Immagine che contiene computer

Descrizione generata automaticamente

Trasferimento file su UDP

Ingegneria di Internet e del Web | Settembre 2020

Caliandro Pierciro

Falcone Gian Marco

Minut Robert Adrian

Sommario

[Traccia del progetto 3](#_Toc51090276)

[Funzionalità del server 3](#_Toc51090277)

[Funzionalità del client 3](#_Toc51090278)

[Trasmissione affidabile 4](#_Toc51090279)

[Architettura e scelte progettuali 5](#_Toc51090280)

[Server 5](#_Toc51090281)

[Implementazione 6](#_Toc51090282)

[Controllo di flusso 6](#_Toc51090283)

[Limitazioni riscontrate 8](#_Toc51090284)

[Testing 9](#_Toc51090285)

[Esempi di funzionamento 10](#_Toc51090286)

[Valutazione delle prestazioni 11](#_Toc51090287)

[Al variare della dimensione della finestra di spedizione 11](#_Toc51090288)

[Al variare della probabilità di perdita dei messaggi 11](#_Toc51090289)

[Al variare della durata del timeout 11](#_Toc51090290)

[Manuale 12](#_Toc51090291)

[Installazione 12](#_Toc51090292)

[Configurazione 12](#_Toc51090293)

[Esecuzione 12](#_Toc51090294)

# Traccia del progetto

Lo scopo de progetto è quello di progettare ed implementare in linguaggio C usando l’API del socket di Berkeley un’applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (socket tipo SOCK\_DGRAM, ovvero UDP come protocollo di strato di trasporto).

Il software deve permettere:

* Connessione client-server senza autenticazione;
* La visualizzazione sul client dei file disponibili sul server (comando list);
* Il download di un file dal server (comando get);
* L’upload di un file sul server (comando put);
* Il trasferimento file in modo affidabile.

La comunicazione tra client e server deve avvenire tramite un opportuno protocollo. Il protocollo di comunicazione deve prevedere lo scambio di due tipi di messaggi:

* messaggi di comando: vengono inviati dal client al server per richiedere l’esecuzione delle diverse operazioni;
* messaggi di risposta: vengono inviati dal server al client in risposta ad un comando con l’esito dell’operazione.

## Funzionalità del server

Il server, di tipo concorrente, deve fornire le seguenti funzionalità:

* L’invio del messaggio di risposta al comando list al client richiedente; il messaggio di risposta contiene la filelist, ovvero la lista dei nomi dei file disponibili per la condivisione;
* L’invio del messaggio di risposta al comando get contenente il file richiesto, se presente, od un opportuno messaggio di errore;
* La ricezione di un messaggio put contenente il file da caricare sul server e l’invio di un messaggio di risposta con l’esito dell’operazione.

## Funzionalità del client

I client, di tipo concorrente, devono fornire le seguenti funzionalità:

* L’invio del messaggio list per richiedere la lista dei nomi dei file disponibili;
* L’invio del messaggio get per ottenere un file
* La ricezione di un file richiesta tramite il messaggio di get o la gestione dell’eventuale errore
* L’invio del messaggio put per effettuare l’upload di un file sul server e la ricezione del messaggio di risposta con l’esito dell’operazione.

## Trasmissione affidabile

Lo scambio di messaggi avviene usando un servizio di comunicazione non affidabile. Al fine di garantire la corretta spedizione/ricezione dei messaggi e dei file sia i client che il server implementano a livello applicativo il protocollo di comunicazione affidabile di TCP con dimensione della finestra di spedizione fissa N (cfr. Kurose & Ross “Reti di Calcolatori e Internet”, 7° Edizione).

Per simulare la perdita dei messaggi in rete (evento alquanto improbabile in una rete locale per non parlare di quando client e server sono eseguiti sullo stesso host), si assume che ogni messaggio sia scartato dal mittente con probabilità p.

La dimensione della finestra di spedizione N, la probabilità di perdita dei messaggi p sono configurabili ed uguali per tutti i processi. I client ed il server devono essere eseguiti nello spazio utente senza richiedere privilegi di root. Il server deve essere in ascolto su una porta di default (configurabile).

Opzionale: Implementare anche il controllo di flusso e il controllo di congestione.

# Architettura e scelte progettuali

## Server

Riguardo l'implementazione del server, naturalmente di tipo concorrenziale, abbiamo deciso di attuare una soluzione ibrida basata sia sui processi che sui thread, così da sfruttare i vantaggi di entrambe le tecnologie.

L’utilizzo di processi ha reso il server più robusto, in quanto un eventuale crash in un determinato processo non comporterebbe danni per gli altri che potrebbero continuare a funzionare senza problemi.

Per alleggerire il carico di lavoro che il server avrebbe dovuto eseguire nel gestire una nuova richiesta di connessione abbiamo implementato il pre-forking, istanziando un numero fisso di processi all'avvio. In questo modo siamo riusciti a limitare l’overhead dovuto alla chiamata della funzione *fork*, cosa che avrebbe altrimenti rallentato il three-way handshake iniziale.

Per far sì che ogni richiesta venga gestita da un diverso processo è stata creata nel processo padre una socket di ascolto principale. Grazie all’utilizzo della funzione *select*, il processo padre può accorgersi dell’arrivo di nuove connessioni e avvisare, con un segnale, un ben determinato processo figlio che sarà incaricato di gestirla.

A questo punto il processo figlio crea un nuovo thread incaricato di mantenere attiva la connessione con il client e di rispondere ai comandi che quest'ultimo invia. Abbiamo scelto di far gestire le connessioni a thread piuttosto che a nuovi processi in quanto creare un nuovo thread è un’operazione meno dispendiosa per la CPU del computer sul quale è in esecuzione il server ed anche molto più veloce.

## Trasmissione

Lavorando a livello applicativo, abbiamo dovuto effettuare delle scelte diverse rispetto a quelle che hanno dovuto effettuare i progettisti del protocollo TCP.

Una delle grosse differenze è che l’unico flusso di dati da trasmettere che è necessario gestire è quello dell’applicazione, nel nostro caso un client ed un server. Inoltre, utilizzando il protocollo UDP al livello inferiore abbiamo potuto trascurare alcuni dettagli per la consegna dei dati e la verifica della correttezza dei dati ricevuti.

# Implementazione

## Controllo di flusso

Per l’implementazione del controllo di flusso abbiamo adottato una strategia leggermente diversa rispetto al protocollo TCP.

Durante lo scambio di messaggi si fa uso principalmente di due funzioni, che abbiamo denominato *send\_tcp* e *recv\_tcp*. Una lato sender e l’altra lato receiver.

Poiché l’obiettivo era effettuare una trasmissione affidabile a livello applicativo, la strategia utilizzata è stata utilizzare il buffer passato nei parametri della funzione *recv\_tcp* come unico buffer per la memorizzazione dei dati ricevuti e di cui si effettua la consegna in ordine, un buffer con dimensione pari ad un intero segmento, compreso di header, per la ricezione dei dati dal livello inferiore su cui si fa uso del protocollo UDP, ed una lista collegata in cui si mantengono tutti i segmenti fuori ordine.

Nella funzione *recv\_tcp* si tiene conto dello spazio rimanente nel buffer passatogli nei parametri utilizzando una struttura dati denominata *recv\_win* che memorizza dati relativi finestra di ricezione, in particolare il campo *rcvwnd*, che tiene conto di quanti dati può ancora memorizzare in ordine nel buffer.

Per effettuare la consegna dei dati in ordine, i segmenti fuori ordine vengono memorizzati in una lista collegata, che viene ispezionata ad ogni arrivo di un nuovo segmento per memorizzarlo in una posizione che tenga conto del suo numero di sequenza e mantenerli così sempre in ordine all’interno della lista.

La lista viene ispezionata periodicamente in base all’arrivo di nuovi segmenti per effettuare la consegna dei dati, a cui segue la liberazione dell’area di memoria e poi l’invio di un ACK al mittente, comunicando qual è l’ultimo byte di cui ha effettuato la consegna in ordine.

Nel momento in cui viene effettuata una consegna in ordine, e quindi vengono copiati i dati all’interno del segmento nel buffer di memorizzazione dei dati, il campo *rcvwnd* della struttura *recv\_win* viene decrementato in base alla dimensione del campo dati del segmento. Il valore all’interno di questo campo viene letto ogni volta che si prepara il header di un nuovo segmento da inviare, comunicando in questo modo al mittente quanti dati può ancora ricevere.

Il mittente estrae questo dato dal header del segmento di riscontro e verifica se può inviare ulteriori dati effettuando la differenza tra l’ultimo byte che ha inviato e che al momento è in volo e l’ultimo byte riscontrato, ottenendo così un valore confrontabile con la *rcvwnd*; se la differenza tra *rcvwnd* e questo valore è maggiore di 0 si procede a inviare altri dati, altrimenti, se minore o uguale a 0 si interrompe la trasmissione dei dati e si ritorna al chiamante il numero di byte che si è riusciti a trasmettere e di cui si è ricevuto il riscontro.

Il chiamante di *send\_tcp* potrà iniziare una nuova trasmissione nel caso debba inviare il resto dei dati, tuttavia la trasmissione verrà sbloccata solamente nel momento in cui il destinatario avrà effettuato una nuova chiamata a *recv\_tcp*, passando un buffer con nuova memoria disponibile.

Server e Client FTP

Le operazioni che il client mette a disposizione dell’utente sono:

* LIST: per visualizzazione i file disponibili sul server;
* GET: per effettuare il download di un file dal server;
* PUT: per effettuare l’upload di un file sul server;
* HELP: per visualizzare il menu e, di conseguenza, i comandi che l’utente può inserire.

Tutti i comandi, oltre al trasferimento del file vero e proprio e ad eventuali messaggi di errore, viaggiano sulla rete grazie ai comandi *send\_tcp* e *recv\_tcp* già descritti in precedenza, naturalmente previa connessione tra client e server grazie all’utilizzo delle socket.

Inoltre, per permettere il corretto trasferimento dei files, si utilizzano due funzioni rispettivamente denominate *RetrieveFile* e *SendFile,* la prima che appunto permette di inviare un file (usata lato server nel caso di *get* e lato client nel caso di *put*), e la seconda per permetterne la ricezione (lato server in caso di *put* e lato client in caso di *get*).

La *SendFile,* dopo aver aperto il file richiesto, entra in un ciclo *while* nel quale legge i bytes del file, li salva in un buffer ausiliario di lunghezza pari al massimo numero di byte che il receiver può accettare e li invia al destinatario sfruttando per l’appunto la funzione *send\_tcp*. Il *while* termina nel momento in cui tutti i bytes sono stati inviati o se ci dovesse essere un errore nella *send\_tcp*.

Analogamente la *RetrieveFile*, dopo aver creato il file, entra in un ciclo *while* nel quale legge i bytes dalla socket grazie alla funzione *recv\_tcp*, li alloca in un buffer ausiliario e li scrive sul file precedentemente creato. Anche questo *while* termina solo nel momento in cui il receiver non trova più byte da leggere dalla socket, cosa che comporta la corretta ricezione del file, o nel momento in cui dovesse avvenire degli errori nella *recv\_tcp* o nella scrittura del file.

# Limitazioni riscontrate

# Testing

Indicazione della piattaforma software usata per lo sviluppo ed il testing del sistema.

# Esempi di funzionamento

# Valutazione delle prestazioni

## Al variare della dimensione della finestra di spedizione

## Al variare della probabilità di perdita dei messaggi

## Al variare della durata del timeout

# Manuale

## Installazione

## Configurazione

## Esecuzione