# Ingegneria di Internet e Web

2019 - 2020

Valerio Crecco - 0239461

Ludovico De Santis - 0244291

**Traccia B2**

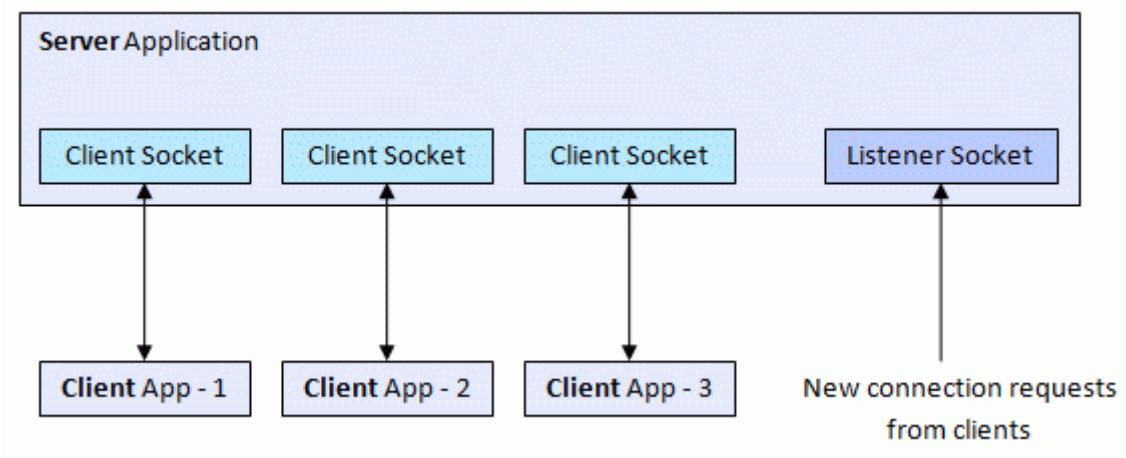
# Indice

## 1. Introduzione

La traccia in esame prevede la realizzazione di un sistema per il trasferimento di file attraverso l'utilizzo di un servizio di rete senza connessione, ovvero l'utilizzo del protocollo UDP al livello di trasporto, basato sull'impiego di API del socket di Berkeley (socket di tipo SOCK_DGRAM per UDP). L'applicazione prevede un sistema di connessione client-server senza autenticazione, l'implementazione delle funzionalità di list, download e upload in modo affidabile, realizzando tale affidabilità a livello applicativo. Tali funzionalità sono state realizzate tramite un insieme di scambi di opportuni messaggi di richiesta e risposta tra client e il server.

## 2. Scelte progettuali

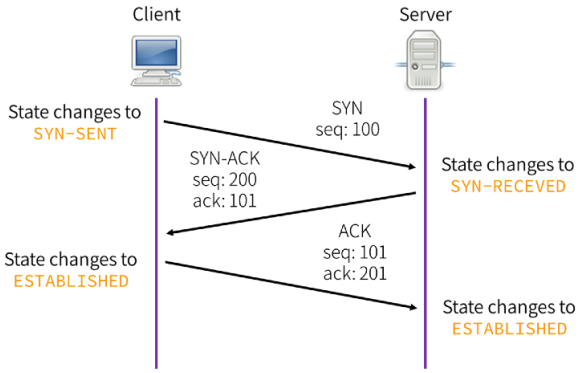
Per lo sviluppo delle varie funzionalità è stata realizzata un'architettura multi-thread. Lato server si è in grado di servire più client in concorrenza. Tale caratteristica è centrale e fondamentale nella realizzazione del progetto e di conseguenza nelle scelte ad essa legate. Nel server, è stato usato il main thread per ascoltare le eventuali richieste di connessione dei clients e le loro eventuali richieste dei comandi di list, download, upload. Il main thread del server, dopo aver ricevuto uno dei comandi, provvede a creare un apposito thread atto a portare a termine l'operazione richiesta dal client. Per la realizzazione della connessione tra client e server è stata implementata a livello applicativo una fase di handshake speculare a quella utilizzata in TCP per garantire un'effettiva instaurazione di una connessione tra le due parti. Analogamente, anche per la chiusura della connessione è stato usato un approccio simile a quello di TCP. Per garantire l'affidabilità nell'esecuzione dei comandi di list, download e upload, sono stati realizzati dei meccanismi per la gestione di un sistema di ack, timeout e numeri di sequenza associati ai pacchetti scambiati, con l'aggiunta di funzioni di bufferizzazione e ritrasmissione dei pacchetti.



# 3. Implementazione

## 3.1 - Handshake

Per quanto riguarda la fase di instaurazione della connessione tra client e server, è stato realizzato un meccanismo di handshake.




Come rappresentato nella figura, il client, che vuole contattare il server, manda inizialmente un pacchetto con il campo SYNbit con valore 1, ed un certo valore di sequenza *x* generato casualmente nel campo *sequence_number*. A tale pacchetto viene associato in fase di invio un timer che consente di ritrasmetterlo se il SYN-ACK non arrivasse prima che il timer associato al pacchetto di richiesta si estingua. Il server, una volta ricevuto il pacchetto contenente il SYNbit settato a 1, provvede a mandare il pacchetto di SYN-ACK, contenente i campi ACKbit e SYNbit messi a 1, il campo ACKnum con il valore di $x+1$, ed un numero di sequenza *y* anch'esso generato casualmente. Anche a questo pacchetto viene associato un timer per la gestione dell'eventuale perdita che verrà eliminato alla ricezione del messaggio di ack finale mandato dal client. Nel caso particolare in cui il server riceva un messaggio di richiesta connessione da parte del client, ma si perda la sua risposta, il server riceverà, causa scadenza del timer lato client, una nuova richiesta di connessione da parte dello stesso client che verrà ignorata, generando un reinvio del pacchetto di SYN-ACK. Una volta ricevuto il pacchetto di SYN-ACK il client passa nello stato di connessione stabilita e procede ad inviare il pacchetto di ACK, contenente il campo ACKbit settato a 1, ACKnum messo a $y + 1$, e procede a creare un timer di lunghezza superiore rispetto ad un timer "ordinario". Questa scelta permette di gestire la perdita dell'ultimo pacchetto inviato dal client. Infatti, non prevedendo tale messaggio un ack da parte del server, in caso di perdita, grazie al timer presente per il pacchetto di SYN-ACK mandatomi dal server, riceverò nuovamente tale pacchetto, in seguito al quale il client invierà nuovamente l'informazione contenente l'ack finale. A questo punto il server passa nello stato finale di connessione stabilita con il client che lo ha appena contattato. Il server, per tener traccia dei client ad esso connessi, allocherà una struttura dati atta a memorizzare le informazioni del client che lo ha contattato (indirizzo IP, porta) e crea per questo un'apposita socket che utilizzerà per contattarlo successivamente.

## 3.2 - List

Per quanto riguarda la funzionalità di list dei file disponibili presso il server, si è deciso di implementarla nel seguente modo: il client che vuole richiedere tale funzione procede a digitare il comando [1] che è associato a tale operazione. Tale valore viene messo all'interno del pacchetto in un campo appositamente creato per il codice dell'operazione richiesta (*operation_no*). All'invio di tale pacchetto viene creato un timer associato, che ha lo scopo di permettere la ritrasmissione del pacchetto di richiesta in caso di perdita, o nel caso in cui il server non risponda entro la scadenza di questo timer. Il server una volta ricevuto il pacchetto contenente il campo *operation_no* con all'interno il valore 1, procede a creare un thread figlio (thread *list_files*) che si occupa di trasmettere una serie di pacchetti verso il client contenenti nel campo *message* i nomi dei file disponibili presso la directory del server. Quest'ultimo thread si occupa inoltre di andare a creare un altro thread figlio (thread *ack_list_handler*), il cui scopo è gestire la corretta ricezione degli ack. Il thread *list_files* ha, inoltre, il compito di comunicare la porta verso la quale il client dovrà mandare gli ack, ovvero la porta sulla quale è in ascolto il thread *ack_list_handler*. Quest'ultimo gestisce la ricezione di ack in ordine e di tutte le possibili casistiche previste (es: ack duplicati, ack cumulativi). In caso di scadenza di uno dei timer, avviene la ritrasmissione del pacchetto associato al timer. Per effettuare tale ritrasmissione, è stata creata una lista collegata per salvare le informazioni relative ai pacchetti inviati ma non ancora riscontrati, cosicché sia possibile effettuare il retrieve dei dati da ritrasmettere. In caso di ricezione di un ack in ordine, il thread va ad eliminare il timer associato al pacchetto appena riscontrato, e la relativa istanza nella lista collegata dei pacchetti inviati, aumentando, quindi, anche lo spazio in finestra disponibile per l'invio. In caso di ricezione di tre ack duplicati, invece, verrà effettuato il reinvio del pacchetto con il più basso numero di sequenza presente in finestra e non ancora riscontrato. Infine, in caso di ricezione di un ack cumulativo si procede ad eliminare tutti i timer esistenti associati ai pacchetti aventi un numero di sequenza inferiore all' ack ricevuto. Il client, nel momento in cui riceve i pacchetti contenenti i nomi dei file disponibili, procede all'invio degli ack per ciascun pacchetto ricevuto verso la porta sulla quale è in ascolto il thread *ack_list_handler*. Nel caso in cui il client riceva dei pacchetti fuori sequenza, provvederà alla bufferizzazione degli stessi, inviando un ack duplicato al server. I pacchetti bufferizzati verranno utilizzati nel momento in cui si riceverà il pacchetto mancante per la corretta ricezione in ordine. Una volta utilizzati, in seguito alla corretta ricezione e conseguente debufferizzazione, vengono liberate le strutture dati allocate. Infine, lato client, nel momento in cui viene ricevuto l'ultimo pacchetto e quindi inviato l'ultimo ack viene creato un timer di lunghezza superiore ad un timer "ordinario", permettendo quindi di essere in ascolto nel caso in cui il server, non avendo ricevuto il nostro ultimo ack, abbia reinviato del contenuto verso il client.

## 3.3 – Download

Per quanto riguarda la funzionalità di download, si è deciso di implementare tale operazione nel seguente modo: il client che vuole richiedere al server tale funzione procederà ad inserire il codice [2] che verrà messo all'interno del campo apposito *operation_no* del pacchetto da inviare al server come pacchetto di richiesta. La funzionalità di download è stata realizzata dividendola in due parti; la prima parte che permette al client di venire a conoscenza dei file disponibili presso il server (implementata in modo analogo alla funzionalità di list), ed una seconda in cui il client, una volta visualizzati i file scaricabili, può inserire il nome del file desiderato. Nella prima parte dell'implementazione, il server, quando riceve un pacchetto contenente nel campo *operation_no* il valore 2, procede a creare il thread *download*, che si occuperà dell'interazione con il client. Il thread *download*, dunque, inzialmente, crea il thread list_files, per fare in modo che client riceva la lista dei file disponibili presso il server, e richiedere ciò che desidera. Quindi, una volta che il client ha visualizzato tutti i file disponibili, può procedere ad inserire il nome del file da scaricare. Tale file richiesto, viene inserito nel campo *message* del pacchetto da inviare, a cui viene associato un timer, il cui scopo è quello di permettere il reinvio di quest'ultimo in caso di perdita dell'ack o del pacchetto stesso. Il server, quando riceve dal client un pacchetto contenente il nome di uno dei file disponibili presso di esso, dopo aver creato il thread *ack_download_handler* per la gestione degli ack, procede nel caso in cui il file sia effettivamente presente, ad inviare inizialmente il nome stesso del file richiesto in un apposito pacchetto, cosicché il client possa andare a creare il file e lo stream per la scrittura su di esso. Il server, inoltre, comunica in tale pacchetto anche la porta verso la quale il client dovrà mandare gli ack relativi al contenuto del file richiesto, ovvero la porta sulla quale è in ascolto il thread *ack_download_handler*. Nel caso in cui il file non sia presente lato server, il thread *download* manda un pacchetto contenente un opportuno messaggio di errore che permette al client di capire che il file richiesto non è corretto. Il thread *download* comunica, inoltre, la dimensione del file richiesto dal client. Per la gestione della ricezione degli ack, bufferizzazione dei pacchetti inviati e per la relativa eventuale ritrasmissione sono state utilizzate le medesime modalità previste per la lista. Il client, nel momento in cui riceve i pacchetti aventi il contenuto del file richiesto in ordine, procede a fare la scrittura sul file. Se il client dovesse ricevere dei pacchetti fuori sequenza, procederà a bufferizzarli in apposite strutture dati, e verranno effettivamente utilizzati nel momento in cui arriveranno i pacchetti che ne permetteranno la scritture sul file in maniera ordinata. I controlli per la ricezione dei pacchetti in modo ordinato vengono fatti controllando i numeri di sequenza. Lato client, nel momento dell'invio dell'ultimo ack, viene creato un timer di lunghezza superiore a quella di un timer "ordinario", permettendo quindi di essere in ascolto nel caso in cui il server, non avendo ricevuto il nostro ultimo ack, abbia reinviato del contenuto verso il client.

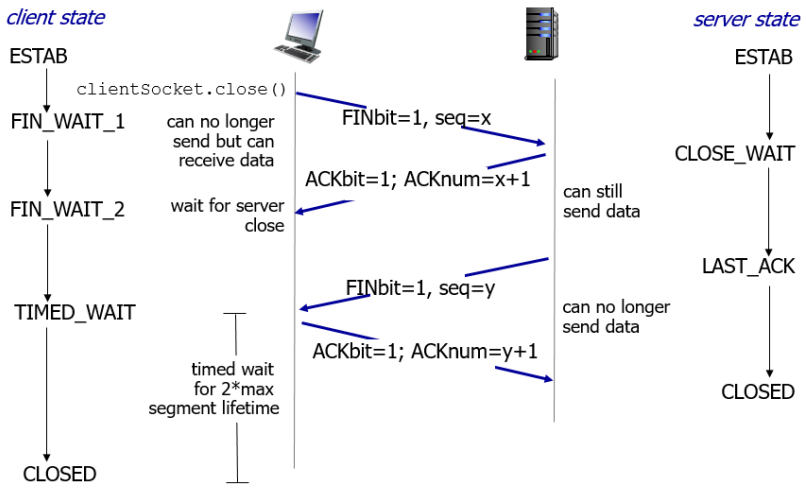
## 3.4 – Upload

Per quanto riguarda la funzionalità di upload, si è deciso di implementare tale operazione nel seguente modo: il client che vuole richiedere un upload al server provvede a digitare il codice [3] che viene messo nell'apposito campo *operation_no* del pacchetto di richiesta da mandare al server. A tale pacchetto viene, inoltre, associato un timer, il cui scopo è quello di permettere la ritrasmissione del pacchetto in caso di perdita o di mancata risposta del server entro un certo periodo di tempo. Il server che riceve un pacchetto contenente una richiesta di upload, procede a creare un thread figlio (thread *upload*), il cui scopo è quello di gestire la comunicazione con il client da quel momento in poi. Tale thread upload, si occuperà di inviare al client un pacchetto all'interno del quale comunica che è pronto a ricevere il contenuto del file da caricare, ed inoltre gli comunica la porta sulla quale è in ascolto per la ricezione. Il client, una volta ricevuto l'ok del server per iniziare l'operazione di upload, crea un thread figlio (thread *ack_upload_handler*), il cui scopo è la gestione della corretta ricezione degli ack relativi ai contenuti inviati al server. Il client, innanzitutto, comunicherà al server il nome del file da caricare e la sua dimensione, cosicché questo possa creare il file ed aprile lo stream, ed inoltre, il client comunicherà la porta sulla quale il server dovrà mandare gli ack (la porta dove è in ascolto il thread *ack_upload_handler*). Dopodiché, il client procederà a mandare il contenuto del file. Il server, quando riceverà i pacchetti in ordine con il contenuto del file da caricare, procederà a scriverlo sul file. I pacchetti ricevuti fuori sequenza verranno bufferizzati per essere effettivamente scritti sul file nel momento della ricezione del pacchetto che ne permetterà la scrittura ordinata. Per ciascun pacchetto ricevuto avente il contenuto da scrivere, il thread *upload* del server, procederà a mandare il relativo ack al thread *ack_uplaod_handler* del client. Le varie casistiche di ack duplicati, ack cumulativi e ritrasmissione sono analoghe a quelle previste per la lista e download. Infine, lato server, il thread *upload* dopo aver inviato l'ultimo ack procede a creare un timer di lunghezza superiore a quella di un timer "ordinario", permettendo quindi di essere in ascolto nel caso in cui il client, non avendo ricevuto il nostro ultimo ack, abbia reinviato del contenuto verso il server.

## 3.5 – Chiusura connessione

Per quanto riguarda la chiusura della connessione si è deciso di implementare tale meccanismo prevendo la possibile terminazione dell'interazione sia da parte del client che da parte del server. Nel caso in cui sia il client a voler interrompere la comunicazione con il server, questo può far ciò sia digitando il comando [4] del menù, sia generando un segnale di SIGINT. In entrambi i casi ciò che accade è che viene mandato al server un pacchetto contenente il campo FINbit settato a 1, ed il campo *sequence_number* popolato con un valore *x* generato casualmente. A tale pacchetto viene associato un timer, che permette la gestione della perdita di questo e della mancata risposta da parte del server entro un determinato periodo di tempo. Il client, dunque, passa nello stato di FIN_WAIT_1. Una volta che il server riceve il pacchetto contenente la richiesta di un client di interrompere la connessione, questo provvede a mandare un pacchetto con i campi ACKbit messo a 1, ACKnum messo al valore di *x* +1, e farà partire un timer di lunghezza superiore a quella di un timer "ordinario" per fare in modo di gestire il caso in cui il client rimandi qualche pacchetto in seguito alla perdita dell'ultimo pacchetto inviato dal server. Il server a questo punto passa nello stato di CLOSE_WAIT, mentre il client nel caso di ricezione di tale pacchetto, passa nello stato di FIN_WAIT_2. Lato server, una volta scaduto tale timer, ovvero dopo aver avuto con alta probabilità la conferma che il client abbia ricevuto il pacchetto di risposta del server, quest'ultimo si occupa di inviare il pacchetto finale con i campi FINbit messo a 1, e il campo *sequence_number* popolato con un valore *y* generato casualmente. A tale pacchetto, anche in questo caso, viene associato un timer il cui scopo è permettere la gestione della perdita di tale pacchetto o la mancata risposta del client entro un determinato periodo di tempo. A questo punto il server è passato nello stato di LAST_ACK in attesa di ricevere l'ultimissimo ack da parte del client. Infine, il client, quando riceve il pacchetto del server con FINbit a 1, passa nello stato di TIMED_WAIT, in cui provvede a mandare il pacchetto finale con il campo ACKbit settato a 1, ACKnum con il valore *y*+1, associando a tale pacchetto un timer di lunghezza superiore a quella di un timer "ordinario" per fare in modo di gestire il caso in cui il server rimandi qualche pacchetto in seguito alla perdita dell'ultimo pacchetto inviato dal client. Il client, una volta scaduto l'ultimo timer creato, può effettivamente considerare chiusa la connessione con il server che, una volta ricevuto l'ultimo ack del client, può eliminare il client dalle strutture dati ad esso dedicate.

Invece, nel caso in cui fosse il server a voler interrompere la comunicazione con i client connessi, questo può farlo esclusivamente generando un segnale di SIGINT. Il meccanismo di gestione della chiusura è analogo a quello previsto nel caso in cui fosse uno dei client a voler interrompere la comunicazione, con la differenza che il server dovrà mandare n pacchetti di chiusura, tanti quanti sono i client che sta gestendo. La gestione della chiusura è effettuata con la creazione di un thread per ogni client da dover eliminare, e seguirà gli stessi identici meccanismi previsti nel caso della chiusura del client a parti invertite. Il server potrà effettivamente chiudere l'interazione solo quando tutti i client gli avranno mandato i relativi ultimi ack.



# 4. Appendice

## 4.1 – Timeout

Per quanto riguarda la gestione dei timeout associati ai pacchetti inviati, sono state utilizzate delle syscall appartenenti alla libreria “time.h”. I timer usati vengono generati dalla funzione *timer_create*() che restituisce l’id univoco del timer. I timer possono essere di due tipi: ordinari e di chiusura. I timer di chiusura vengono usati sono nel caso in cui sono necessarie delle attese più lunghe per i motivi precedentemente elencati nelle funzionalità sopra citate. I timer usati nel sistema sono di tipo *timer_t* e vengono salvati in apposite strutture dati che saranno necessarie per permettere di recuperare le informazioni dei pacchetti da ritrasmettere.

La struttura dati usata per memorizzare i timer e i pacchetti ad essi associati è la seguente:

```
typedef struct timer{
  timer_t  idTimer;
  char SYNbit[2];
  char sequence_number[10];
  char ACKbit[2];
  char ack_number[10];
  char message[MAXLINE];
  char operation_no[2];
  char FINbit[2];
  int  sockfd;
  char port[6];
  char size[15];
#ifdef ADPTO
  float sample_value;
  float last_rto;
#endif
  struct timer *next;
  struct timer *prev;
}t_tmr;
```

La creazione del timer è così strutturata: prima della creazione effettiva con la funzione timer_create(), la quale restituisce un timer_id univoco per ogni timer di tipo timer_t, si invoca la funzione append_id_timer() che ha lo scopo di memorizzare, nella lista degli id, la struttura di memoria alla quale punterà, in caso di eventuale perdita, l’indirizzo *sival_ptr* presente nella struttura dati *siginfo_t* utilizzata per effettuare l’identificazione del timer che è scaduto. Più nel dettaglio, un timer, quando scade,

genera un segnale di tipo SIGRTMIN. La gestione di tale segnale prevede la seguente segnatura:

*sigwaitinfo(&set, &info)*

Il parametro *info* punta alla struct precedentemente impostata in fase di creazione del timer contenente le informazioni del pacchetto relativo al timer appena scattato. Essendo possibile che accadano eventi in parallelo, la lista collegata dei timer esistenti, prima di essere manipolata, necessità della gestione di una sincronizzazione, relativa al singolo client.

## 4.2 - Ritrasmissione

Per quanto riguarda la gestione della ritrasmissione dei pacchetti, come precedentemente detto, è stata gestita attraverso la manipolazione del segnale SIGRTMIN. Allo scattare di un timer viene fatto il retrieve dell'id del timer scaduto cosicché sia possibile individuare le informazioni da ritrasmettere. Tali informazioni erano state precedentemente salvate in una lista collegata al momento dell'invio del pacchetto. Tuttavia, nel caso in cui si ricevessero almeno tre ack duplicati, viene innescato il meccanismo di ritrasmissione selettiva, ovvero si ritrasmette esclusivamente il pacchetto con il numero di sequenza più basso all'interno della finestra di trasmissione. Per spiegare l'implementazione realizzata per la ritrasmissione è necessario innanzitutto premettere che in ambiente linux i segnali vengono gestiti per processo. Il comportamento di un thread appartenente ad un determinato processo, alla ricezione di un segnale, viene definito nello standard POSIX come indefinito, ovvero non è possibile stabilire a priori il cambiamento di schedulazione dei thread una volta terminata la gestione del segnale, che se effettuata con una semplice funzione può risultare bloccante per l'intero processo in determinate casistiche. Per ovviare a tali limitazioni, tutti i thread sono insensibili al segnale SIGRTMIN ad eccezione di un unico thread che risulterà, quindi, l'unico a ricevere tale segnale, attraverso la funzione *sigwaitinfo()*, che è in grado di gestire l'arrivo di segnali dovuti allo scadere dei timer, e la successiva gestione del retrieve delle informazioni attraverso la struct siginfo_t opportunamente impostata. Ogni client ha un thread esclusivo dedicato alla ritrasmissione dei suoi pacchetti. Il thread gestore del segnale SIGRTMIN ha il compito di segnalare ai thread di ritrasmissione dei vari client i pacchetti da ritrasmettere, permettendo così una corretta parallelizzazione della gestione e ricezione dei segnali dovuti alla scadenza dei timer. La segnalazione da parte del thread atto a ricevere il segnale SIGRTMIN verso il thread incaricato della ritrasmissione avviene tramite l'uso della syscall *pthread_cond_signal()* che permette di sbloccare il thread di ritrasmissione che si trova in uno stato di attesa di verifica della condizione (*pthread_cond_wait()*).

## 4.3 - Finestra

La gestione della finestra di trasmissione è stata realizzata tramite l'uso di semafori. In particolare si è deciso di creare un semaforo con N gettoni, dove N è il valore associato alla dimensione della finestra. Tale valore viene decrementato ogni volta che viene fatto un invio di un pacchetto, per essere poi incrementato di una singola unità per ogni ack che riscontra un pacchetto inviato, oppure di x unità nel caso di ricezione di un ack cumulativo che riscontri x pacchetti precedentemente inviati. La struttura dati contenente le informazioni di ciascun pacchetto inviato è la seguente:

```
typedef struct data{
    char SYNbit[2];
    char sequence_number[10];
    char ACKbit[2];
    char ack_number[10];
    char message[MAXLINE];
    char operation_no[2];
    char FINbit[2];
    char port[6];
    char size[15];
    char port_download[6];
}t_data;
```

## 4.4 – Bufferizzazione

La gestione dei pacchetti ricevuti fuori sequenza è stata realizzato implementando un meccanismo di bufferizzazione dei pacchetti. Ogni volta, infatti, che si riceve un pacchetto fuori sequenza viene salvato in un'apposita struttura dati, in modo tale da poter effettuare il corretto utilizzo (debufferizzazione) nel momento in cui si riceverà il pacchetto/i mancante/i, che erano precedentemente andati persi.

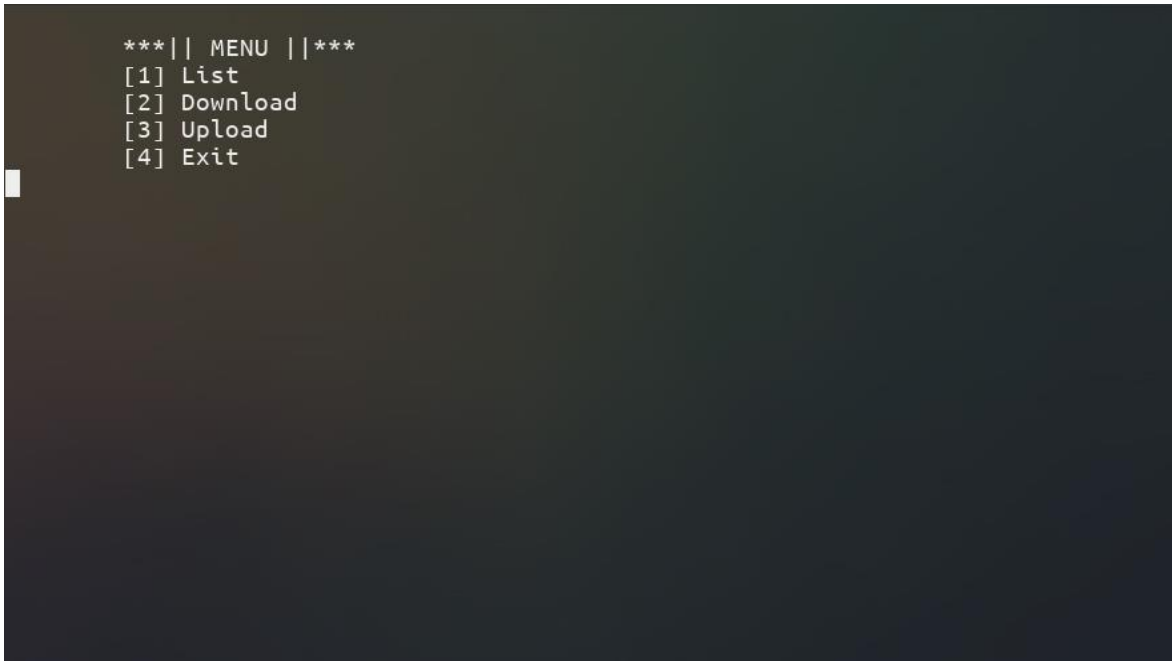
## 4.5 – Gestione dello ‘stato’ del client nel server

Lato server, per poter gestire più client in parallelo, si è deciso, oltre che di salvare le informazioni relative ad indirizzo IP, porta e socket del client, di associare a ciascun client delle apposite variabili di ‘stato’. Ogni client ha informazioni proprie relative a numeri di sequenza ed ack, oltre ad avere propri descrittori per i semafori utilizzati ed avere proprie liste riguardanti timer, pacchetti bufferizzati e pacchetti inviati, in quanto essendo il software strutturato in ambiente multi-threaded è necessario avere informazioni di ‘stato’ specifiche per ogni client, affinché vi possano essere tangibili vantaggi prestazionali. Nei seguenti screenshot è riportata la struttura dati usata per gestire ogni singolo client.

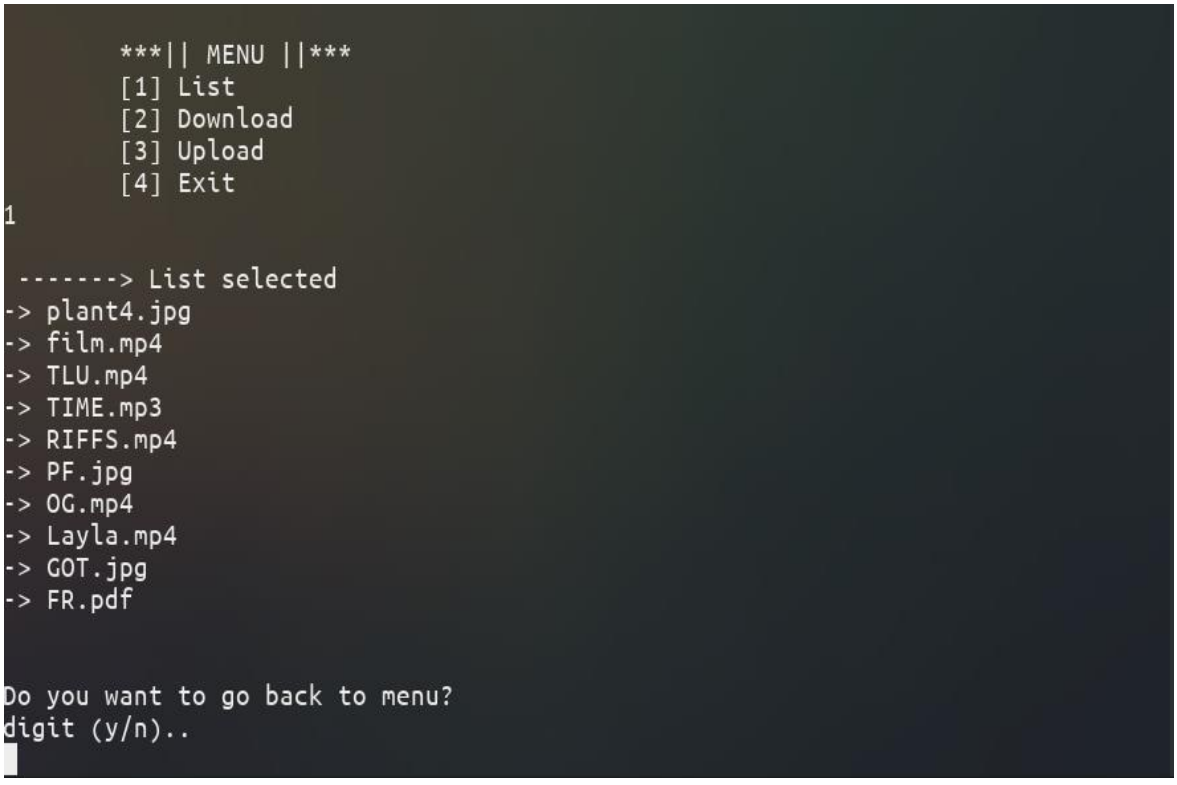
```
typedef struct client_info{
    int             sockfd;
    unsigned short  port;
    int             rand_c;
    char            ipaddr[20];
    int             expected_next_seq_number;
    int             sequence_number;
    int             ack_number;
    int             last_ack;
    int             download_list_phase;
    int             port_download;
    int             sock_filename_download;
    int             sock_list;
    int             sock_download;
    int             retr_phase;
    int             timerid;
    int             operation_value;
    int             semid_shared_queue;
    int             ack_received;
    int             semid_client;
    int             semid_expected;
    int             semid_retrasmission;
    int             semid_retr_turn;
    int             semid_timer;
    int             semid_sum_ack;
    int             semid_upclosing;
    int             semid_window;
    pthread_mutex_t lock;
    pthread_mutex_t lock2;
    pthread_cond_t  condition;
    int             thread_retr_fin;
    int             thread_retr_ord;
    timer_t         shared_tim;
    int             receive_window;
    int             sum_ack;
    int             semid_fileno;
    int             first_pkt_sent;
    int             master_timer;
    timer_t         master_IDTimer;
    int             master_exists_flag;
    int             receiving_ack_phase;
    int             receiving_ack_phase_download;
    int             num_pkt_buff;
    int             flag_last_ack_sent;
    int             flag_upload_wait;
    int             closing_ack;
    int             file_content;
    int             upload_closing;
    t_buffered_pkt  *buff_pkt_client;
    t_tmr           *timer_list_client;
    t_timid         *timer_id_list_client;
    t_sent_pkt      *list_sent_pkt;
    pthread_t       upload_tid;
    int             var_req;
    int             hs;
    long int        file_size;
    int             rand_hs;
    int             flag_close;
    int             second_flag_close;
    int             semid_last_timer;
    int             flag_server_close;
    int             estab_s;
    int             filename_bytes;
    int             ok_phase;
    int             create_file;
    pthread_t       retrasmission_thread;
    t_timid         shared[MAXDIM];
    int             scount;
#ifdef ADPTO
    float           rto;
    float           rto_sec;
    float           rto_nsec;
    float           sample_rtt;
    float           estimated_rtt;
    float           dev;
    int             first_to;
#endif
    struct          client_info *next;
    struct          client_info *prev;
}t_client;
```

## 5. Esempi di funzionamento

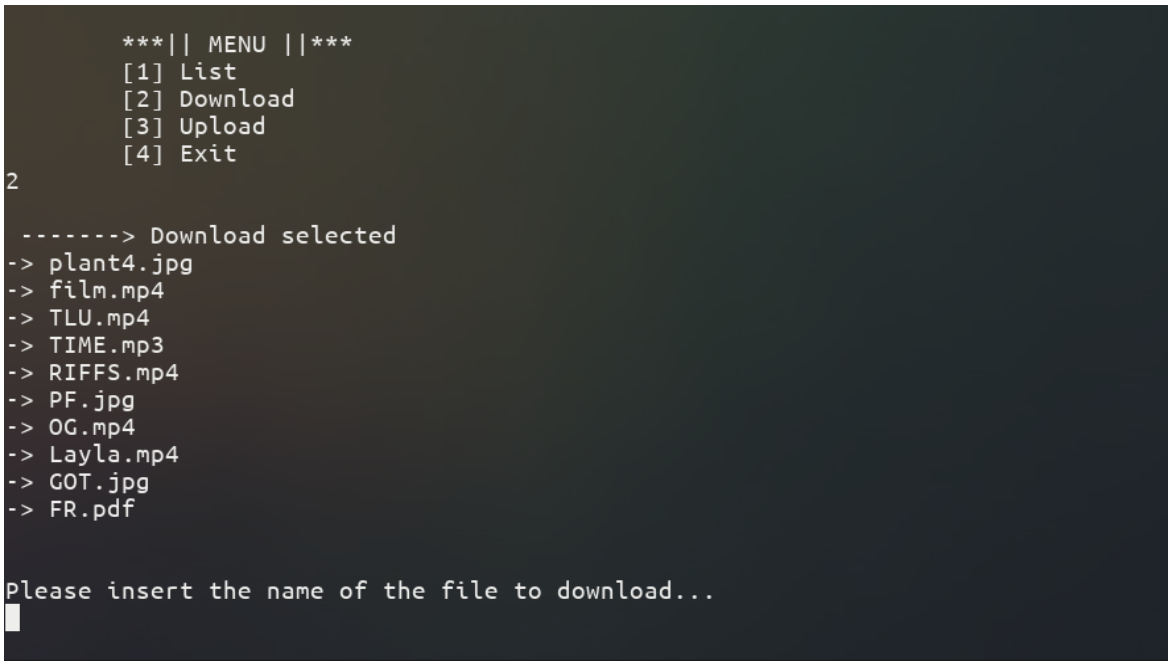
Il client, dopo essere riuscito ad instaurare una connessione con il server, ha il permesso di accedere al menu dell'applicazione in cui sono disponibili le funzionalità di list, download e upload.



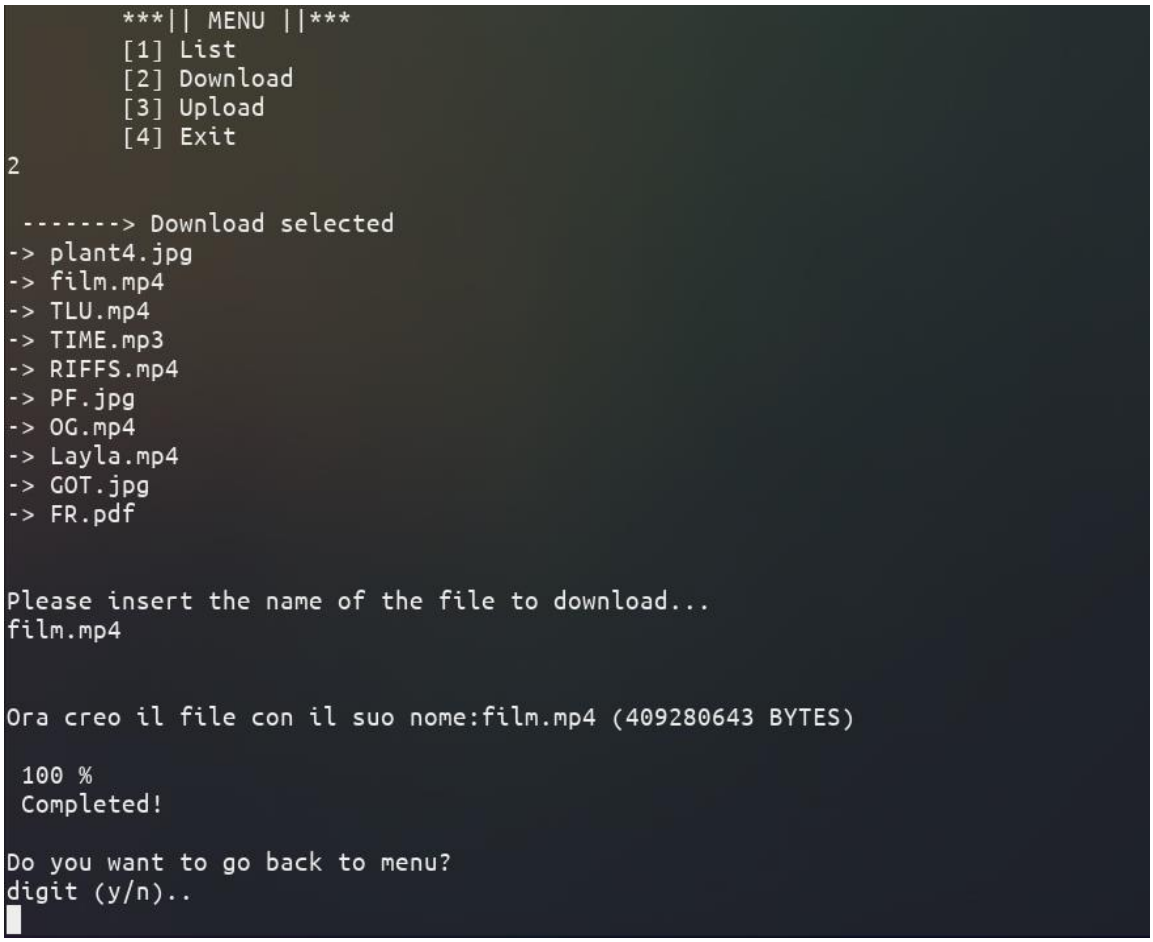

LIST - Il client, selezionando l'opzione [1], può richiedere il comando list, con il quale può ricevere dal server l'insieme dei file disponibili su quest'ultimo.



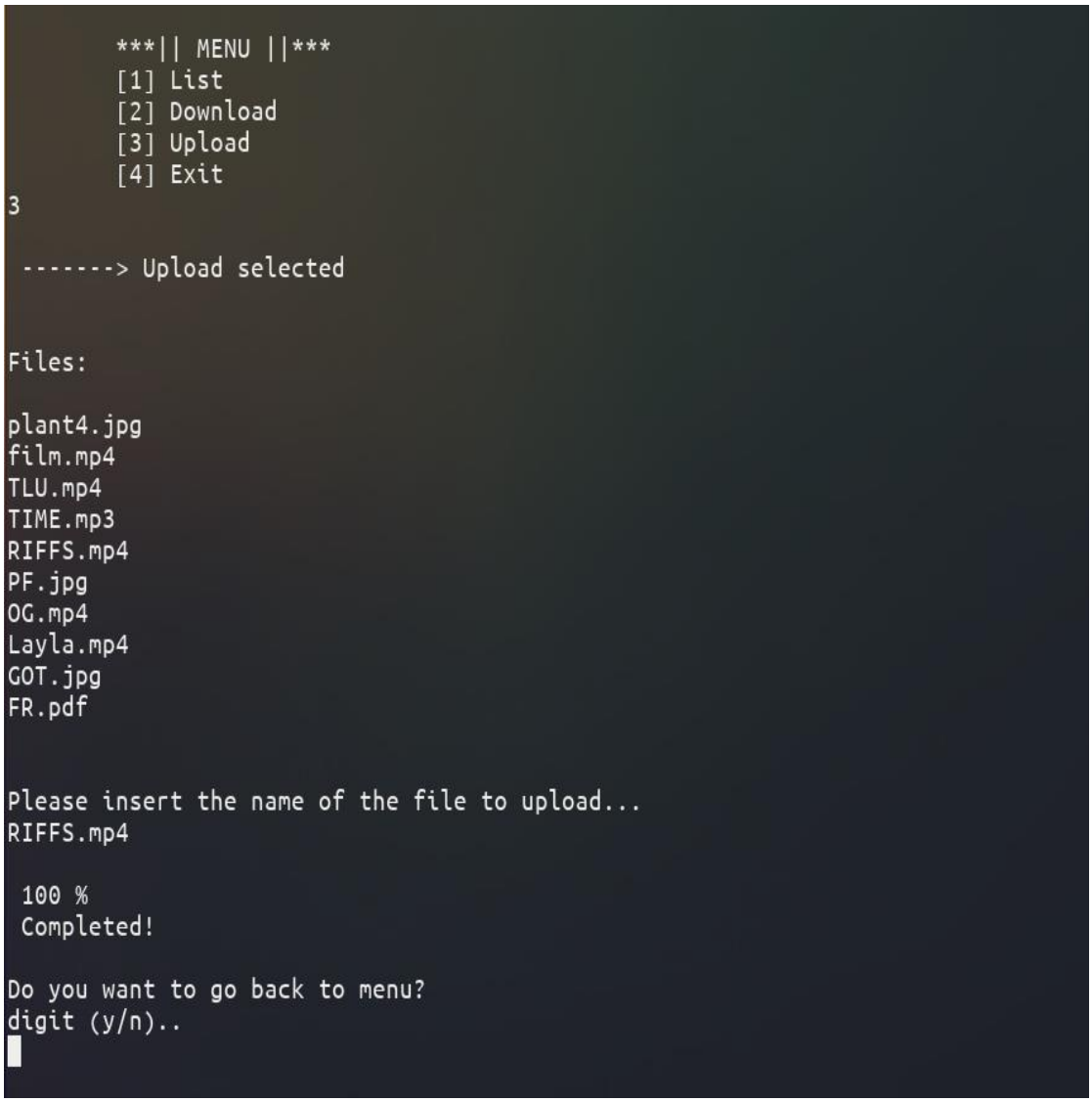
DOWNLOAD PART 1 – Il client, selezionando l'opzione [2], può richiedere il comando get, tramite il quale, inizialmente, il server provvede a fornire al client l'insieme dei files che possono essere richiesti dal client per il download.



DOWNLOAD PART 2 – Il client, una volta ottenuta la lista dei file che può richiedere, può procedere a richiedere uno dei file mostrati, e quindi avviare il download di quest'ultimo.



UPLOAD – Il client, selezionando l'opzione [3], può richiedere il comando put, con il quale può andare a caricare sul server uno dei files disponibili presso la sua directory.



## 6. Prestazioni

| DOWNLOAD FILE 16.4 MB | | | |
|---|---|---|---|
| **to_value (seconds)** | **loss_p** | **window** | **Time (seconds)** |
| 0.2 sec | 5% | 10 | 7.3 sec |
| 0.2 sec | 5% | 15 | 5.2 sec |
| 0.2 sec | 5% | 20 | 4.6 sec |
| 0.2 sec | 5% | 50 | 11.3 sec |
| 0.2 sec | 5% | 80 | 28.4 sec |

| UPLOAD FILE 16.4 MB | | | |
|---|---|---|---|
| **to_value (seconds)** | **loss_p** | **window** | **Time (seconds)** |
| 0.2 sec | 5% | 10 | 12.1 sec |
| 0.2 sec | 5% | 15 | 9.5 sec |
| 0.2 sec | 5% | 20 | 3.6 sec |
| 0.2 sec | 5% | 50 | 21.2 sec |
| 0.2 sec | 5% | 80 | 43.8 sec |

Analizzando i risultati sperimentali ottenuti si può osservare, dalle prime due tabelle, come le prestazioni, a parità di lunghezza dei timer e di perdita, tendano a migliorare fino ad un certo valore della finestra oltre il quale tendono a peggiorare.

| DOWNLOAD FILE 16.4 MB | | | |
|---|---|---|---|
| **to_value (seconds)** | **loss_p** | **window** | **Time (seconds)** |
| 0.2 sec | 2% | 10 | 3.5 sec |
| 0.2 sec | 5% | 10 | 7.2 sec |
| 0.2 sec | 7% | 10 | 16.6 sec |

| UPLOAD FILE 16.4 MB | | | |
|---|---|---|---|
| **to_value (seconds)** | **loss_p** | **window** | **Time (seconds)** |
| 0.2 sec | 2% | 10 | 7.4 sec |
| 0.2 sec | 5% | 10 | 12.1 sec |
| 0.2 sec | 7% | 10 | 28.7 sec |

Invece, mantenendo fissi il valore del timeout e la finestra, si può osservare, nelle seconde due tabelle, come all'aumentare della perdita si può verificare un leggero calo delle prestazioni dovuto alla necessità di dover ritrasmettere il potenziale maggior numero di pacchetti persi.

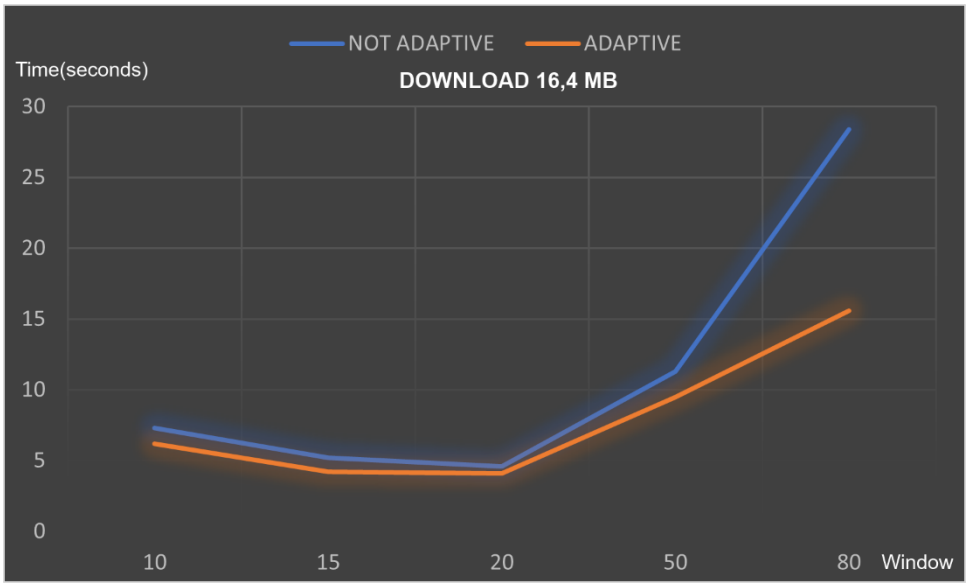
| DOWNLOAD FILE 16.4 MB | | | |
|---|---|---|---|
| **to_value (seconds)** | **loss_p** | **window** | **Time (seconds)** |
| adaptive | 5% | 10 | 6.2 sec |
| adaptive | 5% | 15 | 4.2 sec |
| adaptive | 5% | 20 | 4.1 sec |
| adaptive | 5% | 50 | 9.5 sec |
| adaptive | 5% | 80 | 15.6 sec |

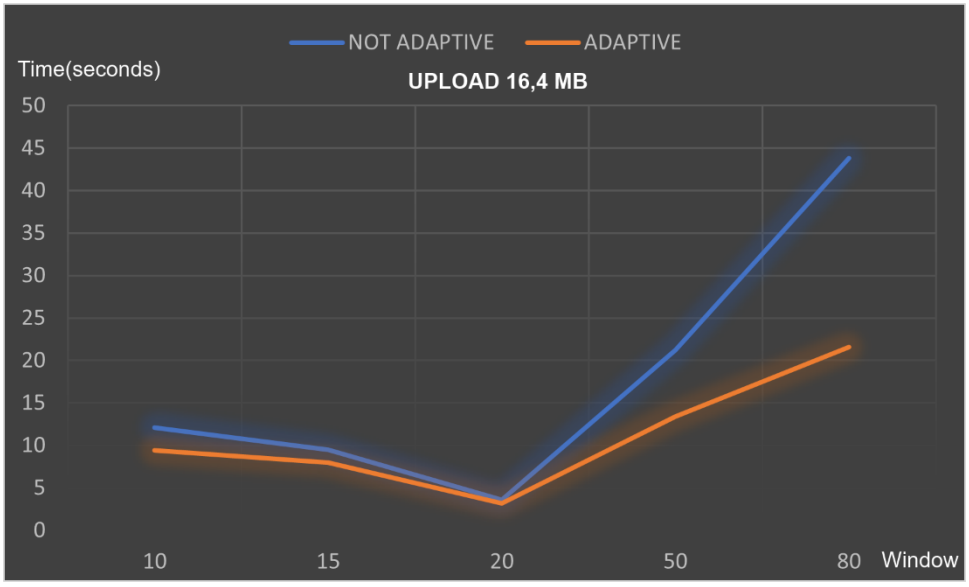
| UPLOAD FILE 16.4 MB | | | |
|---|---|---|---|
| **to_value (seconds)** | **loss_p** | **window** | **Time (seconds)** |
| adaptive | 5% | 10 | 9,4 |
| adaptive | 5% | 15 | 8 |
| adaptive | 5% | 20 | 3,2 |
| adaptive | 5% | 50 | 13,4 |
| adaptive | 5% | 80 | 21,6 |

Infine, nelle ultime due tabelle, si può osservare come usando dei timer adattivi, mantenendo fissi i valori di percentuale di perdita e finestra come nelle prime due tabelle, si ha un miglioramento nelle prestazioni di download e upload.

Analizzando i risultati sperimentali ottenuti si può osservare come le prestazioni, a parità di lunghezza dei timer e di perdita, tendano a migliorare fino ad un certo valore della finestra oltre il quale tendono a peggiorare, mostrando quindi un comportamento tipico dei protocolli a finestra (curva blu).

Usando dei timer adattivi (curva arancione), invece, si ha un miglioramento nelle prestazioni di download e upload dovuto al fatto che il programma si adatta meglio alle varie situazioni che possono verificarsi.



## 7. Manuale

Il programma è stato sviluppato per sistemi Unix-like. Per utilizzare l'applicazione è semplicemente necessario spostarsi nella directory in cui sono presenti i files server.c e client.c e le cartelle FILES_SERVER (in cui vi sono i file che si possono scaricare dal server o quelli caricati dai client) e FILES_CLIENT (in cui vi sono i files del client e quelli scaricati dal server), e a questo punto digitare il comando make per generare gli eseguibili del server e del client.
Per lanciare il server non è necessario specificare argomenti, mentre per il client è necessario specificare l'indirizzo IP del server da contattare.

I parametri per modificare la dimensione della finestra, del timeout, della perdita sono modificabili specificandoli nelle apposite macro all'interno dei sorgenti server.c e client.c.

Infine, di default il Makefile permette di genere degli eseguibili del server e client che operano senza timer adattivi. Per eseguire l'applicazione usando dei timer adattivi è necessario specificare all'interno del Makefile il flag di compilazione -DADPTO.

```
all:
    gcc -g server.c -Wall -Wextra -o server -lpthread -lrt
    gcc -g client.c -Wall -Wextra -o client -lpthread -lrt
clean:
    -rm server
    -rm client
```

```
all:
    gcc -g server.c -Wall -Wextra -o server -lpthread -lrt -DADPTO
    gcc -g client.c -Wall -Wextra -o client -lpthread -lrt -DADPTO
clean:
    -rm server
    -rm client
```