Proxy Server

Lab Project Report

Ruben K. Lopez

CMPE 156: Network Programming, Spring 2014

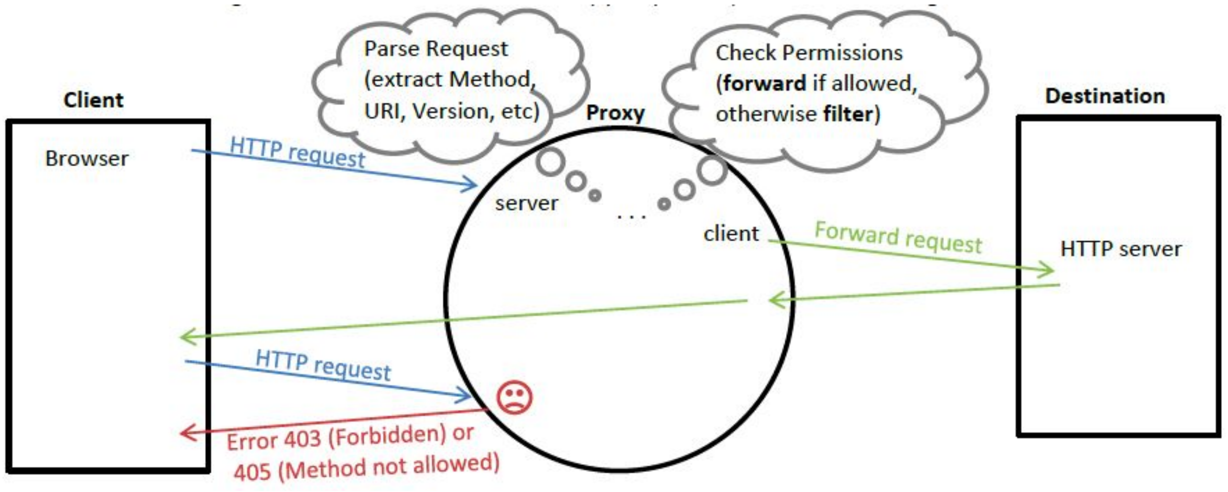
Description

For our Final Project, we were asked to make a proxy server that:

1. Parses HTTP requests from a web browser,
2. checks if the request is permitable (with an Access Control List),
3. and forwards the request to its destination if allowed.

The project is programmed in C, and ran in a UNIX environment. The web browser is configured to use a proxy with an IP address and port number (e.g. IP: 127.0.0.1, Port: 8080).

Block Diagram


Client
Browser
HTTP request
Parse Request (extract Method, URI, Version, etc)
Proxy
server
. . .
Check Permissions (**forward** if allowed, otherwise **filter**)
client
Forward request
Destination
HTTP server
HTTP request
Error 403 (Forbidden) or 405 (Method not allowed)

Software Design

When ProxyServer.c is ran, it parses incoming HTTP requests (for relevant info) and forwards them to their destinations if allowed by the access control mechanism (i.e. by checking "permitted-sites.txt"). It should then forward any data received from the destination back to the BROWSER(client). The proxy server will also Print and Log information for each request, and will return error response codes to the BROWSER (if we Filter a request).

TCP connections on the BROWSER side are managed with the “connected” socket descriptor, which gets established with the return value of our first *accept()*. This happens in the Parent process, before we start forking.

```
}else if( getpid() == parent_pid ){

        /**** This is the PARENT process (recursive) ****/

        /* listen (i.e. wait) for incomming connection */
        listen(sockfd, 3);

        /* accept() connection and assign to connected descriptor */
        connected_fd = accept(sockfd, (struct sockaddr *)&client_addr, &addr_len);
        if(connected_fd < 0)
                error("ERROR on accept()");

        /* recursion (fork again) */
        handleConcurrentConnections();

        closeSocks();
        exit(0);
}
```

On the SERVER (or destination) side, the TCP connection is managed with our “temporary” socket descriptor, which gets created (and closed) every time we Forward a Request (and get the server’s reply). This happens in the child process.

```
if(pid == 0){
        /**** This is the CHILD process ****/
        /* evaluate HTTP Request (parse Header) and
          check if its a valid Method: GET, HEAD, or POST */
        isValidMethod = parseRequest();         //returns true(1) or false(0)
        if(isValidMethod == 1){
                /* check if destination is Forbidden */
                checkPermissions(Host);

                /* forward to destination */
                forwardRequest();
                writeToLog("FORWARD");

        }else{
                writeToBrowser("405");
        }

        closeSocks();
        exit(0);
```

Concurrency is supported by forking.

As you can see in the block diagram, the steps involved in **processing a request** (from the browser) include:

1. Reading from the Browser-side-socket

```
/* Read HTTP request */
if( numbytes = read(connected_fd, buf, MAX_SIZE-1) < 0)
        error("ERROR on read");
```

2. Parse-ing the HTTP Request (to extract the Method, URI, Version, Host, and Path).
3. Checking permissions
4. If PERMITTED, Write to Server-side-socket

```
//WRITE request line to temp_sock (i.e. the HTTP server)
if((numbytes = write(temp_sock, request, strlen(request)) < 0))
                error("write");
```

5. Read reply from Server-side-socket, and Write reply to Browser-side-socket.

```
/***** PROCESS REPLY (read from server, write to browser) ****/
//READ webpage from temp_sock until connection closes (i.e. EOF)
while ((n = read(temp_sock, reply, MAX_SIZE)) > 0) {
        reply[n] = '\0';

        printf("[WRITE]: writing reply to Browser...\n");
        //WRITE it to connected fd (i.e. the Browser)
        if((numbytes = write(connected_fd, reply, MAX_SIZE) < 0))
                error("write");

        shutdown(temp_sock, SHUT_RDWR);
}
```

6. If NOT PERMITTED, Write to Browser-side-socket (Error 403 or 405).

My design deals with **errors** by:

1. Checking for a -1 return code on a read() or write().
2. Using *shutdown(sock, SHUT_RDWR)* to essentially close a connection, so that we can prevent getting stuck in a read()-loop when trying to resolve a webpage (e.g. when we're trying to read the Server's reply).

Usage:

usage: *./proxyServer <port num> <access list>*
example: *./proxyServer 8080 permitted-sites.txt*

Example Setup:
(Tested with Firefox Browser)
Browser**:**
First, configure Firefox to “manually” use a proxy.
Specify the IP address(or hostname) and Port (e.g. IP: 127.0.0.1, Port: 8080)
(e.g. IP: galileo-2.soe.ucsc.edu, Port: 8080)
Then exit the browser.

Terminal:
Next, run the proxy server:
./proxyServer 8080 permitted-sites.txt

Browser:
Now open the browser and try connecting to these sites:
example.com
classes.soe.ucsc.edu/cmpe150/Fall99
classes.soe.ucsc.edu/cmpe110/
classes.soe.ucsc.edu/cmpe151/Spring14/index.html

APPENDIX
ProxyServer.C

```
/****************************************************************
 Author: rklopez@ucsc.edu
 Proxy Server (Phase 1):

 this proxy server parses incoming HTTP requests
 and forwards them to their destinations if allowed by the access
 control  mechanism. (The server checks the Access Control List:
 "permitted-sites.txt")

 It then forwards any data received from the
 destination back to the BROWSER(client).

 the proxy server will Print and Log information for each request.
 It also returns error response codes to the BROWSER.

        usage:    ./proxyServer <port num> <access list>

        ex:                 ./proxyServer 8080 permitted-sites.txt
****************************************************************/

#include <stdio.h>
```

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <time.h>

static char **ACCESS_LIST;
static char *Host;
static char *Path;
static char *method, *version;
static int PORT_NUM, parent_pid;
static int URLcount=0, requestNum;

/* size Definitions */
#define MAX_SIZE 10000

/* socket descriptors and address structs */
static int sockfd, connected_fd;
struct sockaddr_in serv_addr, client_addr, dest_addr;

static socklen_t addr_len;
static int numbytes;

/* functions */
void createSocket();
void getAccessInfo(char *filename);
void checkPermissions(char *hostname);

void error(const char *msg)
{
  perror(msg);
  exit(1);
}

int main(int argc, char *argv[])
{
        //free these when done!
        ACCESS_LIST = (char**)calloc(MAX_SIZE, sizeof(char*));
        Host                = (char*)calloc(256, sizeof(char));
        Path                = (char*)calloc(1024, sizeof(char));
        method              = (char*)calloc(32, sizeof(char));
        version             = (char*)calloc(32, sizeof(char));

        requestNum= 0;

        /*          Error-Check Input Arguements */
        evaluateArgs(argc,argv);

        /*          Read URLs from "permitted-sites" */
        getAccessInfo(argv[2]);

        /* create initial socket and bind() */
        createSocket();
        /* listen() for incoming connections */
        listen(sockfd, 3);
        printf("listening...\n");

        addr_len = sizeof(client_addr);
        /* accept() connection and assign to connected descriptor */
        connected_fd = accept(sockfd, (struct sockaddr *)&client_addr, &addr_len);
        if(connected_fd < 0)
                error("ERROR on accept()");

        printf("Browswer accepted! \n");

        /* store Parent Process ID */
        parent_pid = getpid();
        printf("parent pid = %i \n", getpid()); //show process ID of calling process

        /* Handle forks...(for concurrency) */
        handleConcurrentConnections();

        /* close sockets */
        closeSocks();

        return 0;
}


/******** handleConcurrentConnections() ************
 starts a fork() and checks if the process is the
 CHILD or PARENT.
 ***************************************************/
int handleConcurrentConnections()
{
        int   pid;
        int   isValidMethod = 0;
        requestNum++;


        pid = fork();
        if (pid < 0)
                error("ERROR on creating fork");
        //show process ID of calling process
        if( getpid() == parent_pid)
                printf("current pid = %i (parent)\n", getpid());
        else
                printf("current pid = %i (child)\n", getpid());

        if(pid == 0){
                /**** This is the CHILD process ****/
                /* evaluate HTTP Request (parse Header) and
                   check if its a valid Method: GET, HEAD, or POST */
                isValidMethod = parseRequest();          //returns true(1) or false(0)
                if(isValidMethod == 1){

                        /* check if destination is Forbidden */
                        checkPermissions(Host);

                        /* forward to destination */
                        forwardRequest();
                        writeToLog("FORWARD");

                }else{
                        writeToBrowser("405");
                }
                closeSocks();
                exit(0); //terminate process?

        }else if( getpid() == parent_pid ){

                /**** This is the PARENT process (recursive) ****/

                /* listen (i.e. wait) for incomming connection */
                listen(sockfd, 3);

                /* accept() connection and assign to connected descriptor */
                connected_fd = accept(sockfd, (struct sockaddr *)&client_addr, &addr_len);
                if(connected_fd < 0)
                        error("ERROR on accept()");

                /* recursion (fork again) */
                handleConcurrentConnections();

                closeSocks();
                exit(0);  //terminate process?
        }


}

/******** parseRequest() ***************************
 Extract relevant info from the HTTP request and
 return 1 (if method is valid) or 0 (if it isn't).
 ****************************************************/
int parseRequest() {

        int isValid=0;
        char URI[1024];
        char * ptr;
        char buf[MAX_SIZE];
        bzero(buf,MAX_SIZE);

        /* Read HTTP request */
        if( numbytes = read(connected_fd, buf, MAX_SIZE-1) < 0)
                error("ERROR on read");

        printf("[READ]: packet contains \"%s\"\n", buf);

        sscanf(buf, "%s %s %s", method,URI,version);
        printf("scanned Method: '%s'\n", method);
        printf("scanned URI: '%s'\n", URI);
        printf("scanned Version: '%s'\n", version);


        //Get the Host
        int i = 0;
        ptr  = strstr( buf, "Host: " ); //point to first occurence of "Host: "
        ptr = ptr + strlen("Host: ");
        while(ptr[i] != '\r') {
                Host[i] = ptr[i];
                i++;
        }
        printf("Host string: '%s'\n", Host);

        //Get the Path
        i = 0;
        ptr  = strstr( buf, Host);
        ptr = ptr + strlen(Host); //set ptr to end of host name
        while(ptr[i] != ' ') {
                Path[i] = ptr[i];
                i++;
        }
        printf("Path string: '%s'\n", Path);

        /* Check if valid HTTP request method (GET, HEAD, or POST). */
        if(strcmp(method, "GET") == 0
                || strcmp(method, "HEAD") == 0
                || strcmp(method, "POST") == 0){ isValid = 1;}

        return isValid;
}

/******** forwardRequest() **************************
 WRITE request to the destination server and process
 the reply (i.e READ) in a while loop, until EOF
 ***************************************************/
int forwardRequest()
{
        char **pptr;
        char str[50];
        struct hostent *hptr;

        // get hostent of HTTP destination
        if ((hptr = gethostbyname(Host)) == NULL) {
                fprintf(stderr, " gethostbyname error for host: %s\n",Host);
                exit(1);
        }
        printf("", hptr->h_name);

        //get IP addresses in human-readable form (inet_ntop)
        if (hptr->h_addrtype == AF_INET
           && (pptr = hptr->h_addr_list) != NULL) {
                printf("",
                     inet_ntop(hptr->h_addrtype, *pptr, str,
                                        sizeof(str)));
        } else {
                fprintf(stderr, "Error call inet_ntop \n");
        }

        //Create new socket to connect to HTTP server
        int temp_sock = socket(AF_INET, SOCK_STREAM, 0);
        bzero(&dest_addr, sizeof(dest_addr));
        dest_addr.sin_family = AF_INET;
        dest_addr.sin_port = htons(80);
        //store this IP addr in the destination addr
        inet_pton(AF_INET, str, &dest_addr.sin_addr);

        /* connect() */
        connect(temp_sock, (struct sockaddr *) &dest_addr, sizeof(dest_addr));


        char *request = calloc(MAX_SIZE, sizeof(char));
        char *reply = calloc(MAX_SIZE, sizeof(char));
        ssize_t n;

        /**** FORWARD 'GET' REQUEST (with Host and Path) ****/
```

```
		sprintf(request,
				"GET %s HTTP/1.1\r\n"
				"Host: %s\r\n"
				"User-Agent: Proxy Server (part1) - Ruben Lopez \r\n"
				"\r\n", Path, Host);

		printf("========= Forwarding HTTP Request[%d]... =========\n",requestNum);
		printf("[WRITE]: packet contains \n\"%s\"\n", request);

		//WRITE request line to temp_sock (i.e. the HTTP server)
		if((numbytes = write(temp_sock, request, strlen(request)) < 0))
						error("write");


		printf("[READ]: Processing Server's reply...\n");
		/***** PROCESS REPLY (read from server, write to browser) ****/
		//READ webpage from temp_sock until connection closes (i.e. EOF)
		while ((n = read(temp_sock, reply, MAX_SIZE)) > 0) {
				reply[n] = '\0';

				printf("[WRITE]: writing reply to Browser...\n");
				//WRITE it to connected fd (i.e. the Browser)
				if((numbytes = write(connected_fd, reply, MAX_SIZE) < 0))
						error("write");

				shutdown(temp_sock, SHUT_RDWR);
		}

		printf("========= Processing Finished. Request[%d] is Done. ========= \n",requestNum);
		free(request);
		free(reply);
		close(temp_sock);

}

/******** writeToLog() *******************************
 append Time, method, version, action, etc. to Log File
 *****************************************************/
int writeToLog(char *action) {


		time_t rawtime;
		struct tm * timeinfo;
		char temp[1024];
		FILE *pFile;

		time (&rawtime);
		timeinfo = localtime (&rawtime);
		char *time = asctime(timeinfo);
		time[strlen(time)-1] = '\0';

		if((pFile = fopen ("log.txt", "a")) < 0 ) {
				printf("error opening %s\n", "log.txt");
				error("writeToLog");
		}
		//print statement
		sprintf(temp,		"[Time: %s] [Req: %s] [Vers: %s] [Requesting Host: %s] [Server: %s] [URI: %s] [Action:
%s]\n", time, method, version, inet_ntoa(client_addr.sin_addr), Host, Path, action);

		printf("%s", temp);
```

```
                fwrite(temp, strlen(temp), 1, pFile);

                fclose( pFile );

}


/******** createSocket() **************************
 Handles Socket Setup (family, ip, port).
 note: we want a socket for each client connection.
 **************************************************/
void createSocket()
{
    /* SOCKET(): ALLOCATE SOCKET DATA STRUCTURE (TCP) */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
       error("ERROR opening socket");

                int optval = 1;
                setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

                /* set serv_addr to zeros */
                memset(&serv_addr,0,sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(PORT_NUM);

    /* BIND(): ASSIGN AN ADDR TO EXISTING SOCKET */
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
             sizeof(serv_addr)) < 0)
             error("ERROR on binding");

}



/******** evaluateArgs() *********************
 Check Usage. Check filename. Store Port Num.
 ********************************************/
int evaluateArgs(int argc, char **argv)
{

                char info_filename[255];

    if (argc != 3) {
                                fprintf(stderr,"Arguements are incorrect.\n");
                fprintf(stderr,"Usage:     %s <port num> <filename(of access list)>\n", argv[0]);
                exit(0);
    }

    /* check filename of Access List*/
    if(strcmp(argv[2], "permitted-sites.txt") != 0){
                                fprintf(stderr,"File not found: '%s'.\n", argv[2]);
                                fprintf(stderr,"Usage:     %s <port num> <filename(of access list)>\n", argv[0]);
                                fprintf(stderr,"Hint: use filename 'permitted-sites.txt'\n");
                                exit(0);
                }

                PORT_NUM = atoi(argv[1]);
                printf("Using port: %d and access list: '%s' \n", PORT_NUM,argv[2]);
```

```
}

/******** getAccessInfo() ************************
 Obtains the permitted-sites from the list, and
 stores them in a string array (char **)
 *************************************************/
void getAccessInfo(char *filename)
{

	FILE *pFile;
	char word[255];

	if((pFile = fopen (filename, "r")) < 0 ) {
		error("ERROR opening local file");
	}

	/* disect file for URLs */
	while( fscanf(pFile, "%s", word) != EOF ) {
		ACCESS_LIST[URLcount] = strdup(word)
		URLcount++;
	}
	fclose(pFile);

	int i;
	for(i=0; i < URLcount; i++){
		printf("accesslist[%d] = '%s' \n", i, ACCESS_LIST[i]);
	}
}

/******** checkPermissions ************************
 check if HOSTNAME is a permitted-site
 *************************************************/
void checkPermissions(char *hostname)
{
	printf("checking permissions... ");
	int i;
	for(i = 0; i < URLcount; i++)
		if(strcmp(hostname, ACCESS_LIST[i]) == 0){
			printf("  PERMITTED :)\n");
			return;
		}

	printf("NOT PERMITTED!\n");
	writeToBrowser("403"); //forbidden URL

}

/******** writeToBrowser ************************
 Returns error code to the client(i.e. the browser)
 *************************************************/
int writeToBrowser(char err[255]) {

	char forbidden[255] = "<html>\r\n<h1>403 Forbidden</h1>\r\n<html>";
	char methodNot[255] = "<html>\r\n<h1>405 Method not allowed</h1>\r\n<html>";


	if(strcmp(err, "403") == 0) {
		write(connected_fd, forbidden, strlen(forbidden));
		printf("!!!!!! [WRITE]: to-Browswer: '403 Forbidden' !!!!!!\n");
		shutdown(connected_fd, SHUT_RDWR);
```

```
                writeToLog("403 - FILTER");
                closeSocks();
                exit(0);
        }

        if(strcmp(err, "405") == 0) {
                write(connected_fd, methodNot, strlen(methodNot));
                printf("!!!!!! [WRITE]: to-Browswer: '405 Method not allowed' !!!!!! \n");
                shutdown(connected_fd, SHUT_RDWR);
                writeToLog("405 - FILTER");
                closeSocks();
                exit(0);
        }

}

int closeSocks()
{
        free(ACCESS_LIST);
        free(Host);
        free(Path);
        free(method);
        free(version);
        close(sockfd);
        close(connected_fd);
}
```

Permitted Sites.txt

```
example.com
example.org
www.ucsc.edu
ucsc.edu
www.soe.ucsc.edu
soe.ucsc.edu
courses.soe.ucsc.edu
classes.soe.ucsc.edu
users.soe.ucsc.edu
```