PROJECT 1 : COMPUTER NETWORKS (CS425A)

# PROXY SERVER IMPLEMENTATION

19.08.2016

—

PRAMOD CHUNDURI
ROLL NO 13221
chpramod@iitk.ac.in

IMPLEMENTED OPTIONS :

RECEIVE GET REQUESTS FROM CLIENTS
MAKE APPROPRIATE CHANGES TO GET REQUEST
SEND EDITED REQUEST TO REMOTE SERVER
FORWARD SERVER RESPONSE TO CLIENT
LIMIT ON CONCURRENT CONNECTIONS
INTERNAL SERVER ERROR FOR INVALID REQUESTS

## Overview of Design Choices

### I. Port Number on startup

Port Number is initialized on startup. So, the server expects a command of the format

```
$./proxy port_no
```

### II. 500 Internal Server Error

The proxy server throws Internal Server error for any failed parse requests or any method other than “GET”

### III. Concurrent Connections

The proxy server handles a maximum of 20 concurrent connections and any further connection request will be waiting on the existing connections. The logic is implemented in such a way that the number of childs “cannot exceed” 20, rather than strictly keeping track of number of children at a particular instant.

### IV. Buffer size of incoming requests

The buffer size has been fixed to 8024 bytes, as this has been seen as the maximum buffer size that any of the standard browsers send.

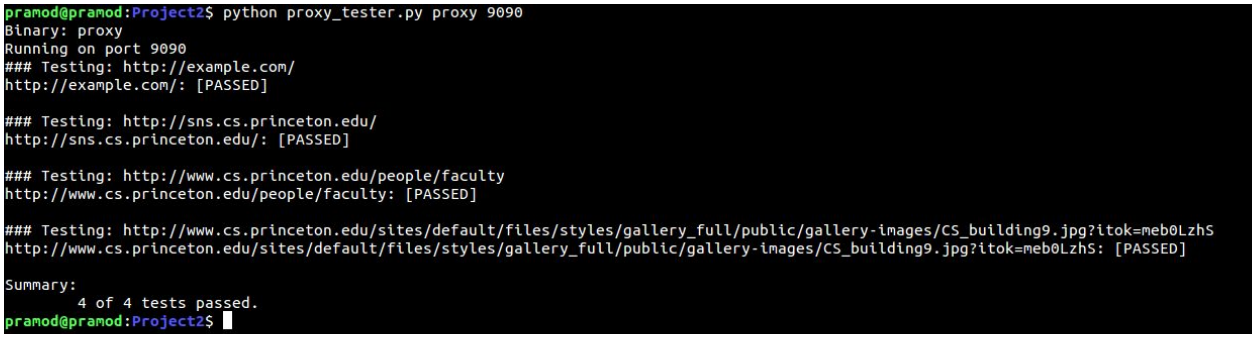
## Testing Results

### I. SUMMARY

All the given test cases have been passed on when run on an no-proxy ironport connection

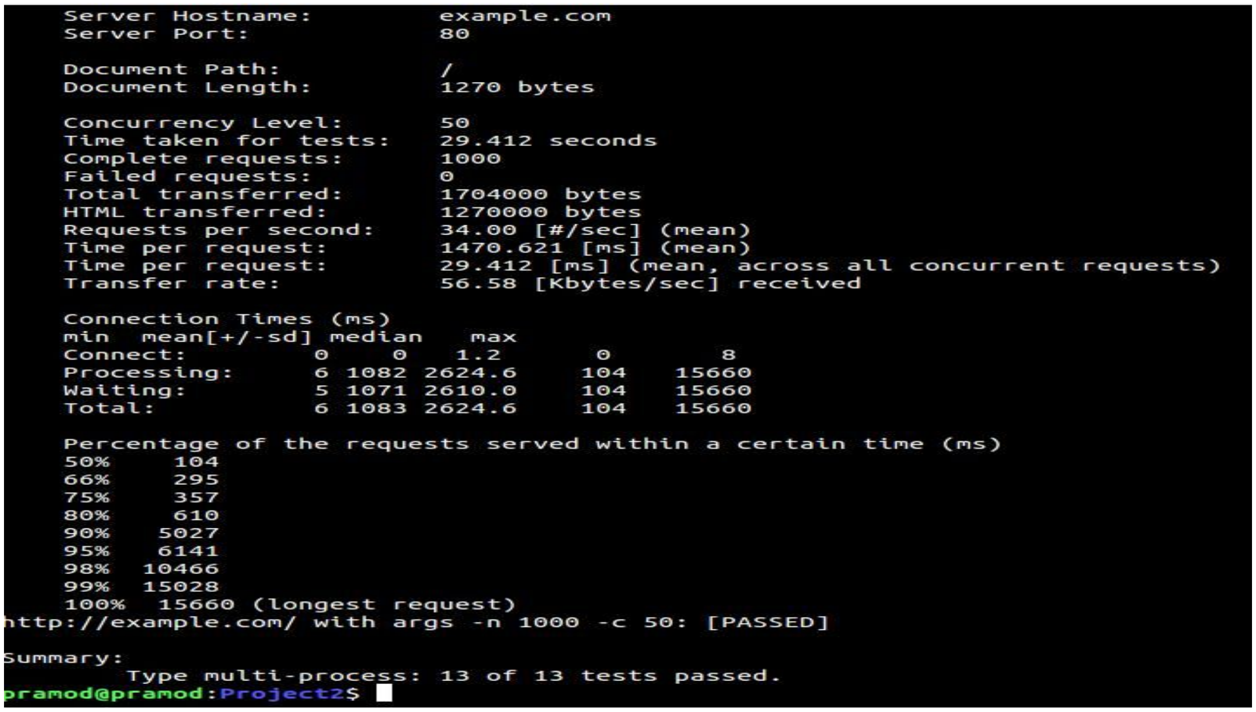
### II. SCREENSHOTS

#### proxy_tester.py test case



```
pramod@pramod:Project2$ python proxy_tester.py proxy 9090
Binary: proxy
Running on port 9090
### Testing: http://example.com/
http://example.com/: [PASSED]

### Testing: http://sns.cs.princeton.edu/
http://sns.cs.princeton.edu/: [PASSED]

### Testing: http://www.cs.princeton.edu/people/faculty
http://www.cs.princeton.edu/people/faculty: [PASSED]

### Testing: http://www.cs.princeton.edu/sites/default/files/styles/gallery_full/public/gallery-images/CS_building9.jpg?itok=meb0LzhS
http://www.cs.princeton.edu/sites/default/files/styles/gallery_full/public/gallery-images/CS_building9.jpg?itok=meb0LzhS: [PASSED]

Summary:
        4 of 4 tests passed.
pramod@pramod:Project2$
```

#### proxy_tester_conc.py test case



```
    Server Hostname:        example.com
    Server Port:            80

    Document Path:          /
    Document Length:        1270 bytes

    Concurrency Level:      50
    Time taken for tests:   29.412 seconds
    Complete requests:      1000
    Failed requests:        0
    Total transferred:      1704000 bytes
    HTML transferred:       1270000 bytes
    Requests per second:    34.00 [#/sec] (mean)
    Time per request:       1470.621 [ms] (mean)
    Time per request:       29.412 [ms] (mean, across all concurrent requests)
    Transfer rate:          56.58 [Kbytes/sec] received

    Connection Times (ms)
    min  mean[+/-sd] median   max
    Connect:        0    0   1.2      0       8
    Processing:     6 1082 2624.6    104   15660
    Waiting:        5 1071 2610.0    104   15660
    Total:          6 1083 2624.6    104   15660

    Percentage of the requests served within a certain time (ms)
    50%    104
    66%    295
    75%    357
    80%    610
    90%   5027
    95%   6141
    98%  10466
    99%  15028
    100%  15660 (longest request)
http://example.com/ with args -n 1000 -c 50: [PASSED]

Summary:
        Type multi-process: 13 of 13 tests passed.
pramod@pramod:Project2$ 
```

## Appendix (Code)

```
#include <bits/stdc++.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <signal.h>
#include <netdb.h>
#include <string>
#include <dirent.h>
#include <sys/stat.h>
#include "proxy_parse.h"
```

```
#include <sys/wait.h>
#include <arpa/inet.h>

#define MAX_PROCS 1000
#define SIZE 1024
#define MAXSIZE 8192
using namespace std;

int serve_the_client(int arg){
        int newsockid = arg;
        char *endIndex;
        char *request = (char *)malloc(SIZE*sizeof(char));
        string port;
        char* curr_ptr = request;
        int bytes_read = 0;
        int recv_curr = 0;
        bzero(request,SIZE);
        //do-while loop to receive GET request
        do{
            recv_curr = recv(newsockid,curr_ptr,MAXSIZE,0);
            bytes_read += recv_curr;
            endIndex = strstr(request, "\r\n\r\n");
            if (recv_curr == 0){
                break;
            }
            curr_ptr = curr_ptr + recv_curr;
            if (recv_curr < 0){
                printf("Error in connection.\n");
                return -1;
```

```
		}
	}while(endIndex==NULL);
	int len = strlen(request);
	if (recv_curr == 0){
		return 2;
	}
	//using the library to parse the request
	ParsedRequest *req = ParsedRequest_create();
	if (ParsedRequest_parse(req, request, len) < 0) {
		//handling internal server error
		char response_header[2048];
		sprintf(response_header,"%s 500 Internal Server 
Error\r\n",req->version);
		sprintf(response_header+strlen(response_header),"Content-Type: 
%s\r\n\r\n","text/html");
		char *curr_ptr = response_header;
		int curr = 0;
		int bytes_left = strlen(response_header);
		while (bytes_left >0){
			curr = send(newsockid,curr_ptr,bytes_left,0);
			bytes_left = bytes_left - curr;
			curr_ptr = curr_ptr + curr;
		}
		return 4;
	}
	if (strstr(req->version,"1.1")!=NULL){
		sprintf(req->version,"HTTP/1.0");
	}
	struct ParsedHeader *r = ParsedHeader_get(req, "Host");
	endIndex = strstr(request, "\r\n\r\n");
```

```
            if (r==NULL){
                        if (ParsedHeader_set(req, "host", req->host) < 0){
                                    printf("set header key not work\n");
                                    return -1;
            }
    }
    if (ParsedHeader_set(req, "Connection", "close") < 0){
                        printf("set header key not work\n");
                        return -1;
}
            int rlen = ParsedRequest_totalLen(req);
            char *string_req = (char *)malloc(rlen+1);
            bzero(string_req,rlen+1);
            if (ParsedRequest_unparse(req, string_req, rlen) < 0) {
              printf("unparse failed\n");
              return -1;
            }
            string_req[rlen]='\0';
            //Opening connection with the server to send the request
            struct sockaddr_in serverAddr;
            struct hostent *serverName;
            //socket creation
            int sockid = socket(AF_INET,SOCK_STREAM,0);
            //defining socket properties
            int port_no;
            if (req->port != NULL)
                        port_no = atoi(req->port);
            else        port_no = 80;
            serverAddr.sin_family = AF_INET;
```

```
                    serverAddr.sin_port = htons(port_no);
                    serverName = gethostbyname(req->host);
                    bcopy((char *)serverName->h_addr,(char
*)&serverAddr.sin_addr.s_addr,serverName->h_length);
                    int connect_status;
                    connect_status = connect(sockid,(struct sockaddr
*)&serverAddr,sizeof(serverAddr));
                    if (connect_status == 0){
                              curr_ptr = string_req;
                              int curr = 0;
                              int bytes_left = strlen(string_req);
                              //sending request to remote server
                              while (bytes_left >0){
                                        curr = send(sockid,curr_ptr,bytes_left,0);
                                        bytes_left = bytes_left - curr;
                                        curr_ptr = curr_ptr + curr;
                              }
                              free(string_req);
                              //sending response to client
                              char response_recvd[2048];
                              curr = 0;
                              bytes_left = sizeof(response_recvd);
                              curr_ptr = response_recvd;
                              do{
                                        bzero(response_recvd,2048);
                                        curr = recv(sockid,response_recvd,bytes_left,0);
                                        if (curr!=0)
                                                  send(newsockid,response_recvd,curr,0);
                              }while (curr > 0);
                              close(sockid);
                    close(newsockid);
            }
            else{
                    return 7;
            }
            return 0;
}

int main(int argc, char *argv[]){
        int numThreads = 0;
        int wait_status;
        int enable = 1;
        wait_status = 0;
        if (argc != 2){
                printf("The program expects two arguments, port number\n");
        }
        else{
                //socket creation
                int port_no = atoi(argv[1]);
                int sockid = socket(AF_INET,SOCK_STREAM,0);
                if (setsockopt(sockid, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) <
0)
        printf("setsockopt(SO_REUSEADDR) failed\n");
                struct sockaddr_in addr;
                struct sockaddr clientAddr;
                //defining socket properties
                socklen_t addrLen;
                addr.sin_family = AF_INET;
                addr.sin_port = htons(port_no);
                addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
    //binding the socket to the network interface
    int bind_status = bind(sockid,(struct sockaddr *)&addr,sizeof(addr));
    if (bind_status == 0){
        int listen_status = listen(sockid, SOMAXCONN);
        if (listen_status == 0){
            while (1){
                //Listen for client requests
                int newsockid = accept(sockid, &clientAddr,&addrLen);
                //Forking a child process for every accepted
connection
                //Parent process runs in an infinite loop until
termination
                while (numThreads >=MAX_PROCS){
                    numThreads--;
                    wait(&wait_status);
                }
                pid_t pid = fork();
                numThreads++;
                if (pid == 0){
                    //Call to the function serving the client
                    int status = serve_the_client(newsockid);
                    //Appropriate messages on the command line,
for different exit statuses
                    if (status == -1){
                        printf("An error occured\n");
                    }
                    if (status == 0){
                        // printf("Successfully served a client\n");
                    }
                    if (status == 1){
```

```
                                        printf("Request to close connection from
the client\n");
                                    }
                                    if (status == 2){
                                        printf("Connection closed by the
client\n");
                                    }
                                    fflush(stdout);
                                    close(newsockid);
                                    exit(0);
                                }
                                else if (pid > 0){
                                    //parent process continues in while loop
indefinitely, until Ctrl + C is found
                                    close(newsockid);
                                    continue;
                                }
                                else
                            {
                              // fork was not successful
                              printf("fork() failed!\n");
                              return 1;
                            }
                        }
                        close(sockid);
                    }
                }
                else{
                    char errMsg[128] = "Could not bind to the specified port\n\0";
                    fwrite(errMsg,1,strlen(errMsg),stderr);
                    close(sockid);
                }
            }
            return 0;
}
```