Università degli Studi di Tor Vergata

facoltà di ingegneria
Corso di Laurea in Ingegneria Informatica
Corso di Ingegneria di Internet e del Web

Relazione al Progetto 'Reliable UDP'

Candidato:
Simone Minasola
Matricola 0132623

Professore:
F. Lo Presti

Anno Accademico 2015-2016

Indice

1 Introduzione

1.1 Contenuto del documento

Questa relazione contiene la descrizione del lavoro svolto per lo sviluppo del progetto finale per il corso di *Ingegneria di Internet e del Web*. Tale documento mostrerà quelli che sono stati i problemi di progettazione riscontrati e i metodi risolutivi adottati, le strutture dati utilizzate e le funzioni che queste svolgono all'interno del programma, così come l'approccio generale e le idee che hanno dato vita al progetto stesso. Queste descrizioni si fermeranno ad un livello di dettaglio distaccato dal codice e dall'implementazione; si è preferito infatti mantenere -in questo documento- una descrizione la più possibile *qualitativa*, lasciando come approfondimento la lettura del codice e i relativi commenti[1].

1.2 Contenuto del CD

Il CD-ROM consegnato insieme a questo documento contiene i codici sorgenti del programma, insieme ad un installer che permette, in automatico, di compilare il codice e creare le directory principali necessarie per il funzionamento del software. Sono presenti inoltre altre cartelle contenenti materiale complementare. Queste ultime verranno elencate in **1.2.1**, mentre la directory contenente i sorgenti (`Reliable UDP`) verrà esaminata in **1.2.2**.

1.2.1 Struttura del CD

```
/
├── LateX source
│   ├── parcolumns.sty
│   ├── xcolor.sty
│   ├── processkv.sty
│   ├── ReliableUDP-frn.pdf
│   ├── ReliableUDP-frn.synctex.gz
│   ├── ReliableUDP-frn.tex
│   ├── ReliableUDP.aux
│   ├── ReliableUDP.log
│   ├── ReliableUDP.synctex.gz
│   ├── ReliableUDP.tex
│   └── ReliableUDP.toc
├── License
│   └── COPYNG.txt
├── Reliable UDP
├── Report
│   └── ReliableUDP.pdf
├── Results
│   ├── Probability of loss.txt
│   └── Window dimension.txt
└── Unused code
```

```
/
└── Unused code
    ├── log_msg.h
    ├── log_msg.c
    ├── log_service.h
    ├── log_service.c
    ├── verbose_service.h
    ├── verbose_service.c
    ├── window.h
    ├── window.c
    ├── window_controller.h
    ├── window_controller.c
    └── p_thraddsem2
```

[1] I commenti nel codice sono stati strutturati nella maniera seguente: ogni *header file* possiede un estratto dove vengono riportate le funzionalità delle strutture dati e funzioni che espone; per una migliore leggibilità del codice, il funzionamento dettagliato di ogni struttura dati e funzione è riportato nello *header file* stesso, mentre nei file *.c* sono commentate quelle funzioni non esposte nei file di intestazione, e sono presenti dei commenti su parti di codice e passaggi importanti.

1.2.2 Cartella `Reliable UDP`

```
Reliable UDP
├── data_example
│   ├── gapil.pdf
│   ├── gettext-0.19.tar.gz
│   ├── IA_relazione_finale.pdf
│   ├── icu4c-56-data.zip
│   ├── icu4c-56-doc.zip
│   ├── install-latex.gz
│   ├── Python-2.7.10.tgz
│   ├── sample_video.mp4
│   └── tor_vergata.jpg
├── help_files
│   ├── help_eng.txt
│   └── help_ita.txt
├── src
│   ├── client.c
│   ├── get.h
│   ├── get.c
│   ├── list.h
│   ├── list.c
│   ├── packet.h
│   ├── packet.c
│   ├── print_messages.h
│   ├── print_messages.c
│   ├── put.h
│   ├── put.c
│   ├── server_status.h
│   ├── server_status.c
│   ├── server.c
│   ├── settings.h
│   ├── strings.h
│   ├── strings.c
│   ├── time_controller.h
│   ├── time_controller.c
│   ├── time_data.h
│   ├── time_data.c
│   ├── timer_wheel.h
│   ├── timer_wheel.c
│   ├── timer.h
│   ├── timer.c
│   ├── utils.h
│   ├── utils.c
│   ├── window.h
│   ├── window.c
│   ├── window_controller.h
│   └── window_controller.c
├── Makefile
├── install.sh
└── README.txt
```

2 Installazione del software

2.1 Compatibilità

Il software è pienamente compatibile con qualsiasi sistema operativo unix-like che supporti le API POSIX. Il programma è stato testato su diversi calcolatori e macchine virtuali con SO Ubuntu 14.04.3 LTS 64 bit. I sistemi operativi della famiglia Mac Os X non sono pienamente supportati, in quanto, in questo ambiente, è stato di recente deprecato l'utilizzo di semafori POSIX anonimi, e la relativa funzione di inizializzazione *sem_init()*. Se si desidera far girare il codice in ambiente Mac Os X, si possono apportare piccole modifiche allo stesso, utilizzando il codice platform indipendent creato dal professore Giuseppe Boccignone (Università di Milano, dipartimento di Scienze Informatiche). Il codice si può trovare nella cartella `Unused code` con il nome *p_thraddsem.c*, e include un piccolo programma di test.

2.2 Procedura di installazione

L'installazione del software è completamente automatica: un piccolo script *bash* compila i sorgenti grazie ad un *MakeFile*, e crea le cartelle necessarie per lanciare il programma. Di seguito sono riportati i passi per l'installazione

1. Copiare la cartella `Reliable UDP` sul proprio computer, in un luogo nel quale si dispone dei privileggi di scrittura e lettura (es. Desktop)

2. Aprire il terminale di sistema, e recarsi (tramite comando *cd*) dentro la cartella appena copiata

3. Lanciare lo script con ***sh install.sh*** ed attendere fino al termine dell' installazione[2]. Qualora si verificasse un problema durante la procedura di installazione, verrà visualizzato a schermo un messaggio di errore

4. Il programma è pronto per essere lanciato

2.3 Alberi delle cartelle generate

L'installer del programma crea due nuove cartelle all'interno di `Reliable UDP`: `server` e `client`. Queste due cartelle contengono rispettivamente il programma lato server e lato client già compilato pronto per essere lanciato, nonché cartelle essenziali per il funzionamento del software. Gli alberi delle directory create in fase di installazione è rappresentato di seguito.

```
Reliable UDP
└── server
    ├── data
    ├── log
    ├── temp
    └── RUDP_server
```

```
Reliable UDP
└── client
    ├── data
    ├── log
    ├── docs
    └── RUDP_client
```

Nelle cartelle `data` verranno inseriti i file scambiati tra server e client. Lato server, la cartella è già riempita con alcuni sample files (presenti in `data_example`). Nelle cartelle `log`, invece, verranno creati e aggiornati i file di log. Nella cartella `temp` (lato server), verrà prima creato

[2]In ambiente Linux, il testo visualizzato sul terminale durante l'installazione è colorato. La colorazione del testo non è supportata dai sistemi operativi Mac Os X. Se quindi si esegue l'installazione su questo ambiente, verranno visualizzati strani caratteri a schermo.

e poi eliminato un file di testo contenente una lista di file presenti in `data`. Questa operazione viene eseguita dal server al ricevimento del comando *LIST*. Tale accorgimento si è reso necessario per evitare che la lista di file includa il file appena generato. Nella cartella `docs` (lato client), in ultimo, sono presenti i file di aiuto in diverse lingue, caricati e mostrati a schermo al ricevimento del comando *HELP*. `RUDP_server` e `RUDP_client` sono i file eseguibili compilati durante la procedura di installazione.

2.4 Configurazione

Prima di eseguire l'installazione del programma, è consigliabile configurare le impostazioni. Se si omette questo passaggio, il software verrà compilato ed eseguito con le impostazioni di default. Per cambiare le impostazioni, aprire il file *settings.h* nella cartella `src`. Il file di intestazione contiene diverse definizioni modificabili dall'utente. Le più importanti sono:

- **WINDOW_DIMENSION**: dimensione della finestra di spedizione
- **LOSS_PROBABILITY**: percentuale che identifica la probabilità di perdita
- **MAX_BLOCK_SIZE**: dimensione in byte del campo *data* dei pacchetti

La modifica di un qualsiasi valore potrebbe creare malfunzionamenti non previsti durante l'esecuzione. Si consiglia di leggere attentamente i commenti e le avvertenze di ogni definizione della quale si desidera modificare il valore di default. I valori di default sono salvati come commento nella parte iniziale del file.
Una volta modificate le impostazioni, eseguire la procedura di installazione come riportato nel paragrafo **2.2**.

2.5 Avvio del programma

2.5.1 RUDP_server

Nella cartella `server`, lanciare il programma con le seguenti diverse opzioni (l'ordine degli argomenti passati non è importante):

- ***./RUDP_server***
  Avvio del programma base senza nessuna funzionalità aggiuntiva.
- ***./RUDP_server -l***
  Avvio del programma con creazione e gestione del file di log. Per il server, il file di log è unico e verrà creato una sola volta. Ogni volta che si avvierà il server con l'argomento ***-l***, il file di log verrà aggiornato con le operazioni correnti. Per consultare il log, aprire il file *LOG.txt* nella cartella `log`.
- ***./RUDP_server -v***
  Avvio del programma in modalità *verbose*. Verranno visualizzati a schermo una serie di messaggi dettagliati riguardo le operazioni in corso nel server. Utile per debug.
- ***./RUDP_server -l -v***
  Avvio del programma in modalità *verbose* e con gestione del file di log.

Per uscire dal programma, terminare il processo con **CTRL+C**.

2.5.2 RUDP_client

Nella cartella `client`, lanciare il programma con le seguenti diverse opzioni (al di fuori del primo, l'ordine degli argomenti passati non è importante):

- ***./RUDP_client 127.0.0.1***
  Avvio del programma base senza nessuna funzionalità aggiuntiva.
- ***./RUDP_client 127.0.0.1 -l***
  Avvio del programma con creazione e gestione del file di log. Per il client, viene creato un nuovo file di log ogni giorno. Ogni volta che si avvierà il client con l'argomento ***-l***, il file di log verrà creato/aggiornato con le operazioni correnti. Per consultare il log, aprire il file *LOG [data odierna].txt* nella cartella `log`.
- ***./RUDP_client 127.0.0.1 -v***
  Avvio del programma in modalità *verbose*. Verranno visualizzati a schermo una serie di messaggi dettagliati riguardo le operazioni in corso nel client. Utile per debug.
- ***./RUDP_client 127.0.0.1 -l -v***
  Avvio del programma in modalità *verbose* e con gestione del file di log.

Il primo argomento rappresenta l'indirizzo IP del server. Essendo stato progettato per lavorare in locale, l'indirizzo IP da passare come argomento è quello di *localhost*.

Per terminare il programma con uno shutdown sicuro, premere **CTRL+D** (EOF stdin).

3 Descrizione del progetto

3.1 Traccia

Lo scopo del progetto è quello di progettare ed implementare in linguaggio C usando l'API del socket di Berkeley un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (socket tipo SOCK_DGRAM, ovvero UDP come protocollo di strato di trasporto). Il software deve permettere:

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

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

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

Funzionalità del server

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

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

Funzionalità del client

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

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

Trasmissione affidabile

Lo scambio di messaggi avviene usando un servizio di comunicazione non affidabile. Al fine di garantire la corretta spedizione/ricezione dei messaggi e dei file, sia client che server implementano a livello applicativo il protocollo *selective repeat* con finestra di spedizione N.
Per simulare la perdita dei messaggi in rete (evento alquanto improbabile in una rete locale per non parlare di quando client e server sono eseguiti sullo stesso host), si assume che ogni messaggio sia scartato dal mittente con probabilità p.
La dimensione della finestra di spedizione N, la probabilità di perdita dei messaggi p, e la durata del timeout T, sono tre costanti configurabili ed uguali per tutti i processi.
Il client ed il server devono essere eseguiti nello spazio utente senza richiedere privilegi di root.
Il server deve essere in ascolto su una porta di default ($configurabile$)

Opzionale

Ciascun processo calcola dinamicamente il valore del timeout T di ritrasmissione in base alla dinamica dei tempi di ritardo di rete.

3.2 Approccio

Per la realizzazione del software non è stato adottato nessun processo di sviluppo vero e proprio. Sono state però seguite delle fasi importanti: in principio sono stati definiti i requisiti richiesti in base al testo dato; è seguita poi una fase di ideazione, durante la quale si è definita l'architettura e delineato i limiti e le funzionalità che il software avrebbe duvuto avere; infine, con la fase di costruzione, sono state codificate le idee derivate precedentemente. Per tutta la fase di costruzione, si è scelto di adottare un **approccio** generale basato su tre principi fondamentali: creare una **architettura di base affidabile**, utilizzare **idee semplici ed efficienti**, e **massimizzare il riuso del codice**.

3.2.1 Scelta dell'architettura

Durante la fase di ideazione, è stato ritenuto opportuno spendere del tempo per selezionare un pattern architetturale che fosse adatto per la realizzazione del progetto. In particolare, l'architettura avrebbe dovuto permettere di scomporre le varie complessità progettuali in piccoli sottoproblemi, in modo tale da poter testare ogni soluzione singolarmente. In questo modo sarebbe stato più facile portare avanti l'intero progetto, in quanto la sua realizzazione non avrebbe potuto disporre delle canoniche 8 ore lavorative giornaliere (causa esami). La scelta è ricaduta sulla categoria di pattern *From Mud to Structure*, in particolare sul pattern **N-layer**. L'uso di questo pattern architetturale ha aiutato a suddividere le varie responsabilità dei componenti software tra vari livelli di astrazione proprio come si desiderava. E' tuttavia importantissimo far notare che i benefici sono stati molti di più di quelli che ci si aspettava. Infatti, è stato possibile sostituire 'parti' del progetto (situate su un determinato livello) con altre, senza dover mettere mano al codice di altre 'parti' su altri livelli di astrazione. L'impatto causato dai cambiamenti è stato ridotto al minimo: questo ha permesso di effettuare diversi test che prevedevano l'uso di algoritmi diversi, al fine di sceglierne il più efficiente (o quello, tra i tanti, funzionante). La descrizione dell'architettura software implementata nel progetto è rimandata al paragrafo **4.1**.

3.2.2 Idee di base

Il progetto è stato affrontato con l'obbiettivo di creare un protocollo di trasmissione affidabile, e non replicare nel dettaglio le scelte implementative del protocollo TCP. Questa idea è rimasta immutata, dall'inizio alla fine dello sviluppo. Sono state comunque utilizzate le idee alla base della trasmissione di dati affidabile: **acknowledgement** e **timeout**. Seppur condividendo questi concetti di base, **TCP** e **RUDP** (**R**eliable **UDP**) ne fanno uso in maniera differente. L'uso di idee già utilizzate e consolidate ha reso solidi i principi con cui si è realizzato questo progetto.

3.2.3 Riuso del codice

Durante lo sviluppo, si è cercato di massimizzare il riuso del codice creando funzioni e strutture dati utilizzabili sia lato client che lato server. L'idea di base era quella di creare un codice il più possibile modulare, ideando e progettando strutture e funzionalità molto generali, facilmente integrabili dal server e dal client. Per esempio, si può notare come una operazione di **GET** richiesta dal client al server, può essere vista come una operazione di **PUT** dal server verso il client. Si è pensato, dunque, di generalizzare il più possibile, scrivendo una volta sola il codice per **PUT** e per **GET**, e concatenarli in maniera efficiente secondo le richieste[3]. Questa strategia è stata perseguita durante tutto lo sviluppo del progetto. Il risultato ottenuto è stato molto più che soddisfacente: client e server condividono più del 90% del codice scritto, mentre le caratteristiche 'particolari' di entrambi risiedono nei file principali (*client.c* e *server.c*).

[3]La descrizione delle operazioni di **GET** e **PUT** è rimandata al capitolo **4**, mentre per maggiori dettagli (vista la complessità delle funzioni *send_file()* e *receive_file()*) è consigliabile la lettura dei commenti e del codice nei file `put.c`, `put.h`, `get.c`, `get.h`.

4 Implementazione

In questo capitolo verranno discusse le scelte implementative e le strategie adottate durante lo sviluppo del progetto. Non verrà fatta una vera e propria analisi del codice (per questo, si consiglia la lettura dei commenti e del codice stesso), bensì verranno 'esplose' tutte le caratteristiche del programma, spiegati i principi e i meccanismi alla base di ogni funzionalità ed esposti i metodi risolutivi ai problemi sorti durante la codifica.

4.1 Architettura N-layer

Come già accennato in **3.2.1**, il progetto si basa sul pattern architetturale **N-layer**, che prevede la scomposizione delle complessità in problemi più piccoli e semplici e, quindi, l'organizzazione a strati delle varie parti che compongono l'intero progetto. In particolare, questa architettura prevede 4 strati, ed è stata adottata per la parte di gestione dei pacchetti e dei timeout.

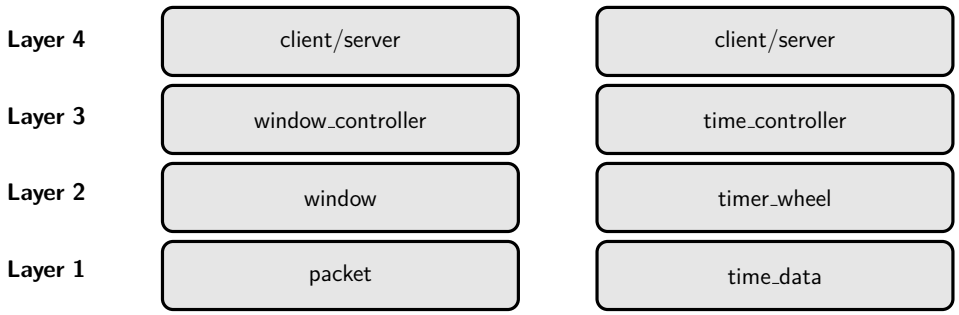

Figura 1: Schema del pattern architetturale a 4 layer adottato mel progetto. Nella colonna di sinistra è rappresentata la gerarchia di layer per la gestione dei pacchetti, mentre a destra è rappresentata la gerarchia di layer per la gestione dei timeout.

Come si può notare dalla *Figura 1*, lo strato più basso dell'architettura è rappresentato dalle due strutture dati che sono alla base dell'intero progetto: **packet** e **time_data**. Queste rappresentano rispettivamente i pacchetti (datagrammi) inviati in rete tra server e client, e la loro componente 'temporale', nella quale vengono memorizzati timeout, tempo di spedizione e tempo di arrivo (capitolo **4.2**). Il layer 2 è composto dalle strutture dati **timer_wheel** e **window**, che possono essere definite come dei 'contenitori' delle strutture basilari del layer 1. La prima realizza la *sliding window* del protocllo *selective repeat*, mentre la seconda realizza una struttura per l'inserimento dei timeout di ogni pacchetto (capitoli **4.3.1** e **4.4.1**). Il terzo livello dell'architettura è occupato da quelle strutture dati che hanno il compito di gestire e controllare le operazioni che fanno uso dello strato sottostante, grazie ad opportuni meccanismi di sincronizzazione e controllo degli eventi: **window_controller** e **time_controller**. La prima viene utilizzata per gestire l'inserimento, l'invio e la cancellazione dei pacchetti nella finestra di spedizione, mentre la seconda viene utilizzata per controllare iterativamente la scadenza dei timeout, per determinare se un pacchetto necessita di un nuovo invio o se il timer deve essere eliminato (capitoli **4.3.2** e **4.4.2**). Il livello più alto è composto dai programmi **client** e **server** veri e propri.
Ogni livello N dell'architettura, offre delle funzioni che permettono al livello N+1 di usare in modo sicuro le strutture dati presenti nel layer N. In questo modo, il client ed il server interagiranno direttamente con le funzioni e le strutture esposte dal layer 3, le quali useranno strutture e funzioni esposte dal layer 2, e così via.

PKT_GET	Usato per inoltrare una richiesta di **GET** al server. In questo caso, il campo **data** del pacchetto contiene il nome del file da inviare.
PKT_PUT	Usato per inoltrare una richiesta di **PUT** al server. In questo caso, il campo **data** del pacchetto contiene il nome del file da inviare, mentre il campo **dimension** contiene il numero totale di pacchetti da spedire per completare l'operazione.
PKT_LS	Usato per inoltrare una richiesta di **LIST** al server. In questo caso, il campo **data** del pacchetto è vuoto.
PKT_HELP	Usato localmente dal client per richiedere la letture del file di aiuto per l'utilizzo del programma.
PKT_LANG	Usato localmente dal client per richiedere un cambio di lingua.
PKT_INFO	Non usato in questa release. Pensato per lo scambio di informazioni generali tra client e server.
PKT_ACK	Identifica un acknowledgment per un pacchetto ricevuto. Se l'ack è stato inviato a seguito di un pacchetto PKT_PUT, il campo **dimension** dell'ACK conterrà la porta disponibile selezionata dal server per il trasferimento del file. Se l'ACK è stato spedito in risposta ad un pacchetto PKT_GET o PKT_LS, allora oltre alla porta verrà inserito il numero di pacchetti da ricevere nel campo **data** dell'ACK. Se l'ACK viene inviato in risposta ad un pacchetto PKT_DATA, allora il campo **seq** dell'ack conterrà il numero di sequenza del pacchetto ricevuto correttamente.
PKT_DATA	Usato per identificare i pacchetti che contengono parti del file da inviare/ricevere. Le parti del file sono inserite nel campo **data** per un massimo di `MAX_BLOCK_DIMENSION` bytes.
PKT_FIN	Individua l'ultimo pacchetto che il server o il client spedisce al termine dell'invio del file.
PKT_FINACK	Individua un ACK speciale per rispondere alla ricezione di un pacchetto PKT_FIN.
PKT_ERR	Usato per informare il client di un errore nel server. Il campo **data** contiene le informazioni riguardo l'errore avvenuto.
PKT_REQ	Usato per richiedere forzatamente l'invio di un determinato pacchetto alla controparte.

Figura 2: Tabella delle varie tipologie di pacchetto utilizzate nel progetto. I primi tre sono molto importanti, e vengono utilizzati per dare inizio ad una determinata operazione. Per ulteriori dettagli si consiglia la lettura del codice e dei commenti della funzione ***sendCMD()*** *nel file* `client.c`.

4.2 Le *fondamenta* del programma

4.2.1 Struttura *packet*

La struttura dati **packet** rappresenta l'unità fondamentale di informazione scambiata tra client e server. I campi di un pacchetto sono mostrati in *figura 3* e verranno analizzati uno ad uno in questo paragrafo.

`data`: contiene i byte del file che si sta ricevento o spedendo. La dimensione massima di questo campo è identificata da `MAX_BLOCK_SIZE`.

`td`: puntatore alla struttura **time_data** che contiene i campi fondamentali per il calcolo del timeout del pacchetto. I dettagli di questa struttura sono rimandati al capitolo **4.2.2**.

`type`: identifica la *tipologia* di pacchetto. Ci sono diversi tipi di pacchetti che il programma può gestire, e ogni tipo ha caratteristiche e significati diversi. In *Figura 2* sono descritti tutte i tipi di pacchetti e il loro utilizzo.

`seq`: è il numero di sequenza del pacchetto. Per ogni pacchetto in invio, esiste una **time_data** legata al pacchetto dallo stesso numero di sequenza (come si può vedere dalla *Figura 3*).

`acked`: rappresenta un marcatore che identifica se, per il pacchetto, è stato ricevuto un ACK o ancora no.

`dimension`: identifica la dimensione reale in byte del campo **data** del pacchetto. Nell'operazione di scrittura o lettura dei dati presenti in un pacchetto, si fa sempre riferimento a questo campo.

`retries`: identifica il numero di tentativi di invio che il pacchetto ha subito. Allo scadere del timeout di un pacchetto, il valore di questo campo viene incrementato di 1, fino al raggiungimento di una soglia limite (`MAX_RETRIES_SENDING_PKT`).

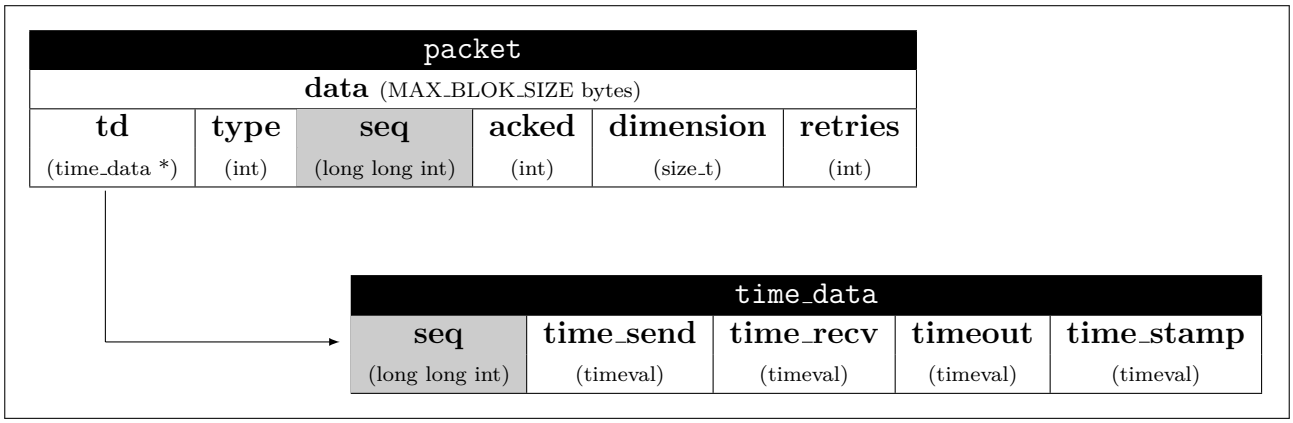

Figura 3: Relazione tra le due strutture dati `packet` *e* `time_data`*. Ogni pacchetto possiede, nel campo* **td***, l'indirizzo della struttura* `time_data` *a lui collegata, usata per il controllo del timeout. Le due strutture verranno gestite in maniera diversa da due strutture dati diverse. E' importante notare che il pacchetto e il timeout sono relazionati da uno stesso sequence number (campo* **seq***).*

4.2.2 Struttura *time_data*

La struttura dati **time_data** contiene dei campi molto importanti per il calcolo del timeout del pacchetto a cui è collegata. Questa struttura viene usata dal server o dal client durante l'invio di un file. Per quanto riguarda la ricezione, invece, questa struttura dati non viene utilizzata. I campi della struttura **time_data** sono mostrati in *Figura 3*, e verranno brevemente descritti di seguito.

`seq`: in questo campo viene memorizzato il numero di sequenza del pacchetto al quale la **time_data** è collegata.

`time_send`: individua l'ora locale di spedizione del pacchetto. Tramite questo campo viene effettuata la verifica sul timeout.

`time_recv`: individua l'ora locale di ricezione del pacchetto. Ogni volta che un ACK per un pacchetto viene ricevuto, si salva in questo campo l'ora locale di sistema. Inoltre viene calcolato l'**RTT** con la differenza tra `time_recv` e `time_send`. Questo permette di stimare dinamicamente il prossimo timeout per i successivi pacchetti da inviare.

`timeout`: in questo campo viene inserito il timeout assegnato al pacchetto.

`time_stamp`: questo campo non è utilizzato nell'attuale release del programma. E' stato pensato per salvare l'ora locale del pacchetto nel momento in cui i dati che contiene vengono scritti nel file di destinazione.

4.3 Gestione dei pacchetti

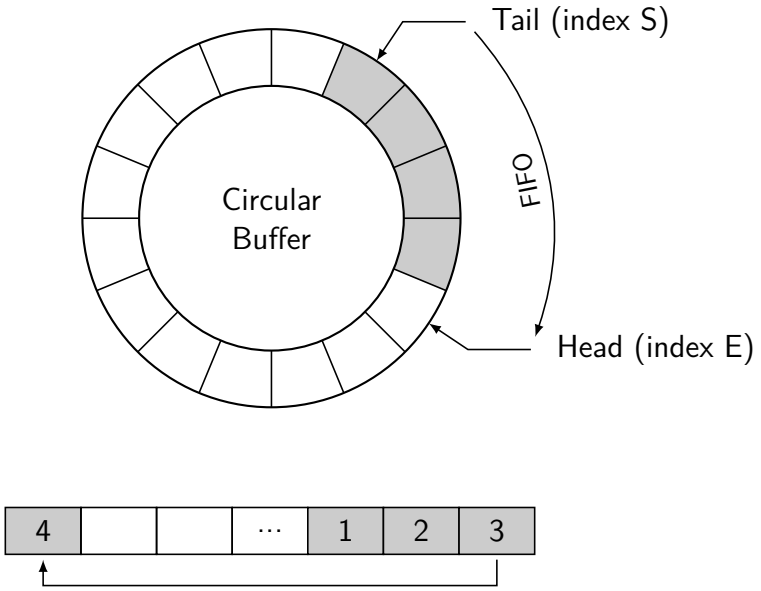

*Figura 4: Struttura circolare utilizzata per **window** e **timer_wheel**. Per la realizzazione è stato utilizzato un semplice array lineare, e la logica circolare è stata implementata all'interno delle funzioni che interagiscono con le strutture stesse. L'indice **S** individua la prima posizione da leggere, mentre l'indice **E** individua la prima posizione libera da scrivere.*

4.3.1 Struttura *window*

La struttura dati **window** rappresenta la finestra di spedizione del protocollo usato da RUDP. La dimensione della finestra è definita da `WINDOW_DIMENSION`[4]. La finestra è stata progettata come un **buffer circolare**: come si può vedere dalla *Figura 4*, i pacchetti inviati/ricevuti vengono inseriti uno di seguito all'altro. La struttura circolare della finestra ha permesso di creare una vera e propria *sliding window*, ponendo anche un limite alla quantità di pacchetti inseribili (e, quindi, dei pacchetti *on-the-fly*). Come si vedrà nel capitolo **4.3.2**, esiste la necessità di riordinare i pacchetti presenti in finestra secondo i loro campi **seq**. Durante l'invio di un file, i pacchetti verranno inseriti nella finestra scorrevole in ordine, ma durante la ricezione, questo non è garantito. L'algoritmo di sorting utilizzato per riordinare i pacchetti è il **quicksort**, e il suo costo di esecuzione è $O(nlogn)$.

Grazie alla struttura a layer del progetto, è stato possibile testare (senza grandi cambiamenti al codice già scritto) il comportamento di un altra struttura dati per rappresentare la finestra di spedizione: l'**albero AVL** (**A**delson-**V**elskij **L**andis). L'idea alla base del test è stata quella di ordinare i pacchetti nel momento in cui questi venivano inseriti, e non nell'stante immediatamente successivo all'inserimento. Infatti, la peculiarità dell'albero AVL è quella di mantenere un' altezza logaritmica rispetto al numero di foglie (migliorando quindi nettamente la ricerca) autobilanciando i nodi con rotazioni indotte dal loro 'peso'. Il prezzo da pagare in termini di efficienza è, appunto, il tempo speso nelle rotazioni (che pur rimane costante con $O(1)$). Le aspettative sull'uso di questa struttura dati non erano delle migliori, date le innumerevoli rotazioni che si supponeva sarebbero dovute essere effettuate.

[4]Trattandosi di un array circolare, la dimensione realmente utilizzata dal programma sarà (`WINDOW_DIMENSION - 1`). Se si usasse la piena disponibilità dell'array, si creerebbe una disambiguità: non si potrebbe più determinare con certezza se la finestra è piena oppure vuota.

In verità, i test condotti hanno evidenziato una inefficienza ancora maggiore di quella aspettata. Si è deciso dunque di lasciare inalterata l'implementazione della finestra come buffer circolare.

Approfondimento: AVL come finestra di spedizione

Si supponga che il client stia inviando (**PUT**) un file al server. Il file viene letto sequenzialmente, e iterativamente vengono memorizzati, all'interno dei pacchetti, `MAX_BLOCK_DIMENSION` byte letti dal file. Essendo creati uno dopo l'altro, i pacchetti da inviare saranno sicuramente in sequenza. Questo implica che, una volta inviati, verranno inseriti in finestra dal client sempre come ultima foglia destra dell'intero albero AVL. Per l'inserimento, quindi, si dovranno scorrere tutti i rami ed arrivare all'ultima foglia, con conseguenti rotazioni di bilanciamento (quindi $O(logn)$ per l'inserimento e $O(logn)$ rotazioni nel caso peggiore). La situazione descritta peggiora con l'aumentare della dimensione della finestra (e quindi all'aumentare dell'altezza dell'albero).

Nello stesso momento, il server che riceve il file, inserisce i pacchetti dento l'AVL. Se la probabilità di perdita di un pacchetto non è molto elevata, si può supporre che la maggior parte dei pacchetti arrivati sia in ordine di sequenza. Questo comporta spesso l'inserimento dell'ultimo pacchetto arrivato come ultima foglia destra dell'albero. Inoltre, mentre un buffer circolare ordinato può essere scorso iterativamente con tempo $O(1)$, un albero AVL necessita di un tempo $O(logn)$ per ricercare il successivo pacchetto con **seq** minore.

Il codice usato per implementare la finestra scorrevole come albero AVL si trova nella cartella `Unused code` del CD-ROM allegato. Si può visionare il codice aprendo i file *window.h*, *window.c*, *window_controller.h*, *window_controller.c*.

Per ulteriori approfondimenti sull'uso di alberi AVL, si propone la lettura della relazione *IA_relazione_finale.pdf* all'interno della cartella `Reliable UDP\data_example` contenuta nel CD-ROM. Tale relazione, è stata scritta dallo stesso autore di questo documento per il progetto finale del corso di *Ingegneria degli Algoritmi*.

4.3.2 Struttura *window_controller*

La struttura dati **window_controller** ha l'onere di gestire la finestra di spedizione. Le operazioni che vengono eseguite su **window** sono rese sicure dalla sincronizzazione tra thread e processi, gestione degli eventi, invio di segnali e attese su condizioni, implementate dentro le funzioni che interagiscono con **window_controller**. Questa struttura dati espone, quindi, funzioni affidabili per eseguire operazioni sulla finestra di spedizione.
window_controller è usata sia dal client che dal server in tutte le operazioni principali. Per esempio, supponiamo che il client invii una richiesta di operazione **GET** al server per un determinato file:

client: ogni pacchetto ricevuto, viene salvato nella finestra scorrevole. Dato che esiste la probabilità che un pacchetto possa non essere stato ricevuto (`LOSS_PROBABILITY`), quelli presenti in finestra possono non essere in ordine di sequenza. Quindi, ogni volta che ne viene aggiunto uno, i pacchetti vengono riordinati. Viene poi scritto il campo **data** di ogni pacchetto in ordine di sequenza sul file di output e spostato iterativamente l'indice che identifica il primo pacchetto da estrarre. Questa operazione di scrittura continua fino

allo svuotamento della finestra, oppure fino al raggiungimento di un pacchetto che non è ancora in ordine.

server: ogni pacchetto inviato, viene salvato nella finestra di spedizione per essere reinviato in caso di scadenza del timeout. Alla ricezione di un ACK, ne viene esaminato il campo **seq** per capire quale dei pacchetti in finestra è stato correttamente ricevuto; quindi, si imposta il campo **acked** del pacchetto a 1. Infine vengono controllati quanti pacchetti sono stati correttamente ricevuti analizzando tutti i campi **acked** di quelli presenti in finestra: tra questi, si eliminano iterativamente tutti i pacchetti in ordine, fino ad arrivare a svuotare completamente la finestra, oppure ad incontrare un pacchetto del quale ancora non si è ricevuto un ACK.

Tutte le operazioni descritte sopra, sono effettuate dalle funzioni che interagiscono con **window_controller**, e quindi sono in grado di operare sulla *sliding window* in modo sicuro.

Una delle funzionalità più importanti di **window_controller**, è quella di calcolare automaticamente il timeout dinamico per ogni pacchetto[5] ogni volta che viene chiamata la funzione *window_controller_set_ack()* all'arrivo di un ACK. Il timeout viene calcolato in base al tempo di spedizione e al tempo di arrivo dell'ACK per ogni pacchetto, stimando il tempo di **RTT** in base alla congestione della rete (simulata da una probabilità di perdita dei pacchetti).

[5]Il calcolo dinamico del timeout è stato realizzato per svolgere anche la richiesta opzionale del progetto dato. Per dettagli sul calcolo dinamico del timeout, leggere il codice e i commenti delle funzioni **window_controller_set_ack()** e **calculate_dynamic_timeout()** in `window_controller.c`

4.4 Gestione dei timeout

4.4.1 Struttura *timer_wheel*

La struttura dati **timer_wheel** serve per contenere e gestire le strutture **time_data** che, come precedentemente detto, racchiudono le informazioni temporali per ogni pacchetto: il timeout. La struttura in oggetto è un buffer circolare (*Figura 4*) proprio come **window**. Quello che rende diverse le due strutture, è il fatto che la **timer_wheel** non necessita di essere ordinata, ovvero i timeout non hanno bisogno di essere in ordine di sequenza. Il motivo è spiegato nel capitolo **4.4.2**. Un'altra sostanziale differenza, è che un timeout può essere eliminato singolarmente, senza bisogno che sia in ordine rispetto agli altri. L'operazione di *delete* è particolare, e per questo viene spiegata brevemente di seguito, con riferimenti ad alcune funzioni descritte in *timer_wheel.h*.

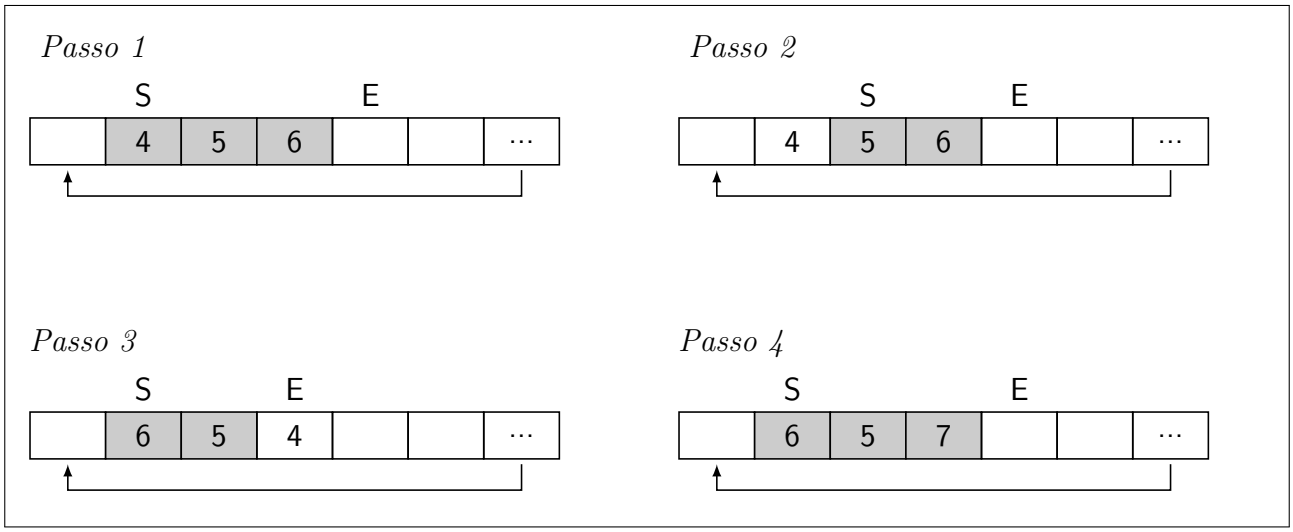

*Figura 5: Sequenza di passaggi che mostra come viene eliminato un timer all'interno della struttura dati **timer_wheel**. Gli slot evidenziati rappresentano quelli che ancora devono essere letti. Si ricorda che l'indice **S** individua la coda dell'array, ovvero la prima posizione disponibile per la lettura, mentre l'indice **E** individua la testa, ovvero la prima posizione disponibile per la scrittura.*

Come si vedrà nel capitolo successivo, gli indici della struttura **timer_wheel** vengono salvati prima di esaminare completamente l'array circolare, e ripristinati all'ultimo, prima di incominciare una nuova iterazione. Nel primo passo della *Figura 5* è rappresentata una ipotetica situazione iniziale, con tre slot della struttura **timer_wheel** ancora da leggere. Nel passo 2, è stato letto il primo slot in coda con la funzione *timer_wheel_get_timer()*, che preleva il timeout da analizzare dallo slot della struttura circolare e sposta l'indice **S** di una posizione, per leggere il successivo slot. Una volta prelevato il timeout, si decide che deve essere eliminato. Si chiama quindi la funzione *timer_wheel_delete_timer()*. Come si vede al passo 3 della *Figura 5*, l'elemento da eliminare viene scambiato con l'ultimo inserito, ed entrambi gli indici vengono spostati indietro di una posizione. In questo modo, l'indice **S** individuerà l'elemento che, prima dello scambio, era l'ultimo immesso (e quindi ancora da esaminare), mentre l'indice **E** individuerà la prima posizione libera, che ora è occupata dall'elemento da eliminare. Quando verrà inserito un nuovo elemento nell'array con la funzione *timer_wheel_add()*, verrà posizionato in **E**, sovrascrivendo l'elemento che era destinato ad essere eliminato (passo 4 della *Figura 5*).

4.4.2 Struttura *time_controller*

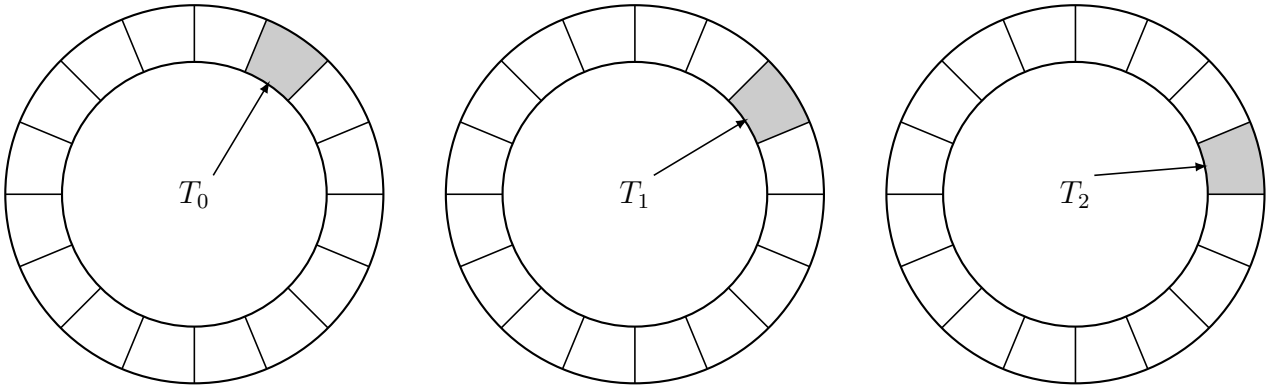

*Figura 6: Rappresentazione del comportamento del thread di controllo di **time_controller**. Come si può vedere, tutti i timeout vengono analizzati uno per uno, uno di seguito all'altro, fino al raggiungimento dell'ultimo timeout presente in **timer_wheel**. Una volta che viene terminato il controllo, il thread dorme per **TIME_CONTROLLER_GRANULARITY** ms prima di ricominciare la prossima scansione completa..*

La struttura dati **time_controller** realizza la gestione e il controllo **automatico** dei timeout inseriti nella struttura **timer_wheel**, garantendo affidabilità e sicurezza, grazie all'implementazione di funzioni thread-safe e l'uso di primitive di sincronizzazione, condizioni e segnali. Come appena detto, il controllo effettuato su ogni singolo timeout è svolto automaticamente: un thread asincrono realizza un vero e proprio controllo iterativo dei timeout presenti nella struttura circolare, e interagisce con la struttura **window_controller** per rispedire, se necessario, un pacchetto. Il comportamento di questo thread di controllo è rappresentato in *Figura 6*. Ogni `TIME_CONTROLLER_GRANULARITY` ms, il thread si attiva e controlla ad ogni passo T un timeout dentro **timer_wheel**, relativo ad un pacchetto presente in finestra: se questo è scaduto, si richiede alla funzione *window_controller_resend_packet()* di ricercare il pacchetto in finestra e rispedirlo, mentre il timeout (che era scaduto) viene ricalcolato, come spiegato nel capitolo **4.5**. Ogni volta che il thread esamina tutti i timeout, si mette a dormire per `TIME_CONTROLLER_GRANULARITY` ms.
Questa struttura dati non viene utilizzata per tutte le operazioni principali, come invece accade per **window_controller**. Infatti, solamente la controparte che invia (**PUT**) il file deve tenere conto dei timeout di ogni pacchetto inviato. Come detto nel capitolo **4.2.1**, ogni pacchetto possiede un puntatore al proprio timeout. Quando un pacchetto viene spedito, il ricevente non usa affatto questo campo[6], in quanto non necessario: è la controparte che si accorge quando il timer è scaduto, e provvede ad inoltrare nuovamente il pacchetto perso.

[6]Anche se volesse usarlo, non potrebbe: il destinatario riceverebbe un puntatore ad un indirizzo di memoria dove (nei casi migliori) non è presente nessun dato.

4.5 Il protocollo ARQ di RUDP

Il protocollo di controllo degli errori adottato in RUDP è del tipo **ARQ** (**A**utomatic **R**epeat-re**Q**uest). In particolare, è stato utilizzato un protocollo **Selective Repeat** con finestra di spedizione fissa **N**: vengono spediti al massimo **N** pacchetti (*on-the-fly*) senza dover aspettare l'ACK di un pacchetto per il successivo invio. La certezza del ricevimento di un pacchetto è data esclusivamente dalla ricezione di un ACK relativo al pacchetto stesso. Ogni pacchetto, infatti, possiede un campo **seq** (*Figura 3*), che permette di identificarlo univocamente.

Le richieste di operazione dal client al server vengono effettuate da pacchetti di tipo `PKT_GET`, `PKT_PUT` e `PKT_LS` (*Figura 2*). Ogni pacchetto di richiesta viene inviato con sequence number 0. Il ricevimento di un ACK con sequence number 0 implica che la richiesta è stata accettata dal server, e che entrambe le controparti possono iniziare la preparazione per l'operazione richiesta. Tutti i pacchetti spediti da questo punto in poi, hanno il campo **seq** crescente a partire da 1.

Gli ACK non sono comulativi: si riceve un ACK per ogni pacchetto inviato. Se un pacchetto non riceve un riscontro entro il tempo di timeout, questo verrà rispedito, e gli verrà assegnato un tempo di timeout raddoppiato. Il motivo dell'aumento del tempo di timeout è semplice: se un pacchetto non viene ricevuto, probabilmente la linea è congestionata; se la linea è congestionata, si ha motivo di credere che il tempo di ricezione di un ACK per il pacchetto sia maggiore di quello stimato. Se un solo pacchetto tra tutti quelli spediti viene reinviato più di `MAX_RETRIES_SENDING_PKT` volte, vuol dire che la rete è troppo satura, ed il trasferimento potrebbe durare troppo tempo. La connessione dunque si interrompe, mostrando a schermo un messaggio utente.

Impostando la dimensione della finestra[7] a 1, il protocollo usato sarà **Stop and Wait**: si invia un solo pacchetto alla volta (proprio perchè la finestra di spedizione avrà un solo slot disponibile) e si attenderà l'ACK relativo al pacchetto prima di inviarne uno nuovo. Questo tipo di trasmissione è molto meno efficiente. Ma è anche vero che la scelta della dimensione della finestra scorrevole è importante: un valore troppo elevato potrebbe garantire trasmissione continua, ma anche svantaggi dovuti alla gestione della moltitudine di pacchetti in finestra.

Approfondimento: dimensione finestra per trasmissione continua

La dimensione della finestra da utilizzare per avere trasmissione continua può essere calcolata con la seguente uguaglianza

$$\frac{W \cdot M}{RTT + \frac{M}{C}} >= C$$

dove:
W: dimensione della finestra scorrevole
M: dimensione del messaggio/pacchetto
RTT: tempo di andata e ritorno
C: capacità del collegamento

[7] Si ricordi che, come spiegato nella nota a piè di pagina del capitolo **4.3.2**, il valore che verrà immesso in **WINDOW_DIMENSION** è, in realtà, usato a meno di 1, per via dell'implementazione circolare delle strutture dati.

4.6 Operazioni principali

In questo capitolo, verranno descritte le tre operazioni principali che il client può richiedere al server. Le operazioni locali del client, come la visualizzazione di un help file o il cambio di lingua, non fanno parte delle operazioni principali.

4.6.1 Operazione PUT

L'operazione di **PUT** viene richiesta tramite il pacchetto `PKT_PUT` al server. Il pacchetto di richiesta porta con se due dati importanti: il **nome** del file che si intende trasferire e il **numero di pacchetti** totali che verranno inviati per completare l'operazione (maggiori dettagli in *Figura 2*). Come si può vedere dalla *Figura 7*, se il server accetta la richiesta (dato che può gestire solamente `MAX_PROCESSES_NUMBER` richieste contemporaneamete), rispedisce al mittente un ACK contenente il numero di porta prescelto per eseguire l'operazione, e crea un **processo figlio**, a cui viene assegnato il compito di completare l'operazione stessa. Successivamente, il client invia tutti i pacchetti al server sulla porta ricevuta in risposta. Quando anche l'ultimo ACK è stato ricevuto, si procede con la chiusura della connessione. Il client invia un pacchetto di tipo `PKT_FIN`, per avvertire il server che la trasmissione è stata completata con successo. Al suo ricevimento, il server invia un pacchetto di tipo `PKT_FINACK` per comunicare l'effettiva chiusura della connessione, e nello stesso tempo chiude la porta utilizzata, rendendola disponibile per una nuova operazione. Inoltre, il processo che si è occupato della ricezione dei dati viene terminato. Una volta ricevuto questo pacchetto, il client ritorna disponibile per richiedere una nuova operazione al server.

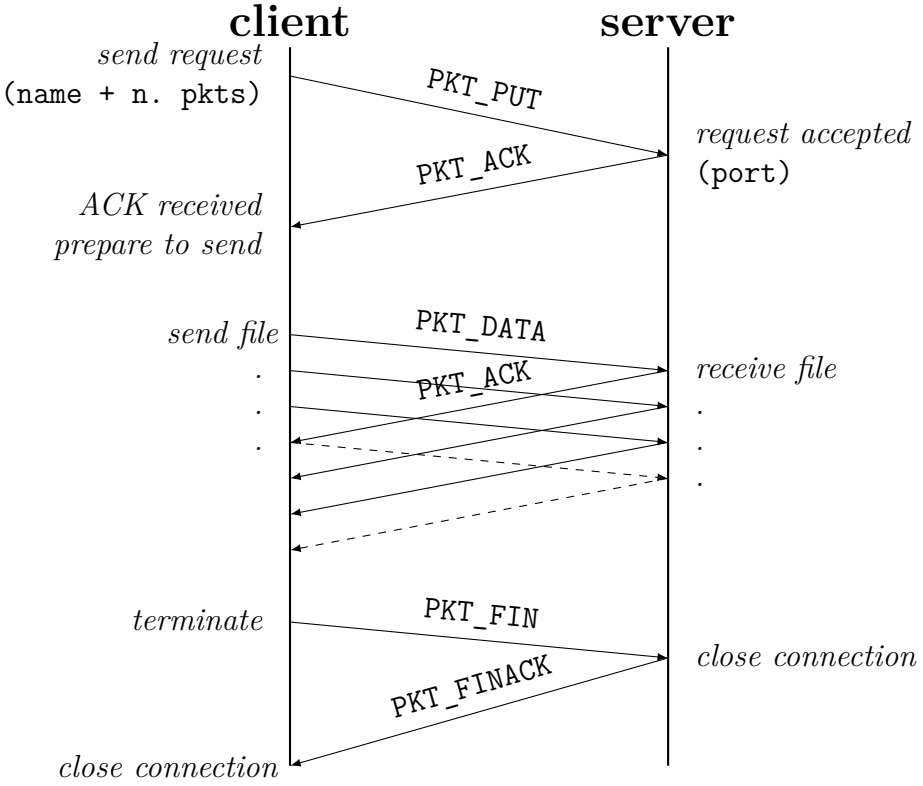

*Figura 7: Rappresentazione dell'operazione **PUT** tra client e server. La parte iniziale corrispondere alla fase di inizializzazione della connessione, quella centrale alla fase di trasmissione dei dati, mentre quella finale alla fase di chiusura della connessione. Per maggiori dettagli circa questa operazione, leggere il codice e i commenti del file **put.c***

4.6.2 Operazione GET

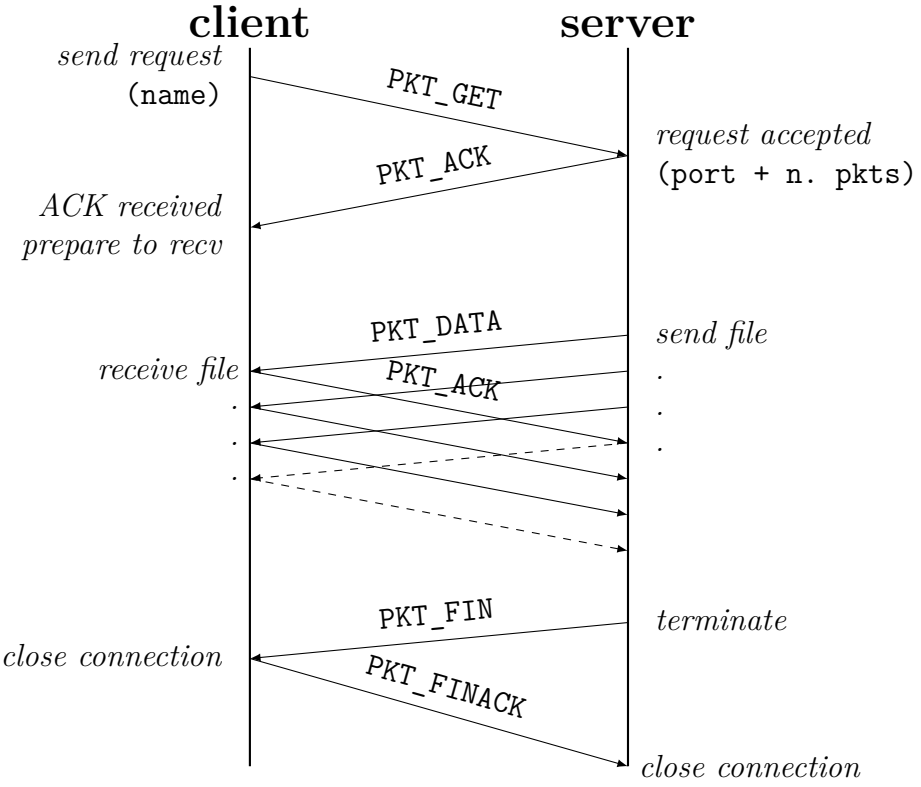

*Figura 8: Rappresentazione dell'operazione **GET** tra client e server. La parte iniziale corrispondere alla fase di inizializzazione della connessione, quella centrale alla fase di trasmissione dei dati, mentre quella finale alla fase di chiusura della connessione. Per maggiori dettagli circa questa operazione, leggere il codice e i commenti del file **get.c***

Come accennato nel capitolo **3.2.3**, la massima attenzione nel riuso del codice, ha portato a creare funzioni le più generali possibili, utilizzate sia dal client che dal server. Per quanto riguarda le operazioni **PUT** e **GET**, queste possono essere viste come faccia della stessa medaglia: se il client esegue una operazione di **GET**, il server invierà i dati con una operazione di **PUT**; se il client esegue una operazione di **PUT**, il server riceverà i dati con una operazione di **GET**. Per questo motivo, i passi eseguiti dall'una o dall'altra sono molto simili, come si nota confrontando la *Figura 7* con la *Figura 8*. Pertanto, la descrizione in questo capitolo sarà breve.
Per quanto riguarda la fase di richiesta, questa viene effettuata tramite l'invio di un pacchetto di tipo `PKT_GET` al server, contenente il nome del file che si desidera ricevere. Il server risponde (se la richiesta viene accettata) con l'invio di un pacchetto `PKT_ACK`, contenente sia la porta prescelta per eseguire l'operazione di invio, sia il numero di pacchetti totali che verranno inviati (dettagli in *Figura 2*). Nello stesso tempo, viene creato dal server un processo figlio, incaricato di seguire l'operazione di invio. Quando anche l'ultimo ACK è stato ricevuto, il server invia un pacchetto `PKT_FIN`. Da parte del client la connessione viene chiusa con la ricezione di questo pacchetto e l'invio di un pacchetto `PKT_FINACK`.

4.6.3 Operazione LIST

La richiesta dei file presenti nel server viene effettuata tramite il pacchetto `PKT_LS`. Il server che riceve il pacchetto si occupa di leggere tutti i file presenti nella cartella `data`, e scrive i loro nomi su un file di testo *.txt*. Il file viene poi spedito come fosse stata richiesta una operazione di **GET** per questo stesso file. Fondamentalmente, anche in questo caso, si è voluto massimizzare il riuso del codice, utilizzando parti di programma già implementate. E' da notare il fatto che, nello stesso istante, il server possa riceve più richieste contemporanee per la lista di file, e che nello stesso server possano essere inviati diversi file ogni istante. La creazione della lista, quindi, viene eseguita istantaneamente a seguito della richiesta. Inoltre, onde evitare che si verifichino conflitti di nome per le liste create, queste vengono nominate univocamente prendendo come riferimento l'ora istantanea locale del sistema per ogni file.
Il file di lista viene creato nella cartella `temp` del server, per evitare che il nome della lista stessa compaia insieme ai file presenti o che possa essere richiesto da un altro client. In ultimo, successivamente all'invio, la lista viene eliminata per recuperare spazio disco.
Il client salva la lista nella cartella `data` con il nome *server_list.txt* (modificabile tramite `LIST_FILE`). Ad ogni nuova richiesta di lista, questa viene sovrascritta.

4.7 Messaggi utente: log file e verbose mode

I programmi server e client non dispongono di interfaccia grafica, ma vengono lanciati e usati da terminale. Le interazioni tra utente e programma avvengono mediante input da tastiera e messaggi testuali. Nella normalità, i messaggi visualizzati a schermo all'utente sono solamente quelli essenziali, che corrispondo alla percentuale di completamento dell'operazione in corso oppure a messaggi di errore o informazione generici. Quando server o client vengono avviati in modalità **verbose**, ovvero con l'argomento **-v**, i messaggi a schermo visualizzati durante tutta la sessione saranno molto più specifici e frequenti. Verrà notificato l'arrivo o l'invio di ogni pacchetto, l'invio o la ricezione di ogni ACK, o ancora la porta selezionata dal server per iniziare il trasferimento. Si è deciso di rendere possibile la visualizzazione di questi messaggi per il test e il debug del progetto. Trasferendo infatti l'output del programma dallo standard output ad un file generico, è stato possibile analizzare il comportamento delle operazioni in varie circostanze.
Oltre ad essere lanciati in modalità verbose, il server e il client possono essere avviati con l'argomento **-l**. In questo caso, il programma gestirà un file di log che includerà tutti i messaggi più importanti visualizzati a schermo per l'utente. Il file verrà salvato nella cartella `log` del programma stesso. Per quanto riguarda il server, esiste un solo file di log (*LOG.txt*), che verrà aggiornato di volta in volta, mentre per il client verranno creati file di log diversi per ogni giorno. I messaggi che vengono visualizzati in verbose mode non sono scritti nel log file, in quanto si è pensato di rendere il file di log leggero e facilmente leggibile, inserendo solo i messaggi riguardanti gli eventi più importanti.
Dato che il client gestisce più thread, e il server gestisce contemporaneamente più thread e più processi, la scrittura dei messaggi nel file di log e a schermo è stata sincronizzata tramite opportune primitive di sincronizzazione (semafori POSIX anonimi di tipo *sem_t* per i processi e mutex di tipo *pthread_mutex_t* per i thread).

Approfondimento: strutture dati *log_service* e *verbose_service*

Durante la fase di ideazione del progetto, sono state ideate e progettate delle strutture dati in grado di gestire in maniera semi automatica la verbose mode e la scrittura sul file di log: **log_service** e **verbose_service**. Queste strutture dati sono state implementate e testate, ma non fanno parte del progetto finale per dei problemi riscontrati durante la fase di test.

Le strutture dati: il codice delle due strutture dati sopra menzionate si può trovare nella cartella `Unused code`. Come si può vedere dal codice stesso, queste gestivano la visualizzazione e la scrittura dei messaggi di testo. Il messaggio vero e proprio veniva rappresentato da un'altra struttura dati: **log_msg**. In questa struttura, venivano memorizzati diversi dati: data e ora del messaggio, tipo di messaggio, testo del messaggio e descrizione opzionale. L'utente poteva scegliere il livello di dettaglio con il quale desiderava far scrivere il messaggio: LOW, MEDIUM, HIGH o EXTREME. Una volta impostate i vari parametri di queste strutture dati, la creazione e scrittura dei messaggi avveniva automaticamente, con ora e data locale.

Test: i test eseguiti con queste strutture dati hanno evidenziato un problema nella rappresentazione dei caratteri a schermo in alcuni sistemi operativi: mentre su Mac Os X ogni carattere veniva correttamente stampato, in Linux, questo non era sempre vero. La colpa era senz'altro attribuibile alla funzione con cui venivano generate e concatenate le stringhe per essere passate come parametro alle funzioni delle strutture dati sopra citate: *create_string()*. Dato che il problema era secondario e che la stima del tempo per la risoluzione non era calcolabile rapidamente, si è deciso di eliminare il codice problematico ed usare un'altra strada, ovvero creare semplici funzioni che hanno il solo scopo di scrivere il messaggio utente. Questo cambiamento è stato possibile senza troppo tempo grazie alla struttura a strati del progetto. Le funzioni che si occupano di scrivere i messaggi a schermo e sul file di log possono essere esaminate leggendo il file *print_messages.c*, mentre la funzione *create_string()* può essere visionata in *utils.c*, dove è stata commentata interamente.

4.8 Impostazioni configurabili: *settings.h*

Come già discusso nel capitolo **2.4**, l'utente ha la possibilità di configurare le impostazioni principali del programma modificando il valore delle definizioni presenti nel file *settings.h*. Questo header file è stato pensato per centralizzare tutte le impostazioni, permettendo all'utente di configurarle accedendo in un unico punto. All'interno del file sono presenti commenti dettagliati di ogni `#define`. E' consigliabile la lettura dei commenti prima di modificare qualsiasi valore. Infatti, inserendo un valore non appropriato, protrebbero verificarsi errori non gestiti correttamente.

4.9 Internazionalizzazione

L'utilizzatore del client ha la possibilità di poter cambiare lingua dei messaggi visualizzati a schermo. E' stato pensato di internazionalizzare solo il lato client dell'intero programma, in quanto verosibilmente, se fosse stato un prodotto commerciale, sarebbe stato utilizzato da molte più persone rispetto al lato server.
In principio l'internazionalizzazione era stata progettata usando tool semi professionali come **ICU** (**I**nternational **C**omponent for **U**nicode, `http://site.icu-project.org`) o **GNU gettext** (`https://www.gnu.org/software/gettext/`). Successivamente, data la natura puramente didattica del progetto, si è deciso di ideare in loco un sistema molto più semplice e minimale, che permettesse, oltre che il cambio lingua, la possibilità di aggiungere semplicemente altre lingue al programma. Per questo scopo non sono stati utilizzati file esterni, in quanto la lettura di file di testo avrebbe portato una visualizzazione di caratteri errati se si fosse sbagliata codifica. Il modo in cui si è internazionalizzato il programma è esposto di seguito.

E' stato creato un array globale di stringhe vuoto, accessibile da qualunque parte e da qualunque thread del programma. Appena si avvia il client, il puntatore di questo array globale viene 'spostato' verso una zona di memoria che contiene un array di stringhe pieno, scritte nel linguaggio scelto (di default, vengono caricate stringhe in inglese). La funzione che permette di caricare una determinata lingua è *select_language()*. Questa, deve essere chiamata come prima funzione del programma, prima di qualsiasi altra funzione, altrimenti non si avrebbe nessuna lingua caricata e il programma si comporterebbe in maniera anomala. Una volta che l'array globale possiede stringhe in una determinata lingua, è possibile estrarle e stamparle a schermo o su file. Per una estrazione della stringa più intuitiva, sono state create delle definizioni in *strings.h* che hanno un nome mnemonico, per facilitare il riconoscimento della stringa stessa. La funzione che si occupa di estrarre la stringa desiderata è *get_text()*[8]. Per esempio, per estrarre la stringa *"server started"*, invece di *get_text(9)* è molto più intuitivo passare come parametro *get_text(STRING_SERVER_STARTED)*. Per cambiare lingua in italiano in tempo reale, l'utente può digitare sul terminale **LANG ita**: viene richiamata la funzione *select_language()*, che sposta il puntatore dell'array in un'area di memoria contenente l'array di stringhe in italiano. Tutte le prossime estrazioni da questo array, dunque, conterranno stringhe tradotte in italiano.

L'aggiunta di una nuova lingua prevede pochi passi. Innanzitutto, occorre creare un nuovo identificatore per la lingua stessa (come **ita** per italiano, **eng** per inglese, ...), inserendolo tra le possibili scelte dentro l'enumerazione *languages* in *strings.h* e aggiungendo un `case` in *select_language()*. Occorre poi tradurre le stringhe nello stesso ordine in cui sono presentate, ricopiando fedelmente la struttura delle funzioni *load_it_lang()* e *load_en_lang()* in *strings.c*. In ultimo, occorre informare l'utente della possibilità di scegliere una nuova lingua, e aggiungere nel `case PKT_LANG` in *client.c* la possibilità di identificare l'id della nuova lingua.

Sicuramente questo metodo di traduzione è molto più ortodosso rispetto a quello utilizzato dai tool professionali, ma è comunque molto semplice ed intuitivo, e per lo scopo per il quale è stato pensato è sicuramente una alternativa valida.

[8]Dato che questa funzione viene usata molto spesso, si è pensato di creare una definizione per abbreviarla: **#define _(str) get_text(str)**

4.10 *Makefile* e script *install.sh*

Per l'installazione del programma, si è scelto di creare un **makefile** per la compilazione dei sorgenti, e un piccolo **script bash** per la generazione delle cartelle necessarie al funzionamento del software e per lanciare automaticamente il comando **make**.

Il file *install.sh* esegue i seguenti passi:

1. pulisce lo schermo e rimuove, se esistono, le vecchie cartelle generate da una precedente installazione
2. esegue il makefile per compilare sia server che client, ottenendo due file eseguibili
3. crea tutte le cartelle e sottocartelle necessarie per il server e per il client
4. muove i file eseguibili nella posizione appropriata, così come tutti i file complementari necessari per il corretto funzionamento (come gli help files e i sample files) nelle cartelle create nel passo precedente.

Il makefile, invece, contiene il codice per trasformare i sorgenti in eseguibili. I codici sorgenti vengono compilati da terminale con il comando **gcc -Wall -Wextra -pthread < .c files > -o < exec name>**. Gli argomenti **-Wall** e **-Wextra** abilitano la visualizzazione di tutti i messaggi di warning, mentre l'argomento **-pthread** è necessario per compilare codice che usa la libreria *pthread.h*.
Il makefile distingue la compilazione del client da quella del server: per compilare il client, viene lanciato il comando `$make CLIENT`, per compilare il server, viene lanciato il comando `$make SERVER`.

Per lanciare la procedura automatica di installazione, seguire i passi elencati nel capitolo **2.2**.

5 Test

5.1 Piattaforme di test

Il programma è stato testato su vari computer e macchine virtuali con SO Linux Ubuntu e Mac Os X. In questo documento, si è deciso di riportare i test effettuati solamente con una delle piattaforme di test, in modo tale da non avere discrepanze tra i risultati ottenuti con hardware diverso. Si è usato, per lo scopo, una macchina virtuale:

HW fisico: MacBook Pro retina, CPU i5 4 x 2.7 ghz, RAM DDR3 8Gb 1867 Mhz
SO primario: Mac OS X Yosemite 10.10.4
HW virtualizzato: CPU i5 2 x 2.7 ghz, RAM DDR3 2Gb 1867 Mhz
SO virtualizzato: Linux Ubuntu 14.04.03 LTS 64 bit
SW di virtualizzazione: Parallels Desktop 10.1.0

5.2 Risultati ottenuti

Di seguito sono riportati i grafici dei risultati ottenuti in fase di test. Sono stati osservati i comportamenti al variare della **probabilità di perdita** di un pacchetto, e al variare della **dimensione della finestra** di spedizione. E' importante notare che, soprattutto per i test riguardanti la probabilità di perdità, non sarà possibile garantire gli stessi esatti risultati su macchine diverse. Il programma è completamente funzionante sulle macchine e sistemi operativi supportati, ma non viene garantito lo stesso comportamento tra calcolatori differenti. Pertanto, è possibile vi siano piccole differenze tra i risultati trascritti in questo capitolo e quelli ottenibili lanciando il programma su un pc.
I test sono stati effettuati tutti sull'operazione **GET** e sul file **gapil.pdf**.

5.2.1 Test: probabilità di perdita variabile

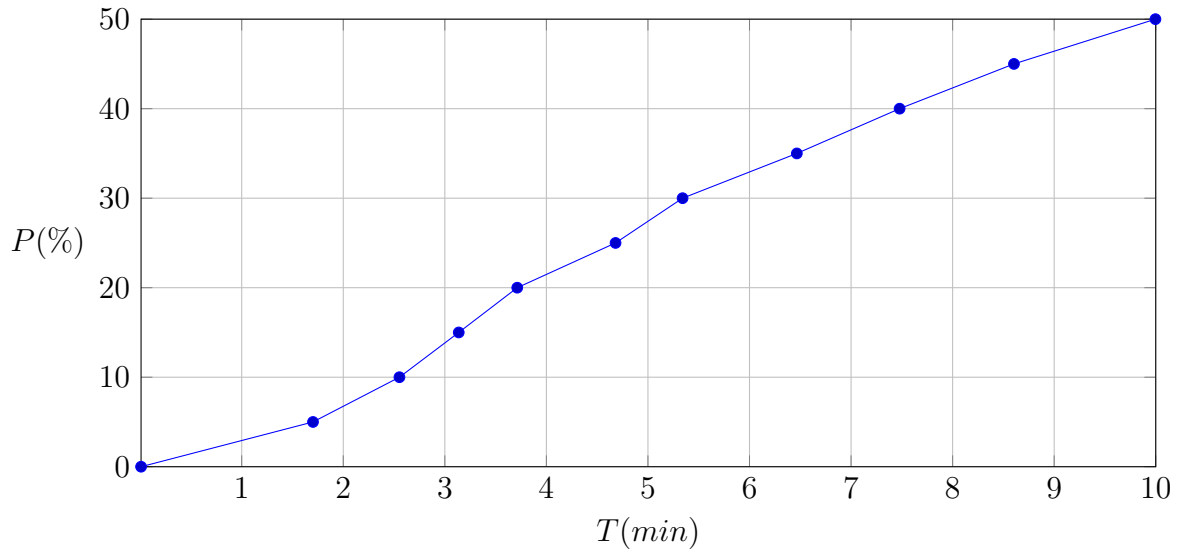

*Figura 9: Tempo totale per l'operazione **GET** sul file **gapil.pdf**. La dimensione della finestra di spedizione è stata lasciata al valore di default: **N** = 30.*

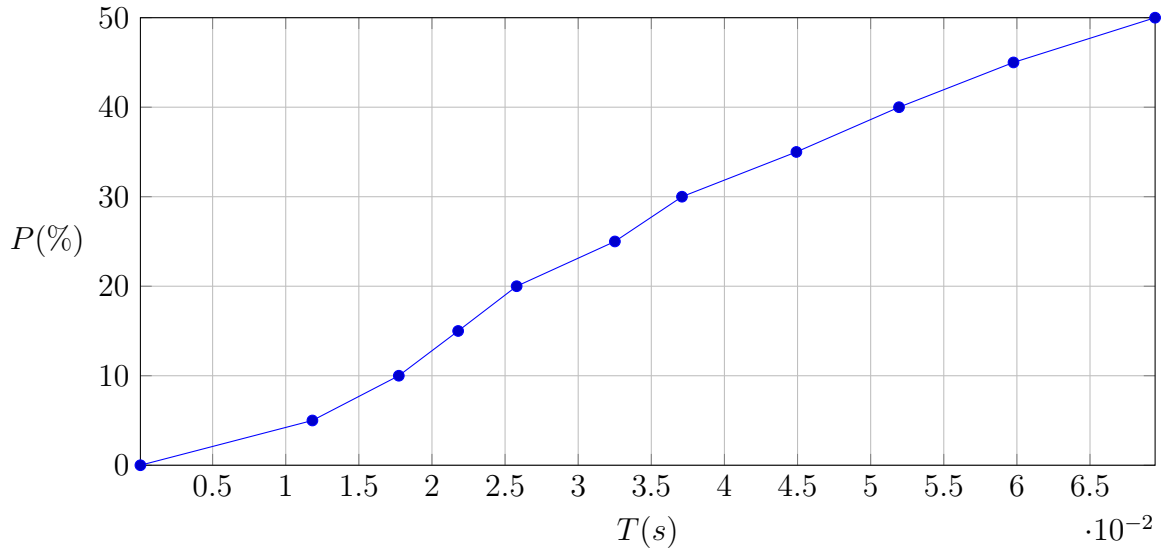

*Figura 10: Tempo medio di ricezione dei pacchetti per l'operazione **GET** sul file **gapil.pdf**. La dimensione della finestra di spedizione è stata lasciata al valore di default: **N** = 30.*

5.2.2 Test: finestra di spedizione variabile

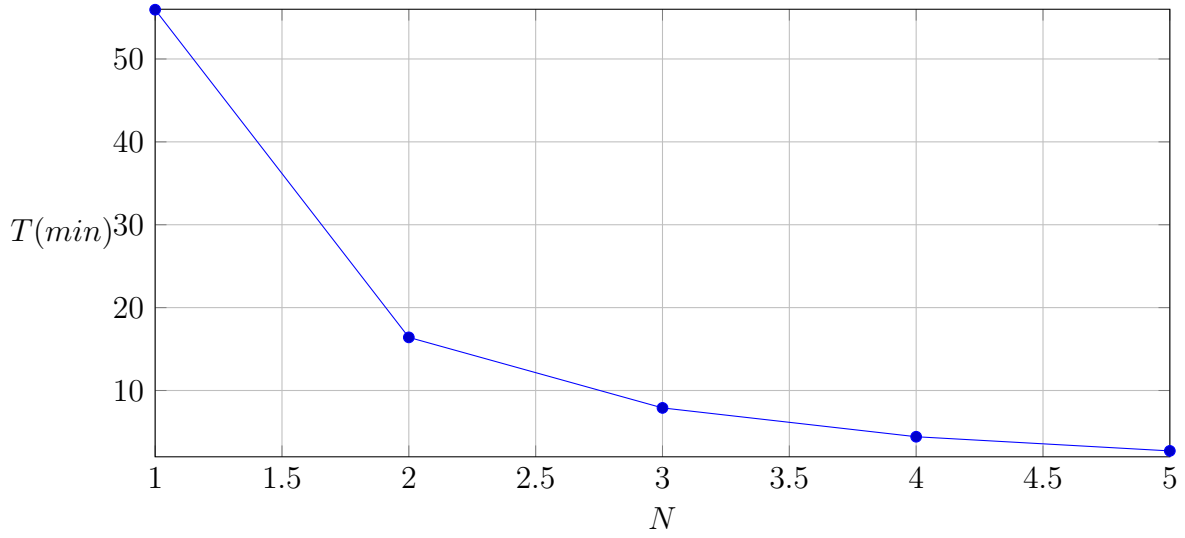

*Figura 11: Tempo totale impiegato per l'operazione **GET** sul file **gapil.pdf** al variare della dimensione della finestra di spedizione **N**. Grafico con **N** da 1 a 5.*

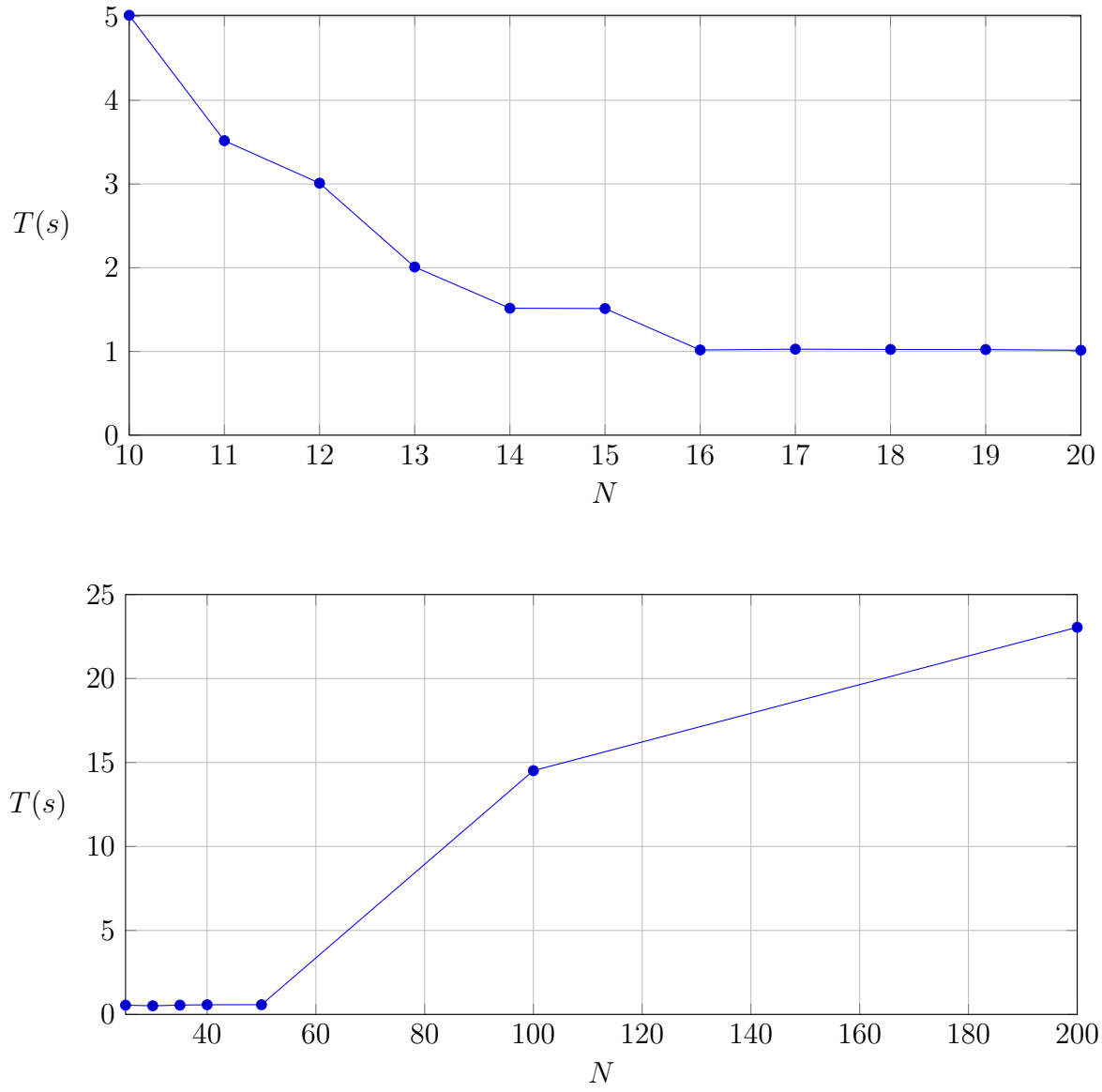

*Figura 12: Tempo totale impiegato per l'operazione **GET** sul file **gapil.pdf** al variare della dimensione della finestra di spedizione **N**. Primo grafico con **N** da 10 a 20, secondo grafico con **N** da 25 a 200.*

5.3 Analisi dei risultati

I grafici mostrano il comportamento del client durante l'operazione **GET**. In particolare, nei grafici di *Figura 9* e *10*, è stata fatta variare per ogni richiesta il valore della probabilità di perdita (`LOSS_PROBABILITY`) da 0 a 50. Gli altri parametri sono stati lasciati invariati al valore di default. Il valore limite della probabilità di perdita entro il quale si è riuscito a ricevere il file è 50%. Dopo questa soglia, è capitato molto spesso che il programma interrompesse la ricezione per due motivi differenti: un pacchetto veniva rispedito troppe volte (`MAX_RETRIES_SENDING_PKT`) oppure non veniva ricevuto un pacchetto per troppo tempo (`MAX_INACTIVITY_TIME`). Come si evince dal grafico, il tempo di completamento cresce all'aumentare della probabilità di perdita: si passa da 0.009 minuti con la probabilità di perdita allo 0% a 9,997 minuti con la probabilità di perdita al 50%. La *Figura 10* mostra il tempo medio di ricezione per singolo pacchetto. La curva generata dai dati segue quella del tempo totale del grafico precedente. E' evidente da entrambe le figure che la probabilità di perdita è senz'altro il fattore che determina maggiormente il tempo di completamento dell'operazione. Con una perdita totale dei pacchetti che va

dal 40% al 50%, diventa quasi impossibile trasferire un file di circa 8 Mb: il tempo cresce di 3 ordini di grandezza rispetto alla probabilità dello 0%. Se si cercasse di rimediare a questa inefficienza aumentando di molto il valore di `MAX_RETRIES_SENDING_PKT` e `MAX_INACTIVITY_TIME`, si potrebbe andare incontro a situazioni di stallo del programma. L'unica soluzione adottabile è quella di rendere la trasmissione dei dati la più affidabile possibile, diminuendo la perdita dei pacchetti derivata dall'infrastruttura e dalla congestione della rete.

Le *Figure 11* e *12*, invece, mostrano il comportamento del client al variare della dimensione della finestra di spedizione. Come si può vedere dal primo grafico, il protocollo stop-and-wait è molto ineficiente: ci vogliono in media 50/55 minuti per un trasferimento completo. Inoltre, basterebbe una piccola variazione sulla probabilità di perdita per peggiorare drasticamente il risultato. Si può notare, però, che all'aumentare di **N**, il tempo di ricezione diminuisca notevolmente, fino ad assestarsi tra i valori di **N** da 15 a 50 e tra 1 e 0.5 secondi. La pendenza della curva in questo tratto di grafico è molto marcata, indice di come bastino pochi slot in più nella finestra di spedizione per migliorare le prestazioni. Si passa infatti da circa 55 a circa 15 minuti totali richiesti per il completamente dell'operazione, aumentando da 1 a 2 la dimensione della finestra. Lo stesso ragionamento, però, non è valido per valori di **N** molto grandi. Infatti, si può notare come l'aumento della dimensione della finestra oltre la soglia di 50, provochi un aumento del tempo medio di completamento per il trasferimento. Quindi una finestra di spedizione troppo grande non porta benefici, ma il contrario. Questo è dovuto al fatto che i pacchetti da gestire in finestra diventano una enormità, e soprattutto la fase di riordino dei pacchetti impiega molto tempo. E' notevole però il fatto che la curva, in questo tratto di grafico, abbia una pendenza minore a quella rilevata tra valori molto bassi di **N**. Quindi si può dedurre che valori molto alti di **N** hanno effetti meno disastrosi di valori molto piccoli. I test effettuati hanno portato ad identificare un buon valore di default per la finestra di spedizione: 30.

6 Conclusioni

In questo documento sono state descritte tutte le funzionalità del programma, le caratteristiche più importanti, le idee e i principi di funzionamento. L'autore, però, è consapevole che questa release è largamente migliorabile, sia per quanto riguarda l'implementazione che le funzionalità. In particolare, potrebbe essere interessante rendere automatica l'impostazione della dimensione della finestra di spedizione di RUDP, ovvero ingrandirla o diminuirla a seconda della congestione della rete, in modo automatico; oppure rendere disponibili nuove funzionalità: per esempio il comando `REMOVE` per eliminare un file sul server, il comando `STATUS` per ricevere lo stato attuale del server, o anche il cambiamento di lingua dei messaggi del server a seconda del linguaggio impostato sul client. In ultimo, si potrebbe cambiare l'algoritmo utilizzato per il riordino dei pacchetti in finestra, ideandone uno che possa essere eseguito *in loco*, senza spreco ulteriore di memoria. Gli obiettivi principali richiesti, però, sono stati tutti raggiunti: è stato realizzato un vero e proprio protocollo di trasmissione dati affidabile, **RUDP**. Il tempo di sviluppo è stato di circa 3 mesi, e le difficoltà affrontate sono state molte. Innanzitutto tutte le fasi dello sviluppo sono state affrontate da una singola persona, e questo è il motivo per il quale sono stati necessari circa 3 mesi di lavoro, oltre il fatto di aver avuto altri esami durante questo periodo. Inoltre, la codifica in linguaggio C ha richiesto lunghi approfondimenti circa le funzioni e le chiamate di sistema che era necessario usare. E' stato comunque molto soddisfacente riuscire a portare a termine un progetto complicato come questo, e sono state acquisite nuove competenze ed abilità grazie al tempo dedicatogli.

Anche la stesura di questa relazione in LaTeX ha richiesto più tempo del normale, circa 3 settimane. L'intento dell'autore è stato quello di rendere questo documento il più professionale

possibile, quasi come fosse un manuale di studio. Inoltre, si è colta l'occasione di sperimentare questo linguaggio di markup ad oggi largamente utilizzato. In ultimo, le immagini presenti sono state pensate per rendere più semplice la spiegazione di alcuni concetti che, altrimenti, sarebbero stati spiegati in molte più righe di foglio .

7 Licenza e redistribuzione

Il software e la relazione sono modificabili e distribuibili secondo la licenza **GNU GPL v3** per il software libero. Una copia della licenza è disponibile nella cartella `License` nel CD-ROM, oppure al sito `http://www.gnu.org/licenses/gpl-3.0.en.html`

8 Bibliografia

Riferimenti bibliografici

[1] S. Piccardi, *GaPil: Guida alla Programmazione in Linux*, revisione 6 febraio 2014

[2] M. Kerrisk, *The Linux Programming Interface*, 2010

[3] C. Demetrescu, I. Finocchi, G. F. Italiano, *Algoritmi e Strutture Dati*, 2008

[4] Al Kelley, Ira Pohl, *C: Didattica e Programmazione*, 2012

[5] G. Varghese, T. Lauck, *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility* (`http://www.researchgate.net/publication/234799806_Hashed_and_hierarchical_timing_wheels_Data_structures_for_efficient_implementation_of_a_timer_facility`), 1987

[6] IBM, *Implement lower timer granularity for retransmission of TCP* (`http://www.ibm.com/developerworks/aix/library/au-lowertime/index.html`)

[7] J. F. Kurose, K. W. Ross, *Reti di calcolatori e internet: un approccio top-down*, 2013

[8] M. Cesati, *Slides per il corso di Sistemi Operativi*, 2014

[9] F. Lo Presti, *Slides per il corso di Ingegneria di Internet e del Web*, 2014

[10] L. Pantieri, T. Gordini, *L'arte di scrivere con LaTeX* (`http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf`), 2012

[11] L. Pantieri, T. Gordini, *L'arte di disegnare con LaTeX* (`http://www.lorenzopantieri.net/LaTeX_files/ArteTikZ.pdf`), 2014