

Relazione progetto B1

Trasferimento file su UDP

Simone Tiberi (M. 0252795)

INGEGNERIA INTERNET & WEB

INDICE

Architettura del sistema e delle scelte progettuali.....	1
Struttura dei messaggi inviati	1
Gestione dell'instaurazione della connessione.....	2
Gestione della fase di chiusura.....	3
Gestione del timer adattativo	3
Implementazione	4
Implementazione delle funzionalità del protocollo	4
Implementazione delle funzionalità comuni a client e server	5
Implementazione del client e del server	5
Operazioni di LIST e GET	6
Operazione di PUT.....	7
Limitazioni riscontrate	7
Specifiche della piattaforma SW utilizzata	8
Esempi di funzionamento	8
Valutazione delle prestazioni	10
Analisi in funzione della dimensione della finestra.....	10
Analisi in funzione dell'entità del timeout	11
Analisi in funzione della probabilità d'invio	11
Manuale per l'installazione, configurazione ed esecuzione	12

Architettura del sistema e delle scelte progettuali

Lo scopo del progetto è quello di realizzare in linguaggio C un'applicazione client-server per il trasferimento affidabile di file che impieghi UDP come protocollo di trasporto.

A tale scopo è stato sviluppato un vero e proprio protocollo applicativo al fine di standardizzare la comunicazione fra sender e receiver relativamente a:

- struttura dei messaggi inviati
- gestione dell'instaurazione e della chiusura della connessione
- gestione del timer adattativo

Struttura dei messaggi inviati

Ciascun messaggio scambiato fra client e server all'interno dell'applicazione è costituito da un header di 4 byte e da un payload di dimensione variabile (da 0 a 512 B).

L'intestazione è così strutturata:

SEQUENCE/ACKNOWLEDGEMENT NUMBER	E	A	L	TYPE
---------------------------------	---	---	---	------

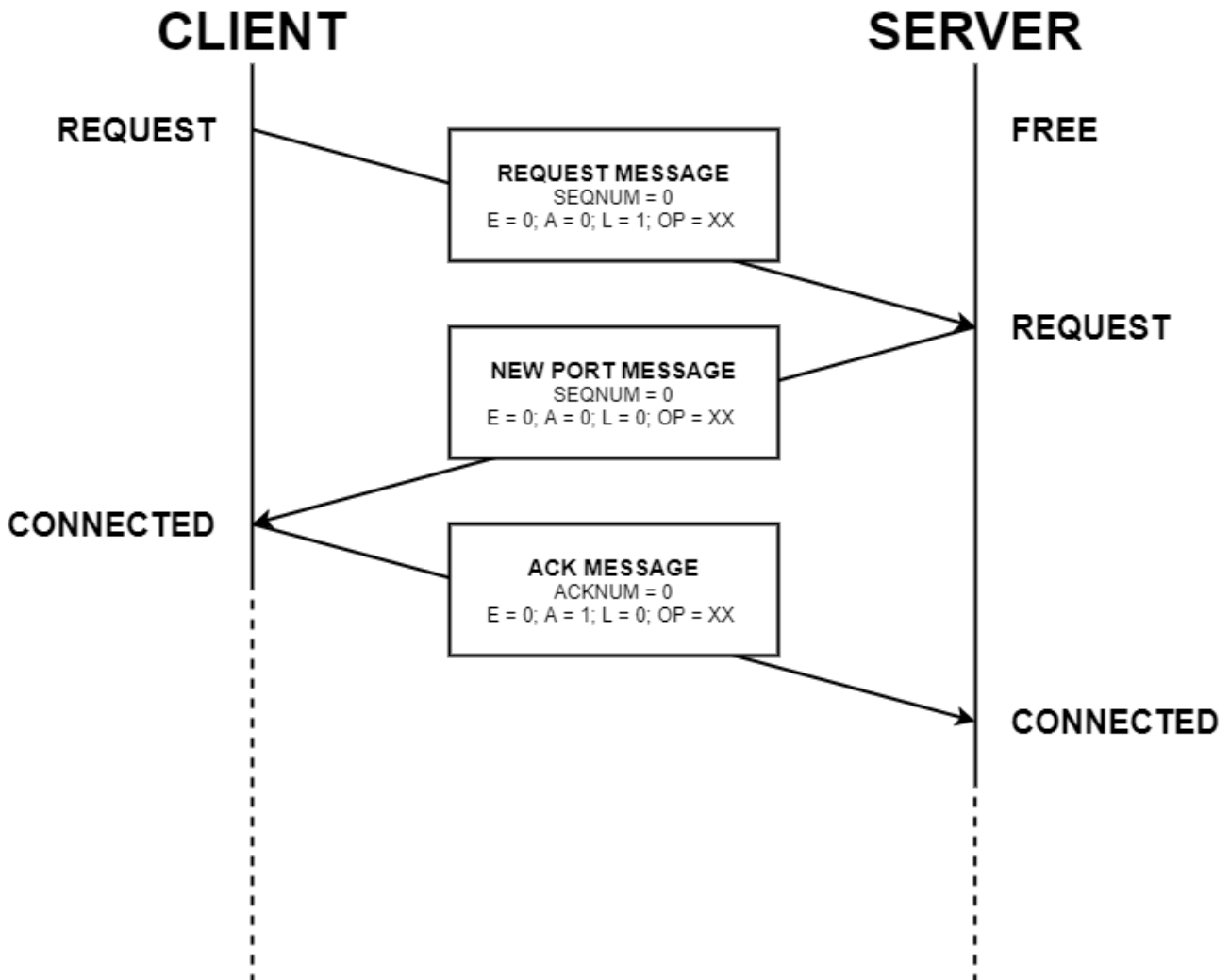
In cui:

Il campo SEQ/ACK number (27 byte) viene utilizzato:

- dal sender per memorizzare il numero di sequenza associato al chunk trasportato.
 - dal receiver per memorizzare il numero di sequenza da riscontrare al sender.
- Il bit di flag E (Error) viene utilizzato nella fase iniziale da parte del server per segnalare un eventuale errore al client:
 - nel caso di richiesta GET viene settato ad 1 se il file richiesto non esiste.
 - nel caso di richiesta PUT viene settato ad 1 se esiste già nel server un file avente lo stesso nome di quello che il client vorrebbe caricare.
- Il bit di flag A (ACKnowledgement) viene utilizzato dal receiver per marcare i propri pacchetti come riscontri.
- Il bit di flag L (Last) viene utilizzato sia dal sender che dal receiver per marcare i propri ultimi pacchetti inviati.
- I 2 bit di flag TYPE vengono utilizzati per discriminare l'operazione a cui sono associati i messaggi scambiati. Le possibili configurazioni sono così gestite:
 - 01 LIST
 - 10 PUT
 - 11 GET
 - a cui si aggiunge la configurazione 00 ZERO utilizzata come reset

Gestione dell'instaurazione della connessione

Il protocollo applicativo sviluppato utilizza un meccanismo di connessione di tipo 3-way handshake la cui struttura è di seguito riportata:



Il client avvia la comunicazione inviando un messaggio di REQUEST al server settando i parametri così come indicato in figura. Quest'ultimo può essere dotato o meno di payload, in particolare se la richiesta è associata ad un'operazione di LIST esso ne sarà sprovvisto, altrimenti conterrà il nome del file da inviare o ricevere.

Il server, una volta ricevuto il messaggio verifica:

- nel caso di una GET se il file richiesto realmente esiste
- nel caso di una PUT se il nome scelto per l'upload non è già presente nella directory public del server.

In caso di esito negativo del test invia al client un messaggio d'errore (flag E = 1) sulla socket d'accettazione altrimenti apre, se possibile, una nuova connessione su una porta differente da quella di accettazione ed invia un messaggio di NEWPORT.

Tale è costituito da un header strutturato come in figura e da un payload contenente la configurazione dei parametri del protocollo utilizzati dal server, di modo che dopo aver ricevuto tale messaggio client e server utilizzino gli stessi parametri così come richiesto dalla specifica.

La fase di connessione termina con l'inoltro di un segmento di ACK da parte del client per notificare l'avvenuta connessione al server.

Per quanto riguarda le azioni di recovery dovute alla perdita di messaggi nella fase di connessione il protocollo opera nel seguente modo:

1. Nel caso di mancata ricezione del messaggio di NEWPORT il client, una volta scaduto il timeout, ritenta l'invio del messaggio di REQUEST aumentando in modo lineare l'RTO fino ad un tetto massimo superato il quale dichiara fallito il tentativo di connessione.
2. Al fine di evitare di aprire più connessioni verso lo stesso client (nel caso in cui il messaggio di NEWPORT vada perso ed il client ritenti l'invio della REQUEST) il server prima di assegnare una nuova socket al client effettua un look-up delle connessioni attive per verificarne l'unicità.
3. Nel caso di mancata ricezione dell'ACK il server invia ripetutamente il messaggio di NEWPORT con le stesse modalità del punto 1.

Gestione della fase di chiusura

Sia il client sia il server sono progettati al fine di accorgersi di presunte disconnessioni della controparte a cui reagiscono disconnettendosi a loro volta.

In ogni caso al fine di diminuire la probabilità di questo genere di chiusura il receiver alla ricezione di un segmento marcato dal flag L invia 4 ACK consecutivi per quest'ultimo prima di chiudere così da aumentare la probabilità di ricezione del riscontro da parte del sender.

Gestione del timer adattativo

Il calcolo del valore migliore per l'RTO è basato su un approccio a stimatori, così come avviene in TCP.

Il valore del Round Trip Time (RTT) viene infatti stimato a campione sull'insieme dei chunk necessari al trasferimento di un intero file. In particolare il calcolo del tempo di interesse viene valutato un pacchetto alla volta, ovvero se X è il numero di sequenza associato ad un pacchetto per cui viene stimato l'RTT, per tutti i pacchetti inviati in concorrenza precedentemente alla ricezione del riscontro di X (o di un pacchetto con $X' \geq X$) tale stima non viene effettuata.

Per quanto riguarda la stima del miglior valore per RTO si utilizza il medesimo approccio di TCP, qui di seguito riportato.

$$SampleRTT = T_{recv} - T_{send}$$

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

$$DevRTT = (1 - \beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT|$$

$$con \alpha = 0.125 \text{ e } \beta = 0.25$$

Nel momento in cui scade il timeout associato ad un pacchetto non ancora riscontrato, il nuovo valore degli stimatori viene assegnato come segue:

$$EstimatedRTT(k + 1) = \min (EstimatedRTT(k), MAXERTT)$$

$$DevRTT(k + 1) = \min (DevRTT(k) * counter, MAXDRTT)$$

dove:

- *counter* rappresenta il numero di timeout consecutivi per la stessa base
- *MAXERTT* rappresenta il massimo tempo d'attesa valutato come il prodotto fra il valore iniziale del timeout e un fattore di scala.
- *MAXDRTT* rappresenta il massimo valore assumibile dalla devianza dello stimatore

Implementazione

Il progetto, così come richiesto è stato interamente implementato in C (fatta eccezione di due script di installazione).

I file sono così organizzati;

- nella cartella bin sono presenti i due eseguibili per il client e per il server
- nella cartella include vi sono gli header file necessari per l'utilizzo delle funzioni scritte per il protocollo realizzato
- nella cartella obj sono salvati i file oggetto (*.o) necessari nella fase di linking della compilazione
- nella cartella script sono memorizzati i due script .sh di installazione
- la cartella src contiene infine i sorgenti del progetto.

L'implementazione dei singoli moduli software del progetto è invece analizzata nei prossimi paragrafi.

Implementazione delle funzionalità del protocollo

Le funzionalità del protocollo applicativo ideato per il progetto sono contenute all'interno dei file gbnftp.h e gbnftp.c.

In particolare nell'header file:

- sono definite le strutture dati per la configurazione del protocollo e per la gestione del timer adattativo
- è presente la definizione del tipo di dato gbn_ftp_header (ridefinendo il tipo uint32_t della libreria stdint.h)
- sono dichiarate due enumerazioni (message_type e connection_status) utili rispettivamente nello scambio dei messaggi e nella gestione del ciclo di vita della comunicazione fra client e server.
- sono infine presenti le dichiarazioni dei prototipi delle funzioni che gli altri sorgenti possono utilizzare includendo tale header

Il file gbnftp.c invece contiene l'implementazione delle firme esplicitate nel precedente file ed in particolare:

- per la gestione delle funzioni getter e setter dei parametri dell'installazione si è fatto uso degli operatori bit a bit come l'AND/OR/NOT logico e gli shift poiché i 32 bit dedicati sono stati interpretati in realtà come una vera e propria struttura dati, così come presentato nella prima sezione, ovvero composta di 27 bit per la memorizzazione numero di sequenza e 5 per i bit di flags.
- la funzione make_segment mappa, tramite allocazione dinamica, un'area di memoria dove viene memorizzato il segmento composto dalla concatenazione dell'header e del payload

- la funzione `get_segment`, duale della precedente, ha il compito di estrapolare il corpo del messaggio e l'intestazione a partire da un segmento ricevuto
- infine le funzioni wrapper per la `send/sendto` e la `recv/recvfrom` permettono di minimizzare il numero di parametri necessari all'utilizzo di tali funzioni e di discriminare se utilizzare primitive di tipo `connection/connectionless` a seconda dei parametri attuali inseriti.

Implementazione delle funzionalità comuni a client e server

Le funzionalità comuni sono tutte raccolte all'interno dei file `common.c` e `common.h`

In particolare, l'header file contiene al proprio interno la definizione di un'enumerazione utile per discriminare le diverse modalità d'uso dell'applicazione a seconda della scelta dei parametri inseriti da linea di comando al momento dell'avvio e una serie di prototipi di funzioni esposte ai sorgenti che lo includono. Sono inoltre presenti le macro

- `perr(mess)` per l'aggiunta della linea e del nome del file dove si è generato l'errore nell'invocazione della funzione che li gestisce.
- `ABS(x)` e `MIN(x,y)` per due operazioni matematiche utili in entrambe le applicazioni

Nel file sorgente C invece sono presenti le implementazioni dei prototipi proposti ed in particolare:

- le funzioni utility di input `multi_choice` e `get_input` vengono utilizzate per effettuare una lettura corretta dallo STDIN e senza possibilità di buffer overflow (di cui invece soffre la funzione `scanf` di base)
- le funzioni `init_configurations` e `elapsed_usec` vengono utilizzate per fattorizzare parti di codice utili sia nel client sia nel server per aumentare la modularità e il riuso.
- la funzione `setup_signals` è responsabile dell'impostazione della maschera dei segnali sia per il main thread sia per i futuri thread spawnati, nonché dell'assegnazione dell'handler per i segnali di job control (`SIGINT SIGQUIT SIGTERM SIGHUP`). L'idea alla base della gestione delle interruzioni all'interno delle applicazioni è la seguente: il main thread è l'unico flusso d'esecuzione in grado di poter catturare un segnale di quelli sopra elencati poiché gli altri thread spawnati bloccano tale lista come prima istruzione della funzione di entry point. Nella `setup_signals` viene anche esplicitamente ignorato il segnale `SIGPIPE` poiché tutti i segmenti inviati sono marcati dal flag `MSG_NOSIGNAL`.
- Sono presenti, inoltre, una serie di funzioni di utility marcate come “_safe” che hanno il compito di svolgere operazioni che interessano sezioni critiche del codice per cui al loro interno invocano primitive di lock e unlock rispettivamente prima e dopo l'esecuzione del blocco di codice.

Implementazione del client e del server

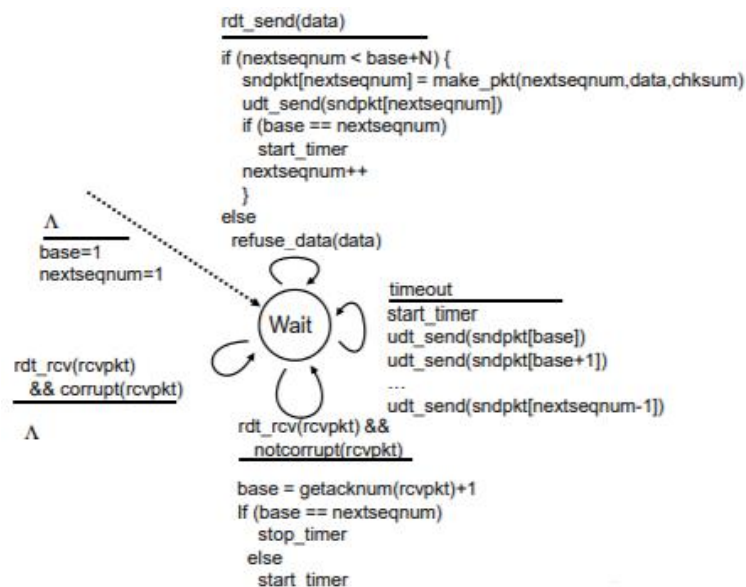
Per quando riguarda l'implementazione del client e del server, in ambedue i sorgenti sono presenti un discreto numero di funzioni di utility su cui è di poco interesse soffermarsi, come ad esempio le funzioni di inizializzazione/distruzione per i vari elementi allocati dinamicamente o le funzioni per il parsing della command line (su cui è presente un'ulteriore piccola digressione nell'ultima sezione).

In questo paragrafo per cui verranno analizzati solo gli elementi di codice realmente di interesse relativamente all'architettura.

Operazioni di LIST e GET

Sia nel caso di richiesta LIST che di GET, il comportamento del server è schematizzabile come segue:

1. All'interno dell'acceptance_loop ricevuta la request da parte del server vengono allocate le risorse necessarie a gestirla (se disponibili) e in tal caso viene spawnato un thread allo scopo di orchestrare l'invio di chunk verso il client.
2. La routine del thread spawnato consiste in un while loop dove a ciascuna iterazione vengono invocate la funzione `send_file_chunk` `handle_retransmit` o `lg_send_port_message` rispettivamente per gli stati CONNECT, TIMEOUT e REQUEST.
3. In piena concorrenza il main thread all'interno della funzione `handle_rcv` (invocata dall'acceptance loop al momento della ricezione di un ACK sulla socket associata alla connessione) aumenta il valore di base al fine di poter far avanzare la finestra e se necessario riavvia il timer.



In definitiva, prendendo di riferimento l'automa del sender del protocollo GBN qui sopra riportato, il thread spawnato, così come spiegato al punto 2, copre le transizioni associate agli eventi di invio dati e timeout, mentre il main thread si occupa di svolgere la funzionalità associata alla transazione di ricezione di un ack non corrotto in maniera concorrente per tutte le connessioni attive. A tale scopo viene utilizzata la funzione `select` che permette di effettuare polling I/O su un insieme di descrittori, diminuendo in tal modo il numero di thread attivi sulla macchina.

Le funzionalità di GET e di LIST differiscono l'un l'altra per il file che viene inviato. In particolare nel caso della LIST viene inviato un file speciale nascosto `.tmp-ls` che viene aggiornato in sezione critica, coperta da `rwlock`, ad ogni operazione di PUT e che contiene la lista di file disponibili nella directory pubblica del server. Nell'operazione di GET invece il file da spedire è specificato all'interno della REQUEST del client.

Il client invece dedica le funzioni `get_file` e `list` all'adempimento di queste due funzionalità. Comune a tutte le funzionalità esposte è la funzione di `request_loop`, la quale tenta la connessione al server inviando ripetutamente messaggi di REQUEST fino alla ricezione del messaggio di NEWPORT o al termine dei tentativi possibili.

In seguito:

- nel caso di richiesta GET vengono scritti su file (tramite la primitiva write) i chunk ricevuti dal server
- nel caso di richiesta LIST vengono stampati a video (sempre tramite primitiva write) i chunk ricevuti contenenti la lista di file disponibili.

La richiesta GET differisce dalla LIST anche nella fase iniziale dove, infatti, viene richiesto all'utente di inserire il nome del file di cui si desidera effettuare il download.

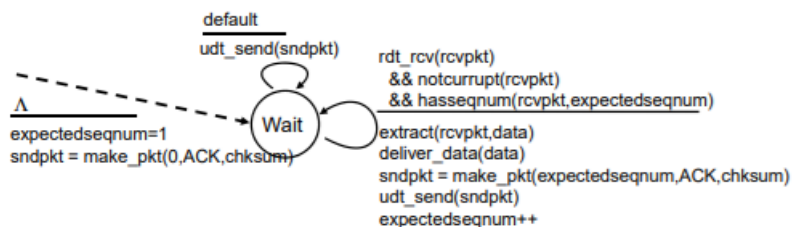
Operazione di PUT

L'operazione PUT prevede un comportamento duale del client e del server. Di fatti mentre LIST e GET sono operazioni di pull quest'ultima è una tipico esempio di push operation.

Il client in questo caso opera in maniera analoga al server ovvero spawnando un thread per la gestione della fase di invio dei chunk. Il server, d'altra parte, dopo aver accettato la richiesta di connessione all'interno dell'`acceptance_loop` spawna anch'esso un thread per la gestione della ricezione e dell'invio dei riscontri.

Nel caso di richiesta PUT il descrittore di file associato alla socket dedicata al client non viene aggiunto al set per la select bensì sarà il nuovo thread a mettersi in ascolto sulla nuova socket sempre tramite select (al fine di avere un tempo di attesa massimo dopo il quale dichiarare chiusa la connessione per abbandono). Alla ricezione di ciascun segmento il server invia riscontri basandosi sui valori di `expected_seq_num` e `last_acked_seq_num`.

Anche in questo caso come nel precedente prendendo come riferimento l'automa del receiver del protocollo GBN qui di sotto riportato il thread spawnato copre interamente le funzionalità default e quella di ricezione di un pacchetto con numero di sequenza atteso.



Limitazioni riscontrate

Sia l'applicazione client che l'applicazione server svolgono correttamente le 3 funzionalità richieste dalla consegna.

Il server utilizza $n + 1$ porte, con n numero di clienti attualmente connessi per soddisfare le richieste. Questo numero pone un upper-bound al numero di connessioni simultaneamente attive pari al numero massimo di porte attualmente disponibili (< 65536).

In realtà il tetto massimo di default utilizzato nell'applicazione è ancora minore e pari al numero di core disponibili sulla macchina moltiplicato per 4. Tale limitazione è stata aggiunta al fine di diminuire l'overhead dovuto al context-switching dei threads fra i core.

Un altro approccio possibile sarebbe stato quello di mantenere una singola socket sia per l'accettazione sia per i client già connessi. In questo scenario sarebbe stato poi necessario smistare i messaggi tramite strutture di ITC (Inter Threads Communication) come, ad esempio, le pipe o buffer di memoria condivisa fra i vari thread attivi.

Questo iter procedurale necessita però di una doppia lettura: una tramite flag MSG_PEEK per mantenere il messaggio nel buffer da parte del thread d'accettazione e una senza necessità di flag effettuata dal thread servente per l'effettiva elaborazione. Queste ultime dovendo essere sincronizzate per evitare sovrascritture della testa del buffer introdurrebbero un collo di bottiglia causando rallentamenti.

Specifiche della piattaforma SW utilizzata

Per lo sviluppo dell'intero progetto è stata utilizzata una macchina Linux con le seguenti specifiche software:

- Sistema operativo: Manjaro Linux x86_64
- Kernel: 5.7.15-1-MANJARO

In aggiunta sono qui riportate le versioni dei tool di compilazione e debugging:

- GCC: versione 10.2.0
- Valgrind: versione 3.16.1
- Terminator: versione 1.92
- Gnome-terminal: versione 3.36.2 con VTE 0.60.3 +BIDI +GNUTILS +ICU +SYSTEMD

Esempi di funzionamento

Di seguito sono riportati alcuni screenshot relativi alle varie funzionalità offerte dall'applicazione.

In tutti gli esempi riportati nella tab superiore di Terminator (il terminal multiplexer utilizzato per il collaudo dell'applicazione) è in esecuzione il server mentre nella tab inferiore vi è il client.

```

/bin/zsh
~/bin/zsh 94x18
~/gmb-ftp/bin >>> ./gmb-ftp-server --verbose
Do you want to see current settings profile? [y/n]: y
List of current settings for server:
N.....: 8
rcvtimeout.: 500 usec
port.....: 2929
adaptive...: true
probability: 0.01
pool size..: 16
Server is now listening ...
[ ]

/bin/zsh 94x18
*** What do you wanna do? ***
[L]IST all available files
[P]UT a file on the server
[G]ET a file from server
[Q]uit
Pick an option [L/P/G/Q]:
  
```

HOME PAGE

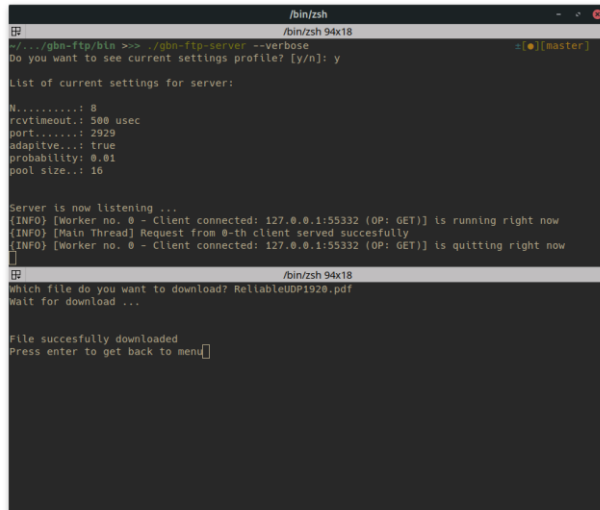
```

/bin/zsh
~/bin/zsh 94x18
~/gmb-ftp/bin >>> ./gmb-ftp-server --verbose
Do you want to see current settings profile? [y/n]: y
List of current settings for server:
N.....: 8
rcvtimeout.: 500 usec
port.....: 2929
adaptive...: true
probability: 0.01
pool size..: 16
Server is now listening ...
(INFO) [Worker no. 0 - Client connected: 127.0.0.1:47413 (OP: LIST)] is running right now
(INFO) [Main Thread] Request from 0-th client served successfully
(INFO) [Worker no. 0 - Client connected: 127.0.0.1:47413 (OP: LIST)] is quitting right now
[ ]

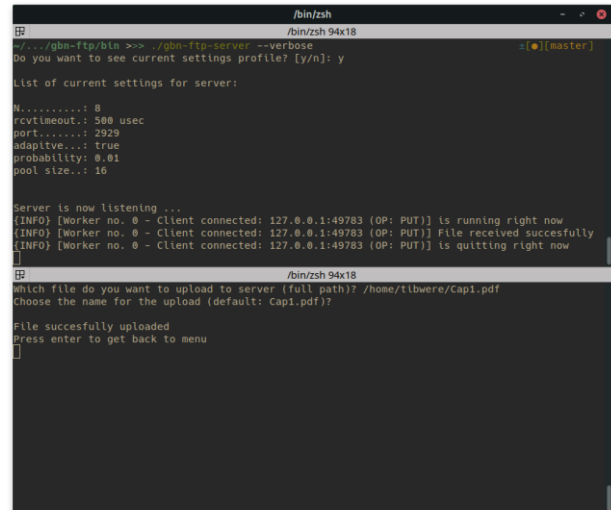
/bin/zsh 94x18
AVAILABLE FILE ON SERVER
README.txt
AUDIO_SEGNALE.zip
cem.pdf
fisica2.pdf
gapil.pdf
ReliableUDP1920.pdf
wallpapers-master.zip
test.iso
Press any key to get back to menu [ ]
  
```

OPERAZIONE LIST

RELAZIONE PROGETTO B1 TRASFERIMENTO FILE SU UDP

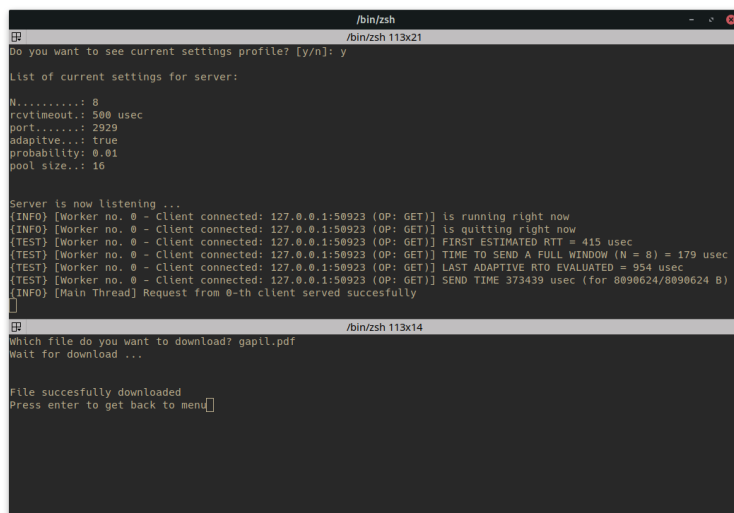


OPERAZIONE GET

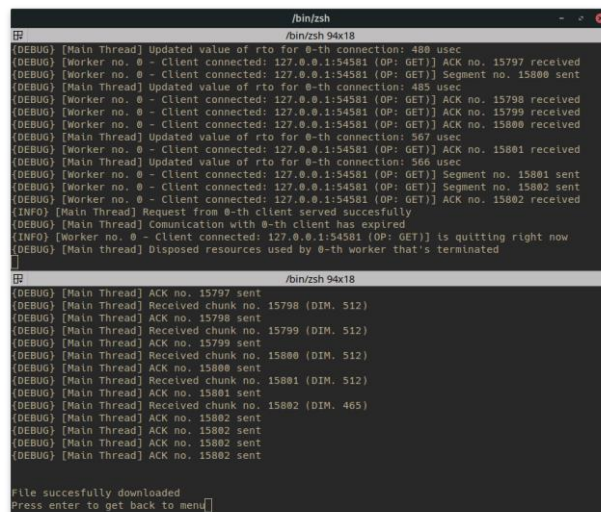


OPERAZIONE PUT

In aggiunta sono riportati di seguito gli screenshot delle modalità debugging e testing usati in fase di progettazione/implementazione:



MODALITÀ TEST



MODALITÀ DEBUG

Valutazione delle prestazioni

Per la fase di test delle prestazioni sono state predisposte alcune funzionalità di calcolo delle tempistiche attivate solo inserendo la macro TEST in fase di compilazione tramite il flag -DTEST nella regola del Makefile. Tale macro inoltre bypassa la randomizzazione del seed al fine di realizzare dei test più affidabili, aventi la stessa serie di esiti positivi/negativi per l'invio dei chunk.

Al fine di testare le prestazioni del protocollo sono stati utilizzati i seguenti 3 file:

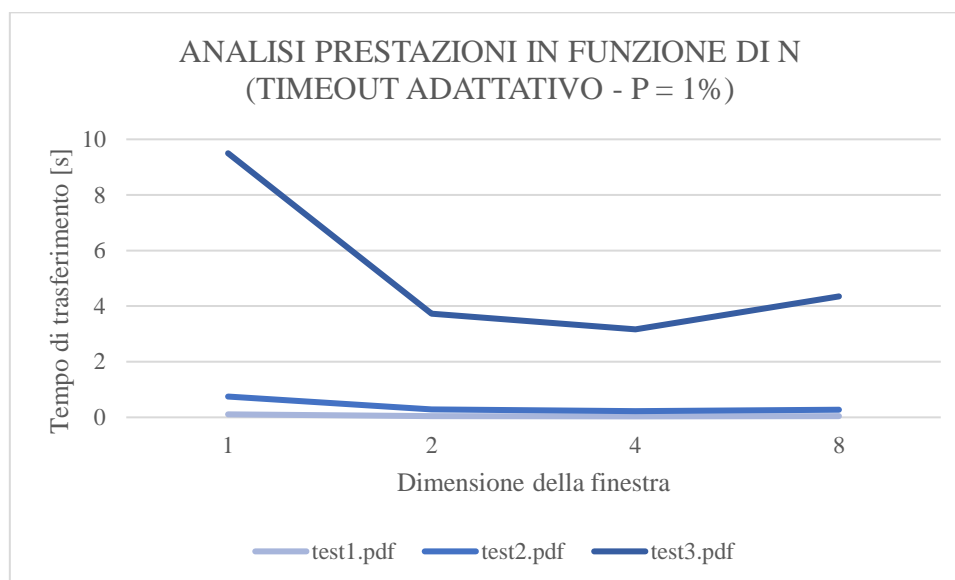
Nome file	Dimensione	Numero chunks
test1.pdf	~ 1 MB (1,083,157 B)	2116
test2.pdf	~ 10 MB (8,090,577 B)	15802
test3.pdf	~ 100 MB (110,638,036 B)	216090

I parametri per qui è stata effettuata un'analisi delle prestazioni sono:

1. Dimensione della finestra (N)
2. Entità del timeout (t)
3. Probabilità d'invio (P)

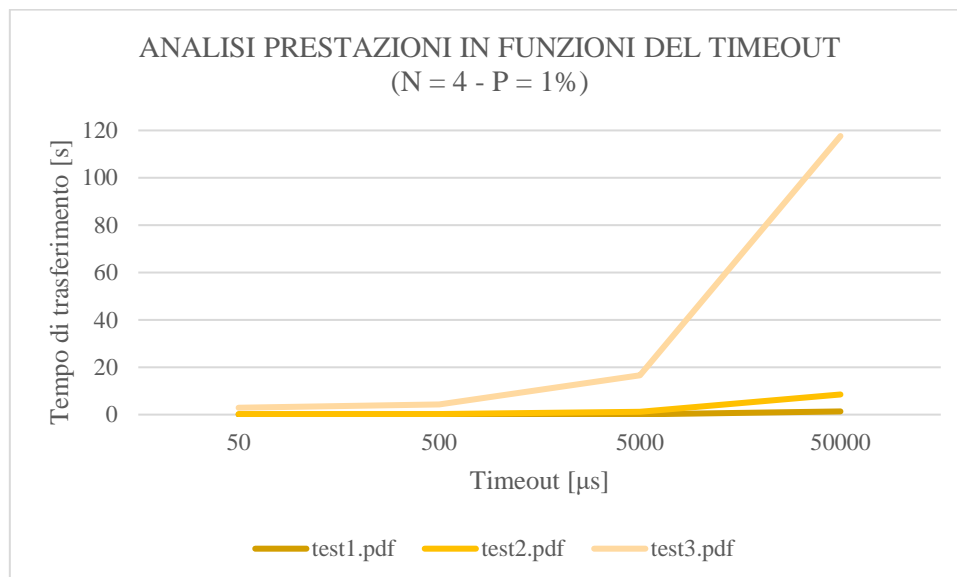
Analisi in funzione della dimensione della finestra

N	test1.pdf	test2.pdf	test3.pdf
1	0,107988 sec	0,748385 sec	9,498714 sec
2	0,046069 sec	0,280757 sec	3,732964 sec
4	0,034576 sec	0,223874 sec	3,165481 sec
8	0,048742 sec	0,27375 sec	4,353436 sec



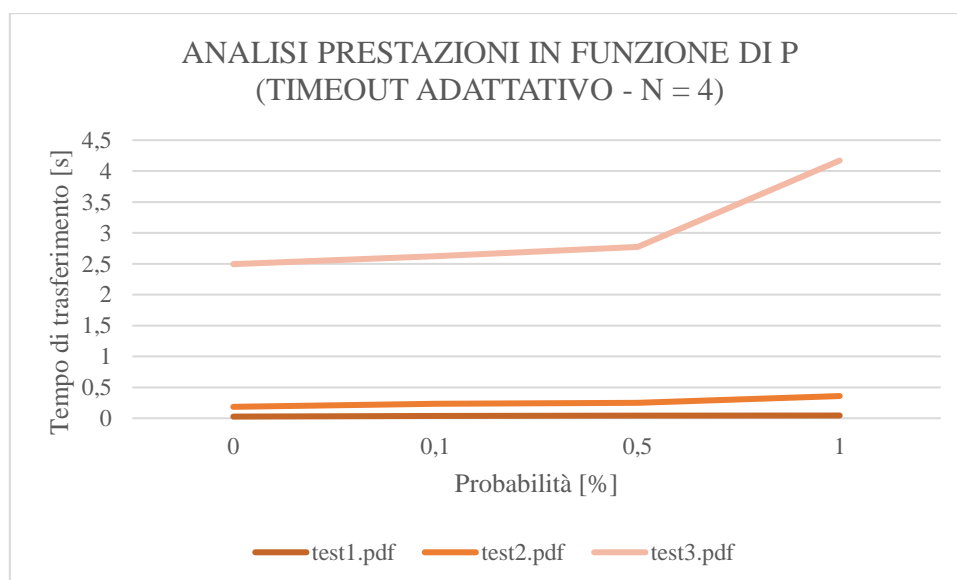
Analisi in funzione dell'entità del timeout

t	test1.pdf	test2.pdf	test3.pdf
50 μ sec	0,043449 sec	0,236393 sec	2,90664 sec
500 μ sec	0,048163 sec	0,303999 sec	4,29539 sec
5000 μ sec	0,184397 sec	1,169922 sec	16,657814 sec
50000 μ sec	1,358286 sec	8,509355 sec	117,623086 sec



Analisi in funzione della probabilità d'invio

P	test1.pdf	test2.pdf	test3.pdf
0 %	0,026677 sec	0,184946 sec	2,494295 sec
0,1 %	0,038842 sec	0,235756 sec	2,622823 sec
0,5 %	0,041272 sec	0,251105 sec	2,773968 sec
1 %	0,044303 sec	0,359431 sec	4,170728 sec



Manuale per l'installazione, configurazione ed esecuzione

Le due applicazioni sviluppate utilizzano un Makefile come facility di compilazione. Quest'ultimo è stato scritto per essere parametrico e basato sulle dipendenze al fine di evitare inutili ricompilazioni dei file oggetto e diminuire il tempo d'esecuzione del comando.

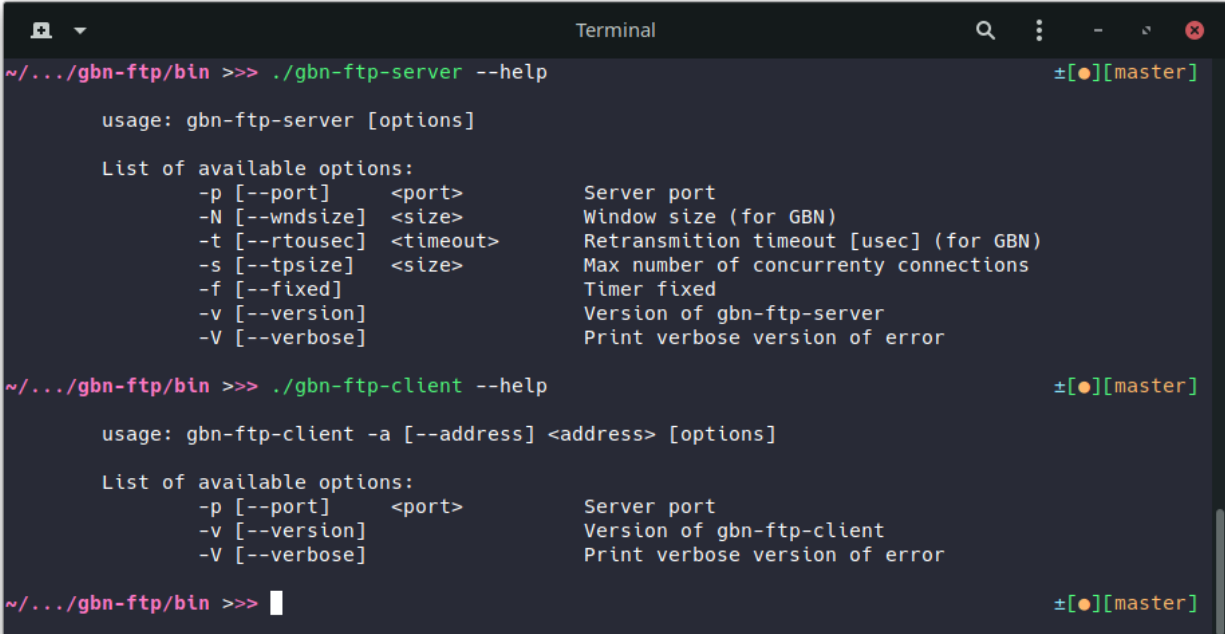
Per compilare i sorgenti è sufficiente utilizzare una qualsiasi shell Linux, spostarsi all'interno della root del progetto /path/to/gbn-ftp e invocare i seguenti comandi:

- make server: per la compilazione del server
- make client: per la compilazione del client
- make: per la compilazione di entrambe le applicazioni
- make clean: per la rimozione dei file oggetto (pulizia della cartella obj)

La fase d'installazione prevede i seguenti step:

- per l'installazione del server è necessario dirigersi sempre tramite shell nella directory script (con il comando cd script/) ed invocare il comando sh install-server.sh. Quest'ultimo è responsabile della creazione della directory pubblica del server: ~/.gbn-ftp-public (eventualmente chiedendo la conferma per una sovrascrittura).
- per l'installazione del client è sufficiente ripercorrere i passi descritti precedentemente invocando questa volta il comando sh install-client.sh. In questo caso verrà creata la directory per la memorizzazione dei file scaricati: ~/.gbn-ftp-download.

Sia il client che il server sono stati concepiti con una gestione della command line conforme allo standard POSIX implementata grazie alla system call getopt. Qui di seguito è riportato uno screenshot dell'help di ambedue le applicazioni:



```

Terminal
~/.../gbn-ftp/bin >>> ./gbn-ftp-server --help ±[●][master]

usage: gbn-ftp-server [options]

List of available options:
      -p [--port] <port>           Server port
      -N [--wndsize] <size>         Window size (for GBN)
      -t [--rtousec] <timeout>     Retransmission timeout [usec] (for GBN)
      -s [--tpsize] <size>         Max number of concurrently connections
      -f [--fixed]                 Timer fixed
      -v [--version]               Version of gbn-ftp-server
      -V [--verbose]               Print verbose version of error

~/.../gbn-ftp/bin >>> ./gbn-ftp-client --help ±[●][master]

usage: gbn-ftp-client -a [--address] <address> [options]

List of available options:
      -p [--port] <port>           Server port
      -v [--version]               Version of gbn-ftp-client
      -V [--verbose]               Print verbose version of error

~/.../gbn-ftp/bin >>> ±[●][master]
  
```