Reliable UDP

Matteo Scarcella 0285629 `scarcella@outlook.it`

Gennaio 2024

Indice

1 Architettura del software

Il progetto consiste in un'applicazione client-server multiprocesso di tipo ricorsivo. Il server tiene attiva una socket di benvenuto, che ascolta le richieste da parte dei vari client. Quando un client fa una richiesta di connessione al server, quest'ultimo esegue una fork(), ovvero crea un processo figlio. Tale processo figlio andrà a creare una nuova socket, tramite la quale comunicherà d'ora in poi in modo esclusivo con il client. Il meccanismo si ripete se qualche altro client effettua una richiesta di connessione.
Durante il ciclo vitale dell'applicazione, il server creerà un certo numero di processi figli, dei quali tiene il conto tramite un'opportuna variabile `active_children`. Al termine di ogni processo server (che avviene in seguito ad una interruzione di connessione con il relativo client) la variabile `active_children` viene decrementata. Quando il server intende chiudersi definitivamente, attiverà il flag di chiusura `quit` visibile a tutti i figli correntemente attivi e attenderà la chiusura di ciascuno di essi. Se in qualche processo figlio è in corso un'operazione, esso terminerà soltanto quando tale operazione sarà conclusa. Una volta che tutti i processi figli sono correttamente chiusi, anche il processo padre può terminare.

1.1 Le funzionalità

Quando client e server sono connessi, possono scambiarsi vari tipi di messaggi, i quali permettono di accedere alle varie funzionalità dell'applicazione:

- List - stampa a schermo il contenuto della directory del server
- Get - scarica in locale un file dal server
- Put - carica sul server un file locale

Più avanti nel documento verranno approfonditi i dettagli sulla corretta sintassi e sul funzionamento di tali funzionalità.

2 Il protocollo

Al di sotto dell'invio di ogni messaggio fra client e server è implementato un protocollo TCP-like, chiamato "pseudoTCP". La libreria omonima, presente nella cartella "lib" del progetto, fornisce una serie di metodi e strutture che permettono uno scambio di messaggi con trasferimento affidabile e controllo di congestione.

2.1 La libreria "pseudoTCP.h"

Anzitutto la libreria contiene 3 principali strutture di dati:

- packet
- cg_control
- adaptive_timeout

La struttura packet è l'unità dati principale che viene scambiata fra client e server. Ogni messaggio infatti è un pacchetto e pertanto deve contenere dei campi che possano garantire il corretto scambio di informazioni. I campi scelti sono:

- `num_seq` - 64 bit (numero di sequenza)
- `acknowledgment` - 1 bit
- `num_ack` - 64 bit (numero di acknowledgment)
- `timer` - 64 bit (un timer per le ritrasmissioni)
- `n_retransmission` - 16 (numero di ritrasmissioni)
- `file_size` - 32 bit (dimensione del file integrale in invio)
- `pkt_size` - 12 bit (dimensione corrente del campo dati)
- `command` - 3 bit (comando da inviare)
- `data` - 1000 bytes (dimensione massima 1 kb)

In particolare, il campo command è composto da 3 bit, pertanto il numero totale di comandi è pari ad 8. Essi sono:

- 000 - NULL
- 001 - SYN
- 010 - FIN
- 011 - LIST
- 100 - GET
- 101 - PUT
- 110 - ERR1 (Errori nelle aperture dei file)
- 111 - ERR2 (Errori relativi alle dimensioni dei file)

Quest'ultimo errore viene trasmesso quando si richiede di trasferire un file che eccede la dimensione massima, posta pari a 1 GB per limitare eventi di congestione. Il primo comando è il comando nullo, e viene impostato quando si invia un ACK senza altre informazioni a suo carico. Tutte le dimensioni dei campi sono state scelte in modo tale da contenere la dimensione massima del valore a cui essi si riferiscono. Ad esempio, i 12 bit del campo `pkt_size` sono più che sufficienti a contenere il numero 1000, che è la dimensione massima del campo dati del pacchetto.

Le strutture cg_control e adaptive_timeout sono invece volte a contenere delle variabili utili rispettivamente per il controllo di congestione e per il timeout adattativo. I campi di cg_control sono:

- `cgwin`
- `sstresh`
- `ack_counter`

Mentre i campi di adaptive_timeout sono:

- `timeout`
- `estimatedRTT`
- `devRTT`

Nei paragrafi successivi verrà illustrato il funzionamento di questi due meccanismi più nel dettaglio.

2.2 Il trasferimento affidabile

Il concetto del trasferimento affidabile si basa sul principio secondo cui "tutti i pacchetti arrivano, e arrivano in ordine". I meccanismi fondamentali adottati per poter raggiungere tale obiettivo sono l'acknowledgment e la ritrasmissione. La loro implementazione è ispirata al funzionamento del protocollo TCP.

Nell'applicazione, client e server possono comportarsi sia da ricevente sia da mittente in base alla specifica funzionalità richiesta. Quindi, per schematizzare il funzionamento del trasferimento affidabile si parlerà piuttosto di mittente e ricevente. Inoltre, si assume al solo scopo di illustrare il meccanismo in modo quanto più chiaro possibile, che non ci siano errori nell'apertura di file, nell'invio dei messaggi tramite socket, o altro. Nell'applicazione finale, tali errori sono stati poi correttamente gestiti.

2.2.1 Algoritmo del mittente

Step 0: Prepara i pacchetti da inviare e li salva in una lista d'invio
Fino a che la lista d'invio non è vuota...
Step 1: Ritrasmette i pacchetti inviati e non riscontrati andati in timeout
Step 2: Invia i primi k pacchetti non ancora inviati, se $k < min\{\texttt{cgwin}, \texttt{max\_window}\}$
Step 3: Ascolta ACK dal ricevente. Se non riceve nulla, riparte da 1.
Step 4: Se riceve un ACK duplicato incrementa un contatore. Al terzo ACK duplicato invia una ritrasmissione rapida e riparte da 1.
Step 5: Se riceve un ACK nuovo, riscontra tutti i pacchetti inviati con numero di sequenza inferiore al numero dell'acknowledgment ricevuto e riparte da 1.

2.2.2 Algoritmo del ricevente

Fino a che il file non è stato ricevuto tutto...
Step 1: Ascolta messaggi dal mittente. Se non riceve niente, riparte da 1.
Step 2: Se riceve un pacchetto vecchio invia un ACK duplicato e riparte da 1.
Step 3: Se riceve un pacchetto nuovo ma con numero di sequenza maggiore di quello atteso, lo salva in una lista di ricezione e invia ACK duplicato e riparte da 1.
Step 4: Se riceve un pacchetto nuovo e con numero di sequenza atteso, invia ACK cumulativo riscontrando tutti i pacchetti in lista di ricezione con numeri di sequenza contingui a quello del pacchetto appena ricevuto e riparte da 1.

Questo è lo scenario di base, quindi senza controllo di congestione, senza timeout adattativo, senza gestione degli errori e senza considerare i dettagli delle funzionalità specifiche, che introducono inevitabilmente dei controlli aggiuntivi sul trasferimento, e rappresenta uno scheletro del funzionamento del trasferimento affidabile, basato su acknowledgment cumulativi e ritrasmissioni.

2.3 Il timeout adattativo

Il timeout è un periodo di tempo nel quale si rimane in attesa che il pacchetto inviato venga riscontrato, dunque dev'essere idealmente pari al Round Trip Time. In una rete complessa però non è sempre noto sapere a priori quale sia il RTT, poichè può subire delle fluttuazioni nel tempo. Se si decide di adottare un timeout fisso, i problemi a cui si va in contro sono sostanzialmente 2:

- Se il timeout scelto è troppo grande, si rischia di sotto-utilizzare la banda disponibile
- Se il timeout scelto è troppo piccolo, si rischia di effettuare eccessive ritrasmissioni inutili, aumentando la congestione di rete.

Inoltre, anche se il timeout scelto dovesse per pura fortuna andare bene in un dato istante, non è detto che esso rimanga la scelta migliore negli istanti successivi, poichè il fenomeno dei ritardi di rete è un evento di natura stocastica.

L'applicazione fornisce quindi la possibilità di mantenere un timeout che si adatti alle circostanze della rete, stimando il tempo di andata e ritorno di un pacchetto, cioè il RTT. La stima del timeout viene eseguita come una media mobile esponenziale nel seguente modo, con i valori delle costanti suggeriti dal RFC 6298:

$$\texttt{estimatedRTT} = (1 - \alpha)\texttt{estimatedRTT} + \alpha\ \texttt{sampleRTT}$$
$$\texttt{devRTT} = (1 - \beta)\texttt{devRTT} + \beta\ |\texttt{estimatedRTT} - \texttt{sampleRTT}|$$
$$\texttt{timeout} = \texttt{estimatedRTT} + 4 \cdot \texttt{devRTT}$$

La variabile `sampleRTT` è stata calcolata con particolare attenzione per evitare ambiguità nei due seguenti scenari:

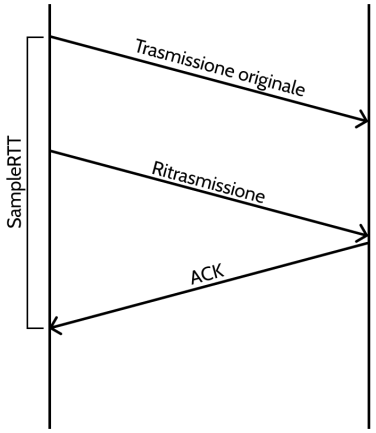

Figure 1: Scenario 1

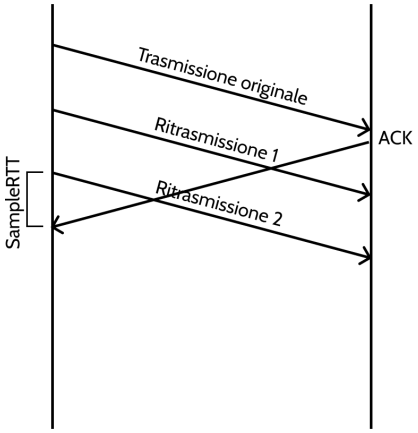

Figure 2: Scenario 2

Nel primo caso, in figura 1, la trasmissione originale non viene correttamente ricevuta dal ricevente, mentre la ritrasmissione arriva e viene riscontrata. Calcolando quindi `sampleRTT` a partire dalla prima trasmissione originale, si ottiene un valore eccessivamente grande, che oltretutto aumenterà all'infinito in caso di perdita costante. Nel secondo caso invece, in figura 2, la trasmissione originale viene ricevuta e riscontrata, ma l'ACK arriva con un ritardo inaspettato al mittente, che è maggiore del valore del timeout adottato. Calcolando in questo caso `sampleRTT` a partire dall'ultima ritrasmissione, si può ottenere un valore eccessivamente piccolo, soprattutto nel caso in cui le ritrasmissioni sono molte e l'ACK si riferisce alla prima.

Dunque in conclusione, non è chiaro se convenga calcolare `sampleRTT` rispetto alla prima trasmissione o rispetto all'ultima, poichè in entrambi i casi ci sono delle ambiguità. Per ottenere un valore di `sampleRTT` quanto più vicino al valore vero del RTT, si è utilizzato quindi un sistema con acknowledgment selettivo nel numero di ritrasmissioni del pacchetto. Il meccanismo funziona nel seguente modo:

- Quando un pacchetto viene ricevuto, viene inviato l'acknowledgment come di consueto, ma viene messo nel campo `n_retransmission` il numero di ritrasmissioni del pacchetto ricevuto. In questo modo l'ACK sarà indicativo su quale delle varie ritrasmissioni del pacchetto andrà a riscontrare

- Quando l'ACK viene ricevuto, si procede a calcolare il valore di `sampleRTT`, che sarà dato dal valore del timer relativo all'ultima ritrasmissione più un contributo aggiuntivo che stima il tempo che il mittente è rimasto in attesa dell'ACK da quando il ricevente ha effettivamente ricevuto il pacchetto, andando a sommare tutti i timeout che hanno generato ritrasmissioni a partire dalla ritrasmissione che è arrivata correttamente al ricevente.

Quest'ultimo valore non può essere calcolato con esattezza, a meno di conservare all'interno di un pacchetto tutti gli istanti di ritrasmissione. Infatti, il timeout può anche cambiare nel corso dell'esecuzione dell'applicazione e, dunque, nel corso dell'invio di un pacchetto. Tuttavia se si accetta un piccolo scarto dato dall'incertezza nella misura, questo meccanismo offre un migliore adattamento alle circostanze di rete sia in caso di perdita sia in caso di ritardi improvvisi e inaspettati. Soprattutto se il numero di ritrasmissioni è molto alto.

In particolare, tale valore è calcolato come la differenza fra il numero di ritrasmissioni del pacchetto riscontrato e il numero di ritrasmissioni dell'ACK ricevuto, moltiplicata per il timeout corrente. In pseudocodice:

```
sampleRTT = get_timer() + (pkt.n_retransmission - ack.n_retransmission)*timeout
```

2.4 Il controllo di congestione

L'applicazione fornisce un semplice meccanismo di controllo di congestione che si basa sull'algoritmo utilizzato da TCP versione Tahoe. Il diagramma a stati finiti che ne rappresenta il funzionamento schematico è il seguente:

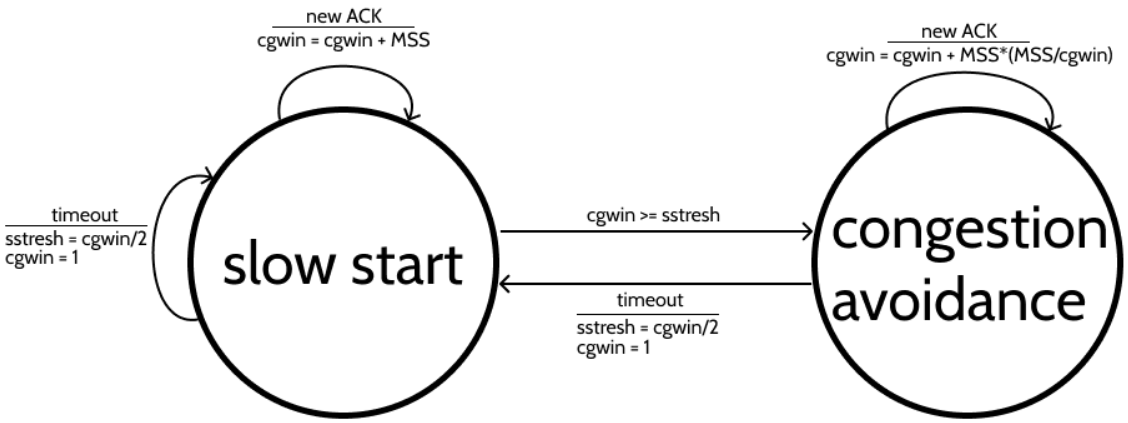

Figure 3: Diagramma a stati finiti del controllo di congestione

L'implementazione specifica prevede delle piccole differenze rispetto all'algoritmo originario. In particolare, la variabile `cgwin` non contiene il numero di bytes disponibili ad essere inviati, ma il numero di pacchetti interi. Di conseguenza, quando si è nella fase di slow start, la finestra di congestione aumenta di 1 pacchetto ogni ACK ricevuto. Nella fase di congestion avoidance invece, nella quale si entra quando il valore di `cgwin` $\geq$ `sstresh`, l'incremento è più cauto e la finestra di congestione aumenta di 1 pacchetto ogni `cgwin` ACK ricevuti. Ad esempio, se la finestra di congestione è pari a 4 significa che occorre ricevere 4 ACK consecutivi senza ritrasmissioni per portarla a 5. Dunque si incrementerà la variabile `ack_counter` fino a che non arriva al valore 4. Da questo momento quindi `ack_counter` viene resettata e occorreranno 5 ACK consecutivi senza ritrasmissioni per portare `cgwin` a 6, e così via.
Un altro dettaglio importante nell'implementazione dell'algoritmo è che le ritrasmissioni rapide non impattano sul valore della finestra di congestione, poichè considerate come eventi abbastanza rari e di minore gravità rispetto alle ritrasmissioni da timeout.

2.5 Handshake e chiusura

L'applicazione è orientata alla connessione. Ciò significa che client e server devono essere connessi l'un l'altro per poter comunicare fra loro, e pertanto è necessario un meccanismo che garantisca la connessione.

L'handshake rappresenta il punto di partenza, ovvero il momento in cui client e server si accordano per stabilire una connessione. È stato quindi adottato l'handshake a 3 vie di TCP, illustrato in figura 4. Il client invia un pacchetto SYN, e attende un ACK particolare, chiamato SYN-ACK, che è un ACK contenente il comando SYN, al quale a sua volta risponderà con ACK. Il server continuerà ad inviare ritrasmissioni del SYN-ACK fintanto che non riceve l'ACK.
La chiusura (in figura 5) è un evento di importanza analoga, se non maggiore, poichè se il client decide di disconnettersi e il server non ne è a conoscenza, c'è il rischio che rimanga un processo attivo in ascolto su una socket che non riceverà mai messaggi, sprecando così risorse utili. Lo stesso vale per il server che se in seguito a un errore si interrompe oppure viene chiuso per manutenzione o altre possibili cause, deve comunicare al client la chiusura, altrimenti quest'ultimo potrebbe rimanere in ascolto perenne senza ricevere mai una risposta. Il client invia il comando FIN al server, e continuerà a ritrasmetterlo fintanto che non riceve l'ACK. Il server, ricevuto il comando FIN, invierà l'ACK e inizierà un periodo di attesa temporizzata in ascolto di possibili ritrasmissioni dal client, per assicurarsi che abbia effettivamente ricevuto l'ACK. Dopo un certo periodo di inattività, deduce che il client si è chiuso, pertanto può terminare. In figura si vede il client inviare il messaggio di chiusura, ma nell'applicazione anche il server può decidere di chiudersi, e il meccanismo è lo stesso, con la differenza che è il server ad inviare per primo il comando FIN.

Oltre alla chiusura è implementato anche un meccanismo di attesa temporizzata durante l'invio di richieste e di errori, che permette di attendere un tempo massimo, posto di default a 15 secondi, durante il quale se non si riceve alcun messaggio si deduce che la connessione si è interrotta, e dunque l'applicazione viene terminata da entrambi i lati. Tale periodo di tempo può essere configurato diversamente, nel caso la rete sia particolarmente lenta e congestionata.

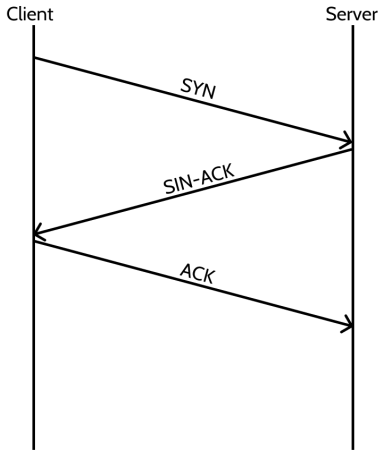

Figure 4: Handshake a 3 vie

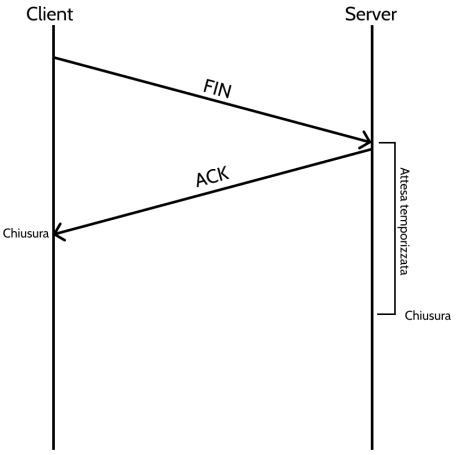

Figure 5: Chiusura

3 Dettagli sull'implementazione

Di seguito, alcuni dettagli implementativi sull'applicazione.

3.1 Modalità d'ascolto

Server e client sono entrambi in ascolto reciproco di possibili comandi. Il modo in cui ascoltano è però differente, poichè il server ascolta soltanto comandi in arrivo dalla socket, mentre il client deve ascoltare sia possibili dati in arrivo sulla socket da parte del server, come per esempio vecchi pacchetti oppure anche errori, sia dati in arrivo su `stdin` immessi dall'utente. Dunque, se lato server è sufficiente implementare una semplice socket in ascolto, ciò non basta per il client. In esso è infatti implementato un multiplexing di IO tramite l'API di sistema `select`, che permette di rimanere in ascolto su più descrittori contemporaneamente, in questo caso appunto la socket e lo `stdin`.

Tutte le socket, sia su server sia su client, sono in ascolto non bloccante, implementato tramite il flag `MSG_DONTWAIT`, per permettere lo svolgimento di altre operazioni durante l'ascolto, ad eccezione della socket principale di ascolto sul client, implementata appunto tramite `select` insieme allo `stdin`.

3.2 `dynamic_list`

Come già accennato precedentemente, l'applicazione invia e riceve pacchetti facendo uso di liste di invio e di ricezione. La struttura `dynamic_list` è un'implementazione di una lista basata su array dinamico di pacchetti, che fornisce una serie di metodi e variabili per l'uso. In particolare, la lista è definita dai campi:

- `int max_size`
- `int len`
- `packet **packets`

Dove `packet **packets` è un array di pacchetti allocato dinamicamente che può aumentare e diminuire la sua dimensione, mantenuta dal campo `len`, entro un certo valore massimo, definito da `max_size`. I metodi forniti sono i seguenti:

- `void create(dynamic_list *list)` - crea una nuova lista
- `void add(dynamic_list *list, packet *p)` - aggiunge un nuovo elemento in coda
- `packet *pop(dynamic_list *list)` - rimuove il primo elemento e lo ritorna
- `void remove_el(dynamic_list *list, int index)` - rimuove l'elemento nella posizione indicata da `index`
- `void sort(dynamic_list *list)` - ordina la lista
- `int contains(dynamic_list list, packet *p)` - ritorna 1 se il pacchetto `p` è contenuto nella lista
- `int is_empty(dynamic_list list)` - ritorna 1 se la lista è vuota

L'ordinamento della lista, effettuato dal metodo `sort` è stato implementato con un algoritmo molto semplice, che esegue ripetute scansioni della lista, calcola il minimo parziale basato sui numeri di sequenza dei pacchetti e lo mette in testa. Tale algoritmo non è certamente fra i migliori in termini di complessità, che è $O(n^2)$. Tuttavia, non è stato ritenuto necessario spingere troppo su questo aspetto poichè la lista a dover essere riordinata è quella di ricezione, che tipicamente è limitata nel numero di pacchetti che può contenere. In particolare, la lista di ricezione può contenere al più un numero di pacchetti pari alla dimensione massima della finestra d'invio del mittente.

3.3 Invio di file pesanti

Così com'è, l'applicazione può già inviare dei file e funziona correttamente. Tuttavia, se si prova ad inviare un file che superi il MB si nota un forte rallentamento o addirittura una chiusura inaspettata. Ciò è dovuto al fatto che quando si tenta di inviare N pacchetti, ciascuno di dimensione L bytes, tutti questi pacchetti vengono inseriti in lista d'invio, cioè in memoria. Questo risulta un grandissimo ostacolo già per file di dimensioni relativamente ridotte nell'ordine di qualche unità di MB.

La soluzione adottata è quella dell'invio a buffer, ed è proprio questo il motivo dell'esistenza del campo `max_size` nella struttura `dynamic_list`. Il meccanismo funziona nel seguente modo:

- Quando il mittente si accinge a preparare i pacchetti per l'invio di un file, non legge l'intero file ma solo una porzione. Tale porzione è esattamente pari alla dimensione massima L del campo dati di ciascun pacchetto moltiplicata per la dimensione massima della lista d'invio K, ovvero il numero massimo di pacchetti che la lista può contenere. Si procede quindi ad inviare i pacchetti attualmente in lista
- Una volta che la lista si è svuotata, se il file non è ancora stato letto tutto, si procede a riempire nuovamente la lista
- Il processo si ripete fino al termine del file.

In questo modo, il caricamento di dati in memoria sarà sempre limitato, e in particolare non può mai superare il valore $K \cdot L$.

3.4 Variabili di sequenza

In ogni istante del ciclo di vita dell'applicazione, client e server mantengono delle variabili di sequenza che permettono il corretto scambio di dati in modo sincrono. Queste variabili sono 3 e sono:

- `num_seq` - il numero di sequenza del prossimo pacchetto da inviare
- `expected_num_seq` - il numero di sequenza del prossimo pacchetto che arriverà
- `current_ack_rcv` - il numero dell'ultimo ACK ricevuto

Le variabili `num_seq` e `expected_num_seq` possono avere un comportamento diverso in base alle circostanze in cui si trovano. Infatti, solitamente esse vengono incrementate rispettivamente nel seguente modo (in pseudo-codice):

```
num_seq = num_seq + length(pkt.data)
expected_num_seq = expected_num_seq + length(pkt.data)
```

Ma in alcuni casi, i pacchetti inviati hanno il campo dati vuoto, ad esempio per i messaggi di errore. Ciò rappresenta un problema, poichè per essere sicuri che il pacchetto ricevuto sia nuovo e non vecchio, le due variabili devono continuamente incrementare. Perciò il problema viene aggirato andando ad incrementarle di 1 in tutti i casi in cui il campo dati risulta vuoto. L'incremento in tal caso diventa rispettivamente:

```
num_seq = num_seq + 1
expected_num_seq = expected_num_seq + 1
```

3.5 Acknowledgment piggybacked

Le tre macro-funzionalità che l'applicazione offre sono la list, la get, e la put, che permettono rispettivamente di stampare a schermo il contenuto della directory del server, di scaricare un file sul client e di caricare un file sul server. Le tre circostanze sono quindi differenti e vedono client e server scambiarsi il ruolo di mittente e di ricevente in base alla funzionalità richiesta.

Tuttavia, è sempre il client che effettua le richieste per primo. In particolare, lo scenario che si va a verificare nel caso della list (o della get, le due funzioni si comportano in modo simile) è quello illustrato in figura 6. Il client deve quindi attendere sia per l'ACK relativo al comando list o get che ha appena inviato, sia per il primo pacchetto di payload che dovrà ricevere (in assenza di errore). Analogamente, il server deve prima inviare un ACK e poi subito dopo inviare un pacchetto di payload. Per evitare l'invio di due pacchetti separati si adotta il meccanismo dell'acknowledgment piggybacked. In sostanza, si inserisce all'interno del campo dati dall'ACK il primo payload disponibile da inviare. In questo modo, il client riscontra il pacchetto inviato e riceve subito il primo pacchetto di dati nello stesso momento, evitando di bloccarsi due volte (figura 7). Per quanto riguarda la funzione put, alla ricezione di tale comando il server può inviare un errore al client, e deve farlo prima che esso cominci ad inviare dati. Pertanto il client dovrà necessariamente prima attendere l'ACK. Perciò l'ACK sarà comunque piggybacked, ma questa volta può trasportare con sè un messaggio d'errore anzichè il payload.

3.6 Paginazione

La funzione list permette di stampare a schermo il contenuto della directory del server, per visualizzare i file. Tuttavia, anche se il nome dei file è limitato a 255 byte su sistemi linux, non c'è un limite su quanti file una cartella possa contenere (a patto che le dimensioni non sforino il limite fisico del disco). Uno scenario del tutto plausibile ad esempio è quello di una directory contenente 5000 file, ciascuno con un nome di 255 bytes,

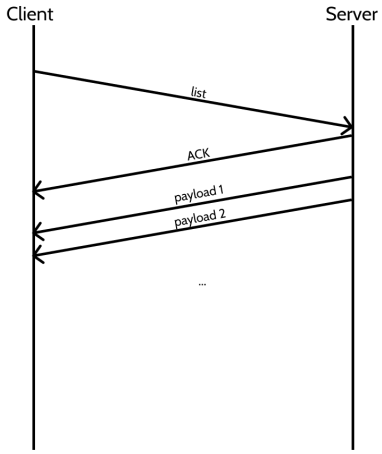

Figure 6: Senza piggybacking

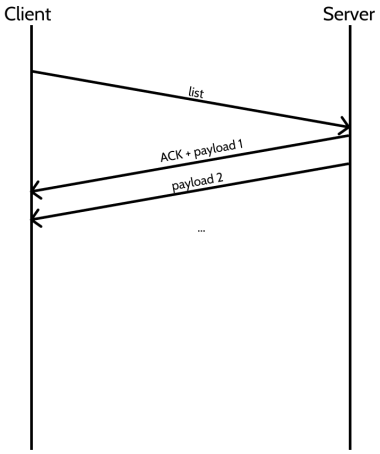

Figure 7: Con piggybacking

tutti contenenti pochi bytes di dati. In questo caso la mole di bytes totali da trasferire da server a client con la funzione list ammonta a circa 1.2 MB. La situazione è quindi del tutto simile a quella del trasferimento di un file molto grande, ma in questo caso non è possibile adottare l'invio a buffer, poichè durante l'esecuzione il contenuto della directory del server può essere modificato dagli altri client attualmente connessi, tramite la funzione put.

Per ovviare a questo problema si è adottato il meccanismo della paginazione. All'apertura dell'applicazione, il server effettua una prima paginazione, cioè salva all'interno di opportuni file dentro la cartella `pages`, denominati con il numero di pagina a cui si riferiscono, il contenuto della directory divisa in più porzioni, se la dimensione eccede il valore $K \cdot L$ (che si ricorda essere il prodotto fra la dimensione massima della lista d'invio e il numero di bytes massimo del campo dati di ciascun pacchetto). Se il client chiama la list e il numero di pagine è pari ad 1, viene inviata la prima ed unica pagina. Se però il numero di pagine è più di 1, viene inviato un errore contenente l'indicazione di specificare un parametro. In questo caso quindi il client dovrà adottare la sintassi "list <numero_pagina>", visualizzando così una pagina per volta tutti i file nella directory del server. Ovviamente vengono implementati gli opportuni meccanismi per la gestione degli errori sull'apertura dei vari file e sul parametro <numero_pagina> inviato dal client. Dopo l'esecuzione di ogni funzione put, il client va ad aggiungere un nuovo file sul server, modificando cosi il contenuto delle pagine, pertanto viene effettuata di nuovo un'ulteriore paginazione. Infine, alla chiusura del server, i file contenenti le pagine vengono eliminati.

3.7 Gestione degli errori

Poichè l'applicazione è orientata allo scambio di file, è sempre possibile che alla chiamata di una delle varie funzionalità dell'applicazione avvengano errori nell'apertura dei file. Tali errori sono gestiti sia stampando un messaggio sullo `stdout` della macchina su cui l'applicazione è in esecuzione, sia inviando un opportuno messaggio d'errore alla macchina con cui si sta comunicando. Il messaggio d'errore è sostanzialmente un pacchetto contente uno dei due comandi:

- 101 - Errore di apertura file o directory
- 111 - Errore dimensione file troppo grande

L'applicazione è stata progettata in modo che solo il server può dover inviare messaggi d'errore al client. Se infatti avvenisse un errore lato client, questi invierà un messaggio al server contenente semplicemente un comando di chiusura, interrompendo la connessione. Dunque, quando il server invia un messaggio d'errore al client, entra in attesa di ricevere l'acknowledgment. Il server deve infatti assicurarsi in modo imprescindibile che il client abbia ricevuto correttamente l'errore. Dall'altro lato, il client non ha alcun interesse ad assicurarsi che il server riceva l'acknowledgment, quindi una volta inviato può terminare, oppure tornare in ascolto di nuove richieste. Infatti, come specificato nel paragrafo 3.1, il client può inviare ACK duplicati anche dopo il termine di una funzionalità, poichè è sempre in ascolto di messaggi da parte del server.

4 Test & Prestazioni

Di seguito verranno illustrati alcuni esempi di test per valutare le prestazioni dell'applicazione in condizioni differenti. Tutti i test sono stati effettuati in ambiente Linux tramite macchina virtuale. Nella directory principale dell'applicazione sono presenti due cartelle, `clientfiles` e `serverfiles`, che contengono rispettivamente i file residenti sul client e sul server. All'interno sono stati inseriti dei file di esempio per testare le funzionalità offerte dall'applicazione.

4.1 Timeout adattativo

Come primo test si effettua una chiamata get su un file di nome "music.mp3" e dalla dimensione di 2 MB. Il timeout è inizialmente fisso con un valore di default pari a 100 ms. Il tasso di perdita è 40%. Il risultato dell'operazione mostra un tempo impiegato pari a 122 secondi, ovvero poco più di 2 minuti (figura 8).
Questo tempo viene drasticamente ridotto se al posto del timeout fisso viene adottato un timeout variabile, con valore di default sempre pari a 100 ms e con tasso di perdita pari a 40%. Si passa infatti da 122 secondi a 29 secondi (figura 9). Si è inoltre graficato l'andamento della variabile timeout (figura 10), mettendo in evidenza il fatto che se il timeout è adattativo è in grado di adattarsi molto bene alla rete sottostante, anche partendo da valori molto alti. In questo caso chiaramente la rete è locale quindi il trasferimento di un pacchetto impiega poco piu di 20 microsecondi per giungere a destinazione, pertanto il timeout tenderà ad avvicinarsi a questo valore.
Chiaramente, il tasso di perdita rimane un fattore di rallentamento anche se il timeout è adattativo. Infatti, con un tasso di perdita ridotto al 20% il tempo d'invio risulta di molto inferiore, arrivando a poco più di un secondo (figura 11).

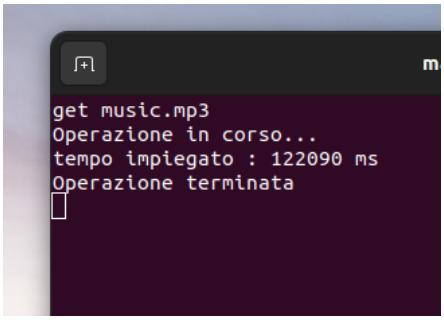

Figure 8: Timeout fisso, perdita 40%

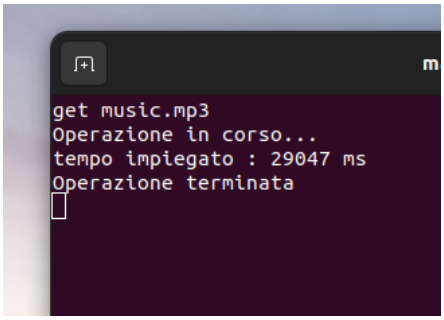

Figure 9: Timeout adattativo, perdita 40%

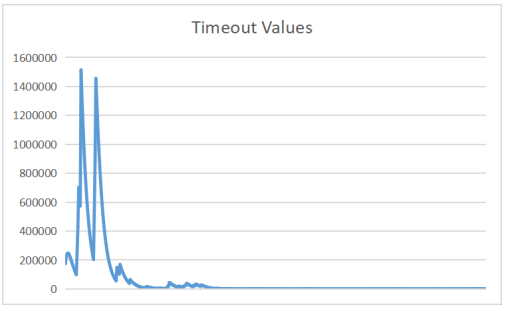

Figure 10: Andamento valore timeout

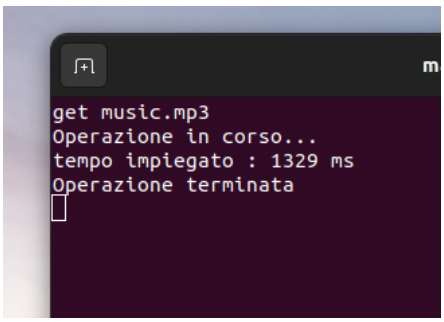

Figure 11: Timeout adattativo, perdita 20%

4.2 Finestra di congestione

Per valutare il funzionamento e le prestazioni del controllo di congestione sono stati fatti 3 test, effettuando il trasferimento del solito file "music.mp3" di 2 MB con tassi di perdita distinti, rispettivamente 40%, 20% e 0%, valore massimo della finestra d'invio pari a 20 e sono stati presi i valori della finestra di congestione in un istante casuale del trasferimento. I risultati sono i seguenti:

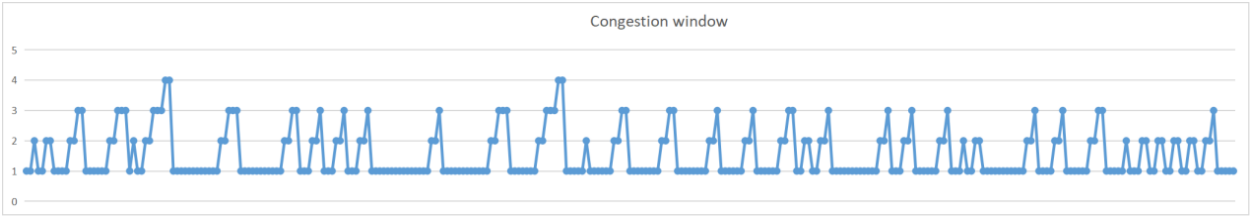

Figure 12: Timeout adattativo, perdita 40%

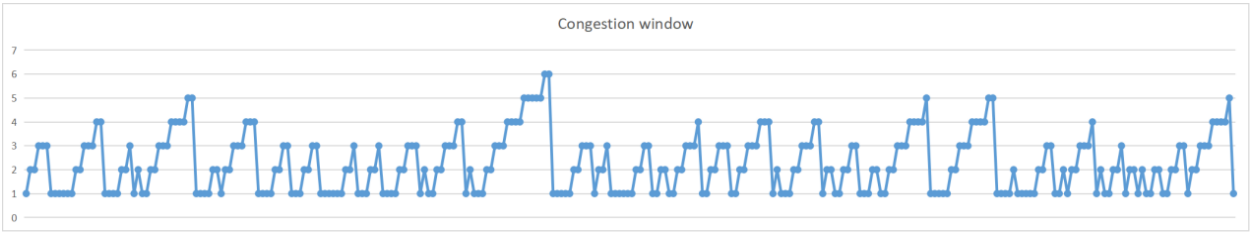

Figure 13: Timeout adattativo, perdita 20%

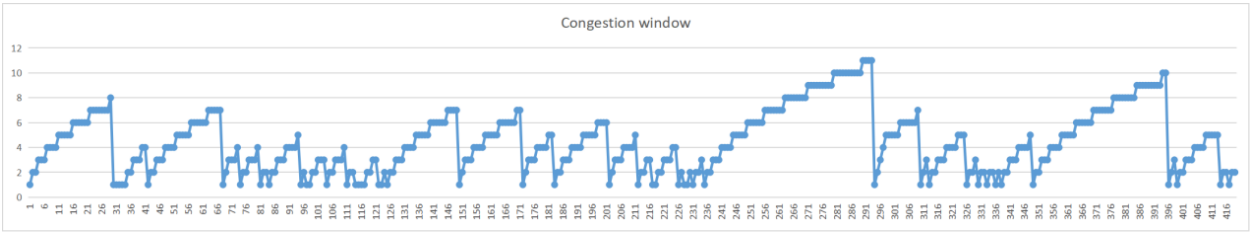

Figure 14: Timeout adattativo, perdita 0%

Com'era atteso, riducendo il tasso di perdita i valori tipici della finestra di congestione risultano mediamente più alti. Notare in particolare che, anche se la dimensione massima della finestra è stata posta al valore 20, tale valore non viene raggiunto nemmeno nello scenario con tasso di perdita allo 0%. Ciò è dovuto al fatto che anche se la probabilità di perdita "artificiale" è nulla, avvengono comunque delle ritrasmissioni per timeout.

5 Installazione e uso

5.1 Installazione

Per installare l'applicazione lanciare il comando `make` nella directory dei file sorgenti. È possibile configurare alcuni parametri aggiuntivi, aggiungendo al comando make i seguenti flag:

- `P_VALUE=<valore>` - Tasso di perdita (compreso fra 0 e 100)
- `TIMEOUT_VALUE=<valore>` - valore del timeout
- `WAIT_VALUE=<valore>` - valore per l'attesa temporizzata
- `AT=1` - Flag per attivare il timeout adattativo (binario)

Se non diversamente specificato, la compilazione avviene con tasso di perdita 0%, valore del timeout a 100 ms, attesa temporizzata di 15 secondi e timeout fisso.

5.2 Avvio

Una volta compilata l'applicazione, è possibile avviarla, con i seguenti comandi:

- `./server` - Per avviare il server
- `./client` - Per avviare il client

5.3 Istruzioni d'uso

La sintassi delle funzionalità disponibili è la seguente:

- `help` - stampa a schermo la lista dei possibili comandi
- `list` - disponibile senza parametro se il numero di pagine è pari ad 1
- `list <numero_pagina>` - parametro obbligatorio se il numero di pagine è maggiore di 1
- `get <nome_file.estensione>` - parametro obbligatorio
- `put <nome_file.estensione>` - parametro obbligatorio
- `quit`

5.4 Chiusura

Per chiudere il server, digitare `CTRL+C`. Per chiudere il client, lanciare il comando `quit`.