Proxy ©2012

Bergami Giacomo - Paolo de Luca - Elena Fabbri

17 febbraio 2012

Indice

Capitolo 1

Introduzione Generale

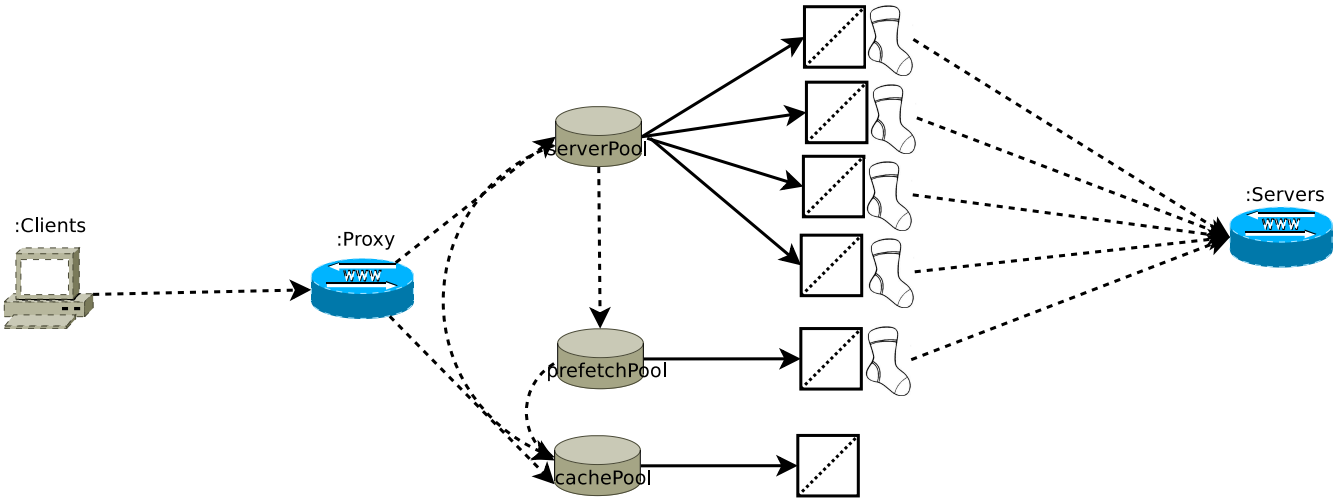

Figura 1.1: *Schematizzazione del meccsnismo di comunicazione Client-Proxy-Server, e rappresentazione ad alto livello dell'architettura del proxy. La semantica di tale rappresentazione è presente in Figura* **??** *a pagina 7..*

Con questa relazione, presentiamo il progetto di Redi di Calcolatori per l'Anno Accademico 2010-2011. Presentiamo la prima versione del proxy, con la quale abbiamo implementato il prefetch su N possibili livelli, con cache implementata tramite file-system.

In questo progetto, abbiamo effettuato una gestione delle richieste tramite una Pool, intesa come "serbatoio" di richieste (che è implementata tramite una lista monodirezionale con sentinella), che può essere utilizzata parametricamente per ogni tipo di richieste che possono essere gestite dai thread gestori, che possono essere utilizzati. Si sono utilizzate queste pool anche allo scopo di evitare il tempo di interleaving dovuti alla presenza di un numero eccessivo di thread all'interno del sistema proxy, ed il tempo della inizializzazione o distruzione delle strutture dati associate all'interno della libreria `pthread`. Per maggiori dettagli, si veda la Sezione 2.2 a pagina 7.

Essendo la cache gestita tramite filesystem, si è rilevato necessario utilizzare una forma di mutua esclusione per effettuare l'accesso alle risorse presenti all'interno della cache su file tramite un meccanismo di hashing: questo permette appunto di effettuare un locking solamente su di una porzione della memoria (ovvero, per tutti gli URL di accesso remoto alla risorsa che collidono con lo stesso numero di hashing), e non sull'intero filesystem. Per maggiori dettagli, si veda il Capitolo 3 a pagina 11.

Sottolineiamo inoltre che la gestione della connessione avviene tramite un numero prefissato di tentativi: se una delle operazione causa l'esaurimento di tutti i tentativi a disposizione, viene effet-

tuato comunque un tentativo di accesso in cache in modo da verificare se la risorsa sia eventualmente a disposizione, altrimenti la comunicazione viene terminata definitivamente.

1.1 Compilazione della guida

La guida è presente sia in questa versione, editata tramite LaTeX, oppure tramite **Doxygen**: per effettuare quest'ultimo tipo di guida, è necessario disporre di `graphviz` e `latex`. Tuttavia, il progetto viene corredato di tutte le guide già compilate.

1.2 Compilazione del progetto

Il progetto include anche la libreria `hashtable` di nostra fattura, con la quale abbiamo gestito il locking sui file della cache. Non sono pertanto richieste librerie aggiuntive fuorchè la `libpthread`. Il progetto principale ha il `main` contenuto all'interno di `init.c`. Sono implementate le seguenti possibilità:

`make clean`**:** Pulisce sia il progetto, sia la libreria.

`make`**:** Effettua la compilazione in *ANSI C*.

`make debug`**:** Effettua la compliazione per il debugging.

1.3 Esecuzione del progetto

Eseguendo il progetto con il comando `proxy -help`, si ottengono le seguenti informazioni:

- Tramite il parametro `-N` o `--tree`, si imposta il livello di profondità richiesto per effettuare il prefetch delle risorse
- Tramite il parametro `-p` o `--port`, si imposta la well-known-port per il proxy
- Tramite il parametro `-i` o `--ip`, si imposta l'indirizzo IP verso il quale effettuare la ricezione delle richieste sulla well-known-port.

Capitolo 2

Dettagli implementativi

2.1 Gestione delle liste

- ◇ **Definizione dei tipi di dato**: `proxy/types.h`
- ◇ **Funzioni utilizzate**: `proxy/uni_list.c`

La gestione delle liste è effettuata in un modo modulare: le funzioni `new`, `dequeue`, `remove_elem`, `enqueue` e `push` trattano esclusivamente la gestione dei tipi di dato delle testate delle liste, che verranno utilizzate per gestire i dati compositi di lista che sono utilizzati in seguito:

Gestione dei JOB : `init_job_queue`, `dequeue_job`, `run_job`, `remove_job`, `alloc_new_job`, `enqueue_job`, `push_job`, `exists_job_res`.

Gestione della coda di prefetch : `update_list`.

2.2 Gestione della Pool

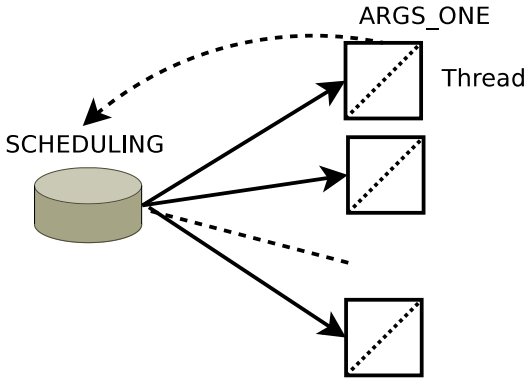

Figura 2.1: *Schematizzazione dell'interazione tra thread che attingono le richieste da una stessa Pool.*

- ◇ **Definizione dei tipi di dato**: `proxy/types.h`
- ◇ **Funzioni utilizzate**: `proxy/thread_functs.c`

Per quanto concerne la gestione delle Pool, ad ogni thread, all'atto della creazione, viene passata staticamente per puntatore una propria struttura dati `ARGS_ONE`, all'interno della quale è contenuto un identificatore numerico del thread correntemente in esecuzione, ed il puntatore per la Pool associata, del tipo `SCHEDULING`. Quest'ultima struttura dati, condivisa tra i thread che si occupano di

gestire le richieste in questione, che vengono prelevate dalla coda di tipo `LIST`, associata al parametro `planning`. L'accesso in mutua esclusione alla lista, è garantita tramite l'uso del mutex `qlock`, mentre si garantisce l'attesa senza busy waiting tramite l'attesa su condizione `non_empty`, sulla quale si ci ferma se non ci sono elementi all'interno della lista (`qsize==0`).

Come elemento della lista, vengono inseriti un puntatore a funzione, e degli argomenti che dovranno essere applicate a questa tramite la funzione `main_add_job`, raccolti all'interno del tipo di dato `JOB`: la chiamata della funzione con i suddetti parametri verrà effettuata dallo stesso thread, che ha come corpo la funzione "server" `thread_memento`, che utilizza un particolare `ARGS_ONE` associato al thread.

Qui sotto forniamo un elenco delle Pool che vengono utilizzate all'interno del nostro progetto:

- *serverPool*: in questa Pool sono inserite tutte le richieste di ottenimento di una risorsa dal server: queste sono gestite da quattro thread, mediante i quali si garantisce che i socket con i quali ci collegheremo per la parte server, siano sempre e soli 4.
- *prefetchPool*: in questa Pool vengono memorizzate quali sono le risorse da scaricare in prefetch: dato il livello di prefetching della risorsa attualmente scandita (es. 1), viene effettuato il parsing della risorsa per ritrovare ulteriori elementi in essi contenuti, per poi riaccodarli se non si è ottenuto il livello prefissato di prefetching. A questa Pool è associato solamente un thread.
- *cachePool*: in questa Pool sono memorizzate tutte le richieste di gestione di ottenimento delle risorse dalla chache: in questa lista vengono anche convogliate le gestioni di invio delle informazioni al client, che inizialmente prevedevano eventualmente un'interazione diretta esclusivamente con il server. Questo verrà dettagliato nella gestione della connessione in Sezione 2.3.

2.3 Gestione delle richieste

◇ **Definizione dei tipi di dato**: `proxy/types.h`

◇ **Funzioni utilizzate**: `proxy/connect.c`

All'interno del `main` effettuiamo il controllo se la risorsa sia già presente in cache: se è presente tale risorsa viene inviata al client, come rappresentato dall'interazione rappresentata dalla Figura 2.2 nella pagina successiva. Se invece non è presente in cache, allora avviene una connessione col server, utilizzando un'interazione descritta dalla macchina a stati della Figura 2.3 a fronte.

La descrizione della comunicazione con il client avviene con la funzione descritta nel file `proxy/toclient.c`, mentre le funzioni di comunicazione con il server sono contenute all'interno di `proxy/toserver.c`: tutte queste funzioni utilizzano la funzione `fetch` e `update_list`, rispettivamente per ottenere la risorsa dal server, e per iniziare ad effettuare il prefetch delle risorse indicate in quella appena scaricata. Sempre facendo riferimento alla Figura 2.3 nella pagina successiva, possiamo innanzitutto precisare come tutte le funzioni presenti nello stato `fetch`, che è di per se una funzione di `proxy/toserver.c`, sono descritte all'interno del file `proxy/connection`, che si preoccupa di gestire la comunicazione con il proxy sia per il prefetch, sia per il download da server, su richiesta di un preciso client.

Le opzioni che vengono impostate sul socket che viene utilizzato per collegarsi con il server, sono le seguenti:

◇ `SO_SNDTIMEO`: se una funzione risulta bloccante, viene settato un timeout, oltre al quale tale funzione restituisce l'ammontare delle informazioni inviate.

◇ `TCP_NODELAY`: disabilita l'algoritmo di Nagle, incrementando conseguentemente la performance, in quanto non consente l'invio di frames parziali, in quanto i parti del buffer vengono inviati istantaneamente, senza aspettare che il buffer venga sufficientemente riempito.

◇ `SO_RCVTIMEO`: analoga alla `SO_SNDTIMEO`, ma utilizzata per la ricezione.

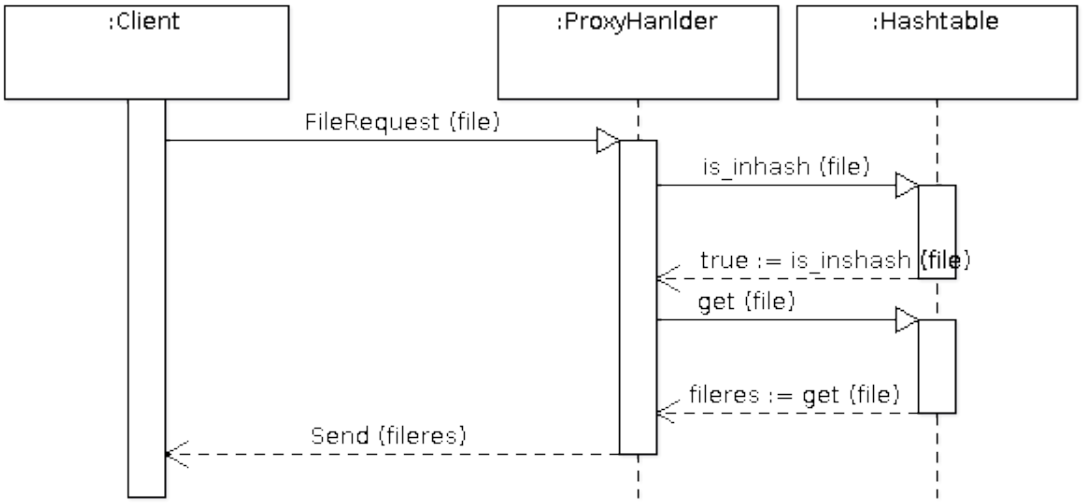

Figura 2.2: *Descrizione dell'interazione Client-Proxy durante una richiesta di una risorsa presente in cache.*

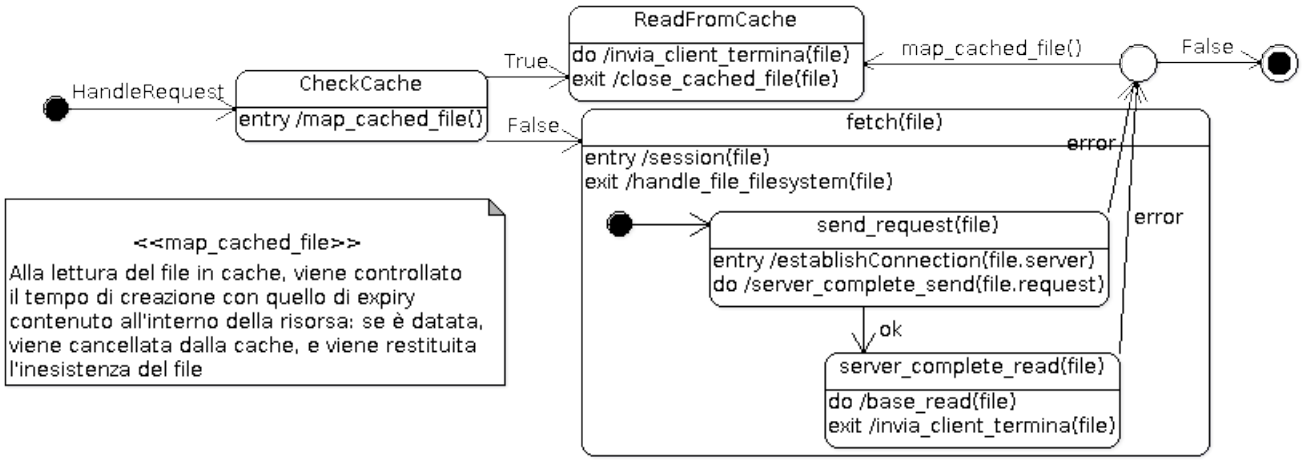

Figura 2.3: *Descrizione dell'automa a stati durante la richiesta di una risorsa non presente in cache.*

2.4 Gestione dello stato del Proxy

◇ **Definizione dei tipi di dato**: `proxy/types.h`

Lo stato del proxy viene mantenuto all'interno di due strutture dati: la prima struttura dati è `struct proxy`, all'interno della quale vengono salvate le informazioni attinenti all'accettazione delle richieste da parte di client(s), l'indirizzo IP e la porta associata, ed il livello di prefetch che viene indicato all'invocazione del programma.

Lo stato della connessione con il server viene invece mantenuto all'interno della struttura dati definita `struct connection`: viene salvato l'indirizzo IP e porta del server al quale connettersi, la

risorsa da richiedere, il numero dei tentativi effettuati ed il file descriptor del client, al quale alla fine eventualmente inviare l'informazione, e del server, al quale inviare appunto la richiesta e ricevere la risposta.

Capitolo 3

Libhashtable

◇ **File di inclusione**: `proxy/libhashtable.h`, `hashtable/libhashtable.h`

Abbiamo deciso di implementare separatamente la hashtable rispetto al proxy, in quanto costituisce un nucelo di codice a sè stante, indipendente dalle funzioni del proxy in senso stretto: unica funzione che è strettamente legata al proxy è `parseHead_time` in `hashtable/hres.c`, in quanto è necessaria per valutare se il file presente in cache sia o meno scaduto.

Come già discusso, lo scopo di utilizzare una hash table è quello di effettuare un locking il più selettivo possibile e indipendente dall'architettura: di fatti, la systemcall `flock` non è implementato in Cygwin, in quanto non appartenente allo standard POSIX[1], mentre nei sistemi Windows è utilizzata `LockFileEx`[2].

Tramite una tabella di hashing, possiamo conseguentemente effettuare il loking in n parti della cache, senza effettuare il lock sull'interno filesystem. Si possono ridurre le collisioni aumentando la dimensione della hash, ma di massima si ridurrebbero solamente le performance del proxy, in quanto si verificherebbero maggiorni collisioni all'interno della tabella di hashing.

Tale hashtable, è implementata come un array di n elementi, dove ciascuna posizione è attribuibile ad un preciso valore di hashing: per quella stessa posizione, sono inoltre indicati se sono presenti elementi o meno; se non è presente alcun elemento, allora non verrà eseguita nessuna syscall, e verrà ritornato un valore negativo. Se invece abbiamo che è presente un elemento al suo interno, allora si deve controllare se questo esiste realmente nella data posizione: questo è necessario in modo da verificare se tra quegli elementi è presente anche la risorsa considerata. Tutte queste operazioni sono effettuate all'interno di `hres.c`.

Riassumiamo ora brevemente il contenuto dei files rimanenti:

filesystem.c : contiene interazioni base con il filesystem, non sempre legate alla hashtable, ma sempre legate al caching: queste funzioni o sono adoperate direttamente a livello di proxy (come `recursiveDelete` che effettua lo svuotamento della cache all'apertura del proxy), o sono adoperate all'interno della libreria

fsys.c : effettua operazioni basi sui files, tra le quali l'allocazione di un'area di memoria atta alla memorizzazione del contenuto del file.

hash.c : contiene le funzioni che permettono di operare sulla struttura dati di hashing in quanto tale, indipendentemente dal fatto che tale struttura dati sia utilizzata allo scopo di effettuare il locking sui files.

lett_scritt.c : sono presenti le funzioni per effettuare il locking.

[1] `http://cygwin.com/ml/cygwin/1999-04/msg00026.html`
[2] `http://goo.gl/Yu031`