UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

INGEGNERIA DI INTERNET E DEL WEB

A.A. 2017/2018

YOUdp - Reliable UDP

Studenti	**Matricola**
Francesco Rosati	0232784
Gabriele Sanelli	0227605
Giuseppe Tomassi	0233229

Indice

1 Presentazione e obiettivo del progetto

YOUdp è un'applicazione di tipo client-server che realizza il trasferimento di file impiegando un servizio di rete senza connessione, utilizzando socket di tipo SOCK_DGRAM, ovvero **UDP** (User Datagram Protocol) come protocollo a livello di trasporto.

L'obiettivo del progetto è quello di rendere il trasferimento dei file affidabile e al fine di garantire ciò è stato implementato a livello applicativo il protocollo **Selective Repeat**, con finestra di spedizione di dimensione N opportunamente scelta per rendere efficiente il trasferimento, il quale gestisce l'eventuale ritrasmissione dei pacchetti persi nella rete.

YOUdp ha un proprio protocollo di comunicazione per far interagire client e server che consiste in scambi di messaggi. In particolare, il client invia dei messaggi di comando per far eseguire un'operazione al server, il quale poi risponderà con un messaggio di risposta al comando con l'esito dell'operazione.

I messaggi di comando principali a disposizione per il client sono i seguenti:

- comando "list", per richiedere al server di inoltrare la lista dei file presenti e disponibili;
- comando "get", per richiedere al server di effettuare il download di un file specificato con il proprio nome;
- comando "put", per richiedere al server di effettuare l'upload di un file specificato con il proprio nome;

Il server alla ricezione dei sopracitati comandi risponde in questo modo:

- comando "list", invio di una lista contenente tutti i file presenti nel server;
- comando "get", invio del file richiesto se presente nel server, altrimenti invio di un messaggio di errore;
- comando "put", invio di un messaggio che segnala l'approvazione a ricevere un file e in seguito un messaggio che comunichi l'esito della trasmissione;

Per migliorare l'esperienza dell'utente con l'applicazione YOUdp è stato definito un ulteriore comando chiamato "stat", il cui compito è quello di raccogliere i messaggi di risposta del server per ogni operazione e mostrare l'avanzamento delle trasmissioni. Questo comando è stato realizzato per evitare sovrapposizioni di stampe sulla console dell'utente e consentire l'immissione di più comandi senza interferenze da parte delle risposte del server.

L'esigenza della creazione del comando "stat" nasce soprattutto poiché YOUdp è un'applicazione che consente sia al client che al server di eseguire più operazioni in modo concorrente. Infatti il client può richiedere di effettuare più download e/o upload contemporaneamente e il server è in grado di gestire più richieste in parallelo.

1.1 Cenni di teoria: UDP e Selective Repeat

UDP è uno dei principali protocolli di trasporto di internet. È un protocollo a pacchetto di tipo connectionless che, a differenza del TCP, non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi, ed è perciò generalmente considerato non affidabile. D'altro canto, è molto rapido ed efficiente per le applicazioni "leggere" o time-sensitive, come la trasmissione di informazioni audio-video real-time.

UDP fornisce soltanto i servizi basilari del livello di trasporto, ovvero la multiplazione delle connessioni, ottenuta attraverso il meccanismo di assegnazione delle porte; la verifica degli errori (integrità dei dati) mediante una checksum, inserita in un campo dell'intestazione (header) del pacchetto, mentre TCP garantisce anche il trasferimento affidabile dei dati, il controllo di flusso e il controllo della congestione.

UDP è un protocollo stateless, ovvero non tiene nota dello stato della connessione dunque ha, rispetto al TCP, meno informazioni da memorizzare: un server dedicato ad una particolare applicazione che scelga UDP come protocollo di trasporto può supportare quindi molti più client attivi.

Selective Repeat è un protocollo applicativo che regola la corretta trasmissione di pacchetti basato sul concetto di finestra scorrevole attraverso cui, sia il mittente che il destinatario memorizzano localmente i pacchetti ricevuti. Utilizzando questo protocollo si può garantire la corretta spedizione e ricezione dei messaggi.

Diversamente dal protocollo Go-Back-N, il Selective Repeat, evita le ritrasmissioni non necessarie facendo ritrasmettere al mittente solo quei pacchetti su cui esistono sospetti di errore (ossia, smarrimento o alterazione). Questa forma di ritrasmissione a richiesta e personalizzata impone al destinatario a mandare acknowledgment specifici per i pacchetti ricevuti in modo corretto. Come in Go-Back-N, il mittente può trasmettere più pacchetti senza dover attendere alcun acknowledgment, ma non può avere più di un dato numero massimo consentito N di pacchetti in attesa di acknowledgment nella finestra. Tuttavia, a differenza di GBN, il mittente potrà ricevere gli ACK di qualche pacchetto nella finestra non necessariamente in ordine.

Pertanto, se definiamo *send_base* come il numero di sequenza del pacchetto presente nella finestra da più tempo che non ha ancora ricevuto un acknowledgment e *nextseqnum* il più piccolo numero di sequenza inutilizzato, ossia il numero di sequenza del prossimo pacchetto da inviare, la visione del mittente della finestra di spedizione sarà la seguente:

- $[0, send\_base - 1]$: pacchetti già trasmessi e che hanno ricevuto tutti acknowledgment.
- $[send\_base, nextseqnum - 1]$: pacchetti inviati, ma che non hanno ricevuto tutti acknowledgment.
- $[nextseqnum, send\_base + N - 1]$: numeri di sequenza utilizzabili per i pacchetti da inviare immediatamente, nel caso arrivassero dati dal livello superiore.
- $\geq send\_base+N$: numeri di sequenza non utilizzabili finché il mittente non riceve l'acknowledgment relativo al pacchetto *send_base*.

Diversamente da Go-Back-N, il destinatario SR non scarta i pacchetti ricevuti fuori sequenza. Infatti, se definiamo *rcv_base* come il numero di sequenza del pacchetto più vecchio ricevuto in ordine, il comportamento del destinatario sarà il seguente:

- $[rcv\_base, rcv\_base + N - 1]$: il pacchetto viene inserito nel buffer, se non era già stato ricevuto e viene inviato un ACK al mittente. Se il numero di sequenza è proprio *rcv_base* allora questo pacchetto e tutti i pacchetti nel buffer aventi numeri consecutivi vengono consegnati al livello superiore.
- $[rcv\_base - N, rcv\_base - 1]$: pacchetto già riscontrato, si invia comunque un ACK al mittente.
- altrimenti il pacchetto viene ignorato.

Si noti come non sempre mittente e destinatario hanno la stessa visuale su cosa sia stato ricevuto correttamente e che quindi nel Selective Repeat le finestre del mittente e del destinatario possono non coincidere. Per una trattazione più ampia del protocollo si rimanda a [1].

2 Architettura del sistema e scelte progettuali

In questo capitolo viene descritto in dettaglio come avviene la comunicazione client-server, il trasferimento vero e proprio dei file e le strutture fondamentali definite in YOUdp.

2.1 Il modello client-server

La struttura dell'applicazione YOUdp è del tipo client-server. Il server si pone in ascolto di richieste su una porta predefinita, la numero 5193, usata dal client per comunicare con esso.
In particolare, è stato adottato un approccio multithread, in cui sia client che server sono composti da un thread principale che invia o gestisce richieste, le quali sono delegate a dei thread secondari, chiamati 'worker'. Questo consente al server di gestire molteplici richieste e al client di inviarne più di una contemporaneamente.

Caratteristiche principali del server:

- Il thread principale, prima di mettersi in ascolto di nuove richieste, crea un numero predefinito di thread secondari, i quali resteranno in vita per tutta la durata del server e in attesa di eseguire nuovi comandi.
- Alla ricezione di una richiesta, il thread principale risveglia uno dei thread 'worker', se ne esiste almeno uno non occupato, altrimenti ne crea uno nuovo (on the fly) per esaudire la richiesta del client.
- Ogni thread secondario ha il compito di creare una nuova socket per la trasmissione di dati, verificare l'esistenza di una nuova richiesta e, se presente, esaudirla.
- Per gestire la richiesta di un client si utilizza la struttura *client_info*, nella quale il thread principale memorizza l'indirizzo del client e il pacchetto ricevuto. Quest'ultimo verrà analizzato e gestito da uno dei thread secondari.

```
struct client_info {
    pthread_t tid;
    struct sockaddr_in addr;
    socklen_t addrlen;
    packet_t pkt;
};
```

- Dopo aver terminato il suo compito, il thread 'on the fly' termina la sua esecuzione e le sue risorse vengono liberate, mentre il thread 'worker' si pone in attesa di processare una nuova eventuale richiesta.

Caratteristiche principali del client:

- Il thread principale crea un numero predefinito di thread 'worker', i quali avranno il compito di eseguire i comandi digitati dall'utente.
- Dopo l'invio di un comando il thread principale ne verifica la sintassi e, se corretta, risveglia uno dei thread secondari per svolgere il compito richiesto. Se tutti i thread 'worker' sono occupati il comando viene ignorato, poiché si è raggiunto il massimo numero di richieste contemporanee.
- Ogni thread 'worker' ha il compito di creare una socket per la trasmissione di dati e, ripetitivamente, verificare l'esistenza di un nuovo comando ed, in caso positivo, eseguirlo.
- I comandi contenenti la richiesta dell'utente vengono inviati alla porta d'ascolto del server, mentre la trasmissione vera e propria avviene su un'altra porta, quella da cui si riceve la riposta al comando dal server.
- Per segnalare lo stato e l'avanzamento di una richiesta al thread principale, ogni thread secondario memorizza informazioni sul progresso della trasmissione a lui affidata nella struttura *thread_element* quali: lo stato del trasferimento, il nome del file da scaricare/caricare, il numero di pacchetti attualmente ricevuti, il numero totale di pacchetti da ricevere, il tempo d'inizio del trasferimento e il tempo attuale.

```
struct thread_element {
    pthread_t tid;
    char status;
    char filename[MAXLINE + 1];
    long cur_pkt;
    long tot_pkt;
    struct timeval start;
    struct timeval now;
};
```

Sarà poi compito del thread principale mostrare all'utente lo stato di tutte le trasmissioni attive.

- Al comando di uscita dalla console il thread principale attende che tutte le trasmissioni in esecuzione, gestite dai thread 'worker', siano concluse e, successivamente, termina l'esecuzione del client.

2.2 I pacchetti

Sia client che server si scambiano dei pacchetti UDP e in questa applicazione sono stati definiti tre nuovi tipi di dato per astrarre, a livello applicativo, la struttura di un datagramma UDP.

- Nella struttura *packet_t*, che definisce il pacchetto vero e proprio, è presente un campo header, struttura nella quale sono memorizzate le informazioni di controllo, e il campo payload, ovvero i dati effettivi che vengono inviati.

```
typedef struct {
    header_t header;
    char payload[MAX_PACKET_SIZE - sizeof(header_t)];
} packet_t;
```

- La struttura *header_t* contiene quindi: un attributo per specificare il tipo del pacchetto; un numero di sequenza, utilizzato per garantire l'identificazione del pacchetto durante la trasmissione; un campo contenente la dimensione effettiva del payload trasmesso nel pacchetto.

```
typedef struct {
    char type;
    long n_seq;
    int length;
} header_t;
```

- La struttura *ack_t* rappresenta invece un pacchetto speciale avente solo il campo header, quindi senza payload, ed è utilizzata per garantire la trasmissione affidabile.

```
typedef struct {
    header_t header;
} ack_t;
```

2.3 La comunicazione

Per instaurare una comunicazione con il server, il client invia dei pacchetti di richiesta. Questi avranno un preciso valore nel campo type dell'header, in base al comando digitato dall'utente.
In particolare si avrà il valore **1**, quando si vuole ottenere la lista dei file disponibili nel server (LIST); **2**, quando il client vuole scaricare un file (GET); **3**, quando si vuole caricare un file (PUT).
L'intestazione del pacchetto di richiesta contiene anche un numero di sequenza di partenza che viene scelto casualmente poiché, dato che le socket utilizzate nella trasmissione vengono riutilizzate nel tempo, è possibile che due host utilizzino una coppia di numeri di porta utilizzati in passato, quindi si vuole ridurre la probabilità che un pacchetto proveniente dalla trasmissione precedente e che sia ancora in viaggio, possa ancora raggiungere il destinatario.

Anche il campo payload del pacchetto differisce in base al comando digitato. Infatti nel caso di una richiesta di download, contiene il nome del file da scaricare; nel caso di una richiesta di upload, contiene il nome del file da caricare con la sua dimensione in byte; mentre nel caso di richiesta della lista dei file, il campo è vuoto.

Alle richieste del client, il server risponde con dei pacchetti di risposta, i quali avranno anch'essi un determinato valore nel campo type dell'header congruente alla richiesta ricevuta.
Il server risponde quindi con il valore **7**, alle richieste della lista dei file presenti (LIST_ACK); **8**, alle richieste di download (GET_ACK); **9**, alle richieste di upload (PUT_ACK).
Il numero di sequenza sarà il successivo di quello presente nel pacchetto ricevuto dal client, mentre il payload di questi pacchetti conterrà nel caso di

- GET_ACK: una stringa formata da "GET" + esito richiesta ("OK" se il file è presente nel server, "ERR" altrimenti) + dimensione del file in byte (-1 se il file non esiste);
- PUT_ACK: una stringa formata da "PUT" + esito richiesta ("OK" se si accetta il file, "FTB" se il file supera la dimensione massima consentita per l'upload);
- LIST_ACK: una stringa formata da "LIST" + dimensione del file list in byte.

A questo punto, per quanto riguarda il comando "put", se il server concede il caricamento, la trasmissione può iniziare immediatamente dato che esso ha ricevuto la dimensione del file da ricevere. Il client quindi inizierà a trasmettere pacchetti DATA (valore **4** del campo type dell'intestazione), i quali contengono i byte del file, non appena riceve la risposta del server.

Per quanto riguarda il comando "get", la situazione è più complessa. Alla ricezione del comando, se il file richiesto è presente nel server, quest'ultimo non può iniziare immediatamente a trasmetterlo, poiché non è certo che al client sia arrivata la sua risposta (esistenza o meno del file e sua dimensione). Perciò, per evitare che i primi pacchetti (DATA) del file vengano persi, il server inizia a trasmetterli solo dopo aver ricevuto un ACK (valore **5** del campo type) con numero di sequenza progressivo al suo pacchetto di risposta, altrimenti la trasmissione del file non ha inizio.

Il comando "list" viene gestito dal server come un caso particolare di "get", dove il file richiesto è una lista contenente nomi di file, creata dal server all'avvio.

Oltre ai tipi di pacchetto già citati, ne vengono utilizzati altri due: FIN (valore **6** del campo type) e FINACK (valore **10**). Questi verranno spiegati in dettaglio in 2.4.4.

2.3.1 Esempi

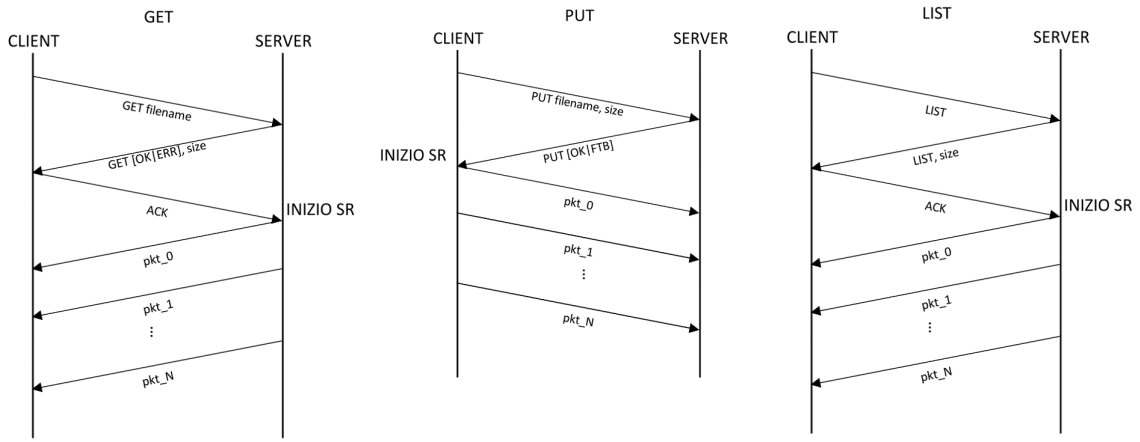


Figura 1: Scambio di pacchetti in condizioni ottimali

Se tutti i pacchetti vengono ricevuti correttamente la situazione è quella rappresentata in Figura 1. Purtroppo però, in una rete reale, non è sempre così. Alcuni pacchetti possono arrivare in ritardo o addirittura essere smarriti. Per ovviare a questa complicazione, il client effettua opportune ritrasmissioni se non riceve una risposta dal server entro un intervallo di tempo stabilito.

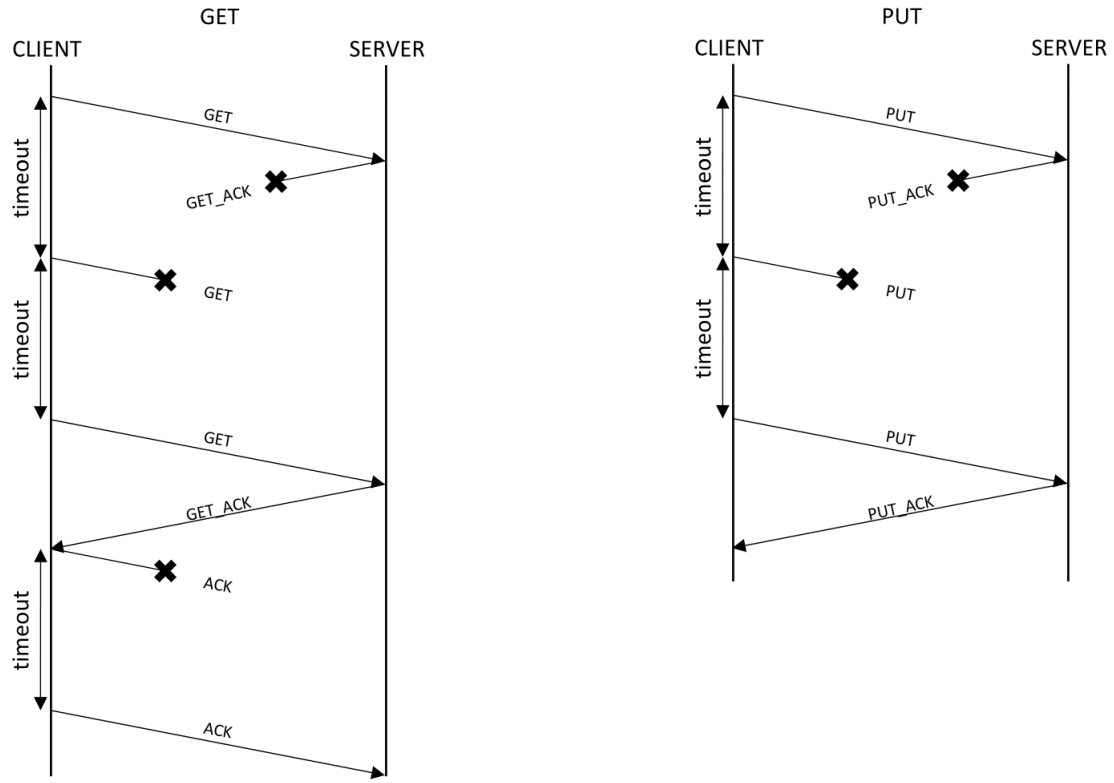


Figura 2: Scambio di pacchetti con perdite

2.4 Il Selective Repeat

Il selective repeat rappresenta il fulcro del progetto. Grazie a questo protocollo infatti, mittente e destinatario, possono scambiarsi pacchetti in modo affidabile. Essendo un protocollo basato sul concetto di finestra scorrevole, sia mittente che destinatario hanno una finestra in cui sono memorizzati i pacchetti da inviare, nel caso del sender, o da ricevere, nel caso del receiver.

2.4.1 Sender

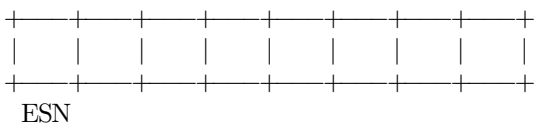
Per gestire i pacchetti che saranno nella finestra di spedizione in YOUdp è stato utilizzato un buffer circolare opportunamente modificato. Pertanto, è stata impiegata la seguente struttura:

```
struct circular_buffer {
    int S;
    int E;
    int N;
    packet_t *window;
    int *acked;
};
```

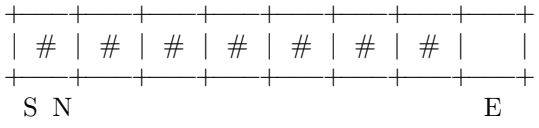
Questa include tre indici numerici, un array contenente i pacchetti da inviare e un altro array per tenere conto dei pacchetti riscontrati.
In generale, per tenere traccia di un buffer circolare, si utilizzano due indici: **S**, il quale indica la posizione del primo pacchetto pronto ad essere inviato; **E**, il quale indica la prima posizione libera in cui salvare nuovi pacchetti. Poiché vi è una possibile ambiguità nel caso gli indici siano nella stessa posizione, ovvero il buffer circolare può essere sia pieno che vuoto, si è aggiunto uno slot in modo tale da evitare sovrapposizioni. La dimensione del buffer circolare quindi sarà di un'unità più grande della dimensione della finestra di spedizione.
Oltre ai classici vi è un ulteriore indice, **N**, il quale indica la posizione del pacchetto più longevo non riscontrato, ovvero quello con il numero di sequenza *send_base*. Questo pacchetto è essenziale in quanto, una volta riscontrato, è possibile muovere la base della finestra verso il pacchetto non riscontrato con il più piccolo numero di sequenza e se ci sono pacchetti non trasmessi con il numero di sequenza che ora ricade all'interno della finestra, questi vengono trasmessi.

Esempio con finestra di dimensione 7:

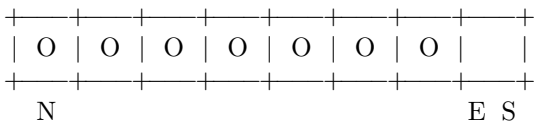
1. Buffer vuoto.



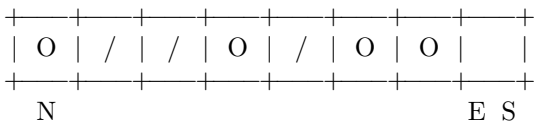
2. Memorizzazione (#) dei pacchetti pronti all'invio.



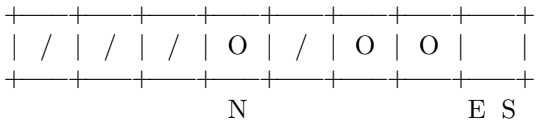
3. Invio (O) dei nuovi pacchetti (#).



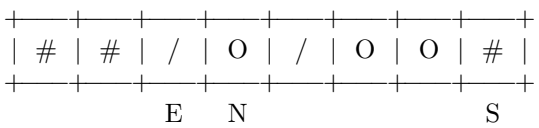
4. Attesa di ACK (/). Non vengono memorizzati (e quindi inviati) nuovi pacchetti finché non si riceve l'ACK relativo al pacchetto in posizione **N** (corrispondente alla base della finestra).



5. Una volta ricevuto tale pacchetto, l'indice **N** si muove del numero di pacchetti consecutivi riscontrati a partire dalla base della finestra.



6. Si memorizzano (sovrascrivendo) tanti nuovi pacchetti quanti sono i pacchetti consecutivi riscontrati a partire da **N**.



7. Si ripete dal punto 3.

Si noti come, al punto 6, non viene sovrascritto il pacchetto in posizione **E**, proprio perché il buffer ha uno slot in più e quindi quella posizione non deve essere utilizzata.

2.4.2 Receiver

Per implementare la finestra di ricezione e il ruolo del receiver l'ostacolo più grande è gestire l'eventualità che i pacchetti possano arrivare in un qualsiasi ordine casuale. Per risolvere questo problema, in YOUdp, si utilizza una linked list ordinata per bufferizzare i pacchetti ricevuti.
Si è quindi impiegata la seguente struttura per memorizzare un nodo della lista:

```
typedef struct node {
    packet_t pkt;
    struct node *next;
} node_t;
```

Detto ciò, il compito del receiver è abbastanza semplice. Quando arriva un pacchetto:

- se il numero di sequenza ricade nell'attuale finestra di ricezione, viene inserito ordinatamente nella lista e inviato l'ACK relativo. Se il pacchetto ha un numero di sequenza uguale alla base della finestra di ricezione, allora il contenuto di questo e di tutti i pacchetti nella lista aventi numeri di sequenza consecutivi viene scritto su file;
- se il numero di sequenza appartiene alla precedente finestra di ricezione, viene inviato comunque l'ACK relativo a quel pacchetto
- altrimenti lo si ignora

2.4.3 Timeout

Per porre rimedio ad un eventuale perdita di pacchetti ed evitare di rimanere in attesa indefinita di acknowledgment, il sender rispedisce ogni pacchetto non riscontrato dopo un certo periodo di tempo, definito dal valore del timeout. Si utilizza, perciò, un contatore 'di finestra'. Ovvero, se il sender non riceve ACK per un tempo maggiore del valore del timeout allora ritrasmette tutti i pacchetti nella finestra non ancora riscontrati.

2.4.4 Chiusura trasmissione

Una volta che il sender ha inviato tutti i pacchetti e il receiver ha spedito l'ACK per ognuno di essi, la trasmissione può terminare. Il sender invia così uno speciale pacchetto di chiusura, denominato FIN, avente numero di sequenza successivo all'ultimo pacchetto inviato. Il receiver, una volta ricevuto tale pacchetto, spedisce al sender un ACK speciale, denominato FINACK, con numero di sequenza progressivo al pacchetto FIN. A questo punto la trasmissione può considerarsi conclusa correttamente.

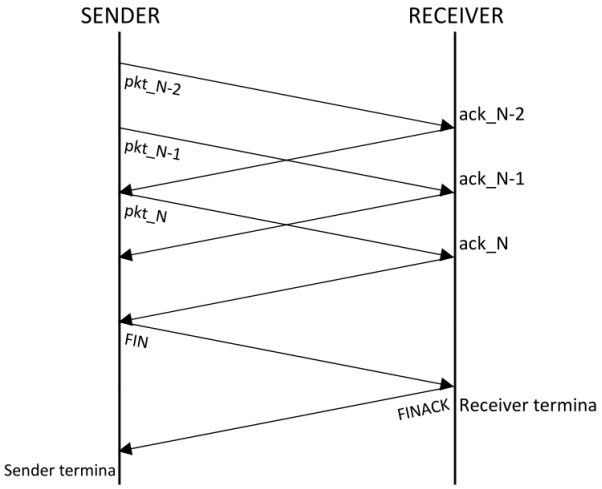


Figura 3: Chiusura della trasmissione

2.5 Scelte progettuali

In questa sezione verranno descritte le principali scelte progettuali effettuate e il motivo che ha portato ad esse.

2.5.1 Numero di thread

Nell'applicazione YOUdp, il numero di thread secondari del client scelto è pari a dieci, mentre nel server è uguale a trenta, proporzionale a quello del client, in modo tale che il server abbia le risorse pronte all'utilizzo e sufficienti a soddisfare tre client a pieno regime.
La scelta di questi valori è maturata a seguito di test empirici e valutazioni prestazionali, mentre la scelta di creare thread subito operativi è stata fatta per avere: un costo di creazione contenuto rispetto alla creazione di thread a tempo di esecuzione; una maggiore efficienza nei tempi di risposta all'istante di arrivo di una richiesta; un parallelismo semplificato che conferisce reattività al sistema.

2.5.2 Limiti di upload

Un'altra scelta effettuata è quella di fissare a 250 MB la dimensione massima di un file che può essere inviato al server, per evitare di impegnare per troppo tempo le risorse del server dato che, notoriamente, la banda di upload risulta essere più limitata rispetto a quella in download.

2.5.3 Numero massimo di ritrasmissioni

Per evitare che il mittente esegua ritrasmissioni dello stesso pacchetto, in modo indefinito, si è scelto di limitare il numero di queste ad un valore predefinito (MAX_RTX), impostato a 15, superato il quale la trasmissione viene abortita. Mentre, per chi riceve, la connessione viene considerata persa dopo un intervallo di tempo t, il quale tiene conto dell'ultima ritrasmissione, secondo la seguente formula:

$$t = (MAX\_RTX \cdot T) + RTT$$

Dove T rappresenta il valore del timeout impostato e RTT indica il tempo richiesto ad un pacchetto per viaggiare da un nodo della rete verso un altro e tornare indietro.

2.5.4 Timeout adattativo

Per gestire l'incremento del timeout, nel caso in cui esso sia adattativo, si è scelto di aumentarlo, in base ai ritardi osservati, secondo la seguente formula:

$$T = T_c \cdot (1 + p_l)$$

Dove T_c è il timeout attuale e p_l è la percentuale di pacchetti persi nella finestra, calcolata nel seguente modo:

$$p_l = \frac{lost\_packet}{window\_size}$$

Si avrà, dunque, un timeout raddoppiato nel caso in cui tutti i pacchetti della finestra vengono persi ($p_l = 1$), costante nel caso in cui non avvengono perdite ($p_l = 0$) o incrementato proporzionalmente in base ai pacchetti persi.
In maniera duale, in caso di decremento, la formula applicata sarà:

$$T = \frac{T_c}{(1 + p_r)}$$

Dove p_r è la percentuale di ack ricevuti per i pacchetti della finestra, calcolata nel seguente modo:

$$p_r = \frac{received\_ack}{window\_size}$$

Per evitare di eseguire innumerevoli variazioni (a volte non necessarie) del timeout, si è scelto di effettuare incrementi, solo se la percentuale di perdita dei pacchetti nella finestra supera una certa soglia (20%); mentre decrementi, solo quando viene ricevuto il pacchetto che identifica la base della finestra e che quindi consente di farla traslare e spedire nuovi pacchetti.
Inoltre si è ristretto l'intervallo dei valori ammissibili del timeout al range $[4, 1000]$ *ms*, per eludere la possibilità che questo aumenti (o diminuisca) eccessivamente.
Anche per il timeout adattativo vale quanto detto in 2.5.3. In questo caso però T corrisponderà al valore massimo dell'intervallo stabilito, ovvero 1 secondo. Questo implica che il destinatario considera la connessione persa dopo non aver ricevuto nulla e atteso un tempo pari circa a 15 secondi.

2.5.5 Rinominazione file con stesso nome

Nel caso in cui venisse scaricato da parte del client un file con un nome già presente nel suo spazio di archiviazione oppure nel caso in cui venisse caricato sul server un file con un nome già presente nella sua directory, si è preferito non sovrascrivere tali file e modificare il nome del file che si sta per scaricare/caricare aggiungendo un numero intero progressivo come prefisso.

3 Implementazione

Per gestire la creazione di nuovi thread, è stata utilizzata la funzione *spawn_threads()* la quale esegue più o meno le stesse istruzioni nel server e nel client.

Lato server, questa funzione, invocata dal thread principale, si occupa di instanziare i thread secondari passandogli come parametro un intero per identificarli come thread worker di default. Questi rimarranno in vita per tutta la durata del server.

```
void spawn_threads(void) {
    int i, res, *is_onthefly;
    pthread_t tid;

    /* thread worker will be alive during server's life */
    *(is_onthefly = allocate_memory(1, sizeof(int))) = 0;
    for (i = 0; i < NUM_THREADS; i++) {
        res = pthread_create(&tid, NULL, thread_job, (void *) is_onthefly);
        abort_on_error(res != 0, "error in pthread_create");
    }
}
```

Figura 4: Implementazione *spawn_threads* nel server

Lato client, invece, il thread principale invoca *spaw_threads()* per creare i thread secondari passando ad ognuno di essi una struttura *thread_element*. Il main thread avrà poi il puntatore alla zona di memoria dove sono memorizzate tutte queste strutture.

```
struct thread_element *spawn_threads(void) {
    int res;
    struct thread_element *te, *p;

    te = (struct thread_element *) allocate_memory(NUM_THREADS,
            sizeof(struct thread_element));
    for (p = te; p < te + NUM_THREADS; p++) {
        res = pthread_create(&(p->tid), NULL, client_job, (void *) p);
        abort_on_error(res != 0, "error in pthread_create");
    }

    return te;
}
```

Figura 5: Implementazione *spawn_threads* nel client

Per gestire la concorrenza ed evitare race condition si utilizza il seguente protocollo, descritto in modo informale in 2.1. Qui di seguito verrà analizzata la parte server, ma per il client il comportamento è analogo.
Per condividere una risorsa tra i vari thread viene utilizzata una variabile globale, la quale conterrà le informazioni ricevute dal client (*client_info*). Occorre però evitare che un thread possa leggere il contenuto di questa variabile mentre un altro la sta modificando. Per garantire quindi la mutua esclusione, ogni thread, prima di accedere a tale risorsa, deve acquisire il relativo mutex. Questo garantisce che durante l'arco di tempo in cui si ha il mutex nessun altro thread potrà accedere alla variabile globale. Si utilizza inoltre una *condition variable*, la quale svolge un compito complementare al mutex. Questa permette infatti ad un thread di informare altri thread che la risorsa condivisa ha cambiato il suo stato e permette a questi di attendere (bloccandosi) tale notifica.

Il thread principale agisce quindi in questo modo:

1. acquisisce il mutex;
2. attende che arrivi una richiesta dal client. Una volta ricevuta la memorizza nella variabile globale;

3. rilascia il mutex;

4. segnala ai thread in attesa su una *condition variable*, se ce ne sono, che la variabile globale è stata modificata. Se tutti i thread sono impegnati ad eseguire altre istruzioni, il thread principale ne crea uno nuovo, il quale dovrà comunque seguire il protocollo stabilito per i thread secondari.

Ogni thread secondario invece:

1. acquisisce il mutex;

2. controlla lo stato della variabile globale. Se non è ancora arriva nessuna richiesta o se quest'ultima è già stata presa in carico da un altro thread, si mette in attesa su una *condition variable*. Una volta arrivato il segnale da parte del thread principale, verifica nuovamente lo stato della variabile globale. Quest'ultimo passo è necessario perché può darsi che un altro thread abbia ricevuto prima il segnale e potrebbe già essersi fatto carico della richiesta;

3. se però questa risulta disponibile allora se ne fa una copia locale e indica la richiesta come già assegnata, in modo tale che nessun altro thread la consideri;

4. rilascia il mutex.

Lo scambio vero e proprio dei pacchetti tramite la rete viene realizzato mediante due funzioni fondamentali: *send_packet* e *receive_packet*.
La prima funzione, scrive sulla socket i byte del pacchetto da inviare. Per simulare la perdita di un pacchetto, evento alquanto improbabile in localhost, si estrae un numero casuale da 1 a 100 (invocando la funzione di libreria *rand()*) e lo si confronta con il valore della probabilità di perdita (PROB). Se il numero estratto ricade nell'intervallo $[0, PROB]$ allora si scarta il pacchetto, altrimenti il pacchetto viene inviato.

```
ssize_t send_packet(int sockfd, const void *buf, size_t len,
        const struct sockaddr *dest_addr, socklen_t addrlen) {
    ssize_t nwritten;
    int rnd;

    /* random number between 1 and 100 */
    rnd = rand() % 100 + 1;
    if (rnd <= PROB) {
        /* simulate packet loss */
        nwritten = 0;
        if (DEBUG) {
            printf("+----------------------+\n");
            printf("|     packet lost!     |\n");
            printf("+----------------------+\n");
        }
    } else {
        /* send packet */
        nwritten = sendto(sockfd, buf, len, 0, dest_addr, addrlen);
        abort_on_error(nwritten < 0, "error in sendto");
        abort_on_error2(nwritten != (ssize_t) len,
                "sendto: packet writing not full!");
    }

    return nwritten;
}
```

Figura 6: Codice della funzione *send_packet*

La funzione *receive_packet* invece, legge dalla socket i byte del pacchetto ricevuto. Qui si gestiscono eventuali errori in ricezione, controllando che i byte ricevuti corrispondano alla dimensione di un pacchetto. Sarà poi compito di chi invoca questa funzione verificare, tramite il campo intestazione, se il numero di byte letti corrispondano ad un ACK o pacchetto dati. Questi dovrà controllare anche, tramite il valore

di ritorno della funzione, se il timeout sia scaduto o meno.

```
ssize_t receive_packet(int socket, void *buf, size_t len,
        struct sockaddr *src_addr, socklen_t *addrlen) {
    ssize_t nread;

    nread = recvfrom(socket, buf, len, 0, src_addr, addrlen);
    /* abort if the error is not timeout expire */
    abort_on_error(nread < 0 && errno != EAGAIN, "error in recvfrom");
    abort_on_error2(nread >= 0 &&
            nread != sizeof(packet_t) && nread != sizeof(ack_t),
            "recvfrom: packet reading not full!");
    return nread;
}
```

Figura 7: Codice della funzione *receive_packet*

Infatti, chi invoca la funzione *receive_packet*, è in attesa di ricevere dei pacchetti ed è possibile che si blocchi in modo indefinito (evento molto probabile se si perdono pacchetti nella rete). Quindi è stato impostato un timeout in ricezione sulla socket tramite la funzione di libreria *setsockopt*, richiamata dalla funzione *set_timeout*. In questo modo si possono gestire le ritrasmissioni e garantire l'affidabilità della comunicazione. La *get_timeout*, in modo speculare, permette di ottenere il valore corrente del timeout impostato sulla socket.

Per implementare il ruolo del mittente e del destinatario nel selective repeat vengono utilizzate le funzioni *sender_job* e *receiver_job*.
La prima gestisce il ruolo del mittente. Il cuore della funzione sta nel ciclo while, in cui viene gestita la finestra di spedizione. Qui vengono letti i byte del file per poi essere impacchettati, memorizzati nel buffer circolare e infine spediti all'host che ha richiesto il file. Dopodiché il mittente si mette in attesa di eventuali riscontri per i pacchetti inviati. Questo compito è assolto dalla funzione *receive_ack*, la quale tiene conto, nel buffer circolare, dei pacchetti riscontrati e non, trasmettendo di nuovo questi ultimi allo scadere del timeout. Questa funzione ritorna solo quando viene riscontrato il pacchetto che identifica la base della finestra, oppure quando si è raggiunto il numero massimo di ritrasmissioni. Nel primo caso, il mittente può inviare nuovi pacchetti e quindi muovere la base della finestra; nel secondo caso, la trasmissione viene abortita.

```
/* SR starts */
send_base = 0;
nextseqnum = 0;
acked = 0;
while (send_base < num_pkt) {
    /* if having pkt to send */
    if (nextseqnum < num_pkt) {
        read_file(fd, buffer, &nextseqnum, num_pkt);
        send_pkt_in_buffer(socket, buffer, dest_addr, addrlen);
    }
    acked = receive_ack(socket, buffer, &send_base, dest_addr, addrlen);
    /* timeout, max retransmission */
    if (!acked) break;
    /* updating progress info */
    if (te != NULL) {
        te->cur_pkt = send_base;
        gettimeofday(&(te->now), NULL);
    }
}
```

Figura 8: Ciclo while della funzione *sender_job*

Se tutto è andato a buon fine, il mittente può inviare il pacchetto finale (FIN) e mettersi in attesa del suo riscontro (FINACK). In questo caso, se il pacchetto conclusivo non viene riscontrato (dopo un massimo numero di ritrasmissioni), la trasmissione viene comunque considerata riuscita. Questo perché, precedentemente, il mittente aveva ricevuto riscontro per ogni pacchetto inviato, si può quindi assumere che il destinatario abbia ricevuto tutti i pacchetti.

```
snd_pkt = (ack_t *) allocate_memory(1, sizeof(ack_t));
(snd_pkt->header).type = FIN;
/* sending FIN */
if (DEBUG) printf("FIN sent\n");
send_packet(socket, snd_pkt, sizeof(ack_t), dest_addr, addrlen);

rcv_pkt = (ack_t *) allocate_memory(1, sizeof(ack_t));
rtx = 0;
for (;;) {
    /* waiting FINACK */
    nread = receive_packet(socket, rcv_pkt, sizeof(ack_t), NULL, NULL);
    if (nread < 0) {
        /* timeout */
        if (rtx == MAX_RTX) {
            if (DEBUG) printf("FIN not acked\n");
            break;
        }
        /* rtx FIN */
        if (DEBUG) printf("timeout: FIN sent again\n");
        send_packet(socket, snd_pkt, sizeof(ack_t),
                dest_addr, addrlen);
        rtx++;
    } else if ((rcv_pkt->header).type == FINACK) {
        if (DEBUG) printf("FINACK received\n");
        break;
    }
}
```

Figura 9: Chiusura congiunta della trasmissione in *sender_job*

La funzione che gestisce il ruolo del destinatario è la *receiver_job*. Qui si utilizza un ciclo indefinito, all'interno del quale il destinatario analizza i pacchetti ricevuti. Se il pacchetto ricade nell'attuale finestra di ricezione, allora viene inserito ordinatamente in una linked list, se non già presente in essa, e viene inviato un riscontro al mittente; se il pacchetto risulta essere della precedente finestra di ricezione (pacchetto fuori sequenza), viene inviato solamente il relativo riscontro; altrimenti il pacchetto ricevuto viene scartato. Il ciclo termina non appena viene ricevuto il pacchetto di chiusura di tipo FIN con il numero di sequenza successivo all'ultimo pacchetto dati ricevuto. A quel punto, il destinatario ne conferma la ricezione e la trasmissione può considerarsi riuscita. Nel caso in cui il destinatario non abbia ricevuto nessun pacchetto per il tempo stabilito dal timeout di connessione (vedi 2.5.3) allora la trasmissione viene abortita.

```
rcv_pkt = (packet_t *) allocate_memory(1, sizeof(packet_t));
/* SR starts */
rcv_base = 0;
for (;;) {
    nread = receive_packet(socket, rcv_pkt, sizeof(packet_t), NULL, NULL);
    if (nread < 0) {
        /* timeout (connection) */
        outcome = 0;
        break;
    } else if ((rcv_pkt->header).type == DATA) {
        /* received data pkt */
        if ((rcv_pkt->header).n_seq >= rcv_base &&
                (rcv_pkt->header).n_seq < rcv_base + WINDOW_SIZE) {
            /* its sequence number falls into the window
               insert pkt into linked list */
            insert(&linked_list, rcv_pkt);

            if (DEBUG) printf("pkt%ld received\t",
                    (rcv_pkt->header).n_seq);
            send_ack(socket, (rcv_pkt->header).n_seq, dest_addr, addrlen);

            if ((rcv_pkt->header).n_seq == rcv_base) {
                /* move window's base of consecutive pkt received from rb
                   and write their payload in the file */
                rcv_base += write_file(fd, &linked_list,
                        (rcv_pkt->header).n_seq);

                /* update progress info */
                if (te != NULL) {
                    te->cur_pkt = rcv_base;
                    gettimeofday(&(te->now), NULL);
                }
            }
        } else if ((rcv_pkt->header).n_seq >= rcv_base - WINDOW_SIZE &&
                (rcv_pkt->header).n_seq < rcv_base) {
            /* its sequence number falls into the previous window */
            if (DEBUG) printf("pkt%ld received out of sequence\t",
                    (rcv_pkt->header).n_seq);
            send_ack(socket, (rcv_pkt->header).n_seq, dest_addr, addrlen);
        } else {
            /*discarding pkt */
            ;
        }
```

Figura 10: Ciclo for della funzione *receiver_job*

Per implementare la comunicazione descritta in 2.3 il server utilizza le funzioni *send_file*, *send_file_list* e *receive_file*. Queste vengono invocate dopo aver ricevuto il pacchetto di richiesta dal client (GET, LIST, PUT).
Le funzioni *send_file* e *send_file_list* svolgono più o meno lo stesso compito, dato che la richiesta della "file_list" è implementata come una richiesta di tipo "get" di un file particolare. L'unica differenza, a parte la sintassi del pacchetto di risposta che verrà inviato, è che la prima controlla l'esistenza del file richiesto, mentre per la seconda questo non è necessario, dato che la "file_list" viene creata dal server all'avvio.
Il server quindi, inizia la trasmissione del file dopo aver ricevuto l'assenso da parte del client.

```c
int send_file(int sockfd, const char *filename,
        const struct sockaddr *dest_addr, socklen_t addrlen, long cur_nseq) {
    int fd, acked, outcome;
    off_t file_len;
    char result[4];
    packet_t *snd_pkt;

    fd = open_file(filename, FILES_DIR, O_RDONLY);
    if (fd == -1) {
        /* file not found */
        file_len = (off_t) -1;
        strcpy(result, "ERR");
    } else {
        file_len = get_file_size(fd);
        strcpy(result, "OK");
    }

    /* build info file packet */
    snd_pkt = (packet_t *) allocate_memory(1, sizeof(packet_t));
    (snd_pkt->header).type = GET_ACK;
    (snd_pkt->header).n_seq = cur_nseq + 1;
    sprintf(snd_pkt->payload, "GET\t%s\t%ld", result, file_len);

    /* send info file pkt to client */
    if (DEBUG) printf("GET_ACK sent, seq: %ld\n", (snd_pkt->header).n_seq);
    //send_packet(sockfd, snd_pkt, sizeof(packet_t), dest_addr, addrlen);
    sendto(sockfd, snd_pkt, sizeof(packet_t), 0, dest_addr, addrlen);
    outcome = -1;
    if (fd > 0) {
        /* waiting ack from client to start trasmission */
        set_timeout(sockfd, &TIMEOUT_CON);
        outcome = acked = wait_ack(sockfd, snd_pkt, GET, cur_nseq + 1,
                dest_addr, addrlen);

        if (acked) {
            /* ack received, trasmission starts */
            set_timeout(sockfd, &TIMEOUT_RTX);
            outcome = sender_job(sockfd, NULL, dest_addr, addrlen,
                    fd, file_len);
        }
    }

    free(snd_pkt);
    close(fd);

    return outcome;
}
```

Figura 11: Codice della funzione *send_file*

La funzione *receive_file* invece, controlla che la dimensione del file da caricare non superi una soglia predefinita (vedi 2.5.2), invia il messaggio di risposta e, se l'upload viene concesso, si mette in ricezione del file. Prima di fare ciò si verifica se esiste già un file con lo stesso nome e in tal caso viene aggiunto un prefisso al nome del file specificato dal client, come già descritto in 2.5.5.

```c
int receive_file(int sockfd, char *filename, off_t file_len,
        const struct sockaddr *dest_addr, socklen_t addrlen, long cur_nseq) {
    int fd, outcome;
    packet_t *snd_pkt;
    char result[4];

    /* check file size */
    if (file_len > MAX_FILE_SIZE) strcpy(result, "FTB");
    else strcpy(result, "OK");

    /* build reply pkt with rqst result */
    snd_pkt = (packet_t *) allocate_memory(1, sizeof(packet_t));
    (snd_pkt->header).type = PUT_ACK;
    (snd_pkt->header).n_seq = cur_nseq + 1;
    sprintf(snd_pkt->payload, "PUT\t%s", result);

    /* send result pkt to client */
    if (DEBUG) printf("PUT_ACK sent, seq: %ld\n", (snd_pkt->header).n_seq);
    //send_packet(sockfd, snd_pkt, sizeof(packet_t), dest_addr, addrlen);
    sendto(sockfd, snd_pkt, sizeof(packet_t), 0, dest_addr, addrlen);

    outcome = -1;
    if (file_len <= MAX_FILE_SIZE) {
        /* check if filename has been already used */
        acquire_mutex(&mtx3);
        fd = check_filename(filename, FILES_DIR, 0777);
        release_mutex(&mtx3);

        /* waiting to receiving file */
        set_timeout(sockfd, &TIMEOUT_CON);
        outcome = receiver_job(sockfd, NULL, dest_addr, addrlen, fd, file_len);

        close(fd);
    }
    free(snd_pkt);

    return outcome;
}
```

Figura 12: Codice della funzione *receive_file*

Il client, dopo aver inviato il comando e aver ricevuta la risposta da parte del server, utilizza le funzioni *get_file* e *put_file*.
Analogamente al server, la richiesta della "file_list" è implementata come una get particolare, dove il nome del file da scaricare è proprio "file_list". Per questo, nella *get_file*, l'unica differenza tra i due comandi sta nel fatto che la "file_list", eventualmente già scaricata, viene sovrascritta, mentre se il nome del file da scaricare è già presente nel client, questo viene modificato aggiungendo un prefisso (vedi 2.5.5). Fatto ciò, il client comunica al server di iniziare a trasmette il file e si pone in ricezione dello stesso.

```c
int get_file(int socket, struct thread_element *te,
        struct sockaddr *dest_addr, socklen_t addrlen, char *filename,
        off_t file_len, long cur_nseq) {
    int fd, outcome;
    ack_t *snd_pkt;
    packet_t *rpl;

    if (strcmp(filename, "file_list.bin") == 0) {
        /* file overwrite */
        fd = create_file(filename, SYSTEM_DIR, 0777);
    } else {
        /* check if filename has been already used */
        acquire_mutex(&mtx3);
        fd = check_filename(filename, DOWNLOAD_DIR, 0777);
        release_mutex(&mtx3);
        strcpy(te->filename, filename);
        te->status = RUN;
    }

    /* client's ready to receive file. Build ACK packet */
    snd_pkt = (ack_t *) allocate_memory(1, sizeof(ack_t));
    (snd_pkt->header).type = ACK;
    (snd_pkt->header).n_seq = cur_nseq + 1;

    /* send ACK to server */
    if (DEBUG) printf("ACK sent, seq: %ld\n", (snd_pkt->header).n_seq);
    send_packet(socket, snd_pkt, sizeof(ack_t), dest_addr, addrlen);

    rpl = wait_reply_packet(socket, snd_pkt, sizeof(ack_t), dest_addr, addrlen,
            NULL, NULL);

    outcome = 0;
    if ((rpl->header).type == DATA) {
        /* DATA pkt received. Waiting for receiving file */
        set_timeout(socket, &TIMEOUT_CON);
        outcome = receiver_job(socket, te, dest_addr, addrlen, fd, file_len);
    }

    free(snd_pkt);
    free(rpl);
    close(fd);

    return outcome;
}
```

Figura 13: Codice della funzione *get_file*

La *put_file*, dopo aver ricevuto la risposta dal server, semplicemente inizia la trasmissione del file.

```c
int put_file(int socket, struct thread_element *te,
        const struct sockaddr *dest_addr, socklen_t addrlen,
        const char *filename, off_t file_len) {
    int fd, outcome;

    fd = open_file(filename, FILES_DIR, O_RDONLY);
    te->status = RUN;
    /* trasmission starts */
    outcome = sender_job(socket, te, dest_addr, addrlen, fd, file_len);
    close(fd);

    return outcome;
}
```

Figura 14: Codice della funzione *put_file*

4 Scenari di funzionamento

Per avviare un client YOUdp occorre digitare su una shell del terminale il nome dell'eseguibile seguito dal flag '-h' e dall'indirizzo IP del server. Indicando solo quest'ultimo si avvia la configurazione di default per la quale la dimensione della finestra è 16, la probabilità di perdita dei pacchetti è nulla e il timeout, adattativo, ha un valore iniziale di 4 millisecondi.
Per cambiare tale configurazione, bisogna indicare un valore associato ai possibili flag:

- '-n' per la dimensione della finestra;
- '-p' per la probabilità di perdita dei pacchetti;
- '-t' per il valore del timeout (fisso per tutta la durata dell'applicazione).

Di seguito vengono illustrati degli esempi di esecuzione dei comandi presenti in YOUdp con probabilità di perdita dell'1% e il resto dei parametri lasciati invariati rispetto alla configurazione di default.

4.1 Comando "get"

```
YOUdp 1.0
Type "help" or "credits" for more information

>>> get GeronimoStilton.pdf
Request scheduled
>>> stat
+-------------------------------------------------------------------------------+
| GeronimoStilton.pdf           34.07%    11 MB/sec     00:00:01 remaining      |
+-------------------------------------------------------------------------------+
>>> stat
+-------------------------------------------------------------------------------+
| GeronimoStilton.pdf           Finished in 02 secs.                            |
+-------------------------------------------------------------------------------+
>>>
```

Figura 15: Esempio di "get" lato client

```
tom@tom ~/Scrivania/YOUdp/SERVER $ ./server -p 1
<#> 127.0.0.1:54299 requested to download 'GeronimoStilton.pdf' file <#>
<!> 127.0.0.1:54299 downloaded 'GeronimoStilton.pdf' file <!>
```

Figura 16: Esempio di "get" lato server

4.2 Comando “put”

```
>>> put Topolino.pdf
Request scheduled
>>> stat
+-------------------------------------------------------------------------------+
| Topolino.pdf                  50.09%    11 MB/sec     00:00:01 remaining      |
+-------------------------------------------------------------------------------+
>>> stat
+-------------------------------------------------------------------------------+
| Topolino.pdf                  Finished in 03 secs.                            |
+-------------------------------------------------------------------------------+
>>>
```

Figura 17: Esempio di “put” lato client

```
tom@tom ~/Scrivania/YOUdp/SERVER $ ./server -p 1
<#> 127.0.0.1:54953 requested to upload 'Topolino.pdf' file <#>
<!> 127.0.0.1:54953 uploaded 'Topolino.pdf' file <!>
```

Figura 18: Esempio di “put” lato server

4.3 Comando “list”

```
>>> list
Request scheduled
+-----------------------------------------------------------+
| GeronimoStilton.pdf                              32.0 MB  |
| tell-me-baby_RHCP_2006_stadium-arcadium.mp3       9.6 MB  |
| unito.pdf                                       303.4 kB  |
| ubuntu-18.04.1-desktop-amd64.iso                  1.8 GB  |
| laziodisu.pdf                                     1.5 MB  |
| divina-commedia.txt                             538.9 kB  |
+-----------------------------------------------------------+
>>>
```

Figura 19: Esempio di “list” lato client

```
tom@tom ~/Scrivania/YOUdp/SERVER $ ./server -p 1
<#> 127.0.0.1:42880 requested file list <#>
<!> 127.0.0.1:42880 downloaded file list <!>
```

Figura 20: Esempio di “list” lato server

5 Testing e analisi delle prestazioni

5.1 Hardware e software

I test, riportati in seguito, sono stati effettuati su Xiaomi Mi Notebook Pro, in ambiente Linux Mint 18.3 Cinnamon 64-bit, processore Intel Core i7-8550U CPU @ 1.80GHz x 4, memoria RAM 16 GB ed m.2 SATA da 250GB.

5.2 Testing

Per testare l'applicazione YOUdp si è focalizzata l'attenzione sui parametri configurabili come la dimensione della finestra di spedizione/ricezione, la probabilità di perdita dei pacchetti e il timeout.
I risultati dei test qui riportati sono stati ottenuti scambiando un file di 32MB tra client e server.

5.2.1 Dimensione della finestra di spedizione/ricezione

Per trovare un valore adatto per la dimensione della finestra di spedizione/ricezione che garantisse prestazioni soddisfacenti è stato scaricato lo stesso file variando la dimensione della finestra con valori crescendi come 2^k, tenendo fissi il valore della probabilità di perdita a 1% e il valore del timeout a 5 millisecondi.

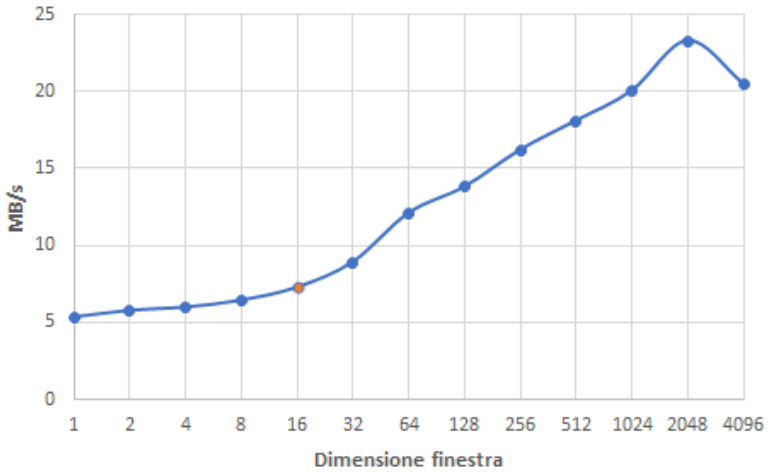


Figura 21: Throughput al variare della dimensione della finestra

Come si evince dal grafico, aumentando la dimensione della finestra si ha un throughput sempre crescente fino a quando non si superi il valore 2048. Avere una finestra di dimensione 2048 è deleterio in quanto, con una probabilità di perdita maggiore, la velocità di trasmissione rallenta in modo consistente e il rischio che tale trasmissione sia abortita è alto.
Effettuando test in localhost si evince che una finestra di spedizione di dimensione 2048 è vantaggiosa rispetto a potenze di 2^k inferiori. Si è notato che effettuando dei test su reti reali con tale finestra, c'era maggiore possibilità che la trasmissione venisse abortita. Si è inoltre notato che al crescere dell'esponente k, il throughput tendenzialmente aumenta del 10-15% e dunque si è deciso di ridurre quest'ultimo a favore di una trasmissione più efficiente. Aumentare la dimensione della finestra potrebbe essere altamente produttivo per reti a banda larga, ma non per le altre reti che ancora sono presenti sul suolo italiano e che sono limitate alla velocità di 10-20 Mbps, quindi si è deciso di fissare a 16 il valore della dimensione della finestra, anche se 32 non sarebbe stata una scelta sbagliata. Con tale finestra di default si ha un valore di throughput accettabile e, in caso di rete congestionata, si riesce sempre a terminare la trasmissione del file.
Dal grafico in Figura 22 si nota che la velocità, con finestra 16 o 32, segue lo stesso andamento al variare della probabilità di errore. È dunque necessario scegliere un valore appropriato della finestra in base alle prestazioni della propria rete; sceglierne uno troppo elevato potrebbe portare alla eccessiva perdita di

pacchetti e dunque alla conseguente interruzione del trasferimento.

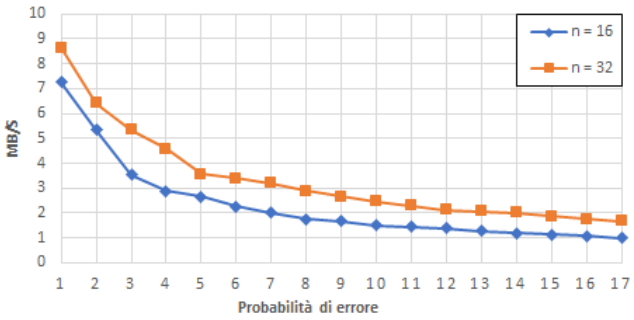


Figura 22: Throughput al variare della probabilità di errore

5.2.2 Comparazione tra timeout adattativo e fisso

L'ultimo test effettuato riguarda le differenze tra l'uso di un timeout adattativo e di un timeout costante, fisso a 5 ms. Si è deciso di impostare il valore di partenza del timeout adattativo a 5 ms con finestra di dimensione 16 e di far variare la probabilità di errore dall'1% al 15%. Si è notato che il throughput è leggermente favorevole al trasferimento con timeout costante, in quanto i picchi di velocità sono più elevati, ma con timeout adattativo c'è meno possibilità di abortire la trasmissione. Con timeout costante, per garantire il successo della trasmissione con una percentuale di errore elevata, occorre aumentare il numero massimo di ritrasmissioni. Si è ovviato per tal motivo all'uso di timeout adattativo, qualora non fosse stato impostato un valore dall'utente.

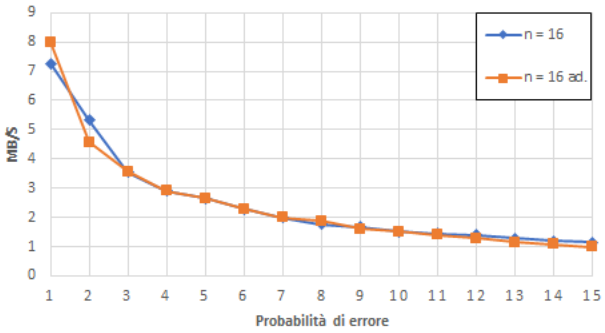


Figura 23: Throughput al variare della probabilità di errore

6 Conclusioni

Lo sviluppo di YOUdp è stato molto formativo e stimolante. Capire come venga effettuato l'invio di pacchetti e come venga regolata la trasmissione affidabile è stato interessante ai fini del percorso accademico e inoltre ha fornito una visione più ampia di come funzionino le trasmissioni a livello globale. Lavorare in gruppo ha reso il lavoro meno duro e molto più produttivo, perché è stato possibile il confronto su molti problemi che potevano essere risolti in maniera differente ed è stato utile per capire alcuni aspetti che non erano ancora del tutto chiari a tutti.

6.1 Limitazioni

Poiché la pratica, a volte, non combacia perfettamente con la teoria, in YOUdp sono state rilassate alcune condizioni.

- *per il pacchetto di risposta al comando (GET_ACK, PUT_ACK, LIST_ACK) non viene simulata la perdita.*
  Questa esigenza nasce perché, al momento della ricezione di una richiesta da parte del client, uno dei thread 'worker' del server verrà risvegliato e incaricato di eseguire il comando. Purtroppo, il client che ha inviato la richiesta, non è a conoscenza da quale socket agisce il thread 'worker' risvegliato. Pertanto, per evitare che un solo client esaurisca (tramite ritrasmissioni del comando) i thread iniziali del server, si assume che il pacchetto di risposta non venga mai perso. Questo pacchetto contiene infatti, oltre alle informazioni richieste dal client, anche la nuova socket su cui avverrà la trasmissione vera e propria.

- *il primo pacchetto dati, nel caso di richiesta di tipo "get", non viene memorizzato (solo la prima trasmissione) dal ricevente.*
  Questa necessità deriva dal fatto che, il client, una volta ricevuto il pacchetto GET_ACK e aver inviato il relativo ACK, non è a conoscenza se questo pacchetto sia arrivato o meno e quindi non può porsi in ricezione degli eventuali pacchetti dati. Il solo modo per rendersene conto è aspettare di riceverne il primo. A questo punto, anche per il ricevente, ha inizio il selective repeat.

- *il valore del timeout non sempre corrisponde a quello desiderato.*
  A causa di un'approssimazione da parte della funzione *setsockopt*, il valore del timeout impostato viene sempre arrotondato al successivo multiplo di 4 *ms*.

6.2 Sviluppi futuri

Realizzare YOUdp è stato un susseguirsi di idee e migliorie ma, per via del tempo incalzante, alcune di queste non sono state implementate. Per questo le citiamo qui.

- **Memory Map**: una modalità alternativa di I/O, che, attraverso il meccanismo della paginazione usato dalla memoria virtuale, permette di mappare il contenuto di un file in una sezione dello spazio di indirizzi del processo che lo ha allocato.
  Questo avrebbe limitato il numero di file descriptor aperti e avrebbe migliorato le prestazioni dell'applicazione in quanto, il memory mapping, permette di caricare in memoria solo le parti del file che sono effettivamente usate ad un dato istante ed effettuare letture e scritture direttamente dalla/nella sezione di memoria mappata.

- **Alberi AVL**: sebbene la ricerca, in una lista ordinata, richieda solo di scorrere puntatori, questa può impiegare nel caso peggiore $O(n)$. Un albero AVL bilanciato garantisce invece, per ogni operazione, un tempo computazionale logaritmico. Ovviamente i vantaggi incrementano con l'aumentare della dimensione della finestra.

- **Download con controllo di versione**: per evitare che un client scarichi nuovamente un file che non è stato modificato dal server, si sarebbe potuto controllare il timestamp dell'ultima modifica di tale file e inviare quest'ultimo al client solo in caso di variazioni dalla sua versione. Realizzando quindi un modello di caching simile a quello utilizzato nei Proxy Server.

Riferimenti bibliografici

[1] J. Kurose & K. Ross, *Reti di Calcolatori e Internet: un approccio Top-Down* (Settima Edizione)

[2] Selective Repeat Applet:
`https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html`

[3] Simone Piccardi, *Guida alla programmazione in Linux (Gapil) - Parte II Programmazione di rete*

[4] Michael Kerrisk, *The Linux Programming Interface*