Indice generale

CAPITOLO 1 - Proxy....3
1.1 Introduzione....3
1.2 Scelte progettuali....3
1.3 Comunicazione tra processi....4
1.4 Configurazione....5

CAPITOLO 2 - Controllo degli accessi....8
2.1 Introduzione....8
2.2 Tipi di filtraggio....8
2.3 Scelte progettuali....9
2.4 Configurazione del controllo degli accessi....12

CAPITOLO 3 - I file di log....17
3.1 Introduzione....17
3.2 Configurazione dei log....17
3.3 Creazione e scrittura dei file di log....20
3.4 I tipi di file di log....22

CAPITOLO 4 - La cache e cache ip....24
4.1 Introduzione....24
4.2 Struttura della cache....25
4.3 Funzione get_shm()....26
4.4 Gestione della cache....28
4.5 Differenza tra cache e cache ip....32

CAPITOLO 5 - Prove d'esecuzione....33

APPENDICE A - Readme.txt....40

APPENDICE B - Install.txt....41

CAPITOLO 1 - Proxy

1.1 Introduzione

La funzione di un proxy server nell'ottica dell'architettura del web è quella dell'intermediario. Un tale tipo di software ha principalmente lo scopo di frapporsi tra un client (o un insieme di client tipicamente facenti parte di una rete locale) ed internet. Sue funzioni tipiche sono quelle di fare il forwarding di una richiesta inoltrata da parte di un client verso il server origine appropriato, fare il caching delle risorse ritrovate (qualora le direttive inserite negli header http lo consentano) e fare il logging delle richieste.

Un'altra funzione importante che un server proxy espleta è quella di consentire una più accurata gestione sia del traffico di rete (ad esempio imponendo una banda massima per ogni connession e effettuata) sia del traffico di livello applicativo (ad esempio consentendo o meno il traffico di pacchetti HTTP in base a determinate regole).

1.2 Scelte progettuali

Le specifiche di progetto richiedono la realizzazzione di un server multiprocesso ed in ambiente Linux. In un server multiprocesso vi è un processo padre che ha il compito di accettare le connessioni da parte dei diversi client. Per ogni connessione accettata il processo padre genera un processo figlio. Il processo figlio si prenderà carico di gestire le connessioni accettate. Con tale sistema operativo si hanno a disposizione sostanzialmente due diverse alternative per la creazione di processi figli da parte di un processo padre:

1. *Utilizzo delle librerie pthread;*
2. *Utilizzo della chiamata di sistema fork;*

Entrambe le soluzioni presentano sia vantaggi che svantaggi.

Utilizzando la libreria pthread vengono generati processi che girano nello stesso spazio degli indirizzi del padre e ne condividono i dati e le risorse, per tale motivo la caduto di un processo figlio causerà la caduta dell'intero programma.

Utilizzando la chiamata di sistema fork(), invece, si ha la generazione di un processo figlio che gira in uno spazio degli indirizzi del tutto diverso da quello del padre, così facendo la caduta di un processo figlio non causerà la caduta dell'intero programma, bensì solo in una gestione non adeguata per la connessione servita da quel particolare processo.

In definitiva si può dire che l'uso di fork() lanci un programma indipendente per ogni connessione gestita.

Poiché un processo con le librerie pthread condivide lo spazio degli indirizzi del padre la creazione di un nuovo processo non comporta operazioni di copia da uno spazio degli indirizzi agli altri e, di conseguenza la creazione di processi risulta più snella di quanto avvenga con l'utilizzo di fork().

La nostra scelta, tuttavia, è ricaduta sulla creazione dei processi figlio attraverso al chiamata di sistema fork().

Infatti il multi-processo ottenuto tramite tale chiamata di sistema rende, a nostro avviso, l'intero programma assai più robusto che con l'utilizzo delle librerie pthread per quanto detto prima al riguardo della caduta dei processi, inoltre l'overhead dovuto alla creazione di un nuovo processo è stato ritenuto del tutto confrontabile con quello causato dalla su citata libreria. Un processo figlio generato con fork() differisce dal padre solo per il valore del suo pid e deve ricevere, quindi, una copia completa dei dati del processo padre.

In realtà l'operazione di copia dei dati non avviene sempre e comunque, ma viene utilizzato un meccanismo di tipo copy-on-write, ovvero un processo figlio riceverà la copia dei dati solo ed esclusivamente nel momento in cui questo richiederà di scrivere su di un particolare dato.

1.3 Comunicazione tra processi

Poiché due o più processi generati con fork sono equivalenti ad avere due o più programmi indipendenti che assolvono al loro particolare compito, ne discende che tali processi non possono comunicare tra loro attraverso l'uso di variabili globali (si ricorda che la scrittura su di una variabile

implica che il processo figlio che scrive su quella variabile ne riceva una copia). Di conseguenza l'utilizzo della chiamata di sistema fork() ha creato la necessità di trovare un appropria meccanismo di comunicazione tra processi, in particolare la comunicazione tra i vari processi generati si rende necessario per realizzare un meccanismo di caching (spiegato in dettaglio nel seguito) efficace. Linux mette a disposizione molti meccanismi di IPC (Inter Process Comunication).

Quello da noi scelto, e ritenuto più appropriato, è quello che fa uso della funzione shm_open().

*int shm_open(const char *name, int oflag, mode_t mode);*

Come si può notare questa funzione ha un prototipo del tutto analogo a quello della classica open, e la semantica dei parametri è esattamente la stessa. Questa funzione restituisce un file_descriptor che rappresenta un segmento di memoria condivisa di lunghezza nulla. Tale file_descriptor si riferisce a dei pseudo-file creati all'interno della dirctory /dev/shm. Dopo la creazione del segmento questo può essere dimensionato opportunamente attraverso l'utilizzo della funzione ftruncate().

Una volta creato e dimansionato opportunamente il segmento di memoria ogni processo che necessità di leggere/scrivere nella relativa area di memoria non deve far altro che mappare nello proprio spazio degli indirizzi lo pseudo-file di interesse.

1.4 Configurazione

La configurazione dei vari aspetti del proxy può essere eseguita attraverso la modifica del file proxy.conf. Tale file è stato suddiviso in sezioni, una per ogni aspetto configurabile del proxy. Nel seguito verrà analizzata la parte relativa alla sezione General, dove trovano posto i parametri di configurazione generali del proxy.

[General]
#Porta su cui il proxy è in ascolto
ListenOnPort= 5051
#Porta dei server origine alla quale verrà forwardata la richiesta dei client
OriginServerPort= 80

#Numero massimo di richieste contemporanee consentite

MaxReq= 150

I commenti apposti nel file di configurazione sono auto esplicativi. Si noti, però che affinchè il proxy sia eseguibile senza la necessità di privilegi da super-utente è necessario valorizzare la direttiva ListeOnPort con un numero di porta corrispondente a porte non privilegiate (tipicamente le porte non provilegiate sono quelle con numero superiore a 1024).

La seconda direttiva di configurazione serve ad impostare il numero di porta sulla quale, di default il proxy tenterà di contattare il server origine. Si noti che tale impostazione può essere sovrascritta automaticamente dal server in due condizioni:

1. Se perviene una richiesta con URL di tipo HTPPS. In tal caso il numero di porta sul quale il proxy tenterà di contattare il server origine sara la 443.
2. Se perviene una richiesta in cui viene specificato una porta alternativa all'interno dell'URL, come ad esempio http://www.foo.com:9878. In tal caso il numero di porta verrà estratto dall'URL e quindi il proxy tenterà di connettersi al server origine utilizzando tale numero di porta.

La terza direttiva, invece, serve ad impostare il massimo numero di connessioni parallele che il proxy è in grado di accettare. Per quanto riguarda la parte implementativa della configurazione abbiamo fatto ricorso alla creazione di una struttura dati atta a contenere le informazioni impostate all'interno del file proxy.conf. La struttura creata ha la seguente forma:

```
struct p_conf {
uint16_t listen_port;
uint16_t origin_server_port;
int max_req;
struct acl_conf conf_acl;
struct log_conf conf_log; };
```

Gli ultimi due campi della struttura sono anch'esse strutture dati complesse che servono per mantenere le configurazioni relative al controllo degli accessi e alla funzionalità di log. Tali strutture verranno analizzate nel dettaglio all'interno delle relative sezioni.

CAPITOLO 2 - Controllo degli accessi.

2.1 Introduzione

Il progetto da noi scelto implica lo sviluppo, oltre che delle funzionalità tradizionali di un proxy quali il forwarding delle richieste al server origine, anche della funzionalità di controllo degli accessi.

Infatti un firewall tradizionale lavora (tralasciando estensioni particolari come ad esempio il modulo layer7 di iptables che lavora, appunto, a livello 7 della pilo ISO/OSI) a livello di rete effettuando il filtraggio dei pacchetti entranti ed uscenti da una rete sulla base di caratteristiche di livello 3 ignorando del tutto la semantica dei pacchetti degli strati superiori (come è giusto che sia).

Un proxy, invece lavora a livello applicativo, ovvero allo stesso livello di HTTP di conseguenza può conoscere la semantica dei dati scambiati a tale livello e quindi effettuare un filtraggio selettivo di pacchetti HTTP.

2.2 Tipi di filtraggio

Come da specifiche sono stati implementati i seguenti tipi di filtraggio:

- *Filtraggio in base all'IP del client che richiede la connessione al proxy;*
- *Filtraggio in base al campo "Host:" contenuto nell'header delle richieste HTTP inoltrate dal client;*
- *Filtraggio in base all'estensione della risorsa richiesta dal client;*
- *Filtraggio in base al campo "Content-Type:" contenuto nell'header di risposta da parte di un server origine;*
- *Filtraggio in base a parole contenute nelle risorse di tipo testuale*

2.3 Scelte progettuali

Per la realizzazione di tali tipi di regole di accesso si è scelto di fare ricorso alle classiche Access Controll List che sono state realizzante utilizzando dei semplici file di testo, uno per ogni tipo di filtraggio effettuato. Le funzioni create per consentire o meno l'accesso ai servizi richiesti sono nella forma seguente:

```
int acl_control_mime(char* response, struct acl_conf* conf){
int allow_result;
if ( conf->acl_state[ACL_MTYPE] == 1){
allow_result = allow(response, conf->acl_path[ACL_MTYPE], "Content-Type:");
if( (conf->default_p == ALLOW) && (allow_result == NO) ){
return ALLOW;
//Se la policy di default è allow significa che ciò che non è esplicitamente
//vietato nella acl è consentito, quindi se non trovo il nome dell'host nella
//acl significa che questo può essere richiesto dall'utente
}
if( (conf->default_p == DENY) && (allow_result == YES) ){
return ALLOW;
//Se la policy di default è deny questo significa che tutto quello che non è
//esplicitamente consentito è vietato. Quindi se non trovo il nome dell'host
//nell'acl corrispondente significa che questo host non può essere acceduto
}
else return DENY;
}
else return ALLOW;
}
```

Tali funzioni (quella sopra citata è a mero titolo di esempio) prendono come parametri la richiesta del client (o la risposta dal server), e la struttura acl_conf (che verrà illustrata con maggior dettaglio nel seguito di questa sezione). Sulla base di tali parametri in ingresso viene invocata una funzione allow_*, ovvero una funzione il cui nome è allow_ seguito da un breve suffisso esplicativo della sua

funzione, ad esempio allow_ext si preoccupa di cercare all'interno del relativo file di testo la presenza o meno del' estensione della risorsa richiesta. Analizzando il codice sorgente si noterà sicuramente che queste funzioni sono estremamente simili tra loro fin quasi a rasentare l'uguaglianza e di conseguenza si potrebbe obiettare che sia stato replicato inutilmente del codice in quanto con piccoli aggiustamenti alle funzioni ci si poteva ricondurre ad una sola funzione. In

realtà quello che ci ha condotto a scrivere il codice in questo modo è stato il pensiero delle possibili, future, modifiche che si potrebbero apportare. Se in una futura fase di sviluppo si decidesse di modificare l'implementazione della funzione per la ricerca del valore del campo "Content-Type" all'interno della relativa ACL tale modifica avrebbe un impatto praticamente nullo sul resto del codice, e, qualora si abbia la possibilità di lasciare invariato il prototipo della funzione, l'unica cosa da fare per integrare la modifica è una semplice ricompilazione.

Continuiamo a prendere come esempio la precedente funzione. Questa si avvale della funzione allow(). Per capire se il valore cercato è all'interno dell'acl o meno e sulla base di tale ricerca prende poi delle decisioni. Vediamo nel dettaglio come questa funzione allow() è stata realizzata:

```
int allow(char* http, char* acl_path, char* header_field){
char value[64];
find_value_field_header(http, header_field, value);

if( find_in_acl_file(value, acl_path ) ){
PRINT_DEBUG_CONF("Trovato nella acl\n");
return YES;
}
else return NO;
}
```

Come si può vedere tale funzione è molto semplice, cerca prima di tutto il calore del campo field (terzo parametro) all'interno della richiesta/risposta HTTP (primo parametro), dopo di che effettua la ricerca dl valore del campo trovato poc'anzi nell'acl che si trova in acl_path. Visto che il grosso del lavoro viene svolto dalla funzione find_in_acl_file() riteniamo sia interessante spiegarla nel dettaglio anche per giustificare la nostra scelta implementativa che, di primo acchito potrebbe

sembrare abbastanza inefficiente).

```
int find_in_acl_file(char* field_value, char* acl_path){
FILE* acl_str;
char buffer[64];
char row_content[64];
if ( ( acl_str = fopen(acl_path,"r") ) != NULL ){
while ( fgets(buffer,64,acl_str) != NULL ){

if (buffer[0] == '#' || buffer[0] == '\n') //Ignora i commenti
continue;
else{
sscanf(buffer,"%s",row_content);
if( !strcmp(row_content,field_value) ) //Ho trovato l'host nel file
return 1;
}

}

return 0;
}
else return 0; }
```

Anche tale funzione, che per altro è alla base di tutto il meccanismo che effettua il controllo degli accessi, è molto semplice. Prima di tutto apre uno stream a file in sola lettura, dopo di che attraverso la funzion fgets legge 64 byte alla volta (in pratica una riga alla volta) dal file. Se la riga letta inizia con il carattere '#' o '\n' la ignora (in pratica '#' è il carattere con cui deve iniziare una riga di commento, mentre '\n' server per ignorare righe vuote.).

Se la riga inizia con un qualunque carattere alfanumerico diverso da quelli precedentemente citati

viene letto tale valore e confrontato con il valore del parametro field_value.

La prima cosa che salta all'occhio leggendo tale funzione è che vi è un accesso a disco per ogni valore che si deve confrontare. La soluzione da noi scelta, in effetti, non è sicuramente la più efficiente possibile, ma si è dimostrata particolarmente efficace soprattutto per fornire all'amministratore del proxy un buon grado di flessibilità. Infatti cosi come abbiamo implementato tale funzione di ricerca l'aggiunta o la rimozione di un campo da una qualsivoglia ACL ha effetto immediato e quindi non si rende necessario il riavvio del servizio per rendere effettive tali modifiche. Ritorniamo a dire che tale soluzione non è sicuramente la più efficiente, ma tenendo conto delle cache ormai universalmente presenti sugli attuali dischi rigidi e considerando anche le cache presenti nel kernel di Linux ci è sembrato un buon compromesso tra efficienza, flessibilità e semplicità implementativa.

2.4 Configurazione del controllo degli accessi.

Questo nostro software consente di configurare pienamente le funzionalità di controllo degli accessi. E' possibile decidere se abilitare o meno un tipo di controllo, configurare il path completo dove risiedono le acl e impostare una policy di default. Si noti che in questo caso per rendere effettive eventuali modifiche apportate al file di configurazione sarà necessario stoppare e quindi riavviare il server.

Riportiamo di seguito il frammento del file di configurazione dedicato a tale configurazione:

[AccessControl]
DefaultPolicy= allow
ACL-Hosts= no
ACL-Ip= no
ACL-MimeType= no
ACL-Extension= no
ACL-Content= no
PathHostsAcl= ./acl/hosts.acl
PathIpAcl= ./acl/ip.acl

PathMimeTypeAcl= ./acl/mime.acl
PathExtensionAcl= ./acl/ext.acl
PathContentAcl= ./acl/word.acl

La direttiva DefaultPolicy serve ad impostare una policy di default per il meccanismo di controllo degli accessi. Si noti che la policy di default impostata vale per tutti e cinque i tipi di filtraggio (che sono in and, ovvero tutte le regole che sono state attivate devono andare a buon fine affinchè le richieste inoltrate al proxy vengano gestite), e non risulta, almeno per ora, impostare policy di default diverse per diversi tipi di filtraggio.

La policy di default, in pratica modifica la semantica delle ACL:

***DefaultPolicy= allow** --> Imposta la policy di default ad allow, ciò significa che viene concesso tutto e viene inibito solo ciò che è riportato all'interno delle acl (che in questo caso assumono il ruolo di black list)*

***DefaultPolicy= deny** --> Imposta la policy di default a deny, ovvero viene vietato tutto e viene permesso solo ciò che è elencato all'interno delle acl (che in questo caso funzionano da white list).*

Ad esempio supponiamo che la acl relativa al filtraggio sul nome degli dell'host cui un client intende connettersi contenga solo il campo www.google.it.

Se la policy di default risulta essere impostata ad allow l'acl funge da block list, ovvero risulterà possibile inoltrare richieste per tutti gli host tranne che per www.google.it che verà invece respinta. Al contrario se la policy di default è invece impostata su deny la acl va funzionae come white list, di conseguenza l'unico host a cui sara possibile connettersi sarà www.google.it, mentre le richieste per tutti quanti gli altri host verranno respinte.

Le direttive:

ACL-Hosts= no | yes
ACL-Ip= no | yes
ACL-MimeType= no | yes
ACL-Extension= no | yes

ACL-Content= no | yes

Servono ad abilitare o meno un particolare tipo di filtraggio, mentre le direttive:

PathHostsAcl= ./acl/hosts.acl
PathIpAcl= ./acl/ip.acl
PathMimeTypeAcl= ./acl/mime.acl
PathExtensionAcl= ./acl/ext.acl
PathContentAcl= ./acl/word.acl
Servono ad impostare il path che il proxy utilizzerà per leggere le acl.

Per memorizzare i valori di configurazione impostati attraverso le direttive sopra elencate è stata creata una struttura dati ad-hoc:

```
struct acl_conf{
int default_p; //Policy di default
int acl_state[ACLNO]; //contiene lo stato delle varie acl
char acl_path[ACLNO][256]; //contiene il path delle acl
};
```

il primo campo, default_p, serve a mantenere l'informazione relativa alla policy di defult. Tenendo conto che le policy di default utilizzabili sono due per mantenere l'informazione relativa a quale delle due viene utilizzata è stato fatto uso di un nuomero intero, cosi se default_p vale 1 la policy di default impostata è allow, se invece vale 0 è deny. Per miglioare la leggibilità del codica sono state definite (attraverso l'uso della diretteva del pre-processore #define) due costanti nel seguente modo:

```
#define ALLOW 1
#define DENY 0
```

Il secondo campo, acl_state, è un vettore di interi che serve a mantenere lo stato di un'acl. ACLNO è anch'essa una costante definita attraverso #define ed è stata impostata a 5, numero delle acl presenti.

Il terzo campo, acl_path, è destinato a contenere il path nel quale il proxy andrà a cercare e quindi

leggere le acl.

Una accortezza ritenuta in questo frangente e pensata appositamente per migliorare la leggibilità del codice è l'utilizzo di una enum:

enum{ACL_HOSTS, ACL_IP, ACL_MTYPE, ACL_EXT, ACL_WORD};

Come abbiamo visto lo stato ed il path di ogni acl sono mantenute all'interno di due vettori. Per conoscere lo stato (se attiva o meno) della acl relativa agli indirizzi ip dei client bisognerebbe scrivere una istruzione del tipo

acl_state[1].......

Ovviamente questo non è auspicabile. Leggendo in mezzo al codice un frammento simile al precedente terze persone, o anche chi ha ideato il codice ma è tanto che non lo rilegge, si troverebbe in sicura difficoltà! Il secondo elemento di acl_state a quale acl corrisponde?

L'introduzione di questa enum ha reso possibile, a nostro avviso, un elevato incremento della leggibilità. Infatti cosi facendo ogni simbolo elencato tra parentesi graffe viene ad avere associato un preciso valore numerico, nella fatispecie.

ACL_HOSTS = 0
ACL_IP = 1
ACL_MTYPE = 2
ACL_EXT = 3
ACL_WORD = 4

La precedente istruzione per conoscere lo stato della acl relativa agli indirizzi ip dei client può quindi essere riscritta come segue:

acl_state[ACL_IP].......

A questo punto risulterà chiaro in qualsiasi momento che in quel frammento di codice ci si sta riferendo allo stato dell'acl relativa agli ip dei client.

CAPITOLO 3 - I file di log

3.1 Introduzione

I log file possono essere definiti come un registro informatico all'interno del quale possono essere memorizzate tutte le operazioni relative alle trasmissioni effettuate(invio,ricezione ecc.) utili per varie operazioni. Essi sono di particolare importanza per i Server Web, Proxy, ecc. per fare delle valutazioni statistiche e per

l'individuazione di errori di diverso genere, in modo da poter rendere disponibile un servizio all'altezza della situazione. In questo capitolo noi descriviamo la creazione e gestione dei file di log del nostro proxy.

3.2 Configurazione dei log

Ogni volta che si usa il proxy, le prime operazioni eseguite sono le configurazioni delle acl, dei log e degli aspetti generali. In questo caso, noi concentriamo la nostra attenzione sulla configurazione dei log.

All'avvio del proxy, come detto precedentemente, viene invocata la funzione

conf_proxy(struct p_conf info_conf)* (in proxy_c.0.0.1.c) la quale ha il compito di iniziare la configurazione del proxy.

```
/*Configurazione proxy e creazione di un file di log(file di log generale).questa funzione torna il descrittore del file di
log oppure -1 in caso di errore nella configurazione*/
int conf_proxy(struct p_conf* info_conf){

        int general_log_fd;
        char msg_log[80];
        //funzione di configurazione proxy che torna 0 se è andata a buon fine
         if ( init_conf_proxy(info_conf,"./proxy.conf") == 0 ){
                //creazione del file di log general.log
                if (info_conf->conf_log.enable == YES)
                        general_log_fd = openLogFile(info_conf->conf_log.log_dir,"general.log");

                sprintf(msg_log,"Configurazione andata a buon fine!");
                //scrittura nel file di log dell'esito dell'operazione del proxy
                writeLog(general_log_fd, msg_log, info_conf->conf_log.dim_file);

                PRINT_DEBUG_CONF("Configurazione andata a buon fine.\n");
                PRINT_DEBUG_CONF("ListenOnPort = %d\n",info_conf->listen_port);
                PRINT_DEBUG_CONF("OriginServerPort = %d\n",info_conf->max_req);
                PRINT_DEBUG_CONF("DefaultPolicy = %d\n",info_conf->conf_acl.default_p);
                PRINT_DEBUG_CONF("ACL-Hosts = %d\n",info_conf->conf_acl.acl_state[ACL_HOSTS]);
                PRINT_DEBUG_CONF("ACL-Ip = %d\n",info_conf->conf_acl.acl_state[ACL_IP]);
                PRINT_DEBUG_CONF("ACL-MimeType = %d\n",info_conf->conf_acl.acl_state[ACL_MTYPE]);
                PRINT_DEBUG_CONF("ACL-HostsPath = %s\n",info_conf->conf_acl.acl_path[ACL_HOSTS]);
                PRINT_DEBUG_CONF("ACL-IpPath = %s\n",info_conf->conf_acl.acl_path[ACL_IP]);
                PRINT_DEBUG_CONF("ACL-MimeTypePath = %s\n",info_conf->conf_acl.acl_path[ACL_MTYPE]);
                PRINT_DEBUG_CONF("Log Abilitato = %d\n",info_conf->conf_log.enable);
                PRINT_DEBUG_CONF("Log Circolari %d\n",info_conf->conf_log.circular);
                PRINT_DEBUG_CONF("Dimensione file di log %d\n",info_conf->conf_log.dim_file);
```

Figura 3.1: Funzione conf_proxy che avvia configurazione proxy

Come si può notare in figura 3.1, la precedente chiamata avvia la funzione

init_conf_proxy(struct p_conf conf, char* conf_file)*

che prende come parametri una *struttura p_conf*, utile a memorizzar sie tutti i dati di configurazione, ed un *file di testo*, dal quale andrà a prelevare tutti i dati per configurare i nostri log.

In figura 3.2 possiamo vedere come è strutturato il file di testo per la configurazione del proxy.

```
[General]
#Porta su cui il proxy è in ascolt
ListenOnPort= 5051
#Porta dei serve1r origine alla quale verrà forwardata la richiesta dei client
OriginServerPort= 80
#Numero massimo di richieste contemporanee consentite
MaxReq= 150
#Policy di Defautl può assumere allow, deny

[AccessControl]
DefaultPolicy= allow
ACL-Hosts= no
ACL-Ip= no
ACL-MimeType= no
ACL-Extension= no
ACL-Content= no
PathHostsAcl= ./acl/hosts.acl
PathIpAcl= ./acl/ip.acl
PathMimeTypeAcl= ./acl/mime.acl
PathExtensionAcl= ./acl/ext.acl
PathContentAcl= ./acl/word.acl

[Log]
LogEnable= yes
LogDir= ./log/
#dimensione massima dei file di log in byte
DimFileLog= 10000
```

Figura 3.2 : file di configurazione del proxy

Quest'ultima funzione permette di leggere il file riga per riga, scartando tutte le righe che rappresentano i commenti ('#') o righe vuote(che iniziano con '\n'). Un particolare importante è l'individuazione, da parte della stessa funzione, delle diverse parti di cui è composto il file ([Log],[AccessControl],[General]).Nel nostro caso [log], ci permette localizzare tutti i dati di configurazione relativi ai log. A questo punto sarà chiamata la funzione

confLog(char field, char* value, struct p_conf* c_str)*

che permette di valorizzare il campo *field* con il valore *value* della struttura dei log(vedi figura 3.3).

```
/*configurazine Log del proxy; ogni volta che prende i parametri dal file proxy.conf
chiama questa funzione per settare i log del proxy*/
void confLog(char* field, char* value, struct p_conf* c_str){

                /*abilita o meno i log*/
        if( !strcmp(field, "LogEnable=") ){
                if (!strcmp(value, "yes") ){
                        c_str->conf_log.enable = YES;
                }
                else c_str->conf_log.enable = NO;
        }
        else{   /*seleziona la directory dove creare i file di log*/
                if( !strcmp(field, "LogDir=") )
                        strcpy(c_str->conf_log.log_dir,value);
                else{


                                if( !strcmp(field,"DimFileLog=") ){
                                        c_str->conf_log.dim_file = atoi(value);
                                }

                }
        }
}
```

Figura 3.3: funzione confLog per la configurazione dei log

I campi della struttura dei log sono, come si può vedere in figura 3.4 ,

- *enable : se i log sono o meno abilitati*
- *log_dir: viene valorizzato con il path di identificazione della directory dei file di log*
- *dim_file: è la dimensione massima dei file di log*

```
struct log_conf{
        int enable;
        char log_dir[256];
        int dim_file;
};
```

Figura 3.4: struttura log

3.3 Creazione e scrittura dei file di log

Come possiamo vedere dalla figura 3.1 se il campo *info_conf->conf_log.enable* è impostato a *YES*, questo significa che il logging è stato attivato e le operazioni successive sono:

openLogFile(char log_dir, char* name)* (vedi figura 3.5)

```
/*Creazione del file di log dove in name deve contenere solo il nome del file di log,
mentre la directory dove viene creato è presa dal file di configurazione,ritorna il file descriptor del file incaso di
correta esecuzione e -1 altrimenti*/
int openLogFile(char* log_dir, char* name){

        char tmp_path[256];
        char tmp_path_data[256];
        tmp_path[0] = '\0';
        tmp_path_data[0]='\0';
        int log_fd;

        strcat(tmp_path,log_dir);
        strcat(tmp_path_data,tmp_path); //il file relativo andra a contenere l'offset nel file di log
        strcat(tmp_path_data,".");
        strcat(tmp_path_data,name);
        strcat(tmp_path,name);
        strcat(tmp_path,"\0");

                /*creazione del file di log */
        if( ( log_fd = open(tmp_path, O_RDWR | O_CREAT | O_APPEND, 0777) ) < 0 ){
                return -1;
        }
        else{
                tmp_path[0]='\0';
                return log_fd;
        }

}
```

Figura 3.5: creazione file di log

e writeLog(int fd, char* msg, int dim_file) (vedi figura 3.6).

```
//Scrive sul file di log corrispondente allo streaming stream. Gestisce anche la circolarità del log
//ritorna -1 in caso di errore 0 altrimenti
int writeLog(int fd, char* msg, int dim_file){

        char tmp_msg[MAX_LOG_LINE];
        struct stat property_file;
        int b_w = 0;

        if( fstat(fd,&property_file) == -1 ){
                return -1;
        }

        if( property_file.st_size > dim_file ){ //Se eccede la massima dimensione impostata
                ftruncate(fd, 0); //Riduco a zero la dimensione del file
        }

                time_t now;
                struct tm now_tm;
                tmp_msg[0]='\0';

                //Prepara la data per da scrivere sul file di log
                time(&now);
                gmtime_r(&now, &now_tm);
                asctime_r(&now_tm, tmp_msg); //la stringa restituita da questa funzione termina con \n

                tmp_msg[strlen(tmp_msg) - 1 ] = '\0'; //elimino lo \n
                 tmp_msg[strlen(tmp_msg)] = '\0';
                 if(strlen(msg)<80){
                         int i = 0;
                         strcat(tmp_msg,msg); //msg ha meno di 80 char

                         for(i=strlen(tmp_msg); i<(80-2); i++)
                                 tmp_msg[i]=' ';
                                 tmp_msg[i]='\n';
                                 tmp_msg[i+1]='\0';

                 }
                 else{
                         strncat(tmp_msg,msg,79); //Se msg eccede gli 80 char lo tronco
                         strcat(tmp_msg,"\n\0");
                 }

        if( (b_w = write(fd, tmp_msg, strlen(tmp_msg))) < 0){ //scrivo indiscriminatamente su file
                    printf("Errore nella scrittura\n");
                     return -1;
                 }
                 else{
                    fsync(fd);
                    return 0;

}
```

Figura 3.6: funzione wirteLog di scrittura dei fila di log

Come è facile intuire dal nome delle funzioni la

openLogFile(info_conf->conf_log.log_dir, "general.log")

prende due parametri il path, dalla struttura dei log, e il nome da assegnare al file di log che verrà creato. Per quanto riguarda la funzione

writeLog(general_log_fd, msg_log, info_conf->conf_log.dim_file),

prende come parametri il file descriptor del file log,che è restituito dalla funzione *openLogFile*, il messaggio da scrivere nei log e la dimensione massima del file di log.

Ogni volta che viene invocata la funzione *writeLog,* viene calcolata, attraverso la funzione *asctime(),* la data giornaliera in quel preciso istante nella forma *Wed Aug 2 13:55:16 2006* per poi concatenarla con il messaggio di log passato alla funzione.

Per una migliore comprensione e lettura dei file di log si scrivono, negli stessi, messaggi non

superiori agli 80 caratteri.

3.4 I tipi di file di log

I file creati in questo proxy sono di due tipi:

- general.log
- [indirizzo_IP].log

Per quanto riguarda i log di tipo general vedi figura 3.7, mentre [indirizzo_IP].log vedi figura 3.8

```
Wed Aug  2 13:55:17 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:17 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:25 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:25 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione accettata per l' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione richiesta dall' IP 127.0.0.1
Wed Aug  2 13:55:26 2006 - Connessione accettata per l' IP 127.0.0.1
```

Figura 3.7: file log di tipo general

```
Wed Aug  2 13:55:26 2006 - Richiesta autorizzata
Wed Aug  2 13:55:26 2006 - Richiesta autorizzata
Wed Aug  2 13:55:26 2006 - Richiesta per l'host www.uniroma2.it
Wed Aug  2 13:55:26 2006 - Richiesta autorizzata
Wed Aug  2 13:55:26 2006 - Connessione con  www.uniroma2.it avvenuta
Wed Aug  2 13:55:26 2006 - Connessione con  www.uniroma2.it avvenuta
Wed Aug  2 13:55:26 2006 - Richiesta per l'host www.uniroma2.it
Wed Aug  2 13:55:26 2006 - Richiesta autorizzata
Wed Aug  2 13:55:26 2006 - Connessione con  www.uniroma2.it avvenuta
Wed Aug  2 13:55:26 2006 - Richiesta per l'host www.uniroma2.it
Wed Aug  2 13:55:26 2006 - Richiesta per l'host www.uniroma2.it
Wed Aug  2 13:55:26 2006 - Connessione con  www.uniroma2.it avvenuta
Wed Aug  2 13:55:26 2006 - Richiesta per l'host www.uniroma2.it
Wed Aug  2 13:55:26 2006 - Richiesta autorizzata
Wed Aug  2 13:55:26 2006 - Richiesta autorizzata
Wed Aug  2 13:55:27 2006 - Richiesta autorizzata
Wed Aug  2 13:55:27 2006 - Richiesta per l'host www.uniroma2.it
Wed Aug  2 13:55:27 2006 - Connessione con  www.uniroma2.it avvenuta
Wed Aug  2 13:55:27 2006 - Connessione con  www.uniroma2.it avvenuta
Wed Aug  2 13:55:27 2006 - Richiesta per l'host www.uniroma2.it
```

Figura 3.8: file di log di tipo ip host

La differenza sostanziale dei due tipi di file di log, sta nel fatto, che il log di tipo general tiene

traccia di tutte le richieste e accettazioni di connessioni da parte di un qualsiasi host connesso al proxy, mentre quello [indirizzo_ip].log tiene traccia di tutte le richieste del host [indirizzo_ip] verso un qualsiasi host, scrivendoci l'esito delle varie operazioni di richiesta e accettazione di connessione.

Nel nostro caso, quest'ultimo file di log è unico, in quanto siamo gli unici host ad utilizzare il proxy quindi genererà un file nominati 127.0.0.1.log. Sostanzialmente verrà creato un file di tale tipo per ogni host che ha richiesto una connessione con il proxy.

CAPITOLO 4 - La cache e cache ip

4.1 Introduzione

Il Proxy è un server che agisce da *filtro* tra le richieste di connessione a siti Internet, provenienti in genere dall'interno della rete LAN o WAN a cui il *proxy* appartiene, ed i siti stessi. La richiesta di accedere ad una risorsa su Internet, proveniente da un computer appartenente ad una LAN o ad una WAN, viene intercettata dal *proxy* di rete in modo del tutto trasparente per l'utente.

Se la pagina richiesta non è presente nella *cache* del *proxy*, la richiesta viene inoltrata al server che ospita la risorsa, così da recuperare la pagina ed inviarla all'utente.

Se, viceversa, la pagina è già presente nella *cache* del *proxy*, questa viene inoltrata direttamente all'utente, senza che occorra inviare alcuna richiesta al server origine che ospita la risorsa stessa.
L'uso di un *proxy* fornisce essenzialmente due vantaggi:

- La possibilità di *filtrare le richieste* provenienti dall'interno della propria rete, in modo da evitare, ad esempio, di soddisfare le richieste di connessione a determinati siti proibiti dalle regole aziendali (per quanto riguarda il nostro proxy ciò è ottenuto con le acl descritte nei primi capitoli).
- La possibilità di *aumentare notevolmente le prestazioni*, risparmiando tempo e banda di connessione: ciò avviene quando una stessa pagina, già presente nella *cache* del *proxy*, viene richiesta da più utenti e quindi inviata loro direttamente dal *proxy* stesso, che evita così di connettersi nuovamente al server remoto che ospita la risorsa.

In questo capitolo ci accingiamo ad illustrare il funzionamento e la struttura della cache per le risorse e cache ip.

4.2 Struttura della cache

Come per la configurazione dei log, anche per la cache, appena viene avviato il proxy tra le prime operazioni eseguite, vi è la creazione della struttura di cache(vedi figura 4.1).

```
/*creazione della hash per la cache*/
cache_resource = init_cache_res(DIM_RES_X, DIM_RES_Y);
/*creazione della hash per la cache degli ip*/
dns_cache = init_cache_ip(DIM_IP_X, DIM_IP_Y);
```

Figura 4.1:funzioni di inizializzazione della cache

Entrambi la cache e la cache ip sono concettualmente concepite con la stessa idea, anche se utilizzate per scopi diversi, rispettivamente sulle risorse e sugli indirizzi ip.

La cache è stata strutturata con con una hash-table composta da DIM_Y liste e ognuna delle quali ha DIM_X elementi.

```
/*Creazione della struttura hash-table per la gestione della cache e ritorna la struttura hash-table*/
struct dorso* init_cache_res(int dimX, int dimY){

        int i=0,j=0;
        char name[256];
        name[0]='\0';
        strcat(name,"ShMemRes");
        /*creazione del dorso(una lista di strutture di puntatori)dell'hash table con la
        funzione get_shm_c() */
        struct dorso* h_table = (struct dorso*)get_shm_c(name, sizeof(struct dorso) * dimY);

        int name_lenght=strlen(name);
        for(i=0;i<dimY;i++){
                sprintf((name + name_lenght), "%d", i);
                strcat(name, "\0");
                /*il for cicla per dimY volte creando puntatori alle liste(corrispondenti alle liste dell'hash)*/
                h_table[i].list=(struct cache *)get_shm_c(name, sizeof(struct cache));
                h_table[i].empty_row_flag = 1;//indica che la lista corrispondente alla posizione i del dorso è vuota

                struct cache* app_list = h_table[i].list;
                app_list->empty=1;//elemento interno alla lista è vuota
                app_list->lru=dimX+1;//inizializo lru come dimX+1
                strcat(name,"_");
                int lenght2 = strlen(name);
                for(j=0;j<dimX-1;j++){
                        /*creazione della lista composta da dimX elementi */
                        sprintf( (name + lenght2), "%d", j);
                        strcat(name, "\0");
                        app_list->nextelem=(struct cache *)get_shm_c(name, sizeof(struct cache));

                          app_list->empty=1;
                          app_list=app_list->nextelem;

                 }

        }
  PRINT_DEBUG_RES_CACHE("X = %d\n", j);
 return h_table;

}
```

Figura 4.2: funzione di creazione e inizializzazione

Come possiamo vedere dalla figura 4.2, la struttura hash table, viene creata dinamicamente. Utilizziamo la funzione get_shm_c(), che spiegheremo successivamente, per creare le dim_x

locazioni di memoria delle dim_y liste.

Ogni locazione di memoria è composta da una struttura dati, tipo quella in figura 4.3,

```
struct cache{
        char resource_name[1024]; //nome completo della risorsa
        char lastmodified[32];
        char expires[256];

        char idfile[64];
        char usedate[256]; //Data di ultimo accesso al blocco di cache
        int empty; //empty=1 indica che il blocco è vuoto
        int lru;//campo di calcolo lru
        struct cache *nextelem;

};
```

Figura 4.3: struttura dati della cache

All'atto della sua creazione viene inizializzato il campo empty =1 per indicare che la locazione è vuota (ovvero non ha nessuna risorsa associata).

4.3 Funzione get_shm()

L' obiettivo principale di questa funzione è quello di creare aree di memoria condivisa da associare alla struttura hash-table. La funzione prende come parametri un *char** ,che è il nome da associare all'area di memoria, e un *int* , che è la dimensione dell'area di memoria che si andrà a creare.

```
/*crea l'area di memoria per l'hash table ritornando il riferimento a l'area di memoria
mappata*/
void* get_shm_c(char* name, int dim){

        void* ret;

        int fd;
        /*creazione dell'areaa di memoria di nome name*/
        if ( (fd = shm_open(name, O_RDWR | O_CREAT, 0777)) < 0 ){
                fprintf(stderr, "impossibile ottenere un segmento di memoria condivisa.\n");
                fprintf(stderr,"Errore %s\n",strerror(errno));

        }
        /*dimensionamento dell'areaa di memoria di lunghezza dim*/
        ftruncate(fd, dim);
        /*mappa l'area di memoria fd */
        ret = mmap(0, dim, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        close(fd);
        return ret;

}
```

Figura 4.4: funzione get_shm crea aree di memoria

Il motivo di questa scelta progettuale è dovuta all'utilizzo dei processi per il proxy e quindi è stato necessario risolvere i problemi di comunicazione tra processi e condivisione della cache tra gli stessi. La shared memory unitamente al meccanismo del memory mapping permettono ai processi

di comunicare in maniera estremamente veloce.

Questo tipo di comunicazione, infatti, é estremamente veloce in quanto la scrittura/lettura dei dati avviene interamente in memoria senza necessitàa disco.

Le funzioni per la gestione della shared memory e dei files mappati permettono di controllare l'accesso alle aree condivise, in modo da coordinarne l'uso. Utilizzando un file mappato in memoria, le modifiche fatte da un processo sul file stesso sono viste da tutti gli altri. Il mappaggio in memoria persiste fino a che tutti i processi cooperanti esistono.

Sia files che aree di memoria condivisa vengono utilizzati con la stessa politica:

- Ottenere un file-descriptor con il comando *open* o *shm_open*
- Mappare l'oggetto con una chiamata alla funzione *mmap*
- Alla fine dell'utilizzo smappare l'oggetto dalla memoria tramite *munma*p
- Chiudere l'oggetto con *close*
- Eliminare l'oggetto dalla memoria condivisa tramite una chiamata a *shm_unlink*, o nel caso di file mappato unlink

Come si può vedere nella funzione in figura 4.4 (*Apertura della memoria condivisa)* utilizza la chiamata *shm_open(name, flags, mode)*: apre un oggetto di tipo memoria condivisa, ritornandone il file descriptor, nel caso che *shm_open* ritorni -1 significa l'impossibilità di aprire un oggetto di questo tipo.

L'oggetto puó essere aperto con diversi flags:

- O_RDONLY apre un accesso con sola lettura
- O_RDWR apre un accesso in lettura e scrittura
- O_CREAT crea un oggetto di tipo memoria condivisa
- O_EXCL usato in unione a O_CREAT, crea un oggetto esclusivo

- O_TRUNC azzera la lunghezza dell'oggetto

Successivamente chiama la *ftruncate(fd,dim)* che permette di dimensionare l'area di memoria creata*shm_open,* identificata dal file descriptor *fd ,* pari a dim.

A questo punto viene invocata la *mmap* che permette di mappare la regione di memoria precedentemente creata all'interno dello spazio degli indirizzi del processo. Cosi facendo tutti i processi che mappano la stessa area di memoria si ritroveranno a lavorare su di una risorsa condivisa.

4.4 Gestione della cache

A questo punto la struttura della cache è stata creata e inizializzata; non resta che gestire gli inserimenti e cancellazione in essa.

Ogni volta che arriva una richiesta di una risorsa, da parte di un client, al proxy, la prima operazione è verificare la presenza della risorsa nella cache chiamando la funzione (vedi figura 4.5)

find_in_cache_res(struct dorso res_cache, char* resource_n, char* id_file_ret,*

int dimY, int find_result)*;

se nel campo *find_result* viene assegnato un valore pari a -1 se la risorsa non

è presente in cache e, di conseguenza, la richiesta sarà inoltrata al server di origine. Comunque può accadere che la risorsa sia stata trovata in cache (cioè campo *find_result* viene assegnato un valore pari a 1) ma la richiesta viene lo stesso inoltrata al server di origine; questo accade se l'header della richiesta contiene il campo *Pragma* valorizzato con *no-cache* (la funzione che mi permette questo controllo è *get_pragma_value(char* header)* e mi ritorna 1 se è valorizzato il pragma a no-cache) ciò sta ad indicare che il client non vuole risorse lette da una cache.

```
find_in_cache_res(cache_resource, res_name, id_file_r, DIM_RES_Y, &f_res);

if( f_res == -1 || get_pragma_value(request_from_client) == 1  ){
        /*Se non trovo niente in cache leggo dal server origine*/
        write_socket(cfd, request_from_client, byte_read_from_client, host_log_fd,&proxy_conf);
        read_no_persistent(c_sfd, cfd, host_log_fd, &proxy_conf, cache_resource, res_name);
}
else{

        if ( revalidatete_cache(cache_resource, res_name, cfd, DIM_RES_Y) == 1){
                read_from_cache(c_sfd, id_file_r);

        }
        else {
              write_socket(cfd, request_from_client, byte_read_from_client, host_log_fd,&proxy_conf);
              read_no_persistent(c_sfd, cfd, host_log_fd, &proxy_conf, cache_resource, res_name);
        }
}
```

Figura 4.5: porzione di programma principale dove viene gestita la cache

Quindi se f_res == -1(cioè campo *find_result=-1 risorsa non cachata*) o la funzione *get_pragma_value(request_from_client)* ritorna valore 1, questo significa che, come si può vedere in figura 4.5, la richiesta è inoltrata al server di origine e successivamente chiamata la funzione

read_no_persistent(int client_socket, int server_socket, int log_fd, struct p_conf conf, struct dorso* cache_res, char* res_name)*

che permette, non solo di inoltrare al client la risposta proveniente dal server di origine, ma anche di gestire e prendere decisioni se inserire la risorsa in cache. La prima cosa che quest'ultima funzione effettua è chiamare la funzione *find_last_modified_value(response_from_server, ret_val_last)* che verifica la presenza nell'header del campo *Last_modified* ,ritornando un valore pari ad 1 in caso affermativo.

Idea di fondo per cachare una risorsa, avviene verificando la presenza nell'header del campo *Last_modified;* una qualsiasi risorsa che non presenta tale campo non sarà cachata. Il motivo principale di questa scelta è dovuta principalmente al problema della rivalidazione della risorsa in cache(vedremo a breve il problema di rivalidazione).

Sempre all'interno della funzione *read_no_persistent(),* una volta verificato la presenza del *Last_modified* nell'header, invoca la funzione in figura 4.6 la procedura di inserimento in cache.

```c
void wrapper_cache_ins(struct dorso* page_cache, char* http_header, int dimY,int dimX, char* resource_name, char* file_id){


        int hash_key=0;

        hash_key = hash_func(resource_name,dimY); //calcolo la funzione di hash
        int result_find = insert_in_cache(page_cache, http_header, resource_name, dimY, dimX, file_id);
        /*Se la riga della cache è piena*/
        if ( result_find == -1 ){
                PRINT_DEBUG_RES_CACHE("Riga di cache piena!!!\n");
                libera_blocco_cache(page_cache, hash_key);
                insert_in_cache(page_cache, http_header, resource_name, dimY, dimX, file_id);
        }


}
```

Figura 4.6: funzione di gestione degli inserimenti in cache

In primis è invocata la funzione *hash_func(),*prende come parametro l'host name della risorsa e ritorna la *Key hash* ottenuta come somma di tutti i caratteri dell' host name(castati a int) modulo la DIM_Y (numero di liste dell' hash table).

Ottenuta questa *Key hash* possiamo inserire, alla prima posizione libera della lista di riferimento, la risorsa.

L'inserimento in cache è effettuato chiamando la

insert_in_cache(struct dorso page_cache, char* http_header, char* resource_name,*

int dimY,int dimX, char id_f)*

che valorizza tutti i campi della struttura in figura 4.3, ossia valorizza il campo *Last-Modified* precedentemente verificato, *resource_name* con il nome del host name, empty con un valore pari a 0 per identificare che nella locazione della cache è contenuta la risorsa(piena) e, infine ma di

fondamentale importanza è, il campo *idfile* nel quale viene identificato con un nome univoco, il nome del file nel quale memorizzare la risorsa fisica cachata(ricordiamo che la risorsa cachata è tutto quello che ritorna dal server di origine senza l'header).Questo idfile è ottenuto in base *key-hash* e la posizione *i* interna alla lista. Nel caso in cui la lista key-hash è tutta piena viene liberato il un blocco, semplicemente impostando il campo empty=1, della lista in base alla tecnica da noi decisa, e successivamente inserisco la risorsa. Se la cache è piena ci troviamo nella situazione di decidere quale elemento eliminare.

La decisione di eliminazione dell'elemento, viene gestito in base al numero di letture (utilizzo della risorsa), cioè ogni volta che si inserisce una risorsa in cache, il campo lru, della struttura in figura 4.3,viene valorizzato a DIM_X, e decrementato ogni volta che riutilizzo la risorsa. Quando all'interno della lista ce un elemento con lru <= 1 esso è il candidato ad essere eliminato dalla cache.

Possiamo concludere,come possiamo vedere dalla porzione di codice principale in figura 4.5,il caso in cui f_res == 1(cioè campo *find_result=1 risorsa è cachata*) o la funzione g*et_pragma_value(request_from_client)* ritorna valore -1. In questo caso sappiamo che la risorsa è in cache ed è possibile prendere la risorsa dalla cache. La prima cosa che viene fatta è la rivalidazione della risorsa in cache prima di inviarla al client, chiamando la funzione

revalidatete_cache(struct dorso cache, char* res_name, int s_fd, int dim_cache_y).*

questa funzione ritorna 1 se la risorsa in cache è ancora buona, -1 altrimenti. Come precedentemente abbiamo già menzionato, precedentemente, il campo LastModified è di primaria importanza per questa operazione. Attraverso la funzione

get_head(int server_fd, char resource_name, char* ret_head)*

chiamata internamente a *revalidatete_cache()* permette di creare un pacchetto con il metodo *head* e ottenere dal server di origine solo l'header della risorsa. A questo punto possiamo estrarre da questo header il campo Last-Modified e confrontarlo con quello della stessa risorsa che si trova in cache; se entrambi i valori coincidono la risorsa in cache è ancora utilizzabile e quindi è possibile riutilizzarla passandola al client che l'ha richiesta, altrimenti siamo costretti richiedere la risorsa completa al server di origine.

4.5 Differenza tra cache e cache ip

I definitiva possiamo concludere che come la cache, anche la cache ip ha la stessa logica di funzionamento, soltanto che quest'ultima a come valori da inserire in cache, gli indirizzi ip degli host-name risolti durante la fase di instaurazione della connessione tra client e proxy o proxy e server di origine.

CAPITOLO 5 - Prove d'esecuzione

Di seguito vengono riportate alcune prove di esecuzione del nostro proxy. Lo scopo di tale capitolo è dimostrare il buon funzionamento del programma

```
paolo@AL9000 ~/Programmazione/ProxyHTTP-OK $ ./proxy
Configurazione del proxy iniziata
Configurazione andata a buon fine!
Configurazione andata a buon fine.
ListenOnPort = 5050
MaxReques= 150
DefaultPolicy = 1
ACL-Hosts = 0
ACL-Ip = 0
ACL-MimeType = 0
ACL-HostsPath = ./acl/hosts.acl
ACL-IpPath = ./acl/ip.acl
ACL-MimeTypePath = ./acl/mime.acl
Log Abilitato = 1
Dimensione file di log 10000
```

Figura 5.1: Inizializzazione del proxy

Nella precedente figura è riportata la fase di avvio del proxy (che deve essere conpilato con l'opzione -D VERBOSE o attraverso il MAKEFILE digitando make verbose_proxy) dove vengono stampate a video alcune informazioni sulla attività di configurazione. Come si può notare il proxy è in ascolto sulla porta 5050 ed è in grado di gestire un massimo di 150 connessioni simultanee. In questa figura non sono riportati tutti i parametri di configurazione, ma solo alcuni a titolo di esempio.

Nella successiva figura viene riportata un esempio di apertura di una pagina web, nella fattispecie l'apertura della pagina www.google.it. La scelta è ricaduta su tale pagina in quanto al suo interno ha un numero limitato di risorse, di conseguenza la verbosità generata dall'operazione di apertura e download è limitata e di ancora facile comprensibilità. Come si noterà viene segnalato l'indirizzo IP del client che ha richiesto la connessione (nei nostri test è sempre 127.0.0.1, ovvero l'indirizzo di loopback in quanto browser e proxy risiedono sulla stessa macchina), viene stampata a video la risorsa richiesta e vengono, infine, segnalate alcune informazione al riguardo della terminazione dei processi generati.

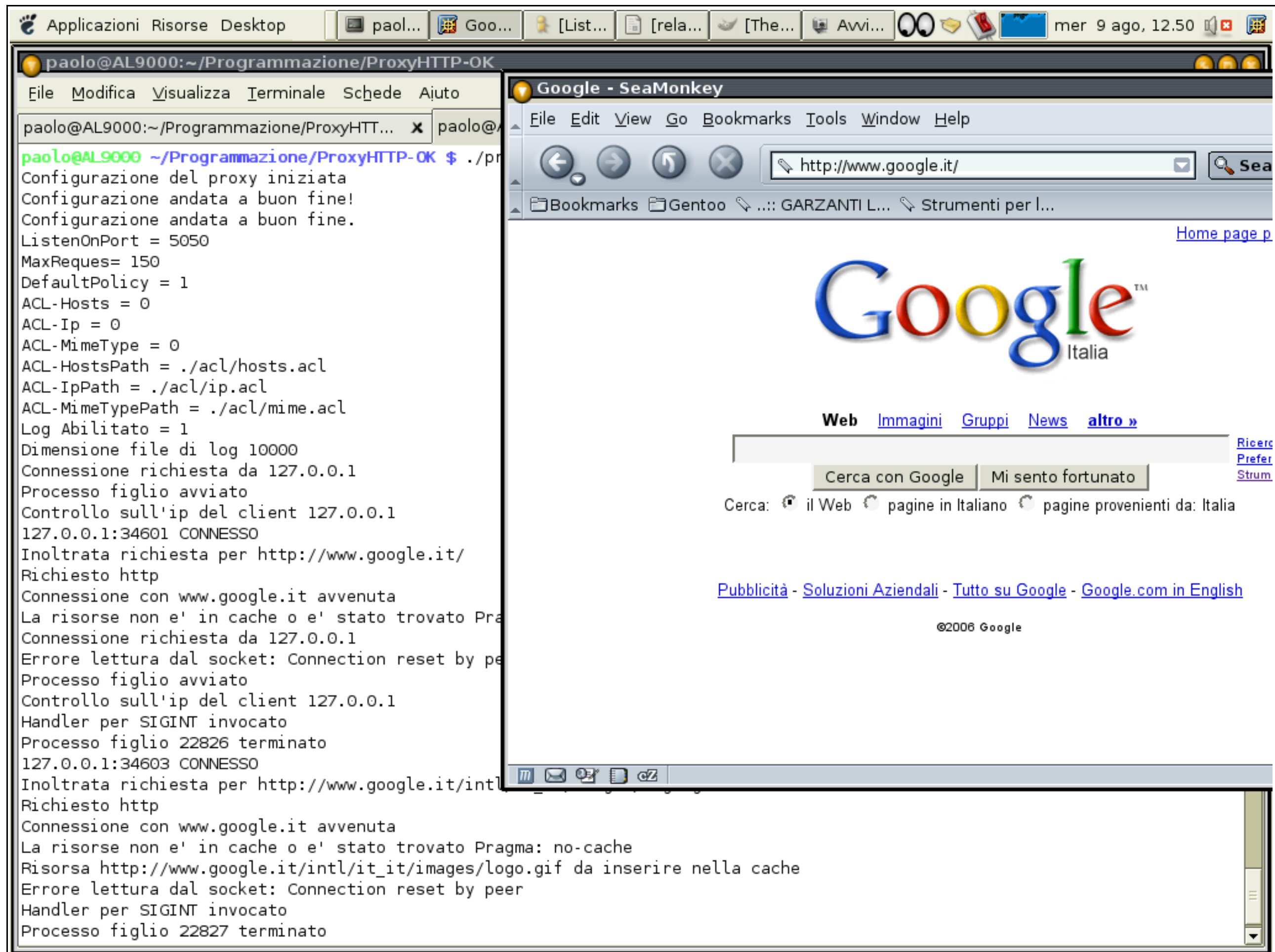

Figura 5.2 - Apertura pagina web

Nelle prossime figure volgiamo dimostrare il buon funzionamento della cache destinata a mantenere le risorse scaricate durante il funzionamento del proxy.

La figura 5.3 riporta il funzionamento del meccanismo di caching. La pagina scelta per effettuare tale test è www.uniroma2.it/didattica, in quanto la maggior parte delle risorse contenute al suo interno rispondono ai criteri di cachabilità da noi scelti. La figura 5.4 dimostra l'effettiva efficacia del nostro meccanismo, infatti tutte le risorse che vengono precedentemente memorizzate nella cache vengono poi effettivamente recuperate, lette e quindi rispedite al client direttamente dal proxy. Per condurre tale esperimento è necessario svuotare preventivamente la cache offerta dal proprio browser per evitare che quest'ultimo rilegga le risorse dalla propria cache.

E' bene altresì notare che tutte le informazioni relative alla cache del proxy vengono perse quando questo viene stoppato, in quanto le aree di memoria condivisa (che ad un elevato livello di astrazione sono file aperti all'interno della directory /dev/shm) vengono rilasciate durante l'arresto

della nostra applicazione.

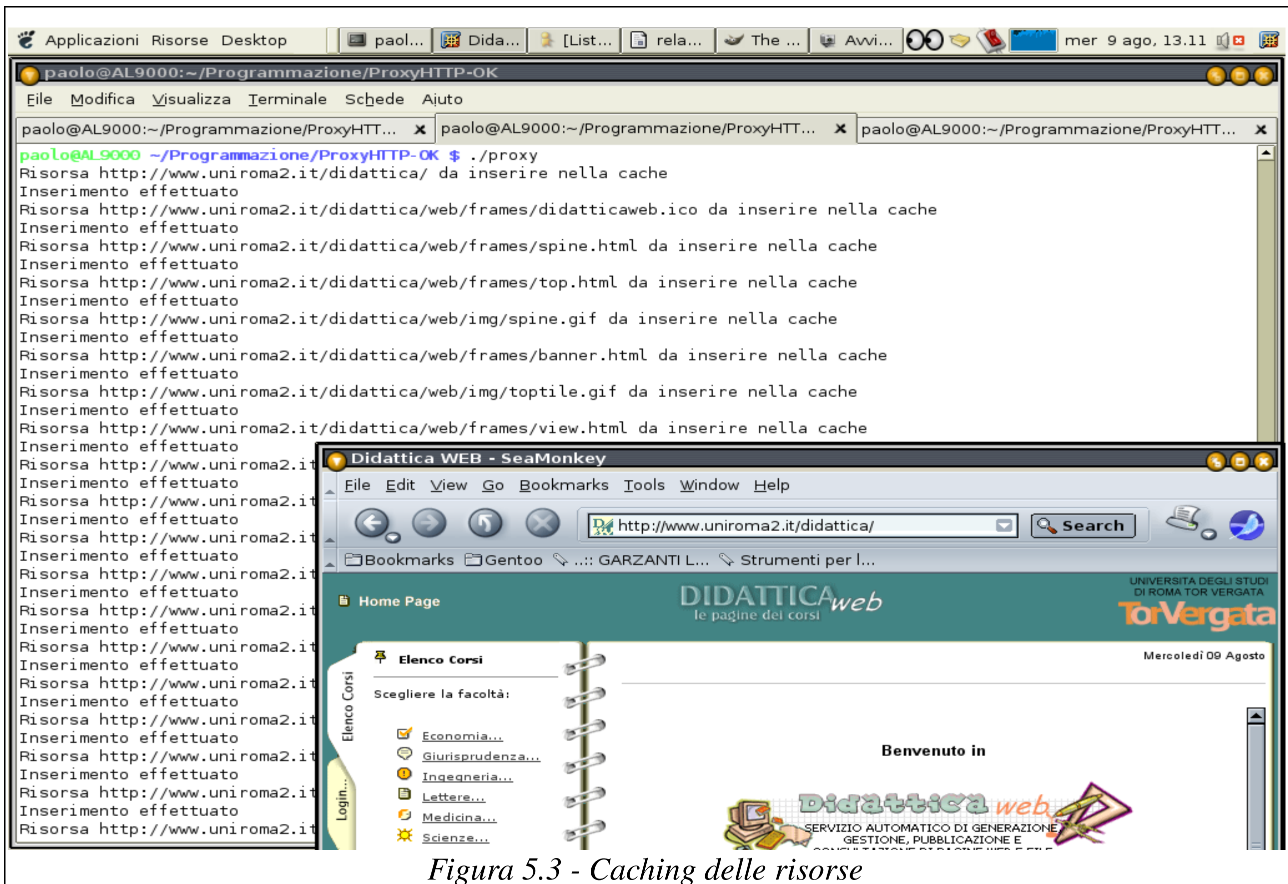

Figura 5.3 - Caching delle risorse

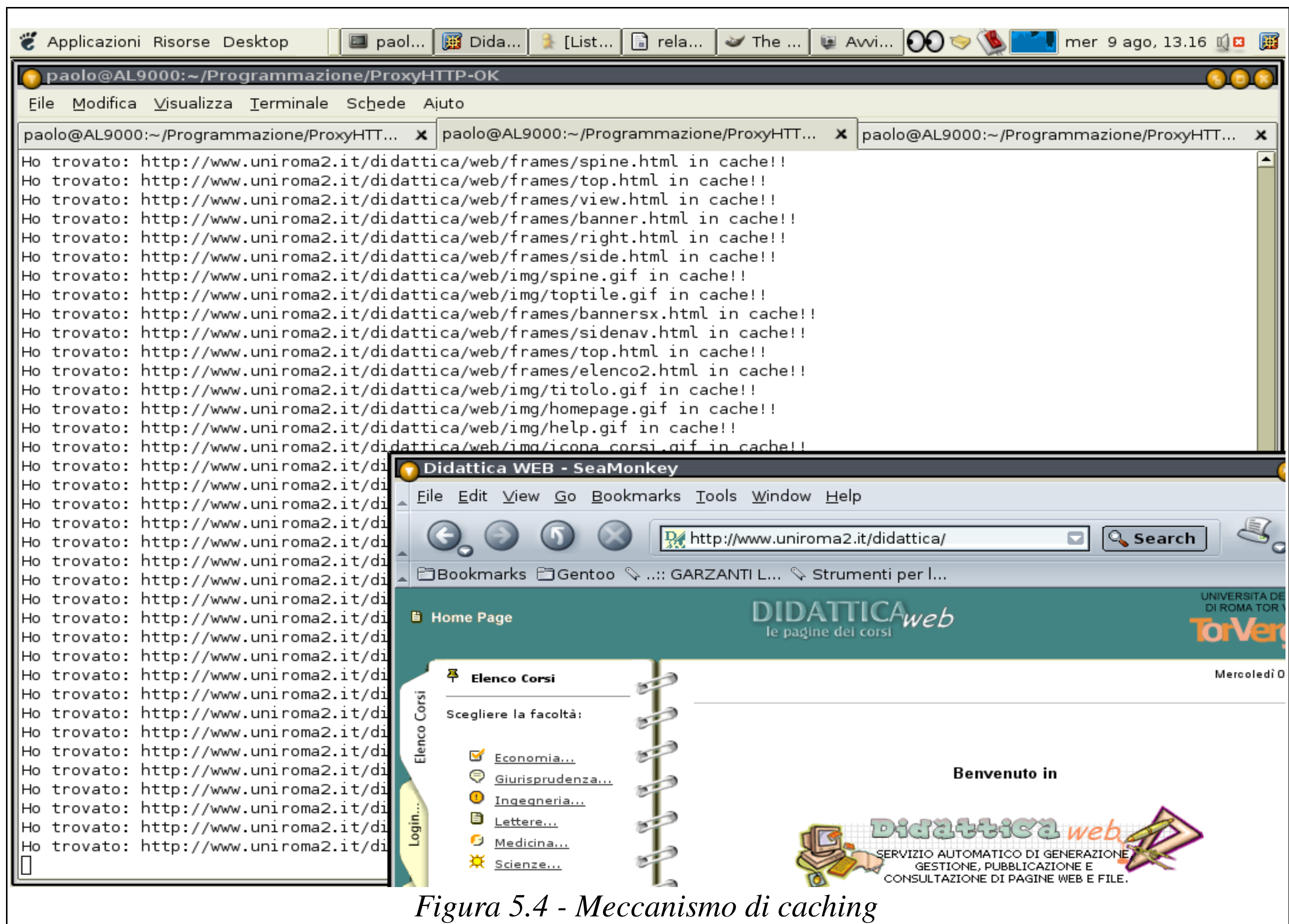

Figura 5.4 - Meccanismo di caching

Dopo aver dimostrato il corretto funzionamento della fase di avvio e del meccanismo di caching, volgiamo ora fornire una dimostrazione del corretto funzionamento dei meccanismi di controllo degli accessi da noi implementati. Tali meccanismi sono 5 (precedentemente descritti), ma per brevità ci limiteremo a dimostrare l'efficacia solo di due di essi.

La prima dimostrazione è stata seguita configurando la sezione relativa al controllo degli accessi nel file *proxy.conf* nel seguente modo:

[AccessControl]

DefaultPolicy= allow

ACL-Hosts= no

ACL-Ip= yes

ACL-MimeType= no

ACL-Extension= no

ACL-Content= no

PathHostsAcl= ./acl/hosts.acl

PathIpAcl= ./acl/ip.acl

PathMimeTypeAcl= ./acl/mime.acl

PathExtensionAcl= ./acl/ext.acl

PathContentAcl= ./acl/word.acl

Come si nota la policy di default impostata è allow ed è stato abilitato il controllo sugli indirizzi ip dei client che richiedono la connessione. Poiché la policy di default è allow la semantica del file *./acl/ip.acl* è di black-list, ovvero tutti gli ip elencati nella lista verranno bloccati. Il risultato di tale situazione è riportato in figura 5.5.

Mantenendo la stessa precedente configurazione, e limitandosi a modificare solo ed esclusivamente la policy di defult impostandola a deny il risultato che si ottiene è diametralmente opposto. Il file *./acl/ip.acl* funge ora da white-list e gli indirizzi IP riportati nella lista sono autorizzati a connettersi.

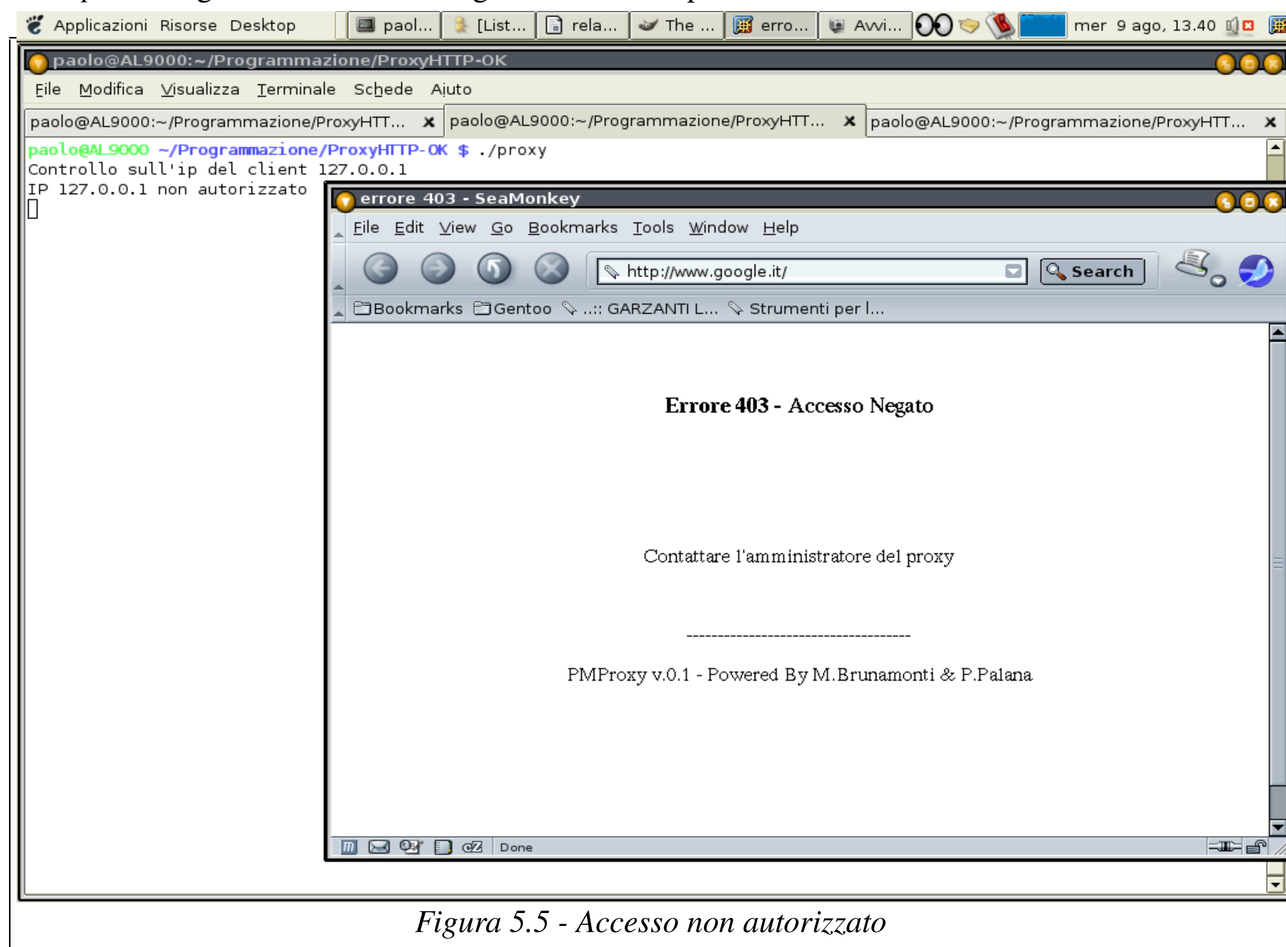

Figura 5.5 - Accesso non autorizzato

Il risultato di tale variazione è riportato in figura 5.6.

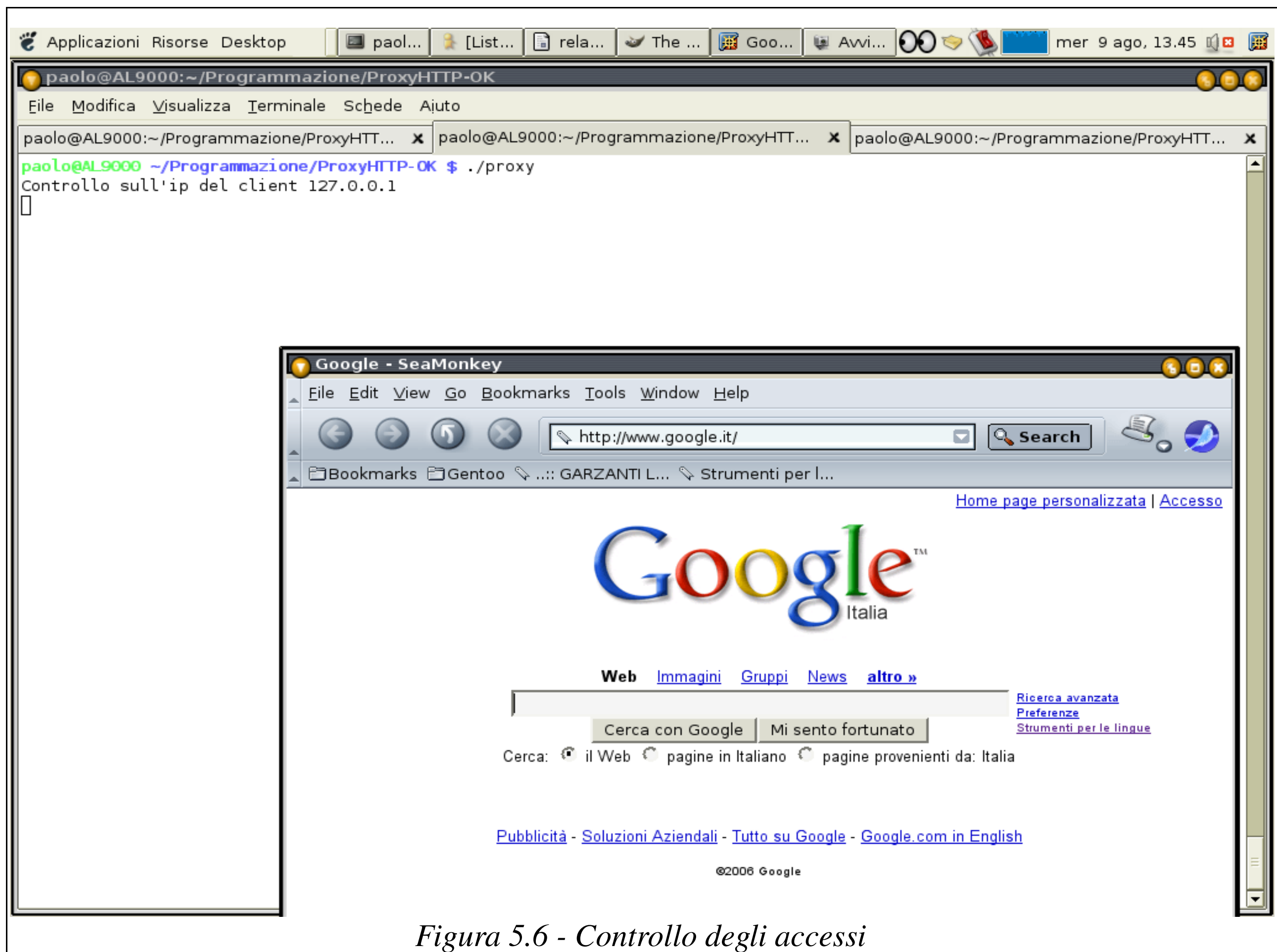

Figura 5.6 - Controllo degli accessi

La seconda dimostrazione, invece, verrà effettuata configurando il proxy nel seguente modo:

[AccessControl]

DefaultPolicy= allow

ACL-Hosts= no

ACL-Ip= no

ACL-MimeType= no

ACL-Extension= yes

ACL-Content= no

PathHostsAcl= ./acl/hosts.acl

PathIpAcl= ./acl/ip.acl

PathMimeTypeAcl= ./acl/mime.acl

PathExtensionAcl= ./acl/ext.acl

PathContentAcl= ./acl/word.acl

Ovvero si è impostata la policy di default ad allow e viene abilitato il controllo sulle estensioni delle risorse per le quali il client effettua la richiesta. Come nel precedente caso con la policy di default allow impostata il file *./acl/ext.acl* assume il ruolo di black-list, di conseguenza non verrà effettuato il download delle risorse la cui estensione comapre all'interno del file. La prova è stata effettuata inibendo il download delle risorse con estensione gif, jpg e jpeg. Il risultato è riportato in figura 5.7.

Il comportamento ottenuto è esattamente quello atteso, ovvero le risorse con estensione gif, jpg e jpeg non sono state scaricate, infatti mancano tutte le immagini.

Modifichiamo ancora una volta la policy di default e impostiamola deny. Ora *./acl/ext.acl* è una white-list, ovvero verrà autorizzato il download solo ed esclusivamente per le risorse con le estensioni elencate nel file.

Se modifichiamo l'acl in maniera tale da consentire il download di sole risorse con estensio html e htm, il risultato che si ottiene aprendo la pagina www.uniroma2.it/didattica è esattamente lo stesso che si era ottenuto nel caso percedente.

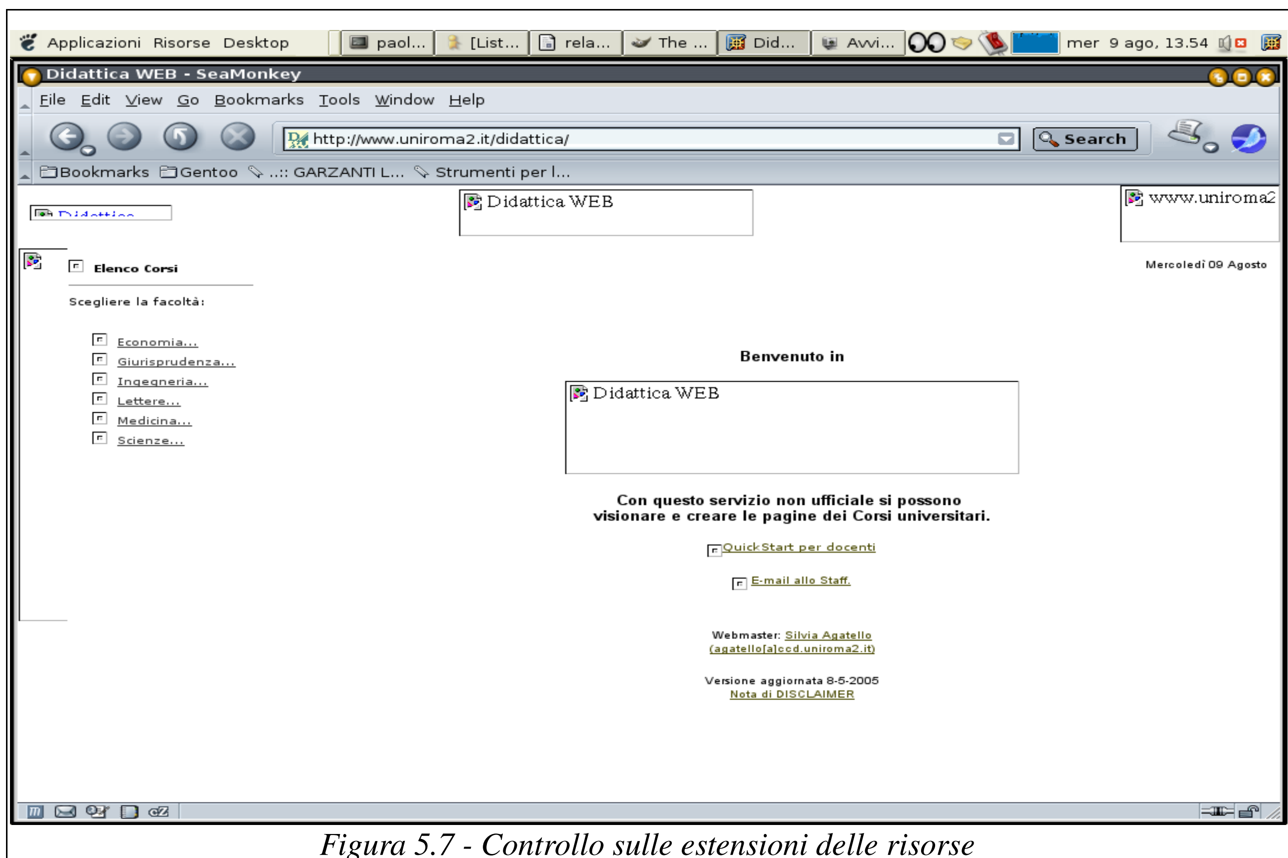

Figura 5.7 - Controllo sulle estensioni delle risorse

APPENDICE A - Readme.txt

Il software contenuto in questo pacchetto, offre le funzionalità di un proxy
con controllo degli accessi basato su:

- *IP dell'host richiedente la connessione*
- *estensione della risorsa richiesta*
- *MIME-TYPE della risposta*
- *nome del server*
- *contenuto testuale della risposta*

Attraveso questo meccanismo consente l'accettazione o meno della richiesta o della risposta.

Un'altra funzionalità offerta dal proxy è la gestione di una cache per una riduzione della latenza percepita dall'utente e del carico della rete.
La politica di rimpiazzamento delle risorse in cache è di tipo LRU
(last recently used).

Il software supporta ed è stato testato con il protocollo HTTP\1.0, quindi per il corretto funzionamento del proxy si consiglia di impostare nel browser di riferimento il protocollo HTTP\1.0, avendo l'accortezza di eliminare l'utilizzo delle connessioni persistenti (vogliamo sottolineare che il proxy funziona anche, utilizzando il protocollo http\1.1 senza connessioni persistenti,ma non viene garantito il corretto funzionamento in tutte le situazioni, in quanto i nostri test sono stati effettuati con le impostazioni sopra citate).

APPENDICE B - Install.txt

Questo file contiene istruzioni riguardo l'installazione del proxy.

Passi da eseguire per la compilazione e installazione del programma

----

PASSO 1 - Compilazione.

Per compilare e quindi generare l'eseguibile del programma ci si può
avvalere del makefile fornito. Questo prevede due diverse alternative.

* Digitando make (o make proxy) si procede alla compilazione e
  generazione dell'eseguibile.

* Digitando make verbose_proxy si procede alla generazione dello stesso
  eseguibile con l'unoica differenza che questo stamperà a video alcune
  informazioni inerenti il funzionamente del programma. Tale funzionalità
  introduce comunque un degrado delle prestazioni a causa della necessità
  di scrivere sullo standard output. Si consiglia dunque di utilizzare tale
  eseguibile solo per scopi di Debug.

PASSO 2 - Installazione.

* Digitando make install si otterrà la generazione delle cartelle (directory)
  necessarie al corretto funzionamento del proxy. E' bene notare che omettendo
  l'esecuzione di tale passo il proxy segnalerà degli errori in fase di
  esecuzione.

PASSO 3 - Esecuzione.

Per eseguire il proxy è sufficiente digitare ./proxy nella directory dove l'eseguibile è stato generato. Una volta avviato il proxy è comunque necessario configurare i client che dovranno utilizzare le funzionalità offerte dal proxy stesso. I parametri necessari (come ad esempio il numero di porta sulla quale il server è in ascolto) sono reperibili nel file proxy.conf.
Si vuole in questa sede anche ricordare che nella versione attuale il software supporta solo HTTP/1.0 e NON SUPPORTA le connessioni persistenti.

PASSO 4 - Disinstallazione

Digitando make clean si ottiene l' eliminazione dei file oggetto, eseguibili e directory create durante la fase di installazione.