Progetto di Reti di Calcolatori: Proxy con Prefetching

Fabian Priftaj <fmp@linuxmail.org>

22 ottobre 2011

Indice

1 Introduzione [*]

1.1 Il modello Client/Server (+Proxy)

E' necessario la realizzazione di un software per la gestione di un sistema **Client/Server**. Il software in questione deve realizzare un *intruso*, detto **Proxy**. Un *proxy* è un programma che si interpone tra un *client* ed un *server* facendo da tramite, ovvero inoltrando le richieste e le risposte dall'uno verso l'altro. Il client si collega al proxy invece che al server, e gli invia delle richieste. Il proxy a sua volta si collega al server e inoltra la richiesta del client, riceve la risposta e la inoltra al client.

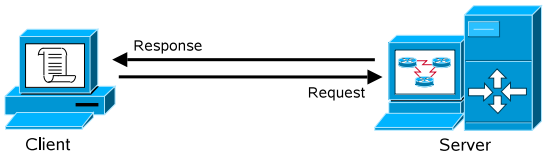


Figura 1.1 Modello Client-Server

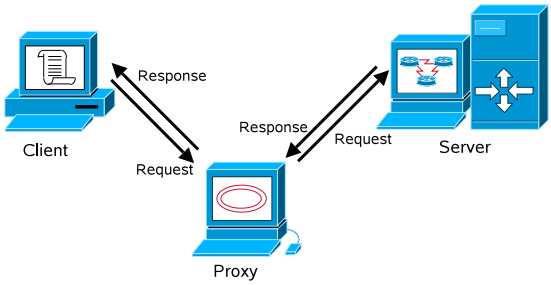


Figura 1.1 Modello Client-Proxy-Server

1.2 Funzionalità del Proxy

L'utilizzo del proxy nasce con l'idea di mascherare al client alcuni dettagli di funzionamento della rete e del server stesso. In particolare il Proxy implementa le seguenti proprieta:

Meccanismo di Caching Può immagazzinare per un certo tempo i risultati delle richieste di un utente, e se un altro utente effettua le stesse richieste il proxy può rispondere senza dover consultare il server originale. Collocando il proxy in una posizione *vicina* agli utenti, questo permette un miglioramento delle prestazioni.

Monitoraggio Permette di tenere traccia di tutte le operazioni effettuate (ovvero di tutte le richieste degli utenti e le eventuali risposte dal server).

1.3 Aspetti tecnici

1.3.1 Compilazione

Il progetto è fornito di un *Makefile* (diverso da quello originale) in modo da facilitarne la compilazione. Per compilare il progetto è sufficiente digitare da terminale il seguente comando:

```
$ make
```

1.3.2 Parametri del programma

`$ proxy --help`
Visualizza l'elenco dei comandi disponibili
`$ proxy --cache-level3`
Abilita il caching di 3° livello (di default caching di 1° e 2° livello sono abilitati)
`$ proxy --no-caching`
Disabilita il supporto della memoria cache
`$ proxy --print-cache`
Visualizza le risorse all'interno della memoria cache durante l'esecuzione
`$ proxy --log logfile`
Abilita il monitoraggio registrando
`$ proxy --maxclients=N`
Numero max. di clients che possono contemporaneamente connettersi al Proxy
`$ proxy --port=N`
Porta utilizzata dal proxy per ricevere le richieste/risposte

E' inoltre possibile combinare l'utilizzo di questi comandi per avere dei comportamenti diversi del Proxy.

1.3.3 Documentazione

La presente documentazione (*.PDF*) spiega quelle che sono state le scelte di implementazione del Proxy; per visionare la documentazione del codice *C* fare riferimento alla documentazione *doxygen*. Per creare la documentazione digitare:

```
$ make doc
```

2 Implementazione

2.1 Parsing

Dato che il compito del Proxy è quello di intercettare le richieste/risposte dal client/server è necessario un attività di parsing in modo da *capire* ciò che viene richiesto dal client durante l'esecuzione dell'applicazione. A seconda delle informazioni estratte dai dati ricevuti, il parsing è suddiviso in tre diverse categorie:

- parsing della risorsa (o indirizzo)
- parsing delle richieste del client
- parsing delle risposte del server

2.1.1 Parsing della risorsa (o indirizzo)

Ogni risorsa è identificata da uno specifico *riferimento,* adatto ad indicare la sua collocazione fisica, composto nel seguente modo:

```
protocol://address:port/[path/]resource.ext
```

L'address-parsing di occupa appunto di estrarre questi tipi di informazioni dal riferimento della risorsa. Le funzioni utilizzate per il parsing dell'indirizzo sono le seguenti:

```
#include <pconst.h>
#include <addrparser.h>

void getResource(char *REF, char *resource);
void getIPaddress(char *REF, char *ipaddr);
int  getPort(char *REF);

boolean isMHTTP(char *REF);
boolean isMHTML(char *REF);
boolean isCORRECT(char *REF);
```

boolean E' un nuovo tipo di dato definito all'interno di `pconst.h` per *semplificare* l'utilizzo di alcune funzioni. La sua struttura è definita come segue:

```
typedef short int boolean;
#define TRUE    1
#define FALSE   0
```

2.1.2 Parsing della richiesta del client

Questo tipo di parsing è necessario per estrarre informazioni da una richiesta ricevuta dal Client.

```
#include <pconst.h>
#include <reqparser.h>
#include <addrparser.h>

boolean isGET(char *request);
boolean isINF(char *request);

void getReqType(char *request, char *type);
void getReqAddress(char *request, char *address);
void getReqRange(char *request, int *start, int *end);
```

Le funzioni `isGET()` e `isINFO()` verificano rispettivamente se una richiesta è di tipo `GET` o `INF`; mentre la funzione `getReqType()` ne estrae direttamente dalla richiesta il tipo e lo memorizza in `*type`. Nel caso in cui il server effettua una richiesta diversa da `GET/INF` o in caso di errori la variabile `*type` assume il valore di `NOT_VALID` (definito in `pconst.h`)

`getReqAddress()` è utile per estrarre l'indirizzo (o il riferimento) della risorsa, memorizzandola in `*address`; in caso di errore `*address` vale `NULL`.

`getReqRange()` serve ad estrarre il range di byte richiesto dal client; l'inizio del range viene inserito in `*start`, mentre la fine in `*end`. Talvolta la richiesta di una risorsa può contenere un range la cui fine non è specificata (ad esempio, `"Range 1-\n`"); in questi casi la variabile `*end` assume valore -1.

2.1.3 Parsing della risposta del server

Questo tipo di parsing è necessario per estrarre informazioni da una richiesta ricevuta dal Server.

```
#include <pconst.h>
#include <
boolean is200(char *response);    /* OK */
boolean is201(char *response);    /* OK RANGE */
boolean is202(char *response);    /* INFO */
boolean is402(char *response);    /* UNKNOWN ERROR */
boolean is403(char *response);    /* WRONG REQUEST */
boolean is404(char *response);    /* FILE NOT FOUND */
boolean is405(char *response);    /* INTERVAL NOT FOUND */

int getResponse(char *response);
int getResLen(char *response);
int getResExpire(char *response);
int getNumResources(char *data);

void getResContent(char *response, char *buffer);

void getINFO(char *text, char *IDX, char *REF, int index);
void getREF(char *text, char *REF);
void getIDX(char *text, char *IDX);
```

Le funzioni `isXXX()` controllano se il codice di risposta è `XXX`; in alternativa si può utilizzare la funzione `getResponse()` la quale effettua il parsing del header del messaggio di risposta e restituisce il codice di risposta del server.

`getResLen()` estrae dal header del messaggio la lunghezza che dovrebbe avere il codice

`getResExpire()` estrae la scadenza dallo header del messaggio

`getNumResources()` restituisce il numero di risorse `IDX+REF` o `REF` all'interno del blocco di dati restituito dal server

`getResContent()` estrae il blocco dati e lo memorizza in buffer; in caso di errori buffer vale `NULL`.

`getINFO()` prende un indice e il testo da cui estrarre le informazioni, e memorizza all'interno di `IDX` e `REF` l'i-*esima* risorsa; in caso di risorse con solo riferimenti `REF`, allora `IDX` assume il valore `NOT_VALID` (definito in `pconst.h`).

2.2 Gestione della memoria cache

La *cache* è una memoria interna al Proxy, utile come supporto per la memorizzazione dei dati ricevuti dal *server*. Essa è implementata come una *lista dinamica* e presenta una struttura dati interna così definita:

```
#include <pconst.h>

typedef struct cache_t {
   char   resource[MAXLENPATH];
   char   response[MAXLENRESP];
   time_t endtime;
   struct server_t sid;
   struct cache_t *next;
}
```

dove:

- `resource` è un unica stringa che rappresenta il percorso relativo della risorsa e il nome della risorsa stessa; questo aiuta ad evitare situazioni in cui il server può contenere diverse directory, dove i file al suo interno presentano contenuti diversi ma nomi uguali;
- `response` indica la risposta completa del server header MHTTP + blocco dati (ovviamente contiene solo le informazioni relative alle risposte con codice **200-OK**)
- `endtime` rappresenta il momento in cui la risorsa scade, ovvero diventa non più valida da fornire al client; (questo comporta la rimozione della risorsa dalla memoria cache);
- `sid` contiene alcune informazioni sull'identificazione del server
- `*next` è un puntatore alla prossima risorsa in cache;

La struttura per l'identificazione del server è la seguente:

```
#include <pconst.h>

typedef struct server_t {
   int  port;
   char address[ADDRLEN];
}
```

dove:

- `port` indica la porta del server a cui il proxy deve collegarsi per richiedere la risorsa (a sua volta richiesta dal client)
- `address` indica l'indirizzo del server
- `ADDRLEN` è una costante che identifica la lunghezza massima di un indirizzo IP (vedere `pconst.h`)

Si potrebbe pensare che l'informazione riguardante la struttura del server (indirizzo e porta a cui appartiene la risorsa) all'interno della cache può risultare inutile; si è deciso di mantenere quest'informazione per evitare situazioni in cui su server diversi possano esistere file diversi ma con percorsi e nomi delle risorse uguali.

2.2.1 Inserire/rimuovere risorse in cache

Per inserire una risorsa in cache è sufficiente fare riferimento alla seguente funzione:

```
#include <cache.h>
#include <pconst.h>
#include <xlib.h>

int cache_insert(struct cache_t **cache, char *resource,
                 char *response, time_t endtime,
                 struct server_t *sid, pthread_mutex_t *mutex);
```

La funzione restituisce 1 in caso di inserimento di una risorsa in cache, altrimenti 0 in caso di un qualsiasi tipo di errore. Gli errori tipici possono essere:
- *la risorsa esiste già in cache*
- *la risorsa esiste in cache ma non è più valida*

I paramteri della funzione hanno lo stesso significato degli attributi della struttura `struct cache_t` vista in precedenza, ad esclusione del parametro `*mutex`. Dato che la memoria cache è una memoria condivisa tra tutti i client (o thread) diventa necessario gestire l'accesso a tale memoria utilizzando i meccanismi della mututa esclusione, per riservare l'integrità dei dati all'interno della cache, ed evitare quindi accessi concorrenziali.

Calcolare il tempo di scadenza Il parametro `endtime` indica il tempo di scadenza di una risorsa, e esso viene rappresentato come un *numero univoco* che identifica un preciso istante di tempo. Per calcolare il tempo di scadenza della risorsa, è sufficiente aggiungere al tempo attuale, ovvero all'istante in cui vengono ricevuti i dati dal server, il tempo di expire. Un esempio è il seguente:

```
...
time_t now;
time_t endtime;

time(&now);
endtime = now + getResExpire("RESPONSE_FROM_SERVER");
...
```

In questo modo posso verificare se una risorsa è scaduta o meno, semplicemente confrontando l'istante in cui la risorsa dovrebbe scadere con l'istante di tempo attuale:

```
...
time(&now);
if (now >= endtime) {
   "RISORSA SCADUTA DA (now-endtime) SEC."
} else {
   "RISORSA ANCORA VALIDA PER (endtime-now) SEC."
}
...
```

Rimozione risorse scadute La rimozione delle risorse non più valide dalla memoria cache viene gestita da un unico thread che rimane in esecuzione durante l'intero ciclo di vita del Proxy. Il thread in questione esegue la funzione `cache_remove_expiry()` (vedere `support/cache.c`) effettuando dei controlli simili a quello appena esposto sul controllo della validità di una risorsa in cache. Dato che le risorse hanno un tempo di validità espresso in secondi, risulterebbe inutile richiamare la funzione di controllo cache più volte in un secondo; perciò la funzione `cache_remove_expiry()` viene eseguita ad ogni secondo. Il corpo della funzione è:

```
while (TRUE) {
   sleep(1);
   cache_remove_expiry(&cache, &mutexcache);
}
```

2.3 Il Proxy

Il Proxy inizialmente inzializza le proprie struttue dati, utili per creare una connessione TCP e successivamente rimane in ascolto per eventuali connessioni dai client.

```
socket();
bind();
listen();

while (TRUE) {
   accept();
   /* gestione del nuovo client connesso */
   ...
}
```

Il Proxy implementa un approcio multi-thread; ad ogni connessione di un nuovo client viene creato un nuovo thread che ha il compito di gestire tutta la fase di connessione/invio/ricezione dati con il singolo client. Lo stesso thread ha anche il compito di connettersi al server, inviare la richiesta e attendere la risposta. Il proxy è in grado di gestire fino ad un massimo di `MAXTHREADS` client contemporanemente (dove `MAXTHREADS` è un limite imposto dal sistema stesso). Le connessioni con il server non possono essere più di 5 contemporaneamente, e per evitare tali situazioni si è deciso di implementare il seguente meccanismo di controllo:

```
pthread_mutex_lock(&mutexdata);
while (contclient > 4) {
   pthread_cond_wait(&condition, &mutexdata);
}
contclient++;
pthread_mutex_unlock(&mutexdata);

/*
 * CODICE DEL PROXY
 */

pthread_mutex_lock(&mutexdata);
contclient--;
pthread_cond_broadcast(&condition);
pthread_mutex_unlock(&mutexdata);
```

Ricezione dati dal server E' noto che il server può introdurre degli errori durante l'invio dei dati al proxy. Questi errori possono essere di 3 tipi:

- *introduzione di ritardi*
- *blocco del server*
- *dati incompleti*

Nel primo e secondo caso viene attuata la stessa politica di gestione: in caso di attesa maggiore o uguale ad un tempo pari ad `INACTIVITY_TIMEOUT_SECONDS=10` la connessione con il client viene interrotta. Nel caso in cui i dati provenienti dal server siano incompleti, si è deciso di effettuare al max. 3 tentavi di ricezione dati, e se dopo questi tentativi i dati risultano ancora incompleti, allora viene scartato ciò che è stato ricevuto fino a quel momento e viene chiusa la connessione con il server segnalando quindi al client che non è stato possibile reperire la risorsa.

```
while (tentativo < 4) {
   ++tentativo;
   if ((nread = data_receive_from_server(serverfd, ...)) != 0)
      break;
   ...
}
```

La funzione `data_receive_from_server()` è così definita:

```
#include <pconst.h>
#include <xlib.h>

int data_receive_from_server(struct server_t *server,
                             char *request, char *response);
```

dove:

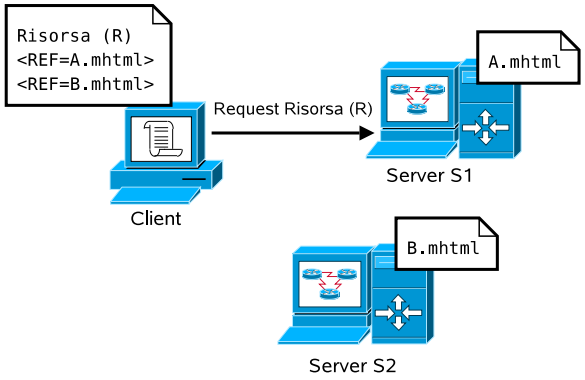
- `server` è la struttura che indica il server da cui ricevere i dati;
- `request` richiesta del client (viene inoltrata al server)

- `response` è il buffer in cui verrà memorizzato la risposta del server

A seconda del valore restituito la funzione 'suggerisce' ciò che è successo durante la ricezione dei dati;

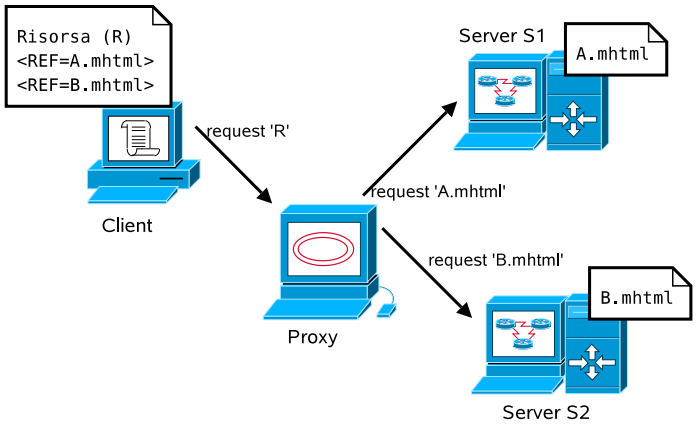
`-2` *impossibile connettersi al server*
`-1` *timeout scaduto (server bloccato)*
`0` *ricevuti dati non completi*
`n` *(n>0) numero di byte ricevuti (risposta corretta)*

Vantaggio Il Proxy permette anche la gestione di alcune situazioni in cui il server altrimenti non è in grado di rispondere adeguatamente al client. Ad esempio supponiamo si verifichi il seguente scenario:



Il client richiede al server **S1** la risorsa *R* la quale al suo interno contiene due riferimenti a due diverse risorse: *A.mhtml* e *B.mhtml*. Il server **S1** al suo interno contiene solo la risorsa *A.mhtml* e non ha alcuna informazione di dove possano trovarsi le altre risorse richieste dal client (in questo caso *B.mhtml*); oltre a questo il server **S1** considera la richiesta del client come non valida e pertanto rifiuta di rispondere in quanto capisce che la richiesta non è a esso indirizzata. Il server **S1** pertanto risponde solo inviando la risorsa *A.mhtml*, ma questo risulta un informazione incompleta per il client il quale segnalerà un errore in quanto non è riuscito ad avere in modo completo l'informazione richiesta. Il server quindi non è in grado di reperire risorse in modo autonomo, ma semplicemenete fornisce al client solo ed esclusivamente le proprie risorse.

Attraverso il Proxy è possibile gestire questo tipo di situazioni, semplicemente effettuando il parsing della richiesta proveniente dal client, ed inoltrandola al server appropriato.



Questo esempio mostra una situazione reale in cui il client è in possesso di informazioni su varie risorse sparse su server diversi, ed in questo modo non deve richiedere le risorse a più server in modo sequenziale e separato, ma può semplicemente inoltrare la singola richiesta al proxy, il quale a sua volta si occupa di gestire tutta la connessione e la richiesta delle varie risorse ai server di competenza.

3 The Source Code