

Proxy HTTP

Cours:	INFOM446 - Systèmes d'exploitation: étude de cas
Professeur:	Jean RAMAEKERS {jra@info.fundp.ac.be}
Assistant 1:	Jean-François Wauthy {jfw@info.fundp.ac.be}
Assistant 2:	Geoffrey Miche {gmi@info.fundp.ac.be}
Année académique:	Master 1 en informatique 2007-2008
Auteur:	Quoc Anh LE
Email:	lequocanh@info.fundp.ac.be

Mise à jour: August 5, 2008

Contents

1	Introduction	2
2	Architecture du système	3
2.1	Architecture générale	3
2.2	Vérification des données disponibles à traiter	5
2.3	Traitement des données disponibles aux sockets	6
3	Règles de filtrage	7
3.1	Chargement dynamique des modules	7
3.2	Fichier de filtrage	7
4	Chaîne des proxies	8
5	Conclusion	9
6	Démonstration	10
6.1	Lancer le proxy serveur	10
6.2	Configurer le navigateur	10
6.3	Chaîne des proxies	11
7	Code	12
7.1	Proxy serveur.c	12
7.2	Module du filtrage.c	31
7.3	Fichier de la configuration du filtrage.txt	33
7.4	Scrip makefile	33

Introduction

Ce document constitue le rapport du travail pratique réalisé dans le cadre du cours de Systèmes d'exploitation: étude de cas.

Dans ce rapport, tous les problèmes rencontrés lors de la réalisation d'un serveur proxy HTTP seront abordés.

Les codes sources seront aussi adjoints à la fin du rapport.

Architecture du système

2.1 Architecture générale

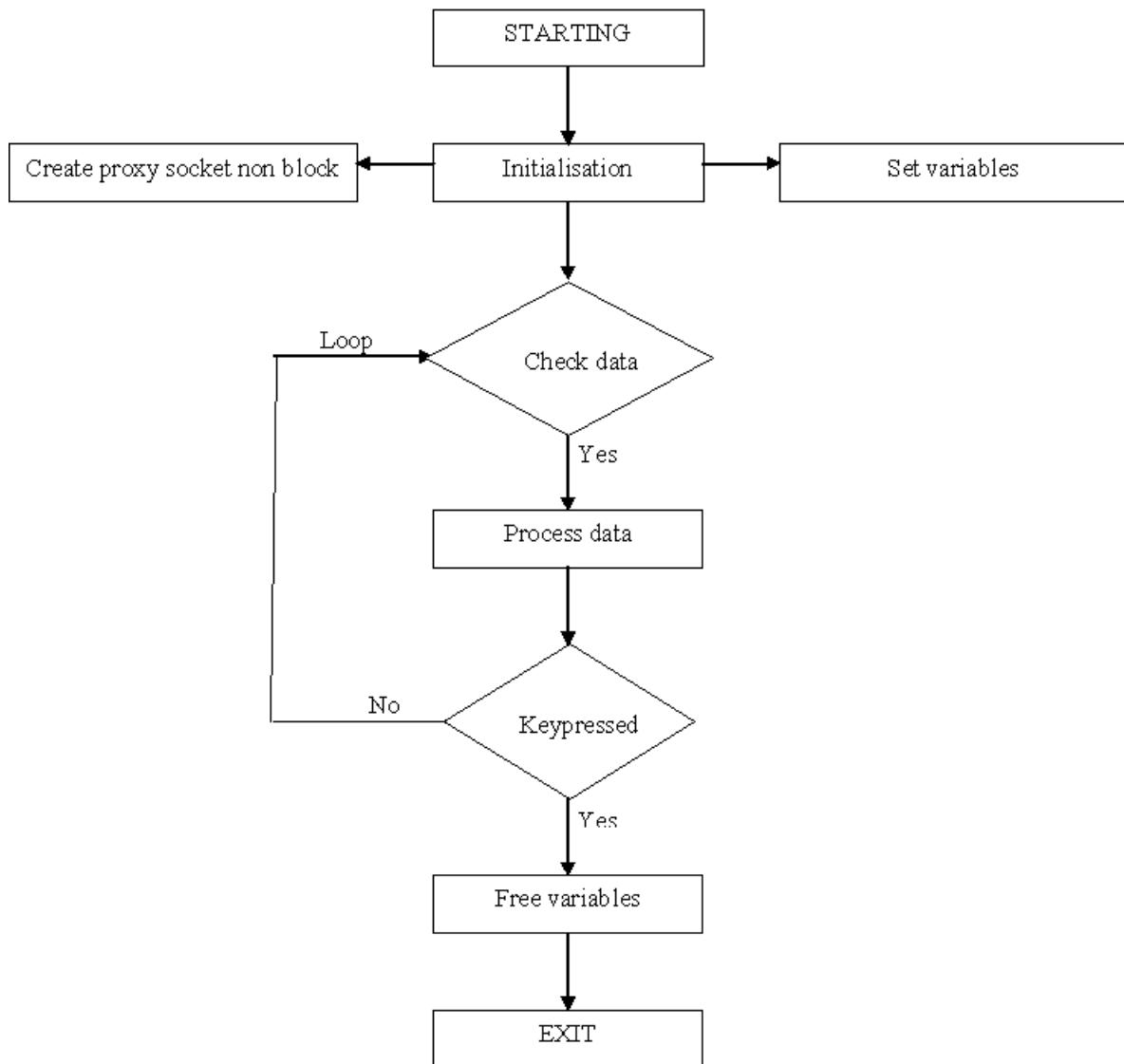


Figure 2.1: Architecture générale du HTTP serveur proxy

L'illustration présente parfaitement le cours du programme.

Dans ce programme, pour gérer facilement les connexions, chaque connexion correspondra à une structuration qui se compose de 8 attributs suivants:

1. int client: identité du client
2. int server: identité du serveur

3. char cBuf[BUF_MAX]: la réserve maximale des données chez client pour la réception
4. char sBuf[BUF_MAX]: la réserve maximale des données chez serveur pour la réception
5. int cBuf_avail: l'index des données totales disponibles chez client pour l'envoi
6. int cBuf_written: l'index des données que le client ont déjà envoyées
7. int sBuf_avail: l'index des données totales disponibles chez serveur pour l'envoi
8. int sBuf_written: l'index des données que le serveur ont déjà envoyées

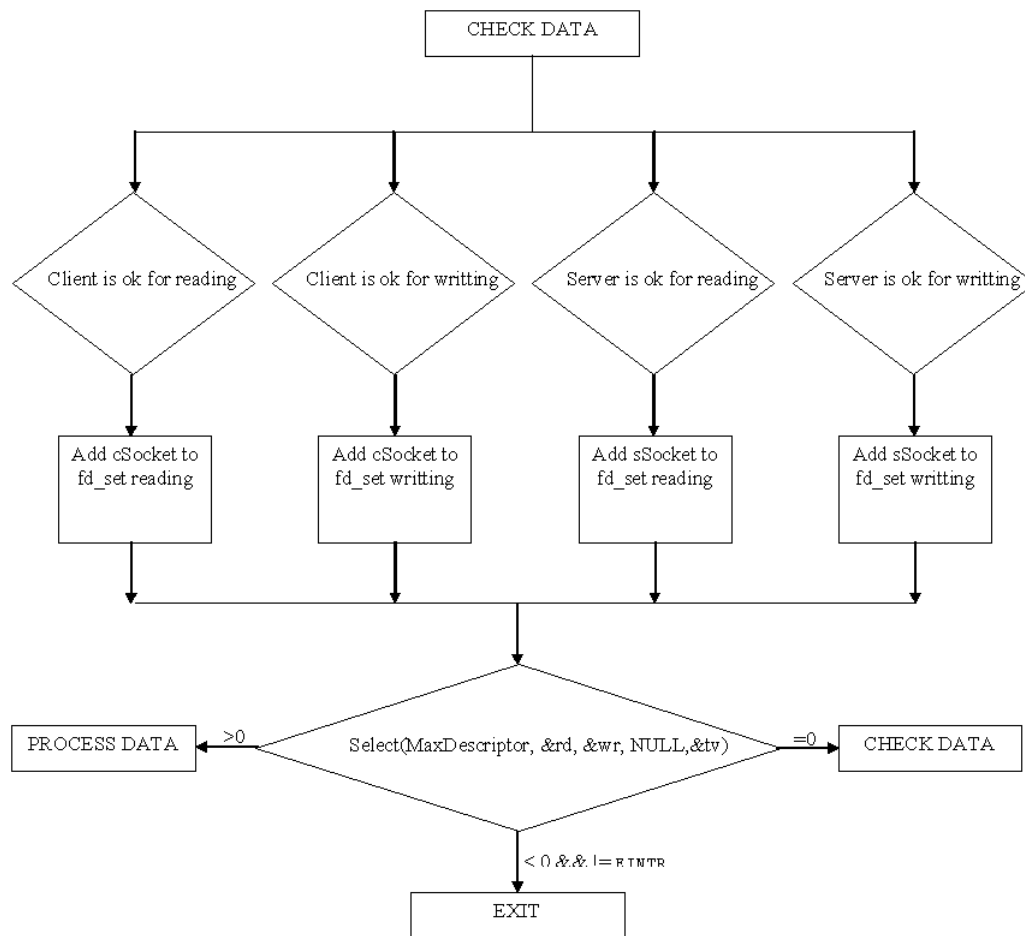
L'explication en détail de ces variables seront décrites dans les sections plus tard. Il faut savoir que c'est la structuration la plus importante pour la réalisation des sockets non bloquant.

Au début, toutes les variables sont établies à zéro. Si l'utilisateur n'indique pas la porte d'attente du serveur proxy et la porte d'attente du serveur remote, alors elles seront respectivement 3005 et 80. Le socket non bloquant du serveur proxy est seul en ce moment.

Le programme s'exécute avec un boucle qui réalise deux processus principaux: Vérifier les données disponibles à traiter chez sockets et traiter ces données si possible.

Le programme se terminera quand une tape sur la touche d'entrer du clavier.

2.2 Vérification des données disponibles à traiter



Client is ok for reading: `Client_Buffer_available < BUF_SIZE`
 Client is ok for writing: `Server_Buffer_available - Server_Buffer_written > 0`
 Server is ok for reading: `Server_Buffer_available < BUF_SIZE`
 Server is ok for writing: `Client_Buffer_available - Client_Buffer_written > 0`

Figure 2.2: Vérification des données disponibles à traiter

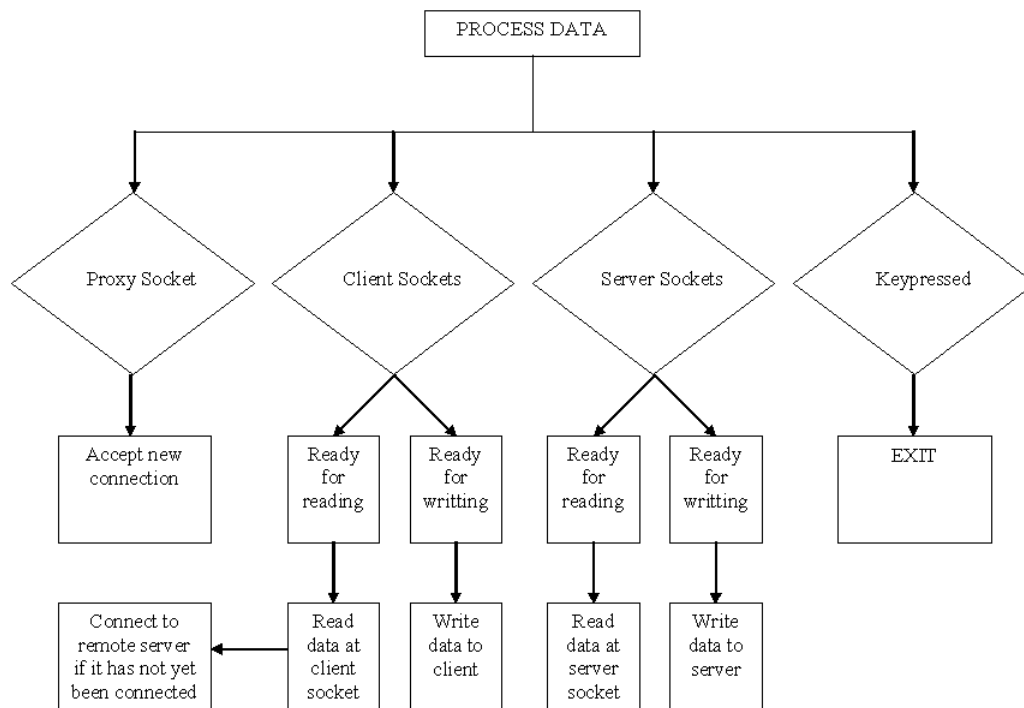
Pour diminuer le nombre des sockets à vérifier, on entre seulement les sockets potentiels à la liste pour la vérification des données disponibles.

Ici, on distingue deux type des données à vérifier, pour l'envoi et pour la réception.

Pour un client, si sa mémoire-tampon n'est pas encore pleine, c'est-à-dire qu'elle peut recevoir plus les données, donc le client sera mis à la liste pour vérifier les données d'arrivée. Si son index des données disponibles est plus son index des données déjà envoyées, c'est-à-dire qu'il peut encore envoyer les données, il sera mis à la liste pour vérifier l'envoi. Idem pour les serveurs.

Le socket proxy est toujours mis à la liste pour vérifier une nouvelle connexion. On vérifie aussi une tape sur la touche d'entrer du clavier.

2.3 Traitement des données disponibles aux sockets



Read data at client socket: `r = read(client, cBuf + cBuf_avail, BUF_SIZE - cBuf_avail);`
`cBuf_avail = cBuf_avail + r;`

Write data to server socket: `w = write(server, cBuf + cBuf_written, cBuf_avail - cBuf_written);`
`cBuf_written = cBuf_written + w;`

Read data at server socket: `r = read(server, sBuf + sBuf_avail, BUF_SIZE - sBuf_avail);`
`sBuf_avail = sBuf_avail + r;`

Write data to client socket: `w = write(client, sBuf + sBuf_written, sBuf_avail - sBuf_written);`
`sBuf_written = sBuf_written + w;`

Connecto remote server: This server is maybe a http server or another proxy.
 Before connecting, the request will be applied by a http filter

Figure 2.3: Traitement des données disponibles aux sockets

Après la commande SELECT, si les données sont disponibles à traiter, ce processus sera réalisé.

Si une requête arrive chez socket proxy, on crée une nouvelle connexion et l'associe avec la structuration abordé à la première partie.

Si l'utilisateur tape sur la touche d'entrer du clavier, le programme libère toutes les ressources alloués durant son exécution avant de se terminer.

Parce qu'un socket peut envoyer ou recevoir une différente quantité des données, on doit utiliser la quantité réelle d'envoi ou de la réception pour calculer les paramètres des clients et des serveurs. Cela permet assurer l'intégrité des données d'envoi et de la réception.

La formule est calculée comme dans le figure.

Règles de filtrage

3.1 Chargement dynamique des modules

Afin de charger dynamiquement les règles de filtrage, j'ai utilisé les fonctions *dlopen*, *dlsym* et *dlclose* qui permettent de charger dynamiquement une librairie à l'exécution du programme.

Le but du module est vérifier les requêtes entrées si elles sont satisfaites les règles de filtrage ou non. Ce sont trois règles suivants:

1. Blocage de certains navigateurs
2. Blocage d'URL contenant au moins un mot-clé interdit
3. Blocage de pages sur base de leur nom de domaine. Par exemple, `www.info.fundp.ac.be` et `leibniz.info.fundp.ac.be` seront bloqué si `info.fundp.ac.be` est dans la liste des noms de domaine refusés. Par contre, les requêtes ver `www.fundp.ac.be` seront acceptées.

La requête est analysée afin de fournir les informations correspondantes avec les règles. Ici, elles sont le nom de hôte, le nom de navigateur et le contenu complète de l'adresse du site demandé.

3.2 Fichier de filtrage

La configuration des différentes règles de filtrage est stockée dans un fichier qui s'appelle *filter.txt*. Les règles qui sont enregistrés dans ce fichier doivent être observés strictement comme la suivant:

- <Navigators> IE Elisa <Navigators>
- <Keywords> sex prof <Keywords>
- <Domains> info.fundp.ac.be lesoir.be <Domains>

Chaîne des proxies

Ce server pourra être déployé de manière chaînée avec d'autres serveurs proxy.

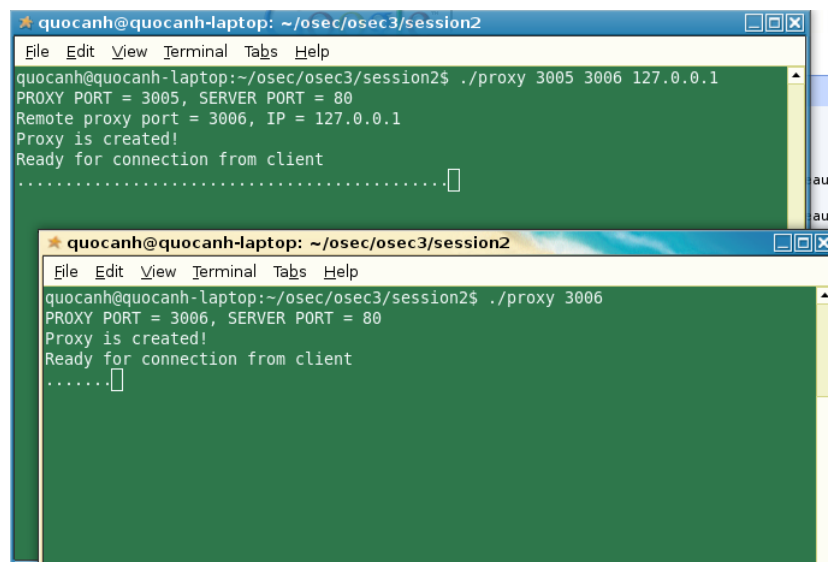
Alors, au lieu de faire un appel le serveur remote WEB correspondant avec la requête, le programme convoque un autre serveur proxy et envoie la requête à ce serveur.

Donc, on doit fournir les informations du deuxième serveur proxy en incluant une porte et son adresse lors de lancement du programme.

Les autres procédures se passent comme normal.

J'ai déjà testé sa capacité de façon que j'ai lancé deux serveurs proxy dans une même machine. Le premier serveur proxy va connecter avec le deuxième serveur proxy, et le deuxième serveur proxy marche comme normal.

Voilà un exemple du lancement des deux serveurs proxy:



```
quocanh@quocanh-laptop: ~/osec/osec3/session2
File Edit View Terminal Tabs Help
quocanh@quocanh-laptop:~/osec/osec3/session2$ ./proxy 3005 3006 127.0.0.1
PROXY PORT = 3005, SERVER PORT = 80
Remote proxy port = 3006, IP = 127.0.0.1
Proxy is created!
Ready for connection from client
.....

quocanh@quocanh-laptop: ~/osec/osec3/session2
File Edit View Terminal Tabs Help
quocanh@quocanh-laptop:~/osec/osec3/session2$ ./proxy 3006
PROXY PORT = 3006, SERVER PORT = 80
Proxy is created!
Ready for connection from client
.....
```

Figure 4.1: Lancer le proxy serveur

Conclusion

Ce programme marche bien avec toutes les fonctions abordées dans l'énoncé sous l'environnement Linux (les autres environnement ne sont pas encore testés).

On peut facilement modifier le module du filtrage et aussi modifier le fichier de configuration pour ajouter plusieurs règles différents.

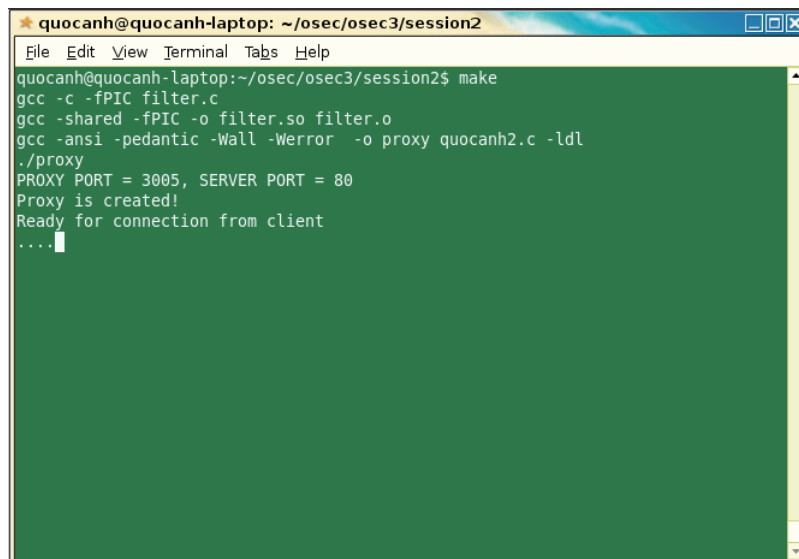
Pour le code source, le programme observe strictement les règlements de l'énoncé. Tous les sockets sont non bloquant et le programme appelle seulement une fois la commande "Select()".

Les ressources allouées dynamiques durant son exécution sont bien libérés.

Faire attention: Il faut exporter le chemin LD_LIBRARY_PATH avant de lancer le programme.

Démonstration

6.1 Lancer le proxy serveur



```
quocanh@quocanh-laptop: ~/osec/osec3/session2
File Edit View Terminal Tabs Help
quocanh@quocanh-laptop:~/osec/osec3/session2$ make
gcc -c -fPIC filter.c
gcc -shared -fPIC -o filter.so filter.o
gcc -ansi -pedantic -Wall -Werror -o proxy quocanh2.c -ldl
./proxy
PROXY PORT = 3005, SERVER PORT = 80
Proxy is created!
Ready for connection from client
....
```

Figure 6.1: Lancer le proxy serveur

6.2 Configurer le navigateur

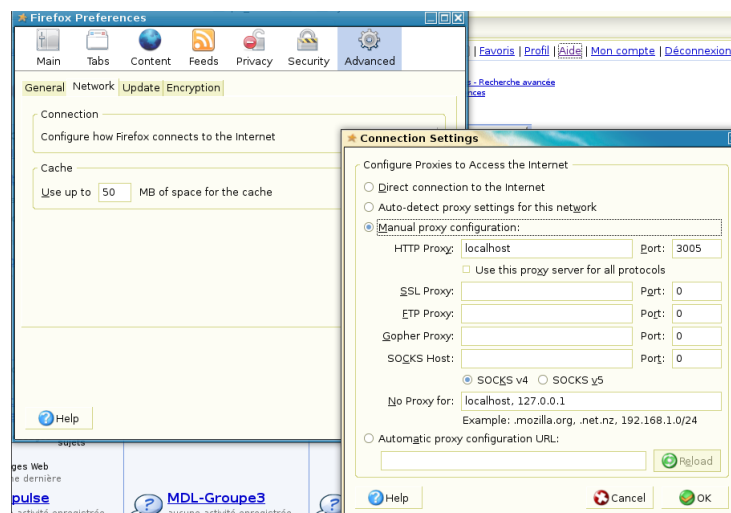


Figure 6.2: Configurer le navigateur (firefox)

6.3 Chaîne des proxies

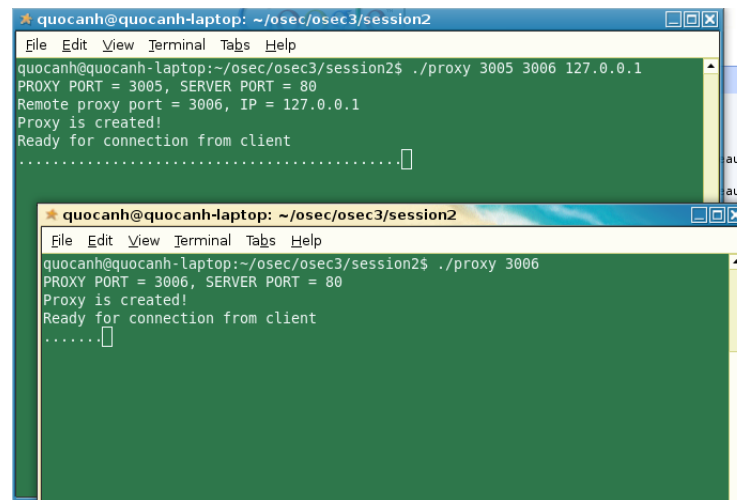


Figure 6.3: Chaîne des proxies

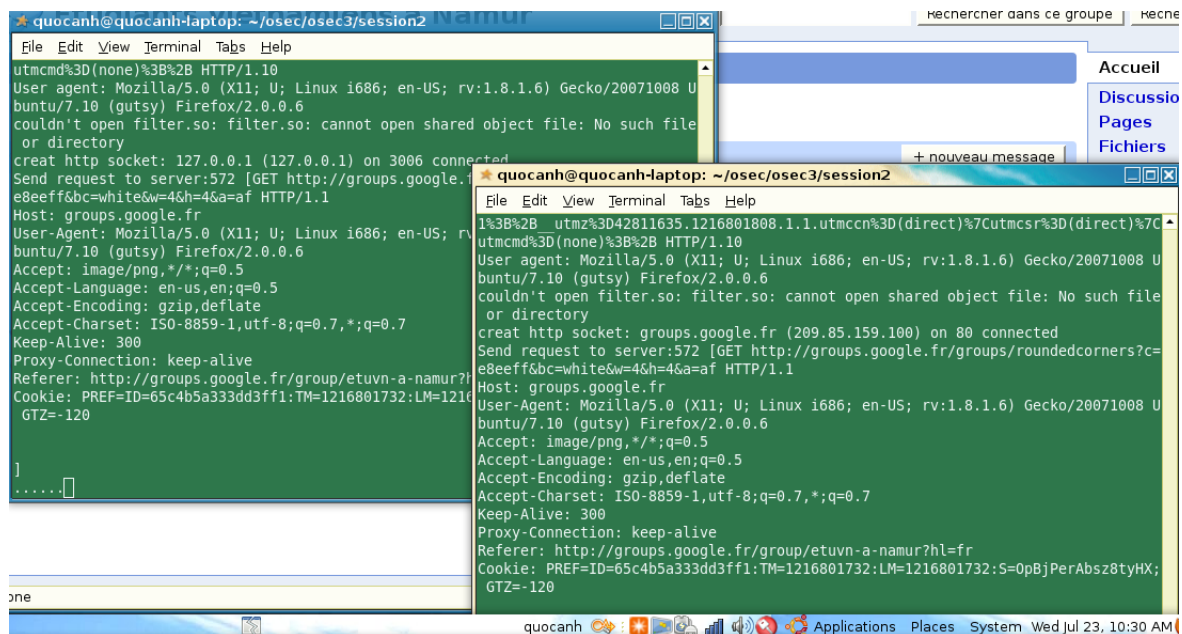


Figure 6.4: Chaîne des proxies

Code

7.1 Proxy serveur.c

```

/*****
 * Proxy HTTP 1.1
 * Done by Quoc Anh LE
 * Finished August 18, 2008
 * Reported on Wednesday, August 27, 2008
 *
 * Fonctions:
 * - Proxy Server
 * - Chain of proxy
 * - Filter by dynamic moduls
 * - Proxy-Cache
 *
 * example: man select_tut 2
 *
 *          ls -als core
 * debug: gdb ./proxy ./core
 *          valgrind --tool=memcheck --leak-check=yes -v ./proxy
 *
 *****/
#include <stdio.h>          /* for printf() */
#include <stdlib.h>         /* for atoi() and exit() */
#include <string.h>         /* for memset() */
#include <sys/ioctl.h>
#include <net/if.h>
#include <netdb.h>
#include <arpa/inet.h>      /* for sockaddr_in and inet_ntoa() */
#include <netinet/in.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>     /* for socket(), bind(), and connect() */
#include <time.h>

#include <dlfcn.h>

/* According to POSIX.1-2001 */
#include <sys/select.h>

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>

```

```

#include <unistd.h>
#define _XOPEN_SOURCE 600
#include <sys/select.h>

/* Defines */

#define MAX_CONNECTION 250 /* Maximum of request */
#define BUF_SIZE 102400 /* Maximum of packet */

#undef max
#define max(x,y) ((x) > (y) ? (x) : (y))

/* Global variables */
struct connection
{
    int client;
    int server;

    char cBuf[BUF_SIZE+1];
    char sBuf[BUF_SIZE+1];

    int cBuf_avail, cBuf_written;
    int sBuf_avail, sBuf_written;
};

char *msg_busy=
    "<html><head><title>Error</title></head><body>Sorry, this server is too busy."

char *msg_interdict =
    "<html><head><title>Error</title></head><body>Sorry, this site or this navigat

char *msg_bad_request = "<html><head><title>Error</title></head><body>BAD REQUES

char *msg_not_found = "<html><head><title>Error</title></head><body>NOT FOUND</b

int PROXY_PORT = 3005; /* Proxy server port */
int SERVER_PORT= 80; /* Remote server port */

int PORT2 = 0;
char PROXY2[255];

struct connection lstCon[MAX_CONNECTION];
time_t start_time;
int ProxySock;
fd_set rd, wr;
int running;

/* Prototypes */

```

```

int create_socket_proxy();
int create_socket_http(char *hostname, int port);
int check_data();
int filter_and_check(char *buf);

void initialisation(int argc, char *argv[]);
void loop();
void accept_new_connection();
void close_all();
void close_connection(int i);
void shut_sock(int sock);
void process_data();
void connection_remote_server(int index);
void get_host_name(char* host_name, char* buffer);
void set_non_blocking(int sock);

int (*function_filter)(char*, char*, char*);
void *module_filter;

/* Start here */

int main(int argc, char *argv[])
{
    initialisation(argc, argv);
    ProxySock=create_socket_proxy();
    loop();
    close_all();
    return 0;
}

/*****
***          INITIALISATION
*
* Description: Initialise variables
* Arguments   : Arguments from command line
* Returns     : None.
* Note        : cmd: ./> make run 3005 80
***
*****/

void initialisation(int argc, char *argv[])
{
    int k;

    time(&start_time);

    running = 1;

```

```

if(argv[1] != NULL)
    PROXY_PORT = atoi(argv[1]); /* First arg: local port */

printf("PROXY PORT = %d, SERVER PORT = %d\n",PROXY_PORT, SERVER_PORT);

if(argc > 2)
{
    PORT2 = atoi(argv[2]); /* Second arg: remote proxy port */
    strcpy(PROXY2,argv[3]); /* Third arg: remote proxy address */

    printf("Remote proxy port = %d, IP = %s\n",PORT2,PROXY2);
}

for(k=0;k<MAX_CONNECTION;k++)
{
    lstCon[k].client = 0;
    lstCon[k].server = 0;
    lstCon[k].cBuf_avail = 0;
    lstCon[k].cBuf_written = 0;
    lstCon[k].sBuf_avail = 0;
    lstCon[k].sBuf_written = 0;
}
}

/*****
***          CREATE A PROXY SOCKET
*
* Description: Create a socket non blocking for proxy server
* Arguments  : Port
* Returns    : Descriptor if succes or exit programme if failed
* Note      :
***
*****/

int create_socket_proxy()
{
    int sock;
    int rc, on=1;
    struct sockaddr_in serveraddr;

    /*
    * AF_INET address family sockets can be either
    * connection-oriented (type SOCK_STREAM) or they
    * can be connectionless (type SOCK_DGRAM).
    * Connection-oriented AF_INET sockets use TCP as
    * the transport protocol. Connectionless AF_INET

```



```
* sockets use UDP as the transport protocol. When
* you create an AF_INET domain socket, you specify
* AF_INET for the address family in the socket program.
* AF_INET sockets can also use a type of SOCK_RAW.
* If this type is set, the application connects
* directly to the IP layer and does not use either
* the TCP or UDP transports.
*/

/*
* The socket() function returns a socket descriptor
* representing an endpoint. The statement also
* identifies that the INET (Internet Protocol) address
* family with the TCP transport (SOCK_STREAM) will be
* used for this socket.
*/
sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock < 0)
{
    perror("socket() failed");
    shut_sock(sock);
    exit(EXIT_FAILURE);

}
else
    printf("Proxy is created!\n");

/*
* Set socket non block
*/
set_non_blocking(sock);

/*
* The setsockopt() fonction is used to allow the local
* address to be reused when the server is restarted
* before the required wait time expires.
*/
rc = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
                (char *)&on, sizeof(on));
if(rc < 0)
{
    perror("setsockopt(SO_REUSEADDR) failed");
    shut_sock(sock);
    exit(EXIT_FAILURE);

}
```

```

/*
 * After the socket descriptor is created , a bind()
 * function gets a unique name for the socket.
 * s_addr is set to zero , which allows connections
 * to be established from any client that specifies
 * port PROXY_PORT
 */
memset(&serveraddr , 0, sizeof(struct sockaddr_in));
/* Internet address family */
serveraddr.sin_family      = AF_INET;
/* Local port */
serveraddr.sin_port        = htons(PROXY_PORT);
/* Any incoming interface */
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* Bind to the local address */
rc = bind(sock , (struct sockaddr *)&serveraddr ,
           sizeof(serveraddr));
if(rc < 0)
{
    perror("bind() failed");
    shut_sock(sock);
    exit(EXIT_FAILURE);
}

/*
 * The listen() function allows the server to accept
 * incoming client connections.
 * The backlog means that system will queue
 * CONNECTION_QUEUE_SIZE incoming connection requests
 * before the system starts rejecting the incoming
 * requests.
 */
rc = listen(sock , MAX_CONNECTION);
if(rc < 0)
{
    perror("listen() failed");
    shut_sock(sock);
    exit(EXIT_FAILURE);
}
else
    printf("Ready for connection from client\n");

return sock;
}

/*****

```

```

***                LOOP
*
* Description: Repeat checking and processing the data until keypressed (Enter)
* Arguments  : None
* Returns    : None
* Note       :
***
*****/

void loop()
{
    while(running)
    {
        if(check_data() > 0)
            process_data();
    }
}

/*****
***                CHECK DATA
*
* Description: Using one only command Select() to check available data
*              for reading and writting
* Arguments  : None
* Returns    : Positive if data is ready for processing
* Note       : Keypress is also checked
***
*****/

int check_data()
{
    struct timeval tv;
    int retval;
    int maxDes;
    time_t ctime;
    int t,k;

    /* First put together fd_set for select(), which will
     * consist of the sock variable in case a new connection
     * is coming in, plus all the sockets we have already
     * accepted
     */

```

```

/* FD_ZERO() clears out the fd_set called socks, so that
 * it doesn't contain any file descriptor
 */

FD_ZERO(&rd);
FD_ZERO(&wr);

/* FD_SET adds the file descriptor "sock" to the fd_set,
 * so that select() will return if a connection comes in
 * on that socket (which means you have to do accept(),
 * etc.
 */

FD_SET(STDIN_FILENO, &rd); /* check keyboard */

FD_SET(ProxySock, &rd);    /* check new connection */

/* wait 0.01s */
tv.tv_sec = 0;
tv.tv_usec = 10000;

maxDes = ProxySock;

/* Loops through all the possible connections and adds
 * those sockets to the fd_set
 */
for (k=0; k<MAX_CONNECTION; k++)
{
    if (lstCon[k].client > 0 &&
        lstCon[k].cBuf_avail < BUF_SIZE)
    {
        FD_SET(lstCon[k].client, &rd);
        maxDes = max(maxDes, lstCon[k].client);
    }

    if (lstCon[k].server > 0 &&
        lstCon[k].sBuf_avail < BUF_SIZE)
    {
        FD_SET(lstCon[k].server, &rd);
        maxDes = max(maxDes, lstCon[k].server);
    }

    if (lstCon[k].client > 0 &&
        lstCon[k].sBuf_avail - lstCon[k].sBuf_written > 0)
    {
        FD_SET(lstCon[k].client, &wr);
        maxDes = max(maxDes, lstCon[k].client);
    }
}

```

```

        if (lstCon[k].server > 0 &&
            lstCon[k].cBuf_avail - lstCon[k].cBuf_written > 0)
        {
            FD_SET(lstCon[k].server, &wr);
            maxDes = max(maxDes, lstCon[k].server);
        }
    }

    retval = select(maxDes+1, &rd, &wr, NULL, &tv);

    if ((retval == -1) && (errno != EINTR))
        return -1;
    else if (retval < 0)
    {
        perror("select() failed");
        close_all();
        exit(EXIT_FAILURE);
    }
    else if (retval == 0)
    {
        time(&ctime);
        t = ctime - start_time;
        if (t % 5 == 0)
        {
            printf(".");
            fflush(stdout);
            start_time = ctime + 1;
        }
        return -1;
    }

    return 1;
}

/*****
PROCESS DATA
*
* Description:  if keypress then exit
*               if new connection then create new connection
*               if data is ready for reading then read data
*               if data is ready for writting then write data
* Arguments   :
* Returns     : None
* Note        :
*****/
*****/

```

```

void process_data()
{
    int r,k;

    if(FD_ISSET(0,&rd)) /* Enter keypress */
    {
        running = 0;
        getchar();
        return;
    }

    if(FD_ISSET(ProxySock,&rd)) /* New connection */
    {
        accept_new_connection();
        return;
    }

    for(k = 0;k<MAX_CONNECTION;k++)
    {
        if(lstCon[k].client > 0)
            if(FD_ISSET(lstCon[k].client , &rd))
            {
                r = read(lstCon[k].client ,
                        lstCon[k].cBuf + lstCon[k].cBuf_avail ,
                        BUF_SIZE - lstCon[k].cBuf_avail);
                if(r < 1)
                {
                    printf("read(client)\n");
                    close_connection(k);
                }
                else
                {
                    lstCon[k].cBuf_avail = lstCon[k].cBuf_avail + r;

                    if(lstCon[k].server == 0 ||
                       (lstCon[k].cBuf[0] == 'G'))
                        connection_remote_server(k);
                }
            }

        if(lstCon[k].server > 0)
            if(FD_ISSET(lstCon[k].server , &rd))
            {
                r = read(lstCon[k].server ,
                        lstCon[k].sBuf + lstCon[k].sBuf_avail ,
                        BUF_SIZE - lstCon[k].sBuf_avail);
                if(r < 1)
                {

```

```

        printf("read(server)\n");
        close_connection(k);
    }
    else
        lstCon[k].sBuf_avail = lstCon[k].sBuf_avail + r;
}

if(lstCon[k].client > 0)
    if(FD_ISSET(lstCon[k].client, &wr))
    {
        r = write(lstCon[k].client,
            lstCon[k].sBuf + lstCon[k].sBuf_written,
            lstCon[k].sBuf_avail - lstCon[k].sBuf_written);

        if(r < 1)
        {
            printf("write(client)\n");
            close_connection(k);
        }
        else
            lstCon[k].sBuf_written=lstCon[k].sBuf_written+r;
    }

if(lstCon[k].server > 0)
    if(FD_ISSET(lstCon[k].server, &wr))
    {
        r = write(lstCon[k].server,
            lstCon[k].cBuf + lstCon[k].cBuf_written,
            lstCon[k].cBuf_avail - lstCon[k].cBuf_written);

        if(r < 1)
        {
            printf("write(server)\n");
            close_connection(k);
        }
        else
        {
            printf("Send request to server:%d [%s]\n",
                lstCon[k].cBuf_avail-lstCon[k].cBuf_written,
                lstCon[k].cBuf+lstCon[k].cBuf_written);

            lstCon[k].cBuf_written=lstCon[k].cBuf_written+r;
        }
    }

}

/* check if write data has caught read data */
if(lstCon[k].cBuf_written >= lstCon[k].cBuf_avail)
    lstCon[k].cBuf_written = lstCon[k].cBuf_avail = 0;

```

```

        if (lstCon[k].sBuf_written >= lstCon[k].sBuf_avail)
            lstCon[k].sBuf_written = lstCon[k].sBuf_avail = 0;

        /* if client has closed the connection ,
         * the server must be closed
         */
        if (lstCon[k].client <= 0)
            close_connection(k);
    }
}

/*****
***          ACCEPT A NEW CONNECTION
*
* Description: Create new connection for new request
* Arguments   : None
* Returns     : None
* Note        :
***
*****/

void accept_new_connection()
{
    int cSock;           /* Socket descriptor for client */
    struct sockaddr_in cAddr; /* Client address */
    unsigned int cLen;     /* Length of address structure */
    int k;

    cLen = sizeof(cAddr);

    if ((cSock = accept(ProxySock, (struct sockaddr *) &cAddr,
                        &cLen)) < 0)
    {
        perror("accept() failed");
        close_all();
        exit(EXIT_FAILURE);
    }

    printf("Handling client %s\n", inet_ntoa(cAddr.sin_addr));

    /*
     * Set socket non block
     */
    set_non_blocking(cSock);

    for (k=0; (k<MAX_CONNECTION)&&(cSock != -1); k++)
        if (lstCon[k].client == 0)

```



```

    {
        lstCon[k].client = cSock;
        printf("\nConnection accepted: FD=%d; Slot=%d \n",
               cSock, k);

        cSock = -1;
    }

    if (cSock != -1)
    {

        printf("No room left for new client.\n");
        send(cSock, msg_busy, strlen(msg_busy), 0);

        printf("send back: %d = %s\n", strlen(msg_busy), msg_busy);
        shut_sock(cSock);
    }

}

/*****
***          FILTER
*
* Description: Load dynamique module and
               check acceptation conditions for a request
* Arguments  : A request
* Returns    : 0 if request is ok and otherwise
* Note       :
***
*****/

int filter(char *userAgent, char *command, char *hostname)
{
    int ret = 0;

    /* Load modul here */
    module_filter = dlopen("filter.so", RTLD_LAZY);

    if (!module_filter)
    {
        fprintf(stderr, "couldn't open filter.so: %s\n", dlerror());
        return 0;
    }

    /* Find the address of function */

```

```

*(void **>(&function_filter)=dlsym(module_filter,"check_filter");

if(dlerror() != NULL)
{
    perror("Couldn't find function check_filter");
    return 0;
}

ret = (*function_filter)(userAgent,command,hostname);

printf("Bo loc tra ve gia tri = %d\n",ret);

if(dlclose(module_filter) != 0)
    perror("dlclose() failed");

return ret;
}

```

```

/*****
***          GET INFORMATIONS ABOUT REQUEST
*
* Description: Get hostname, agent, ...
* Arguments  : Port
* Returns    : Descriptor if succes or exit programme if failed
* Note       :
***
*****/

```

```

void get_host_name(char* host_name, char* buffer)
{
    char *line;
    int i;
    i = 0;
    line = strstr(buffer, "Host: ");
    line = line+6;
    while(*line && *line != '\r'){
        host_name[i++] = *line++;
    }
    host_name[i] = 0;
}

```

```

void get_user_agent(char* userAgent, char* buffer)
{
    char *line;
    int i;
    i = 0;
    line = strstr(buffer, "User-Agent: ");

```

```

        line = line+strlen("User-Agent: ");
        while(*line && *line != '\r'){
            userAgent[i++] = *line++;
        }
        userAgent[i] = 0;
    }

/*****
***          CREAT HTTP SOCKET
*
* Description:
* Arguments  : hostname, (Port is a global var)
* Returns    :
* Note       :
***
*****/

int create_socket_http(char *hostname, int port)
{
    int sock;
    struct sockaddr_in address;
    struct hostent *hp;

    if(hostname == NULL) return -1;

    hp = gethostbyname(hostname);

    if(hp == NULL) return -1;

    sock = socket(AF_INET, SOCK_STREAM, 0);

    if(sock <= 0)
    {
        perror("create_socket_http() failed");
        shut_sock(sock);
        return -1;
    }

    address.sin_family = AF_INET;
    address.sin_port = htons(port);
    memcpy(&address.sin_addr.s_addr, hp->h_addr, hp->h_length);

    printf("creat http socket: %s (%s) on %d",
           hostname, inet_ntoa(address.sin_addr), port);

    if(connect(sock, (struct sockaddr*)&address,
               sizeof(struct sockaddr_in)) < 0)

```

```

        {
            perror("Connect to http server failed");
            shut_sock(sock);
            return -1;
        }
    else
        printf(" connected\n");

    set_non_blocking(sock);

    return sock;
}

/*****
***          CHECK BAD REQUEST
*
* Description:
* Arguments  : request
* Returns    :
* Note       :
***
*****/

int check_bad_request(char *httpRequest,
                     char *command,
                     char *userAgent,
                     char *hostname)
{
    int i = 0;
    size_t size = strlen(httpRequest);

    if(size <= 0)
        return 1;

    if ((httpRequest[0] != 'G') ||
        (httpRequest[1] != 'E') ||
        (httpRequest[2] != 'T'))
        return 1;

    /* command */
    while(httpRequest[i] != '\r' && httpRequest[i] != '\n')
    {
        command[i] = httpRequest[i];
        i++;
        if(i >= size)
            return 1;
    }
}

```

```

    command[i] = '0';

    get_host_name(hostname, httpRequest);

    get_user_agent(userAgent, httpRequest);

    if((hostname == NULL) || (userAgent == NULL))
        return 1;

    return 0;
}

/*****
void connection_remote_server(int index)
{
    int httpSock=0;

    char command[2048];
    char userAgent[1024];
    char hostname[1024];

    if(check_bad_request(lstCon[index].cBuf,
                        command, userAgent, hostname)==1)
    {
        send(lstCon[index].client, msg_bad_request,
            strlen(msg_bad_request), 0);
        close_connection(index);
        return;
    }

    printf("Hostname: %s\n", hostname);
    printf("URL: %s\n", command);
    printf("User agent: %s\n", userAgent);

    if(filter(userAgent, command, hostname)==1)
    {
        send(lstCon[index].client, msg_interdict,
            strlen(msg_interdict), 0);
        close_connection(index);
        return;
    }

    if(PORT2 == 0) /* No remote proxy */
        httpSock = create_socket_http(hostname, SERVER_PORT);
    else
        httpSock = create_socket_http(PROXY2, PORT2);
}

```

```

    if(httpSock > 0)
        lstCon[index].server = httpSock;
    else
        lstCon[index].server = 0;
}

/*****/
void set_non_blocking(int sock)
{
    /*
     * Under Linux, select() may report a socket file descriptor
     * as "ready for reading", while nevertheless a subsequent
     * read blocks. This could for example happen when data has
     * arrived but upon examination has wrong checksum and is
     * discarded. There may be other circumstances in which a
     * file descriptor is spuriously reported as ready.
     * Thus it may be safer to use O_NONBLOCK on sockets that
     * should not block.
     */
    int opts;

    opts = fcntl(sock,F_GETFL);
    if (opts < 0) {
        perror("fcntl(F_GETFL)");
        exit(EXIT_FAILURE);
    }
    opts = (opts | O_NONBLOCK);
    if (fcntl(sock,F_SETFL,opts) < 0) {
        perror("fcntl(F_SETFL)");
        exit(EXIT_FAILURE);
    }
    return;
}

/*****/
void shut_sock(int sock)
{
    if(sock >=0)
    {
        shutdown(sock,SHUT_RDWR);
        close(sock);
    }
}

/*****/
void close_all()
{
    int k;

```

```
    shut_sock (ProxySock);

    for (k=0;k<MAX_CONNECTION;k++)
        close_connection (k);
}

/*****/
void close_connection (int i)
{
    if (lstCon[i].client != 0)
    {
        /* Disables further send and receive operations */
        shut_sock (lstCon[i].client);
        lstCon[i].client = 0;

        printf("\nClose connection %d\n", i);
    }
    if (lstCon[i].server != 0)
    {
        shut_sock (lstCon[i].server);
        lstCon[i].server = 0;
    }

    lstCon[i].cBuf_avail = lstCon[i].cBuf_written = 0;
    lstCon[i].sBuf_avail = lstCon[i].sBuf_written = 0;
}
```

7.2 Module du filtrage.c

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define TAG_NAV "<Navigators>"
#define TAG_KEY "<Keywords>"
#define TAG_DOM "<Domains>"

int get_word(char* line , char* word, int pos, char* tag)
{
    int i=pos;
    int j =0;
    int len = strlen(line) - strlen(tag);

    while(line[i] == ' ' && i < len) i++;

    while(line[i] != ' ' && i < len) word[j++]=line[i++];

    word[j]= 0;

    if(i < len)
        return i;

    return 0;
}

int check_filter(char* userAgent, char* command, char *hostname)
{
    char navigator[255];
    char keyword[255];
    char domain[255];
    int n;

    /* read configuration file */
    FILE *f;
    char* line = NULL;
    ssize_t read;
    size_t len = 0;

    f = fopen("filter.txt","r");
    if(f == NULL)
    {
        perror("fopen() failed");
        return 0;
    }
```

```

while((read = getline(&line , &len , f)) != -1)
{
    printf("Retrieved line of length %zu: \n",read);
    printf("%s",line);

    if(strstr((char*)line,TAG_NAV) != NULL)    /* Block IE */
    {
        n = strlen(TAG_NAV);
        while((n=get_word(line , navigator ,n,TAG_NAV)) > 0)
        {
            printf("%s %d\n",navigator ,n);
            if ( strstr( (char*) userAgent ,navigator) != NULL)
                return 1;
        }
    }
    else if(strstr((char*)line ,TAG_KEY) != NULL) /* keyword */
    {
        n = strlen(TAG_KEY);
        while((n=get_word(line , keyword ,n,TAG_KEY)) > 0)
        {
            printf("%s %d\n",keyword ,n);
            if ( strstr( (char*) command ,keyword) != NULL)
                return 1;
        }
    }
    else if(strstr((char*)line ,TAG_DOM) != NULL) /* domain */
    {
        n = strlen(TAG_DOM);
        while((n=get_word(line , domain ,n,TAG_DOM)) > 0)
        {
            printf("%s %d\n",domain ,n);
            if ( strstr( (char*) command ,domain) != NULL)
                return 1;
        }
    }

}

if(line)
    free(line);
fclose(f);

return 0;
}

```

7.3 Fichier de la configuration du fitrage.txt

```
<Navigators> IE Elisa <Navigators>
<Keywords> sex prof <Keywords>
<Domains> info.fundp.ac.be lesoir.be <Domains>
```

7.4 Scrip makefile

```
export LD_LIBRARY_PATH=./usr/local/lib:/opt/lib
CC = gcc
CFLAGS = -ansi -pedantic -Wall -Werror
EXEC = proxy
DEBUG = -ggdb

all:
    $(CC) -c -fPIC filter.c
    $(CC) -shared -fPIC -o filter.so filter.o

    $(CC) $(CFLAGS) -o $(EXEC) quocanh2.c -ldl

    ./$(EXEC)

run:
    ./$(EXEC)

debug:
    $(CC) $(CFLAGS) $(DEBUG) -o $(EXEC) quocanh2.c

clean:
    if [ -e $(EXEC) ]; then rm $(EXEC); fi
    rm filter.so
    rm filter.o
```