Despina Patronas
CSE 130

Design Document Assignment 3: httpproxy

Introduction

I. **Goals and Objectives:**
This assignment is to implement *httpproxy* as a multi-threaded HTTP proxy server.

II. Usage and Scope of *httpproxy*

i. *httpproxy* responds to GET commands for files

ii. *httpproxy* reponds to malformed / erroneous requests w/o crashing.

iii. The port number is specified on the command line as a mandatory user arg.
Port number must be positive 16 bit integer. Server closes with invalid port #

iv. *Httpproxy* accepts command line args: -N threads, -r frequency.

v. *httpproxy* supports persistent connection to Clients, able to respond to multiple requests that arrive in the same connection.

vi. *httpproxy* persistently stores files in the same directory the server is run.
Ie: server can be restarted with the same files
or run on a directory that already has files.

vii. *httpproxy* exhibits HTTP protocol as a Client and Server by coordinating message passing along a web of connections.

viii. *httpproxy* keeps track of servers health periodically for port forwarding purposes

ix. *httpproxy* load balances servers work

Proxy as a Client: (Proxy<->Server)

1. Sends requests to Server via http
2. Receives responses from Server

Proxy as a Server: (Client<->Proxy)

3. Receive requests from Client
    a. Parse simple simple HTTP requests
4. Forwards valid requests to Server

(Client<->Server)

5. Sends responses from Server to Client

x. Example of HTTP Protocol

1. Client sends request to Server :
    a. Request a file: client <- proxy **(GET – file from proxy)**

2. Proxy does the following:
    Receives (a) UNSUCCESSFULLY
        i. Respond with error message
            1. Server closes current client connection
                a. Idles until another client connects

    b. Receives (a) successfully
        i. Forward client request appropriately
            1. Idles for:
                a. Another request from current client
                b. OR if Client closes connection:
                    i. Idles until another connection

    c. Port Forwards Messages to best server for load balancing
        i. Determined by healthcheck.
        Updates every -r frequency of requests

    d. Healthcheck operation:
    Healthcheck every -r frequency of responses handled.
            1. Receive logs from servers
                a. If server log is corrupt or invalid,
                skip server until next healthcheck

                b. Else If server does not respond, skip
                c. Else Determine server to port forward
                    i. Lowest entries
                    ii. Lowest failures

III. Design

httpproxy may NOT use HTTP libraries.
Httpproxy may NOT use FILE * calls (except for printing error messages)

CLIENT HTTPS REQUESTS:

i. GET: ***Assume headers are no longer than 1KiB. Body may be any size.***
***Client* connects and sends (ASCII text) request:** *Note: \r\n == CRLF == \n*

GET /Object HTTP/1.1*(\r\n)*	--request
Host: 10.0.0.5:8080*(\r\n)*	--header
Content-Length: 3*(\r\n)*	--size of body sent
(\r\n)	--empty line

1. HEADER:
   a. Ascii text
      i. Request line. *(3 components space separated parts)*
         1. commands = GET, PUT, HEAD
         2. /Object = Should match a valid object name.
         3. HTTP = version used. ***Asgn uses HTTP/1.1***

      ii. Header. *Must check host & value are present and valid*
         Must be present in all requests
         1. HOST:value *(value should be valid)*

      iii. One empty line (only CRLF) ends HEADER

ii. Responding to Client
1. If proxy detects a command it does not support (ie: NOT GET)
   a. Return a 501 "Not Implemented" to Client

2. else If proxy cannot parse header from Client.
   a. Return 400 "Bad Request" to Client

3. ***else Proxy* Response to successful client request:**

   Forward entire Client Request to a Server.
   Let Server reply directly to Client

i. Proxy GET healthcheck logs from Server:

Terms:
Healthcheck: Health check diagnoses the number of failed logs per total requests and returns that information back to client.

Server Logs: Logging is done for each request while a log file argument exists.
1. Logs contain information about the request line from client as well as content length and a hex representation of first 1K or less bytes process.

2. Logs also document failed requests.
3. Logs return a body with <errors>\n<entries>\n format

***Proxy acts as Client* connects and sends request to a Server:**

Note: no content-length or body

GET /healthcheck/Host1.1*(\r\n)*	--request
localhost: serverport*(\r\n)*	--header
(\r\n)	--empty line

***Server* Response to proxy for healthcheck:**

HTTP/1.1 200 OK*(\r\n)*
Content-Length: sizeof_file_requested(\r\n)
Last-Modified: Day, DayofMonth Month Year H:M:S GMT (\r\n\r\n)
errors(\n)
entries(\n)

Healthcheck cases for Server Status:

If Server fails to respond or connect, it can be considered offline,
If Server response code !200 or body format is malformed,
log can be considered corrupt (unaccessable)

Communication Functions (system calls):

ii. Send()
Usage: *ssize_t send(int sockfd, const void *buf, size_t len, int flags);*
Sends message to another socket when connected ie: recipient is known.
On success, returns number of bytes sent.
On error, -1 is returned, and errno is set appropriately.

iii. Recv()
Usage: *ssize_t recv(int sockfd, const void *buf, size_t len, int flags);*
Receives message from another socket when connected ie: recipient is known.
On success, returns number of bytes received.
On error, -1 is returned, and errno is set appropriately.

***iv.* Open()**
Usage: *int fd = open(const char *filename, int flags);*
If successful, returns a file descriptor for the opened file,
otherwise, returns -1

***v.* Read()**
Server uses directory which it is run to Read files from GET (sent files)
Server reads HEAD requests.
Usage: *int r_count = read(int fd, void *buff, int count);*
If successful, returns number of bytes read from file descriptor to buff
(0 for End of File (EOF)), otherwise returns -1.

shoulders: Read will be done in a loop, with constant buffer size until EOF. Note: the size which you read in is subjective and performance will vary based on your implementation.

vi. **Write()**

Server uses the directory which it is run to Write files from PUT requests

Usage: *int w_count = write(int fd, void * buff, int count);*

If successful, returns number of bytes written to file descriptor from buff, otherwise returns -1.

Design Document Diagram:

https://lucid.app/lucidchart/046cd71c-2aa7-4889-a560-2717d239ceda/view?page=0_0&invitationId=inv_f78165bd-5646-405a-a7ee-0c6639170974#