Progetto di

Laboratorio di Programmazione di Reti

Anno Accademico 2001/02

Studenti:

- Michetti Alessandro
- Pegoraro Alessandro
- Valenti Roberto

Indice:

- Obiettivo P. 2
- Prime considerazioni P. 2
- Proxy P. 3
- Client P. 5
- Conclusioni P. 6

Obiettivo

Realizzazione di due entità inserite in un'architettura volta al trasferimento di file, operanti concorrentemente su hosts diversi.

Prime Considerazioni

Viste le specifiche del progetto, alcuni punti sono stati chiari fin dall' inizio, e si è proceduto con la stesura di opportuni "impacchettatori" per creare gli header necessari. La comunicazione avviene infatti tramite scambio di dati incapsulati all'interno di pacchetti predefiniti in base alla richiesta, e queste funzioni creano l'header di tali pacchetti.
In seguito, abbiamo ritenuto fosse necessaria la presenza (come sul server) di funzioni di lettura e scrittura sicure, che effettuassero diversi cicli per inviare/ricevere tutti i byte gestendo alcuni tipi di errori.
La parte del progetto che richiedeva più attenzione ed impegno è comunque la comunicazione Proxy/Server, vista l'esigenza di parallelismo con più server. Nonostante la nostra limitata esperienza in programmazione concorrente, abbiamo cercato di gestire fork() e comunicazioni inter-processo.
Ci siamo prefissati, come da specifiche, di soddisfare due requisiti: l'inizio spedizione del file al Client prima del termine della totale ricezione del file sul proxy, ed il mantenimento di una sola connessione attiva con ogni server, per ciascun Client connesso. Non sono mancati i problemi causati da questo tipo di scelta, nonché i dubbi su quale fosse la miglior implementazione per garantire affidabilità e prestazioni.
Dopo alcuni tentativi con pipe(), file tables condivise e select() si è optato per un' area di shared memory per mettere i processi in comunicazione tra loro.

Di seguito descriviamo il comportamento generale delle entità create, una spiegazione più dettagliata e di tipo tecnico/implementativa e' lasciata ai commenti scritti direttamente sui codici sorgenti.

Proxy

Usage :

proxy listenport server1 port1 server2 port2 server3 port3 [server_n port_n]

Il Proxy è l' entità posta al centro dell' architettura: essa scambia pacchetti in entrambe le direzioni con più Client e più Server; accetta richieste di trasferimento dai primi e per ciascuna di queste crea dei processi che a loro volta instaurano delle connessioni con i Server per ottenere dei segmenti di file.
Per mantenere il main() snello e leggibile sono state scritte varie funzioni, atte ad inoltrare o ricevere i pacchetti su socket già collegati e passati come argomento. In particolare, le due funzioni gestore_server() e gestore_client() vengono usate qualora vi sia la necessità di ottenere informazioni dalle rispettive entità; essi invocano, a loro volta, un'altra delle funzioni di gestione I/O, in dipendenza dal tipo di pacchetto, per poi rimanere in attesa o inviare delle informazioni.
Vista la disponibilità che il Proxy deve fornire contemporaneamente a più richiedenti, si è pensato all' implementazione di un listening socket sempre aperto (fino ad n connessioni) su un processo padre, che genererà processi figli per ogni connessione ricevuta.
Dovendo realizzare questa funzionalità abbiamo optato per una fork() che duplica il processo padre. Uno dei due processi (il figlio) chiuderà il listening socket ereditato, mentre l'altro chiuderà la connessione instaurata, rimanendo così pronto ad accettare altre connessioni dal listening socket; il processo figlio e' ora pronto per comunicare con i server.
Quest'ultimo cerca di sapere la dimensione del file richiesto provando ad instaurare una connessione con un qualsiasi server; appena ricevuta crea dei "processi-segmento" tramite un'altra fork(), chiamati così in quanto si preoccuperanno di gestire tutto quello che riguarda il singolo segmento richiesto (attribuito appunto ad un singolo processo), che va dalla richiesta al server, alla gestione degli errori, alla consegna al client in modo ordinato.
La creazione di questi particolari processi e' limitata dal processo padre: esso testa infatti la disponibilità dei server ad accettare nuove connessioni, verificando che non ci siano troppi processi attivi per una singola richiesta e solo allora creerà un nuovo processo-segmento.

La vita di un processo-segmento e' così strutturata:

- nasce
- cerca un server libero e funzionale (ciclo indeterminato)
- occupa il server
- instaura una connessione (se fallisce ricomincia da capo)
- invia la richiesta al suddetto server (se fallisce ricomincia da capo)
- legge la risposta (se fallisce ricomincia da capo)
- se durante le operazioni suddette scade il timeout si "uccide" e termina la connessione con il client
- libera il server
- si mette in attesa che il sia il suo turno per la spedizione al client (se scade timeout si "uccide" e fa cadere la connessione con il client)
- passa il turno al successivo
- muore

Per poter implementare questo schema e' necessario l'uso di comunicazione tra processi (IPC).
La comunicazione inter-processo è garantita da due aree di memoria ad accesso condiviso: sulla prima viene memorizzato un intero che esprime il numero di segmenti ordinati già spediti, mentre la seconda (un array di byte) descrive lo stato delle connessioni coi server. In tal modo, ogni processo-segmento può ottenere un "lock" sulla connessione instaurata con il Server, settando il valore corrispondente ad 1; ricevuto il segmento il processo lo invia al client se, alla lettura del contatore condiviso (che controlla con cicli while) , risulti essere il suo turno. Terminato l'invio la variabile condivisa viene incrementata in modo da poter permettere al segmento successivo di essere a sua volta inviato.

Per comprendere meglio la struttura interna del proxy e' possibile consultare la seguente immagine:

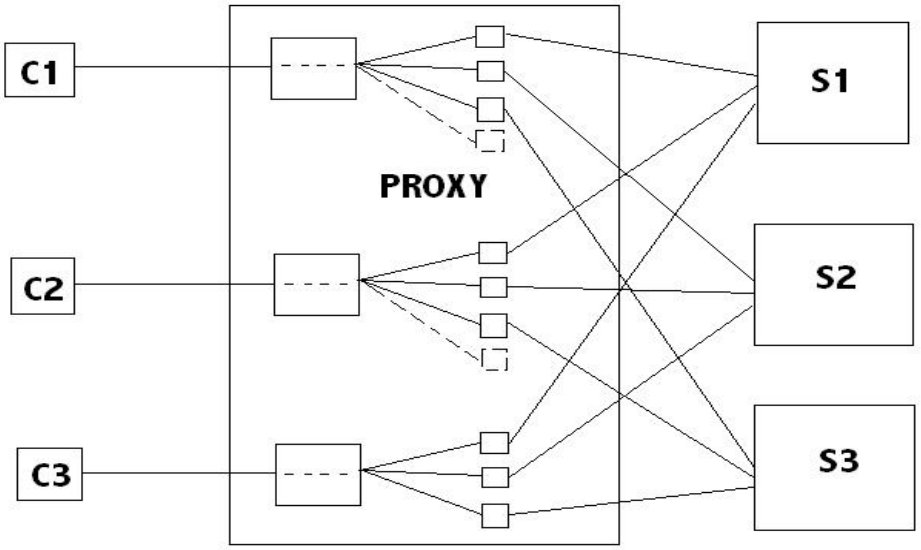

Figura 1: schema del proxy; i vari Cn sono i client, mentre gli Sn sono i server che comunicano col proxy, l'entità centrale.

Client

Usage:

client ServerAddress ServerPort File OutputFile

Il Client rappresenta l' entità più semplice delle tre. Esso tenta una connessione con il proxy specificato a riga di comando (indirizzo e porta) e, creato un segmento tipo GetRequestHeader con una chiamata a makerequest(), lo trasmette e rimane in attesa del file richiesto. Il Proxy risponde dapprima con un invio della dimensione totale, e poi con i segmenti ordinati (durante questa fase il Client sa quanti byte deve attendere). Al termine della ricezione dei segmenti dal Proxy, il Client effettua una chiamata a save() per creare il file ottenuto.

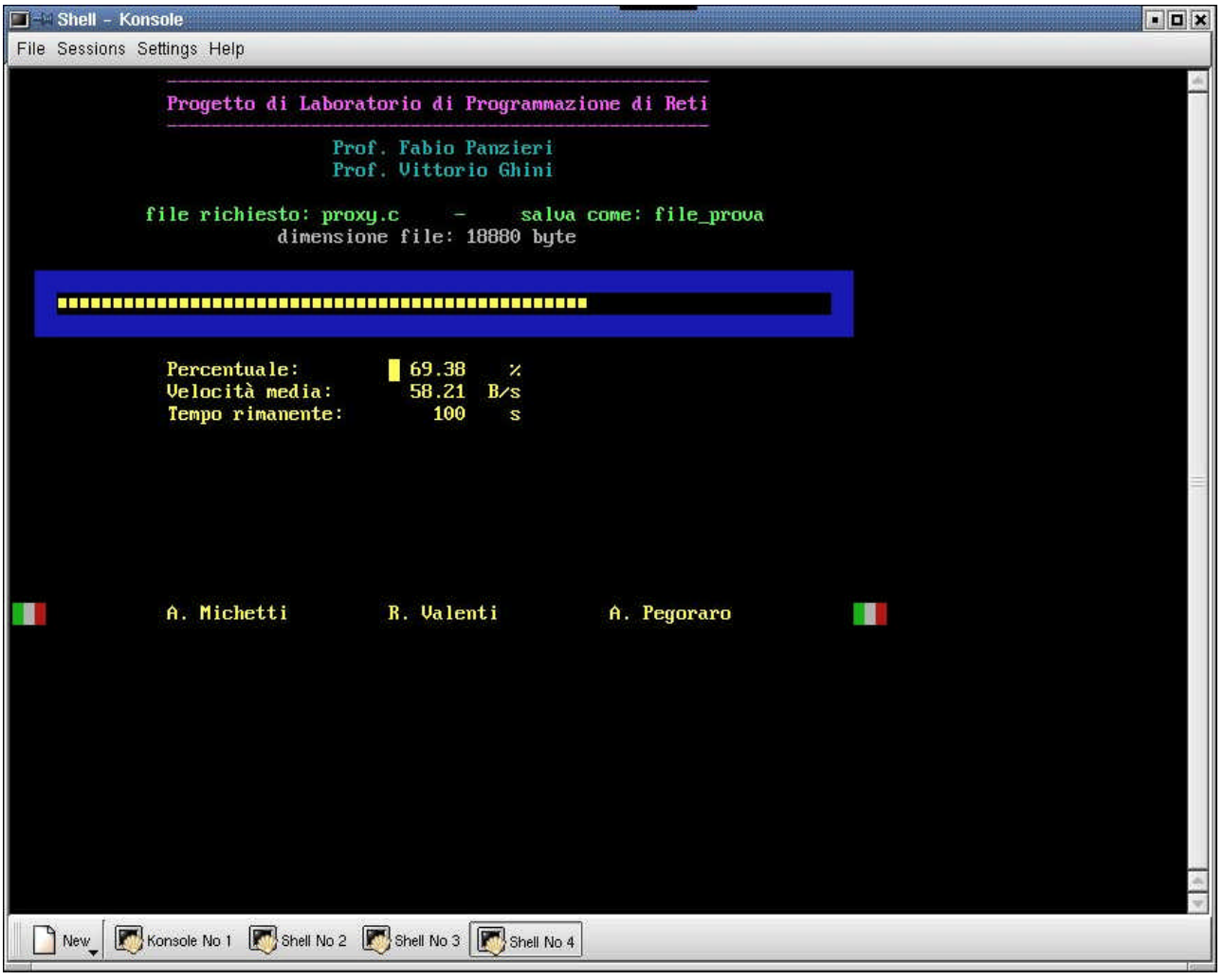

Figura 2: la schermata del client durante il download di un file

Conclusioni

Una volta terminato ci siamo chiesti se non fosse un problema, per il Proxy, avere dei processi che ciclino in un while infinito controllando ripetutamente una variabile (per capire se e' il proprio turno per l'invio del segmento al client). Le prestazioni sembrano in effetti afflitte da questo compromesso adottato per colmare una lacuna in capo di programmazione concorrente, però lo spostamento dei segmenti avviene in modo parallelo, serializzando solo gli invii al Client (ovvio, visto che devono essere ordinati) e questo procedimento inizia prima che l'intero file sia stato ricostruito sul Proxy, come suggerito nelle specifiche. Con la fase di debug abbiamo eliminato alcuni problemi (permessi per le variabili condivise, limitazione dei processi attivi, uso ottimizzato dei server error-prone, etc.), e l'architettura sembra funzionare in tutte le condizioni: uccidendo un server le connessioni si spostano sugli altri, e ri-eseguendolo di nuovo il proxy se ne accorge rimandandogli nuove richieste. Client multipli sono supportati, così come un numero di Server maggiore di tre, fino ad un massimo di MAXSERVERS (definito in "define.h").
Abbiamo anche deciso di impiegare un po' del tempo con funzioni grafiche del Client: è stato utile per dare impatto visivo e per distrarsi dai frustranti tentativi con le fork().
Crediamo di aver svolto un discreto lavoro e, anche se l'applicazione è cpu-bound, dimostra di funzionare variando i limiti dei segmenti (< 100 B) e la velocità segnalata riporta attendibilmente un incremento proporzionale (non lineare) a quello del MAXSEGMENTSIZE.

Michetti Alessandro
Pegoraro Alessandro
Valenti Roberto