Relazione progetto: Trasferimento file su UDP (B1)
Reti di Calcolatori ed Ingegneria del Web

Studente: Pierpaolo Spaziani
Matricola: 0243651
Elaborato: Codice

Descrizione

Il progetto realizzato è un'applicazione multi-processo client-server per il trasferimento di file impiegando il servizio di rete UDP.
Implementata nel linguaggio C per macchine UNIX e utilizzando l'API del socket di Berkeley, l'applicazione offre un servizio affidabile dato dal protocollo Go-Back-N implementato a livello applicativo.

Manuale

La cartella del progetto è così organizzata:
- *server.c/client.c* : contengono il main lato server/client;
- *server_folder/client_folder* : contengono i file che l'applicazione lato server/client utilizza;
- *server_file/client_file* : contengono i file disponibili allo sharing lato server/client.

Eseguire "*gcc server.c -o server*" e "*gcc client.c -o client*" da riga di comando per compilare il codice ed avere i due eseguibili nominati *server* e *client*.

Entrambi utilizzano la stessa sintassi:

./server <Chunk Size> <Window Size> <Loss Rate> <Timer>

- **Chunk Size** : intero compreso tra 1 e 512, specifica la grandezza del campo *"Data"* dei Pacchetti utilizzati;
- **Window Size** : intero maggiore o uguale di 1, specifica la grandezza della finestra utilizzata nel Go-Back-N;
- **Loss Rate** : decimale compreso tra 0.0 e 1.0, specifica la probabilità che un Pacchetto/ACK venga perso. Opzionale, pari a 0.0 se non specificato;
- **Timer** : decimale, specifica il tempo atteso prima che il Pacchetto venga considerato perso. Opzionale, se non specificato ne verrà calcolato uno (versione ottimale).

Per il client basta sostituire *server* con *client*.

IMPORTANTE : server e client devono avere stessa Chunk Size!
Se specificato, il Timer in ricezione deve essere minore di quello di chi spedisce.

Non è necessario specificare indirizzo IP e numero di porta poiché, essendo un'applicazione sviluppata per scopi universitari, si presuppone vengano eseguiti su macchina locale sia lato client che server ed è quindi possibile utilizzare parametri fissi.

Formato Messaggi

Prima di analizzare il funzionamento dell'applicazione è utile vedere il tipo di messaggi che vengono scambiati.

Pacchetto :

TIPO	NUMERO DI SEQUENZA	CLIENT PID	SERVER PID	DATA

Tipo :

- **0** : Pacchetto di richiesta;
- **1** : Pacchetto della sequenza;
- **2** : il file è vuoto;
- **3** : il file non esiste;
- **4** : ultimo Pacchetto.
- **5** : Pacchetto di conferma ricezione ultimo ACK;

ACK :

TIPO	NUMERO DI SEQUENZA	CLIENT PID	SERVER PID

Tipo :

- **0** : il file già esiste;
- **1** : la trasmissione può procedere (utilizzato dopo un messaggio di richiesta);
- **2** : ACK di risposta;
- **4** : ACK per l'ultimo pacchetto (risposta al Pacchetto di Tipo 4).

In entrambi, i campi *Client Pid* e *Server Pid* rappresentano i *pid* dei processi che stanno comunicando e vengono utilizzati rispettivamente dal client e dal server per riconoscere se i messaggi ricevuti sono per loro o per altri processi.

Funzionamento

Avviato, il **server** è pronto a ricevere e gestire le richieste dei client.
Per ogni richiesta ricevuta crea un processo figlio tramite la *fork* con il compito di gestirla, per poi terminare una volta soddisfatta.
In questo modo il server è sempre in ascolto pronto a ricevere nuove richieste.

Avviato il **client**, l'utente ha a disposizione 3 comandi:

- *list* : permette di ricevere la lista di file disponibili per il download;
- *get 'file_name'* : richiede il download del file specificato;
- *put 'file_name'* : richiede di fare l'upload del file specificato.

Ogni volta che l'utente digita un comando, il client analizza il testo ricevuto riconoscendo, se presente, quale dei 3 comandi è stato scelto e genera un processo figlio con la *fork* incaricato di soddisfare la richiesta dell'utente.
Terminato il suo compito, con o senza successo, il processo figlio termina mandando un segnale di SIGCHLD al padre che lo intercetta e lo gestisce per non lasciare che i processi zombie sovraccarichino il sistema (stessa cosa accade nel server). In questo modo il client è in grado di svolgere più mansioni in parallelo.

Se l'utente digita un comando corretto, il processo figlio invia al server attraverso la funzione *sendto* un messaggio di richiesta di Tipo "Pacchetto" con campi:

0	**0**	**Pid del processo figlio**	**0**	**Input dell'utente**

In questo modo il client invia al server la richiesta dell'utente specificando il Tipo 0 e comunicando inoltre il suo *pid*.
Il server tiene traccia di questa informazione e la inserisce in ogni risposta nell'apposito campo, sia Pacchetto che ACK, aggiungendo inoltre nel campo *Server Pid* il pid del processo appena creato.
A questo punto il client conosce il pid del processo server a cui è stato assegnato e lo inserirà nei messaggi successivi, e viceversa.

Entrambi quando ricevono messaggi con la *recvfrom* utilizzano il flag MSG_PEEK, in questo modo il messaggio non viene rimosso dalla coda ed effettuano un controllo sul pid per verificare se il messaggio ricevuto è per loro; solo in caso affermativo viene eseguita una nuova *recvfrom* con flag 0 per rimuovere il messaggio dalla coda per poi procedere.

Analizziamo adesso le 3 funzionalità dell'applicazione separatamente guardando in parallelo lato server e lato client.

LIST

SERVER	**CLIENT**
	Invia la richiesta di *list*, parte il timer nel caso il server sia offline oppure vengano persi i messaggi e rimane in attesa di una risposta.
Ricevuta la richiesta, esegue l'accesso alla directory dei file disponibili per lo sharing e, allocando spazio dinamicamente, concatena i nomi dei file in una stringa inserendo un separatore tra ognuno per creare in output un elenco puntato. In questo modo nel caso fosse necessaria una ritrasmissione, basta riposizionare il puntatore sulla stringa. Quindi crea e invia i *Pacchetti* di Tipo 1 contenenti i vari frammenti della lista rispettando il protocollo Go-Back-N. Terminato l'invio della stringa crea e invia il *Pacchetto Terminale*, ovvero di Tipo 4 con campo "*Data*" vuoto.	

	Per ogni Pacchetto ricevuto, il client crea e invia un messaggio di *ACK* di Tipo 2 stampando in output il contenuto del campo *"Data"* del Pacchetto. Tenendo traccia del *"Numero di Sequenza"* dell'ultimo Pacchetto ricevuto in ordine, evita di stampare nuovamente dati di pacchetti ricevuti in precedenza in caso di ritrasmissione da parte del server. Dopo ogni invio viene reimpostato il timer per controllare che il server sia ancora online. Una volta ricevuto il *Pacchetto Terminale* il processo termina dopo aver creato e inviato il corrispettivo *ACK* con Tipo 4.
Ricevuto l'*ACK* con Tipo 4 il processo termina, invece nel caso venisse perso, il protocollo Go-Back-N permette di non rimanere bloccati poiché, dopo un certo numero di ritrasmissioni (che non arriveranno mai a destinazione essendo il corrispettivo processo client terminato), il processo termina come se il client non fosse più raggiungibile anche se ha ricevuto con successo tutti i Pacchetti, ma ciò non è un problema poiché il client vede la sua richiesta soddisfatta.	

GET

SERVER	CLIENT
	Invia la richiesta di *get*, crea un nuovo file vuoto per copiare il file richiesto, parte il timer nel caso il server sia offline oppure vengano persi i messaggi e rimane in attesa di una risposta.
Ricevuta la richiesta, controlla se il file sia presente nella cartella dei file per lo sharing.	

- In caso negativo crea e invia un Pacchetto di Tipo 3 con campo“*Data*” vuoto;
- Se il file esiste ma è un file vuoto crea e invia un Pacchetto di Tipo 2 con campo “*Data*” vuoto;
- Se invece il file esiste e non è vuoto, rispettando il protocollo Go-Back-N, procede con l'invio di Pacchetti di Tipo 1 contenenti nel campo *“Data”* i frammenti di file prelevati utilizzando la funzione *read*. Terminato l'invio del file, crea e invia il *Pacchetto Terminale*, ovvero di Tipo 4 con campo “*Data*” vuoto. Nel caso fosse necessaria una ritrasmissione non si verificano problemi, poiché, prima di eseguire la lettura della specifica porzione di file, viene sempre eseguita la funzione *lseek* con flag SEEK_SET per riposizionare il puntatore del file nella posizione giusta calcolata moltiplicando la *Chunk Size* per il *Numero di Sequenza* dell'ultimo ACK ricevuto.

- Se il *Pacchetto* ricevuto è di *Tipo 3*, il file richiesto non esiste. Il processo termina stampando in output il responso;
- Se il *Pacchetto* ricevuto è di *Tipo 2*, il file richiesto esiste ma è un file vuoto e viene quindi salvato come tale. Il processo termina stampando in output il responso;
- Se il *Pacchetto* ricevuto è di *Tipo 1*, il file richiesto esiste.
  Per ogni Pacchetto ricevuto, il client crea e invia un messaggio di *ACK* di Tipo 2 scrivendo sul file creato precedentemente il contenuto del campo *“Data”* del Pacchetto utilizzando la funzione *write*.

	Mantenendo traccia del *"Numero di Sequenza"* dell'ultimo Pacchetto ricevuto in ordine, evita di scrivere nuovamente sul file dati di pacchetti ricevuti in precedenza in caso di ritrasmissione da parte del server. Dopo ogni invio viene reimpostato il timer per controllare che il server sia ancora online. Una volta ricevuto il *Pacchetto Terminale* il processo termina dopo aver creato e inviato il corrispettivo *ACK* con Tipo 4.
Ricevuto l'*ACK* con Tipo 4 il processo termina, invece nel caso venisse perso, il protocollo Go-Back-N permette di non rimanere bloccati poiché, dopo un certo numero di ritrasmissioni (che non arriveranno mai a destinazione essendo il corrispettivo processo client terminato), il processo termina come se il client non fosse più raggiungibile anche se ha ricevuto con successo tutti i Pacchetti, ma ciò non è un problema poiché il client vede la sua richiesta soddisfatta.	

PUT

SERVER	CLIENT
	Invia la richiesta di *put*, parte il timer nel caso il server sia offline oppure vengano persi i messaggi e rimane in attesa di una risposta.
Ricevuta la richiesta, controlla nella directory dei file disponibili per lo sharing se il nome del file esiste già. • In caso affermativo, crea e invia un ACK di Tipo 0; • Se invece non è presente alcun file con quel nome, crea e invia un ACK di Tipo 1, indicando quindi che la trasmissione può procedere. Rimane quindi in attesa dei	

Pacchetti da parte del client.

- Se l'*ACK* ricevuto è di *Tipo 0*, il server ha già un file denominato allo stesso modo e non è quindi permesso procedere. Il processo termina stampando in output il responso;
- Se l'*ACK* ricevuto è di *Tipo 1*, è possibile procedere con l'upload del file rispettando il protocollo Go-Back-N, con l'invio di Pacchetti di Tipo 1 contenenti nel campo *"Data"* i frammenti di file prelevati utilizzando la funzione *read*. Terminato l'invio del file viene creato e spedito il *Pacchetto Terminale*, ovvero di Tipo 4, con campo "*Data*" vuoto. Nel caso fosse necessaria una ritrasmissione non si verificano problemi, poiché, prima di eseguire la lettura della specifica porzione di file, viene sempre eseguita la funzione *lseek* con flag SEEK_SET per riposizionare il puntatore del file.

Per ogni Pacchetto ricevuto, il server crea e invia un messaggio di *ACK* di Tipo 2 scrivendo sul file creato precedentemente il contenuto del campo *"Data"* del Pacchetto utilizzando la funzione *write*. Mantenendo traccia del *"Numero di Sequenza"* dell'ultimo Pacchetto ricevuto in ordine, evita di scrivere nuovamente sul file dati di pacchetti ricevuti in precedenza in caso di ritrasmissione da parte del client. Dopo ogni invio viene reimpostato il timer per controllare che il client sia ancora online. Una volta ricevuto il *Pacchetto Terminale* il processo crea e invia il corrispettivo *ACK* con Tipo 4.

Ricevuto l'*ACK* con Tipo 4 crea e invia un *Pacchetto* di Tipo 5 per confermare la ricezione dell'ultimo ACK. Il processo quindi termina.

Ricevuto il Pacchetto di Tipo 5 il processo termina, invece se viene perso, il protocollo Go-Back-N permette di non rimanere bloccati poiché, dopo un certo numero di ritrasmissioni (che non arriveranno mai a destinazione essendo il corrispettivo processo client terminato), il processo termina come se il client non fosse più raggiungibile, ma ciò non è un problema poiché la richiesta del client è stata soddisfatta.

Cartella Temporanea

Quando il **server** sta ricevendo un file, lo posiziona in una cartella utilizzata per file temporanei in modo tale che, se riceve una richiesta *get* o *list*, non lo troverà tra i disponibili. Una volta terminata la ricezione lo sposta tra gli altri.
Specularmente il **client** utilizza la cartella di file temporanei sia, quando sta ricevendo un file, per evitare la disponibilità allo sharing di file non completi, sia quando li invia, per evitare cambiamenti al file che sta trasferendo. In entrambi i casi, terminata l'operazione, il file viene spostato tra gli altri e reso disponibile.

Go-Back-N

Come già anticipato, l'applicazione offre un servizio affidabile utilizzando il protocollo Go-Back-N.

Per semplicità, chiamiamo il lato che manda Pacchetti **sender** e il lato che li riceve, rispondendo con gli ACK, **reciver**.

Il protocollo nel sender è progettato per spedire Pacchetti in sequenza finché la finestra (*Window Size*) non è piena oppure finché non terminano i Pacchetti da spedire.
La prima condizione è stata realizzata controllando che il *Numero di sequenza* dell'ultimo Pacchetto inviato sottratto al *Numero di sequenza* dell'ultimo Pacchetto di cui è stato ricevuto l'ACK sia al massimo uguale alla dimensione della finestra. La seconda, invece, è stata ottenuta facendo un controllo tra il *Numero di sequenza* dell'ultimo Pacchetto inviato e il numero di segmenti in cui viene diviso il file da inviare (quantità trovata dividendo la lunghezza del file per la *Chunk Size*).
Nel momento in cui una delle due condizioni non è più soddisfatta, il sender attende le risposte da parte del reciver, ovvero gli ACK relativi ai Pacchetti la cui

ricezione non è ancora stata confermata, per un tempo massimo regolato da un timer.
Appena il sender riceve un ACK azzera il contatore delle ritrasmissioni (utilizzo descritto successivamente), poiché ricevere una risposta, giusta o sbagliata che sia, indica che il reciver è ancora online. Successivamente controlla il *Numero di sequenza* dell'ACK per verificare se è uno di quelli attesi. In caso affermativo reimposta il timer, aggiorna il *Numero di sequenza* dell'ultimo Pacchetto di cui ha ricevuto risposta per liberare la finestra e procede con l'invio dei nuovi come in precedenza. Altrimenti, attende la scadenza del timer per poi procedere con la ritrasmissione di tutti i pacchetti presenti nella finestra.
In caso di ritrasmissione viene incrementato di 1 il contatore delle ritrasmissioni consecutive. Se il numero di ritrasmissioni supera il massimo consentito, il processo termina e si comporta come se il reciver non fosse più raggiungibile.
Ricevere l'ACK di Tipo 4 significa che la trasmissione è andata a buon fine e il processo può quindi terminare.

Essendo Go-Back-N un protocollo con ACK cumulativi, ricevere un ACK vuol dire che il Pacchetto relativo ad esso è stato ricevuto correttamente e, in modo indiretto, comunica che lo sono stati anche tutti quelli precedenti ad esso, quindi anche se dovesse mancare l'ACK di qualche Pacchetto, non è un problema a condizione che arrivino quelli successivi.

Dall'altra parte, il reciver è sempre in attesa del Pacchetto successivo all'ultimo ricevuto della sequenza. Per ogni Pacchetto che riceve risponde con il relativo ACK, a meno che siano fuori dall'ordine. In questo caso invia al sender ACK con *Numero di sequenza* pari all'ultimo Pacchetto ricevuto correttamente. In questo modo comunica qual è l'ultimo Pacchetto ricevuto in sequenza e che è ancora in ascolto, nonostante non stia mandando gli ACK che il sender si aspetta di ricevere.

NOTA : In TCP ricevere 3 volte lo stesso ACK non atteso porta ad una ritrasmissione rapida, ovvero il sender non attende la scadenza del timer per ritrasmettere se riceve 3 ACK uguali, poiché significa che il reciver sta ancora ascoltando, ma non ha ricevuto un Pacchetto in sequenza. Avendo utilizzato UDP, tale funzionalità non è stata implementata, ma nel caso fosse stata richiesta, bastava aggiungere un contatore che andava incrementato ogni volta si ricevesse un ACK con *Numero di sequenza* pari a quello atteso meno 1, e una volta ricevuti i 3 ACK uguali fuori sequenza iniziare la ritrasmissione azzerando il contatore e facendo ripartire il timer.

Calcolo probabilità di perdita

Come detto precedentemente, si presuppone che sia lato client che lato server vengano eseguiti su macchina locale. E' quindi molto difficile che dei messaggi vengano persi. Per simulare questo evento è stata utilizzata una funzione che

prende come parametro un numero decimale compreso tra 0 e 1 che rappresenta la probabilità che un messaggio venga perso. Restituisce 1 se il messaggio non va spedito, ovvero se ne è stata simulata la perdita, 0 se va spedito. Per fare ciò è stata utilizzata la funzione *rand48* che genera un numero decimale pseudo-randomico compreso tra 0 e 1 che viene confrontato con il parametro *Loss Rate* inserito al momento del lancio dell'applicazione; se il numero generato è minore la funzione restituisce 1, altrimenti 0.

Calcolo Timer

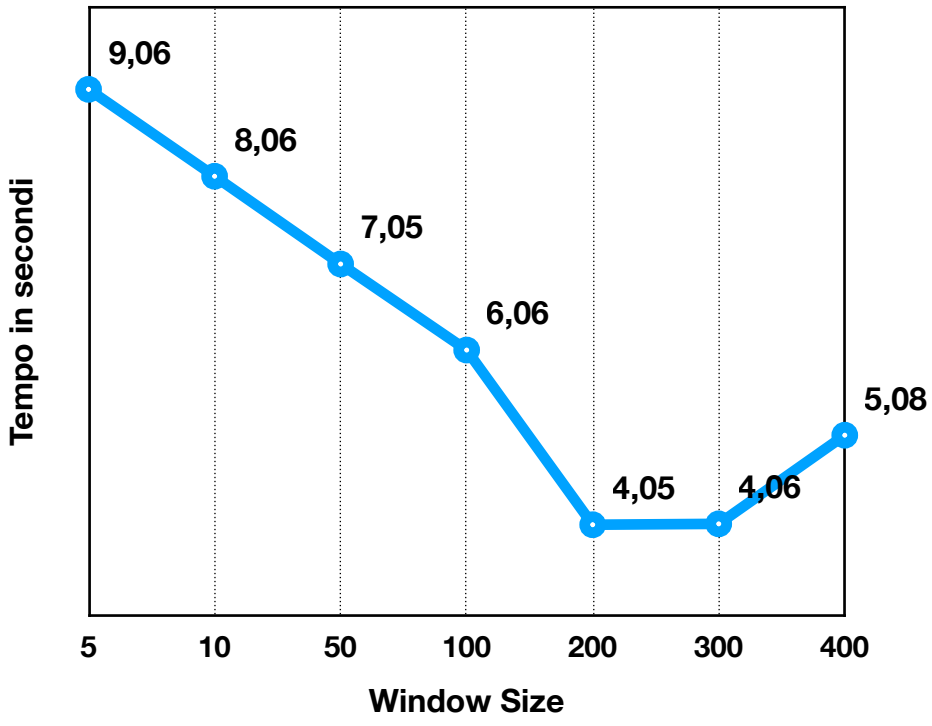
L'applicazione è stata progettata per utilizzare, lato **sender**, un Timer calcolato dal sistema nel caso che l'utente non abbia voluto specificarlo al momento dell'avvio. Per rendere la trasmissione più efficiente possibile viene calcolato un RTT indicativo facendo la differenza tra il momento dell'invio del Pacchetto di richiesta e quello della ricezione del relativo ACK. Il triplo di questo valore viene impostato come Timer. È stato scelto 3 volte il valore trovato per dare margine di ricezione ad ACK successivi a quello atteso nel caso che questo sia stato perso, ovvero per far funzionare gli ACK cumulativi. Non è stato preso un valore maggiore poiché se neanche i successivi sono validi, attendere più tempo porta ad un eccessivo rallentamento della trasmissione. Il Timer relativo al Pacchetto di richiesta è pari ad 1 secondo. Nel caso di ritrasmissione, il Timer viene aumentato ogni volta sommando se stesso tante volte quanto è il numero di ritrasmissione, per poi essere reimpostato alla durata iniziale quando si riceve un ACK in sequenza.

Il **reciver**, se non specificato dall'utente, utilizza un Timer fisso di 3 secondi, poiché indica solamente dopo quanto tempo considerare il sender offline. Se un ACK viene perso non necessita di essere ritrasmesso e quindi neanche del Timer.

Prestazioni

Comando: PUT

File Size	397 KB
Chunk Size	500
Loss Rate	0,01
Timeout	1



Nell'esempio riportato è stato effettuato l'upload di una foto utilizzando un Timeout grande inserito dall'utente, considerato che per fare la stessa operazione con il Timeout calcolato dal sistema, la trasmissione avviene in 0,13 secondi.

Come ci aspettavamo e come si evince dal grafico, le prestazioni migliorano aumentando l'ampiezza della finestra per poi stabilizzarsi e successivamente peggiorare a causa della saturazione del canale.

Esempio di esecuzione

CLIENT

```
pierpaolospaziani@MacBook-Pro-di-Pierpaolo Progetto % gcc client.c -o client
pierpaolospaziani@MacBook-Pro-di-Pierpaolo Progetto % ./client 500 5 0.01

 ---------------------- Welcome! ----------------------

 Available commands:

  -      list       : returns a list of available files

  - get 'file_name' : download a file from the server

  - put 'file_name' : upload a file to the server

 ------------------------------------------------------

 Type one of the 3 commands:

> list

 The file list has been requested ...

 - file.txt
 - empty.txt
 - sfondo.jpg
 - s.jpg


 Type one of the 3 commands:

> get sfondo.jpg

 This file already exist! Do you want to overwrite it? [Y or N]
> y

 Ok, the file will be overwritten!

 Your file has been requested ...

 File downloaded successfully!


 Type one of the 3 commands:

> put foto.jpg

 Trying to upload your file ...

 File uploaded successfully!
```

SERVER

```
pierpaolospaziani@MacBook-Pro-di-Pierpaolo Progetto % gcc server.c -o server
pierpaolospaziani@MacBook-Pro-di-Pierpaolo Progetto % ./server 500 5 0.01

 ---------------- Server started successfully! ---------------

 Operation:   list
 List sent!

 Operation:   get
 Required :   sfondo.jpg
 File sent!

 Operation:   put
 Uploading:   foto.jpg
 File received!
```