void processRequest (HttpOutputStream out) throws IOException {
        out.setHeader ("Content-type", HTTP.guessMimeType (file.getName ())); 50
        out.setHeader ("Content-length", String.valueOf (file.length ()));
        if (out.sendHeaders ()) {
            FileInputStream in = new FileInputStream (file);
            out.write (in);
        }
    }
}
public class RhttpClose implements HttpProcessor {
    protected File file;
    SocketList socketList;
    Hashtable controlHash;
    Socket client;
    FileInputStream in;
    String aID;

    public boolean keepAlive = false;

    public RhttpClose (HttpServletRequest in, SocketList socketList,
            Hashtable controlHash, Socket client) throws IOException {
        this.socketList = socketList;
        this.controlHash = controlHash;
        this.in = in;
        this.client = client;

        // we assume that path begins with /close/

        if (!in.getPath().startsWith(HTTP.CLOSE_BIN))
            throw new HttpException (HTTP.STATUS_BAD_REQUEST, in.getPath ());
    }
}

A.11  RhttpClose.java
if (in.getPath().length() < HTTP.CLOSE_BIN.length())
    throw new HttpException (HTTP.STATUS_BAD_REQUEST, in.getPath ());

int indexOfClose = HTTP.CLOSE_BIN.length ();
this.aID = in.getPath().substring(indexOfClose);
System.out.println("Agent-ID in PAC: "+aID);
}

public void processRequest (HttpOutputStream out) throws IOException {

if (in.getMethod () == HTTP.METHOD_GET) {
    deleteServer(aID);
    sendResponse(out);
}

}

void deleteServer(String aID) {
    ServerSocket ss = (ServerSocket) socketList.get(aID);
    InetAddress address = ss.getInetAddress();
    int port = ss.getLocalPort();

    // shutdown safely the controlThread for the server
    ProxyFile controlThread = (ProxyFile) controlHash.get(aID);
    controlThread.shutdown = true;
    try {
        Socket shutdownSocket = new Socket(address.getLocalHost(), port);
    } catch (IOException e) {
        System.err.println("Couldn't connect to control proxy to shut it down!");
        e.printStackTrace();
    }
    controlHash.remove(aID);
    socketList.remove(aID);
}

void sendResponse(HttpOutputStream out) throws IOException {
A.12 RhttpControl.java

```
package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * The HttpProcessor that handles requests to to initiate and reinitiate the control connection.
 *
 * author &lt;a href="mailto:esit@mit.edu">Eric Sit&lt;/a&gt;
 * version %I%, %G%
 * (c) Eric Sit 1999
 */
public class RhttpControl implements HttpProcessor {
    protected File file;
    SocketList socketList;
    SocketList browserHash;
    Hashtable controlHash;
    Socket client;
    HttpInputStream in;

    public boolean keepAlive = false;

    public RhttpControl (HttpInputStream in, SocketList socketList,
        SocketList browserHash, Hashtable controlHash,
        Socket client) throws IOException {
        this.socketList = socketList;
        this.browserHash = browserHash;
        this.controlHash = controlHash;
        this.client = client;
    }
```
```java
if (in.getMethod() == HTTP.METHOD_POST) {

    // get the output stream
    OutputStream o = client.getOutputStream();
    HttpOutputStream out = new HttpOutputStream(o, in);
    // send an OK back
    out.write("HTTP/1.0 " + HTTP.STATUS_OKAY + " Ok\r\n");
    out.write("\r\n");
    out.flush();
    keepAlive = true;

    // Find the agent ID.
    String aID = in.getHeader("Agent-ID");

    ServerSocket ss = null;

    if (!socketList.containsKey(aID)) {
        // we are adding a new server
        ss = addServer(aID);

        // start proxy
        ProxyFile toControl = new ProxyFile(ss, client, browserHash, true);
        toControl.setPreprocess("encodeAsResponse");
        toControl.start();

        controlHash.put(aID, toControl);
    } else {
        // we are trying to reinit a dead connection
        System.out.println("Attempting to reinitialize.");

        ss = (ServerSocket) socketList.get(aID);

        ProxyFile newToControl = (ProxyFile) controlHash.get(aID);
        newToControl.setOutSocket(client);
    }
}

public void processRequest(HttpOutputStream out) throws IOException {
```

---

The code snippet begins with a check to see if the incoming method is `POST`. If it is, it proceeds to handle the request. The method `processRequest` is defined with a return type of `null` and a single parameter `HttpOutputStream out`, with the `throws` clause listing `IOException`. The code handles the creation of an output stream, writes an OK response back to the client, and includes logic for managing the agent ID and establishing a new server socket if necessary. It also contains a mechanism to handle reinitialization of a dead connection.
ServerSocket addServer(String aID) {

    // now we want to create a serverSocket to handle requests for this server
    // first pick a port to use
    boolean unique = false;
    ServerSocket ss = null;

    while (!unique) {
        try {
            ss = new ServerSocket(0);
            unique = true;
            System.out.println("ServerSocket at "+ss.getLocalPort());
        } catch (IOException e) {
            System.err.println("Could not start server on port: "+ss.getLocalPort());
            System.out.println("Trying a different port.");
            // try to get a different port
        }
    }

    // store association between new server socket to the connected client socket
    socketList.put(aID, ss);

    System.out.println("Associating RHP port "+ss.getLocalPort()+" with client.");

    return ss;
}

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.*;

/**
 * The HttpProcessor that handles incoming data.
 * These are the responses to previously made requests.
 * author <a href="mailto:esit@mit.edu">Eric Sit</a>
 * version %1%, %G%
 */
public class RhttpData implements HttpProcessor {
    protected File file;
    Socket client;
    Socket bSocket;
    String bID;
    String aID;
    SocketList socketList;
    SocketList browserHash;
    ServerSocket ss;
    HttpInputStream in;
    OutputStream to_server;
    HttpOutputStream http_to_server;
    
    public boolean keepAlive = false;

    // we are guaranteed that client is distinct because we just got a new socket from RHPWeb's accept call
    
    public RhttpData (HttpInputStream in, SocketList socketList, SocketList browserHash, Socket client) throws IOException {
        this.socketList = socketList;
        this.browserHash = browserHash;
        this.client = client;
        this.in = in;

        System.out.println("Begin RhttpData.");

        // first create the streams
        try {
            to_server = client.getOutputStream();
            http_to_server = new HttpOutputStream (to_server, in);
        } catch (Exception e) {
            System.err.println("System could not connect to server.");
            e.printStackTrace();
        }
    }
}
if (in.getMethod() == HTTP.METHOD.POST) {
    // we should be handling a continuation of a previous communication

    System.out.println("RHA POST request should contain DATA.");

    bID = in.getHeader("Browser-ID");

    // System.out.println("The browser ID in the incoming request is " + bID);

    // using the browser ID coming in, try to match it up with the original browser socket
    bSocket = (Socket) browserHash.get(bID);

    // determine agent-id
    aID = in.getHeader("Agent-ID");
    System.out.println("The agent ID of this RHA request is " + aID);

    // determine appropriate server socket
    ss = (ServerSocket) socketList.get(aID);

    keepAlive = true;
}

public void processRequest(HttpOutputStream out) throws IOException {

    // make a proxy to original browser socket
    Proxy toBrowser = new Proxy(client, bSocket, false);
    toBrowser.setPreprocess("decodeFromRequest");
    toBrowser.setContentLength(Integer.parseInt(in.getHeader("Content-length")));
    toBrowser.start();

    // wait for everything to be sent back before proceeding.
    try {
        toBrowser.join();
    } catch (InterruptedException e) {
        System.out.println("Rhttpdata back to browser interrupted!");
        e.printStackTrace();
    }

    // get rid of the browser socket entry in the hash table since we don't need it anymore.
    browserHash.remove(bID);

    if (!toBrowser.dataSent) {

System.err.println("Couldn't send data back to browser!");
// for some reason, we couldn't send the data to the browser.

// A possible reason for this is that the browser has already closed the socket.
// For example, the user clicked on another link before it received a response to an earlier request.

// we don't care about the data anymore. we consider it dead.

// send back an error along the data connection
http_to_server.write("HTTP/1.0 " + HTTP.STATUS.INTERNAL_ERROR + " 0\r\n");
http_to_server.write("\r\n");
http_to_server.flush();

// we want to close connection to RHA
keepAlive = false;

// Now, the assumption is that the control connection is still up.
// The plan is to ignore this particular bit of data coming in along the data connection.
// We'll just wait till another request comes along the control connection.

} else {
// success

http_to_server.write("HTTP/1.0 " + HTTP.STATUS.NO_CONTENT + " 0\r\n");
http_to_server.write("\r\n");
http_to_server.flush();

}

System.out.println("End RhttpData.");
System.out.println();

}

}

A.14 RhttpPac.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.*;
/**
 * The HttpProcessor that handles requests for PAC files. PAC files are "Proxy automatic
 * configuration files."
 *
 * This processor depends on a file "proxy.cfg" which contains site specific proxy
 * information.
 *
 * author <a href="mailto:esitemit.edu">Eric Sit</a>
 * version %I%, %G%
 * (c) Eric Sit 1999
 */

public class RhttpPac implements HttpProcessor {
    protected File file;
    HttplnputStream in;
    SocketList socketList;
    String host;
    int localPort;
    String localHost;

    public boolean keepAlive = false;

    public RhttpPac (HttplnputStream in, SocketList socketList) throws IOException {
        this.in = in;
        this.socketList = socketList;
        System.out.println(in.getPath());

        // we assume that path begins with /pac/ and ends with .pac

        if (!in.getPath().startsWith(HTTP.PAC-BIN))
            throw new HttpException (HTTP.STATUS.BAD-REQUEST, in.getPath ());
        if (!in.getPath().endsWith(".pac"))
            throw new HttpException (HTTP.STATUS.BAD-REQUEST, in.getPath ());
        if (in.getPath().length() < HTTP.PAC.BIN.length())
            throw new HttpException (HTTP.STATUS.BAD-REQUEST, in.getPath ());

        int indexOfPac = HTTP.PAC-BIN.length();
        int indexOfExt = in.getPath().lastIndexOf(".pac");

        String aID = in.getPath().substring(indexOfPac, indexOfExt);

        host = aID;
ServerSocket ss = (ServerSocket) socketList.get(aID);

localPort = ss.getLocalPort();

// figure out the localhost
try {
    localHost = InetAddress.getLocalHost().getHostName();
} catch (UnknownHostException ex) {
    localHost = "localhost";
}

public void processRequest (HttpOutputStream out) throws IOException {

    File file = new File("proxy.cfg");
    FileInputStream from_file;

    try {
        from_file = new FileInputStream(file);
    } catch (FileNotFoundException e) {
        System.err.println("Proxy config file unavailable!");
        throw new IOException(e.toString());
    }

    System.out.println("Generating PAC.");

    out.setHeader ("Content-type", "application/x-ns-proxy-autoconfig");
    out.sendHeaders();
    out.write("function FindProxyForURL(url, host)\
    {\
        if (shExpMatch( host, ""+host+"\*")\
            return "PROXY "+\
            localHost +":"+localPort+";"\n            ");

    // copy rest of proxy config from file
    byte[] buffer = new byte[4096];
    int bytes_read;
    try {
        while ((bytes_read = from_file.read(buffer)) != -1) {
            out.write(buffer, 0, bytes_read);
            out.flush();
    }
catch (IOException e) {
    System.err.println("Error writing proxy info to socket!");
    try {
        from_file.close();
    } catch (IOException ef) {
        System.err.println("Couldn't close file!");
        ef.printStackTrace();
    }
    e.printStackTrace();
    throw new IOException(e.toString() + "Error writing proxy info to socket");
} finally {
    try {
        from_file.close();
    } catch (IOException e) {
        System.err.println("Couldn't close file!");
        e.printStackTrace();
    }
}

out.write("\n");
out.flush();
}

---

**A.15 RhttpServers.java**

```java
package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * The HttpProcessor that handles requests for a list of available servers.
 * 
 * @author <a href="mailto:esit@mit.edu">Eric Sit</a>
 * @version %I%, %G%
 * (c) Eric Sit 1999
 */
public class RhttpServers implements HttpProcessor {
    protected File file;
    SocketList socketList;
```

99
public boolean keepAlive = false;

public RhttpServers (HttpInputStream in, SocketList socketList) throws IOException {
    this.socketList = socketList;
    this.in = in;
}

public void processRequest (HttpOutputStream out) throws IOException {
    if (in.getMethod () == HTTP.METHOD_GET) { 
        listServers(out);
    }
}

void listServers(HttpOutputStream out) throws IOException {

    // figure out the localhost
    String localhost;
    try {
        localhost = InetAddress.getLocalHost ().getHostName ();
    } catch (UnknownHostException ex) {
        localhost = "localhost";
    }

    // display server list
    System.out.println("Showing server list.");
    out.setHeader ("Content-type", "text/html");
    out.sendHeaders();

    out.write("<h1>Available Servers</h1>

    if (socketList.isEmpty())
        out.write("No servers have been added.");
    else {
        Enumeration sEnum = socketList.keys();
        while (sEnum.hasMoreElements()) {
            String aID = (String) sEnum.nextElement();

            out.write("<hr>");
            out.write("For RHA " + aID + ", we have allocated a tunneling proxy at ");
            // figure out which local port has been allocated to mirror it.

    }
}
ServerSocket localSS = (ServerSocket) socketList.get(aID);
out.write("<a href=http://"+ localHost +":" + localSS.getLocalPort() +"/>");
out.write("http://"+ localHost +":" + localSS.getLocalPort() +"/<a><br>");
out.write("You need change your proxy configuration to use this proxy.<p>");

// write direction to get to a PAC file
out.write("Use this PAC file: <a href=http://"+ localHost +"/pac/"+ aID +".pac"+">");
out.write("http://"+ localHost +"/pac/"+ aID +".pac"+"</a><p>");
out.write("Then browse: <a href=http://" + aID +">http://" +aID+ "</a><p>");
out.write("Kill connection: <a href=http://"+ localHost +"/close/" +aID+ "">");
out.write("http://"+ localHost +"/close/" +aID+ "</a><p>");

}
}

A.16 SocketList.java

package rhttp;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * A class containing a table of connections to servers
 *
 * author <a href="mailto:esitemit.edu">Eric Sit</a>
 *
 * version %I%, %G%
 *
 * (c) Eric Sit 1999
 */
public class SocketList extends Hashtable {

  public void SocketList() {
  }

  public synchronized Object ask(Object key) {
    try {
      wait();
    } catch (InterruptedException e) {
      System.err.println("Interrupted Exception caught");
    }
    return get(key);
  }
public synchronized Object deliver(Object key, Object value) {
    Object obj = put(key, value);
    notify();
    return obj;
}

public String listAll() {
    return this.toString();
}

A.17 HttpException.java

package rhttp;
import java.io.*;

/**
 * An exception handling whenever something unusual needs to be sent to browser.
 * 
 * author <a href="mailto:esit@mit.edu">Eric Sit</a>
 * 
 * version %I%, %G%
 * (c) Eric Sit 1999
 */
public class HttpException extends IOException implements HttpProcessor {
    protected int code;

    /**
     * Creates an HttpException, given an HTTP error code and a location.
     */
    public HttpException (int code, String detail) {
        super (detail);
        this.code = code;
    }

    /**
     * Sends back an HTML page giving error.
     */
    public void processRequest (HttpOutputStream out) throws IOException {
        out.setCode (code);
        out.setHeader ("Content-Type", "text/html");
        if (out.sendHeaders ()) {
            out
        }
    }
}
String msg = HTTP.getCodeMessage (code);
out.write ("<HTML><HEAD><TITLE>
" + code + " " + 
msg + "</TITLE></HEAD>\n" + "<BODY><H1>" + msg + "</H1>\n" + 
getMessage () + "</BODY></HTML>\n");
}

A.18 RedirectException.java

package rhttp;
import java.io.*;

/**
 * An exception handling whenever redirect is needed to be sent to the browser.
 *
 * author <a href="mailto:esitmit.edu">Eric Sit</a>
 *
 * version %I%, %G%
 * (c) Eric Sit 1999
 */
public class RedirectException extends HttpException {
    protected String location;

    /**
     * Creates a RedirectException, given an HTTP error code and a location.
     *
     * @param code HTTP error code
     * @param location new location
     *
     */
    public RedirectException (int code, String location) {
        super (code, "The document has moved <A HREF="" + 
                location + "">here</A>.");
        this.location = location;
    }

    /**
     * Sends headers telling the browser the new location of the object.
     *
     */
    public void processRequest (HttpOutputStream out) throws IOException {
        out.setHeader ("Location", location);
        super.processRequest (out);
    }
}
References


