

Relazione progetto IIW

A.A. di riferimento 2021/2022

Oggetto: Trasferimento file affidabile tramite UDP + Selective Repeat

Partecipanti: Samuele Taglienti [0270075], Ferdinando Ottaviani [0266984]

● Scelte progettuali:

Obiettivi:

- connessione senza autenticazione
- garantire trasmissione affidabile di messaggi/file su un servizio di comunicazione non affidabile (UDP), usando come scheletro lo stesso concetto del Selective Repeat.
- comando di <LIST> (identificato nel nostro progetto come LIST, key=1)
- comando di <GET> (identificato nel nostro progetto come DOWNLOAD, key=2)
- comando di <PUT> (identificato nel nostro progetto come UPLOAD, key=3)

Parametri del Selective Repeat:

- finestra di spedizione di dimensione fissata, scelta dall'utente al lancio dell'applicazione "client".
- timer adattivo o statico, scelto dall'utente al lancio dell'applicazione "client".

Struttura dei pacchetti:

in primo luogo, abbiamo definito i pacchetti da inviare/ricevere tramite una struct indicata come segue:

```
struct packet{
    int num_pkt;           // Numero di pkt
    int ack_rec;           // Identifica la presenza/assenza del relativo ack
    char payload[SIZE]     // SIZE→ Macro modificabile
    struct packet* next;   // Puntatore al nodo successivo
};
```

Un'ulteriore precisazione sul puntatore "next", questo serve per poter implementare una lista, tramite la quale avverrà tutta la gestione del selective repeat (e.g. ack, riTx ...).

Server concorrente:

quest'ultimo è in grado di gestire contemporaneamente richieste di più client, infatti una volta ricevuta una request, esegue una **fork()** e delega la gestione di tale richiesta al figlio, il quale terminerà all'invocazione del comando "exit" [key = 4] dal lato client; infine il numero di porta "server figlio" è funzione del numero di porta del server principale e dei vari client.

Implementazione selective repeat:

- 1 – creazione e gestione della finestra di spedizione
- 2 – invio e ricezione dei pacchetti
- 3 – invio e ricezione degli ACK
- 4 - gestione del timer di rispedizione

1.

La finestra di spedizione del mittente è stata realizzata mediante una struct come segue:

```
struct window{
    int start;
    int end;
};
```

con window.start sempre pari ad 1 al lancio del comando (put|get) e window.end posto al valore scelto dall'utente per la grandezza della finestra. Per questioni puramente visive è stato scelto di non seguire il pattern classico della finestra (es. dim = 4 → 0123|0123|...) bensì window.end incrementa fino a coincidere con l'ultimo pacchetto da inviare (es. dim = 4 → 0123|4567|...) ovviamente basando il tutto sullo scorrimento della finestra grazie a window.start e mantenendo quindi il concetto alla base.

2.

La spedizione di pacchetti non viene eseguita dal main thread, bensì per ogni pacchetto da inviare viene lanciato un nuovo child_thread che si occuperà, oltre che dell'invio del pacchetto, della ricezione dell'ack e dell'aggiornamento delle strutture dati. [Presenza di semafori]

L'invio avviene tramite una sendto(), la quale prende in input una struct packet popolata precedentemente (in particolare il payload tramite una fseek() seguita da una fread()). La struct usata dal lato mittente, viene situata in un'area di memoria allocata dinamicamente e messa in coda ad una lista a puntatore singolo [con ack_rec=0].

Una volta inviato correttamente il pacchetto, partirà il timer annesso (implementato con la funzione alarm()) e gestione del relativo segnale SIGALARM) eventualmente preso in carico dalla funzione funct_gest_timeout(); questo fa sì che, avendo utilizzato i segnali, la ritrasmissione di pacchetti ha priorità sul resto.

La ricezione dei pacchetti avviene tramite una recvfrom(), dalla quale, presa la struct packet in ingresso, viene estratto il payload e scritto nell'apposito file (tramite una fseek() seguita da una fwrite()).

3.

Lato destinatario, una volta ricevuta la struct packet, viene resa disponibile come informazione anche il num_pkt che verrà utilizzato per inviare al mittente l'ack corretto. In seguito all'invio di quest'ultimo, verrà aggiornata la struttura dati ack_is_in[]; questa viene utilizzata come check dopo ogni arrivo di pacchetto per catturare un'eventuale ritrasmissione, la quale necessita del solo invio dell'ack senza salvataggio del pacchetto, perché già presente in memoria.

Lato mittente dati, come anticipato, saranno i thread a gestire l'arrivo degli ack. Una volta ricevuto l'ack_#i per il pkt_#i, il thread_#i andrà ad invocare la funzione check_list() che provvederà ad aggiornare la relativa struct packet (in particolare ack_rec 0 → 1) oltre che a fare un check sull'eventuale possibilità di spostare in avanti la finestra di trasmissione. Questo spostamento va di pari passo con quello della testa della lista, così da ridurre il numero di accessi in memoria ad una prossima invocazione di check_list().

4.

Come detto poc'anzi la gestione dei timer e delle relative ritrasmissioni avviene tramite la funzione funct_gest_timeout() dopo la ricezione del segnale di alarm().

Questa funzione va a fare un check della lista per trovare la vittima con ack_rec=0, rispedisce il pacchetto e riattiva il timer.

● Analisi strutturali:

Inizialmente l'applicazione chiede di inserire i vari parametri (dimensione della finestra, probabilità di perdita e timer) per il corretto funzionamento di quest'ultima. Apertosi il menù sul terminale ed una volta scelto il comando da eseguire partirà lo scambio di messaggi che andremo ad analizzare qui di seguito.

CASO LIST [1].

Dal lato mittente troviamo una recvfrom() seguita da una printf().

Dal lato server viene creata una fileList[] tramite un ciclo while unito a strcat() e spedita tramite sendto().

Il client è subito pronto per eseguire un nuovo comando

CASO DOWNLOAD [2].

Una volta selezionato il file da scaricare, scelto dalla lista stampata a schermo, dal lato mittente (server) Il primo passo per inviare un file è quello di suddividerlo in pacchetti utilizzando la variabile **div_packets**, calcolata con l'ausilio della funzione **ftell()** e dividendola per SIZE (la dimensione prefissata per il pacchetto). Successivamente div_packets viene condivisa con il destinatario (client) tramite una sendto() e recvfrom().

Come anticipato verranno lanciati un numero di threads pari a div_packets, ognuno dei quali eseguirà in ordine le seguenti mansioni:

stima del timeout da lanciare in seguito, creazione della struttura **packet** (contenente il numero di sequenza (int), la presenza o assenza del relativo ACK (int), il payload[SIZE] che contiene i dati da inviare alla sendto() (char) ed infine struct packet* next, ovvero il puntatore al nodo successivo) più inserimento nella lista; questa struttura verrà inviata da un lato e ricevuta dall'altro, facendo partire l'apposito timer. Il thread, lato mittente (server), è

quindi pronto per ricevere l'ack, aggiornare il relativo campo `ack_rec` nella struct `packet`, modificare eventualmente la window e la struttura dati `ack_is_in[]` che il main thread server usa con cadenza periodica per verificare la terminazione dell'invio del file (implementata strutturalmente con un ciclo `while()`). Se questo controllo dovesse avere esito positivo il client (e/o server) è pronto per eseguire un nuovo comando. Nel caso in cui dovesse scadere un timer arriverà un segnale `SIGALARM` passando così immediatamente il controllo alla funzione `funct_gest_timeout`, che scorrerà la lista, troverà la vittima senza ack ricevuto, spedisce il pacchetto facendo partire il nuovo timer e cederà il controllo all'indietro.

Dal lato ricevente (client) come nel main thread del mittente (server), c'è una struttura dati `is_in[div_packets]` usata anche qui periodicamente come controllo di un'eventuale terminazione del comando.

Estratto il pacchetto dalla struttura dati ricevuta, verrà scritto il relativo payload su file ed inviato l'ack corrispondente con aggiornamento dell'array sopra citato.

Nel caso in cui dovesse arrivare un pacchetto già arrivato in precedenza (saremo in grado di capirlo tramite `is_in[]`) questo verrà scartato perché già scritto in memoria, mentre il relativo ack verrà rispedito.

Al termine del comando, il file verrà chiuso, le strutture dati allocate dinamicamente verranno deallocate tramite `free()` ed i semafori usati per la consistenza dei dati cancellati.

CASO UPLOAD [3].

Una volta selezionato il file da inviare, dopo averlo scelto dalla lista stampata a schermo, dal lato mittente (client), il primo passo per inviare un file è quello di suddividerlo in pacchetti utilizzando la variabile **`div_packets`**, calcolata con l'ausilio della funzione **`ftell()`** e dividendola per `SIZE` (la dimensione prefissata per il pacchetto).

Successivamente `div_packets` viene condivisa con il destinatario (server) tramite una `sendto()` e `recvfrom()`.

Come anticipato verranno lanciati un numero di threads pari a `div_packets`, ognuno dei quali eseguirà in ordine le seguenti mansioni:

stima del timeout da lanciare in seguito, creazione della struttura **`packet`** (contenente il numero di sequenza (int), la presenza o assenza del relativo ACK (int), il payload[`SIZE`] che contiene i dati da inviare alla `sendto()` (char) ed infine struct `packet* next`, ovvero il puntatore al nodo successivo) più inserimento nella lista; questa struttura verrà inviata da un lato e ricevuta dall'altro, facendo partire l'apposito timer. Il thread è quindi pronto per ricevere l'ack, aggiornare il relativo campo `ack_rec` nella struct `packet`, modificare eventualmente la window e la struttura dati `ack_is_in[]` che il main thread usa con cadenza periodica per verificare la terminazione dell'invio del file (implementata strutturalmente con un ciclo `while()`). Se questo controllo dovesse avere esito positivo il client è pronto per eseguire un nuovo comando.

Nel caso in cui dovesse scadere un timer arriverà un segnale `SIGALARM` passando così immediatamente il controllo alla funzione `funct_gest_timeout`, che scorrerà la lista, troverà la vittima senza ack ricevuto, spedisce il pacchetto facendo partire il nuovo timer e cederà il controllo all'indietro.

Dal lato ricevente (server) come nel main thread del mittente (client), c'è una struttura dati `is_in[div_packets]` usata anche qui periodicamente come controllo di un'eventuale terminazione del comando.

Estratto il pacchetto dalla struttura dati ricevuta, verrà scritto il relativo payload su file ed inviato l'ack corrispondente con aggiornamento dell'array sopra citato.

Nel caso in cui dovesse arrivare un pacchetto già arrivato in precedenza (saremo in grado di capirlo tramite `is_in[]`) questo verrà scartato perché già scritto in memoria, mentre il relativo ack verrà rispedito.

Al termine del comando, il file verrà chiuso, le strutture dati allocate dinamicamente verranno deallocate tramite `free()` ed i semafori usati per la consistenza dei dati cancellati.

CASO EXIT [4].

Lato client viene chiusa la socket e successivamente viene terminata anche l'applicazione.

Lato server viene chiusa la socket relativa a quel dato client, il server figlio viene terminato, ma il main server è sempre disponibile alla ricezione di richieste; quest'ultimo può essere terminato solo tramite comando CTRL+C da terminale

CASO CHIUSURA FORZATA. [CTRL+C] v [CTRL+\].

Nel caso in cui venisse chiuso forzatamente uno tra client e server, la chiusura di uno implica la chiusura dell'altro. In particolare nel caso in cui venga chiuso forzatamente il client, verrà chiuso forzatamente il server figlio, mantenendo attivo il server principale; cosa che non accade invece a parti inverse, dove un segnale di SIGINT nel server tira giù tutta l'applicazione.

LIMITAZIONI.

La nostra applicazione client/server non è in grado di gestire la consistenza tra un upload/download (o download/upload) in contemporanea dello stesso file su due o più client diversi.

● Test e approfondimenti sui risultati ottenuti

A questo punto inizia la fase di test del sistema che è stata realizzata mediante l'impiego di Excel.

I test effettuati riguardano l'analisi del tempo impiegato per un file di prova per essere trasferito da client a server e viceversa a seconda dell'ampiezza della finestra e del tipo di timer utilizzato.

In questo primo grafico mostriamo l'andamento del tempo di trasferimento e possiamo notare che all'aumentare della dimensione della finestra migliorano le statistiche.

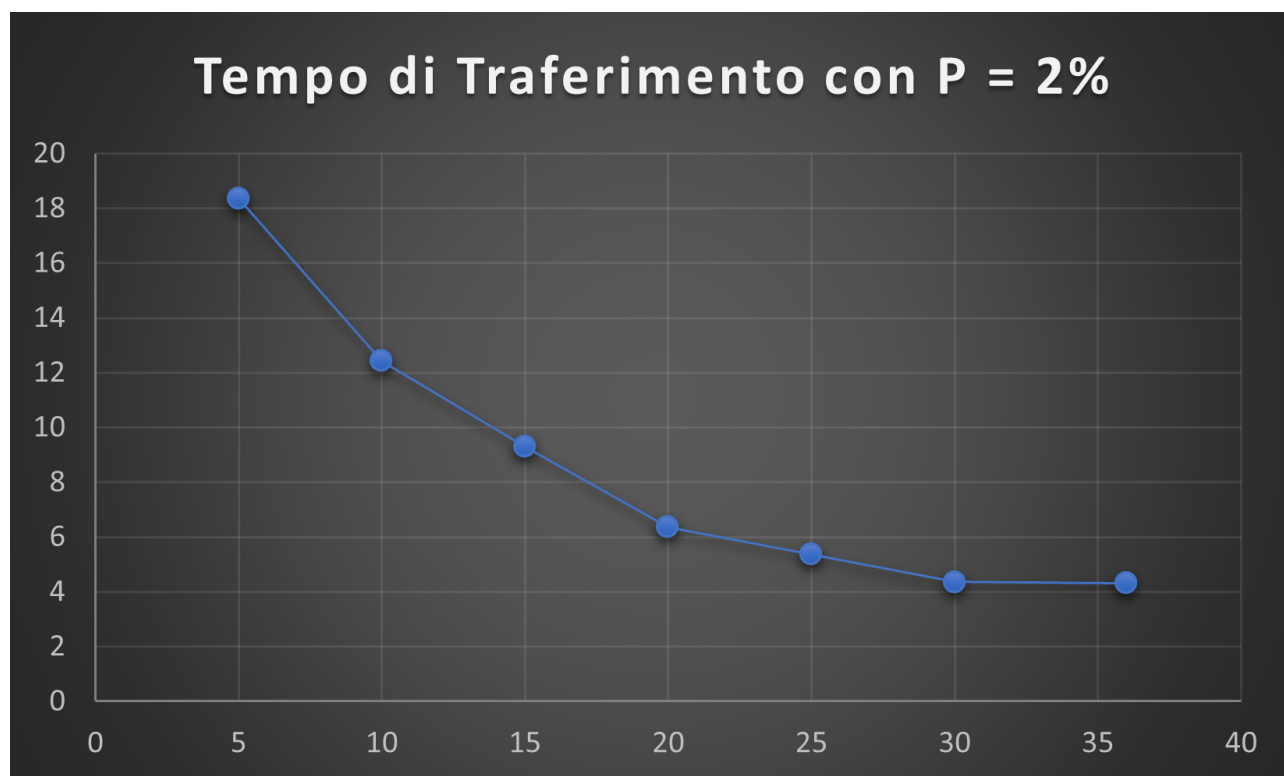
Questo grafico è stato realizzato tenendo in considerazione una probabilità di perdita (P) pari al 20%, timer adattivo ed una window variabile di 5 in 5 da test a test.

Il file campione utilizzato è l'immagine denominata lucky.jpeg di dimensione pari a 35,2 KiB divisa in 36 pacchetti di 1024 bit.

Sull'asse delle X abbiamo la dimensione della finestra, mentre su quelle delle Y abbiamo il tempo.



Il secondo grafico prende in input gli stessi parametri del precedente, si assume però la presenza di un canale ottimo con una percentuale di perdita bassa pari al 2%.



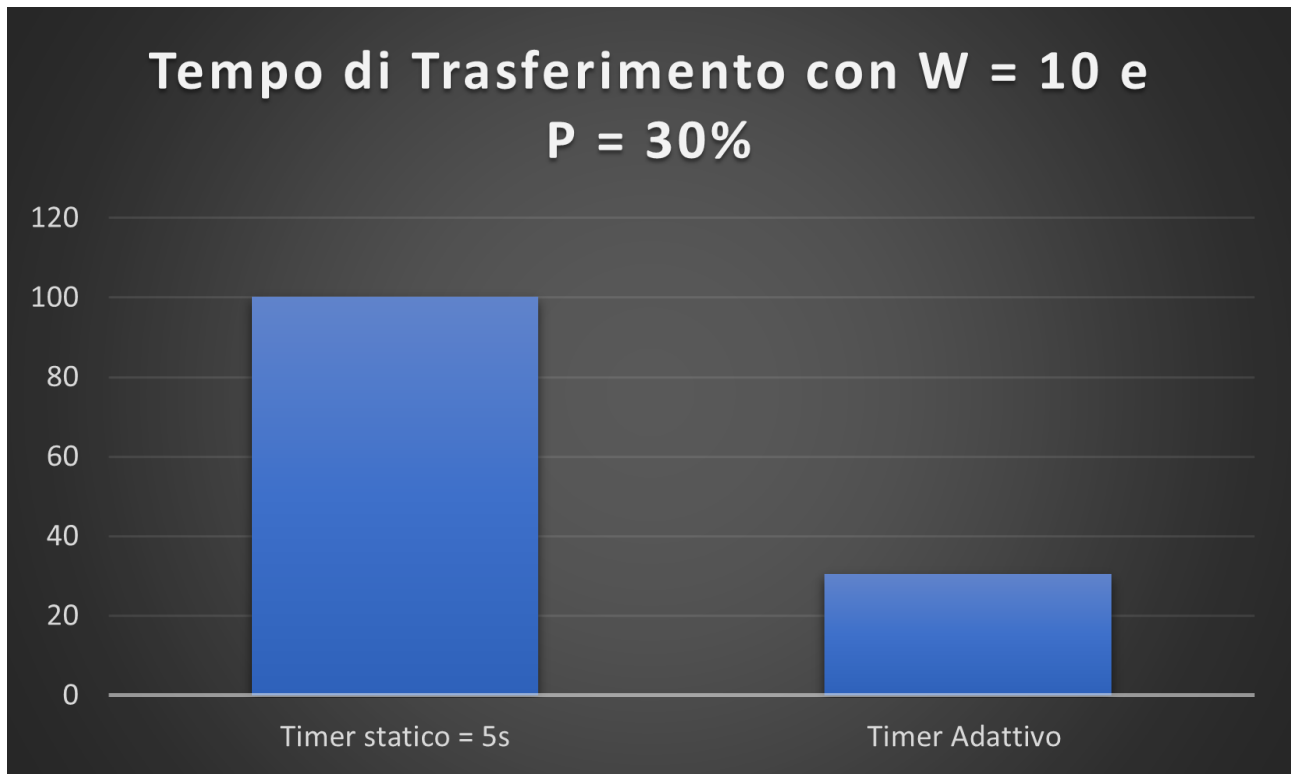
Il terzo grafico, stesso file, ma con window pari a 10, ed insieme all'utilizzo del timer adattivo, mostra il tempo di trasferimento al variare della probabilità di perdita P .

Verrà preso in esame un caso con perdita 0% come riferimento del test per poi andare a maggiorare fino ad arrivare ad una perdita dell'80%.

Sull'asse delle x abbiamo i vari tassi di perdita mentre su quella delle y abbiamo il tempo. Possiamo notare come all'aumentare della probabilità di perdita il tempo di trasferimento aumenti notevolmente.



Il quarto ed ultimo grafico va ad analizzare la differenza nel tempo di trasferimento, tra la scelta del timer statico (impostato a 5s) ed il timer adattivo. Il test è stato eseguito sempre con lo stesso file .jpeg con finestra $W = 10$ e perdita $P = 30\%$. Come si può ben notare la scelta del timer adattivo impatta considerevolmente il tempo di trasferimento del file.



● Piattaforma Software utilizzata

Per sviluppare il seguente progetto sono state utilizzate due macchine in configurazioni diverse. La prima è una macchina virtuale tramite VMWare Workstation 16 con KUbuntu 22.10, la seconda invece virtualizzata con Oracle Virtual Box 6.1.40 con Ubuntu 22.04.1. In entrambi i casi per comodità è stato utilizzato come editor Visual Studio Code in modo tale da trasferire il lavoro senza utilizzare programmi diversi. Per quanto riguarda il testing delle applicazioni invece è stato utilizzato il semplice terminale (Konsole per Kubuntu, Terminale per Ubuntu). Per il debug è stato utilizzato il tool GDB che ci ha permesso di evidenziare e fixare eventuali errori all'interno del codice.

● Esempi di funzionamento

● Compilazione ed Esecuzione del Programma

Tramite il comando

```
gcc -o server server.c -lpthread e/o gcc -o client client.c -lpthread
```

vengono compilati entrambi i programmi.

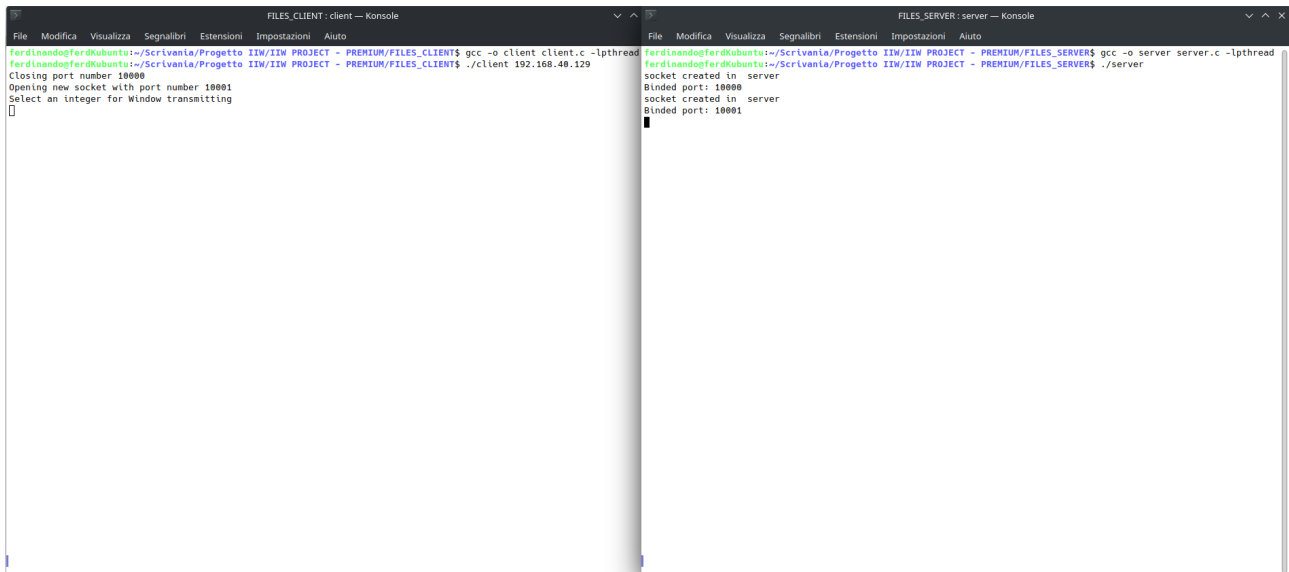
Con

```
./server
```

```
./client <ip_address>
```

vengono entrambi lanciati.

IMPORTANTE: per la buona riuscita dell'applicazione client-server è **RIGOROSAMENTE NECESSARIO** lanciare come primo programma `./server` e solo successivamente i vari `./client`

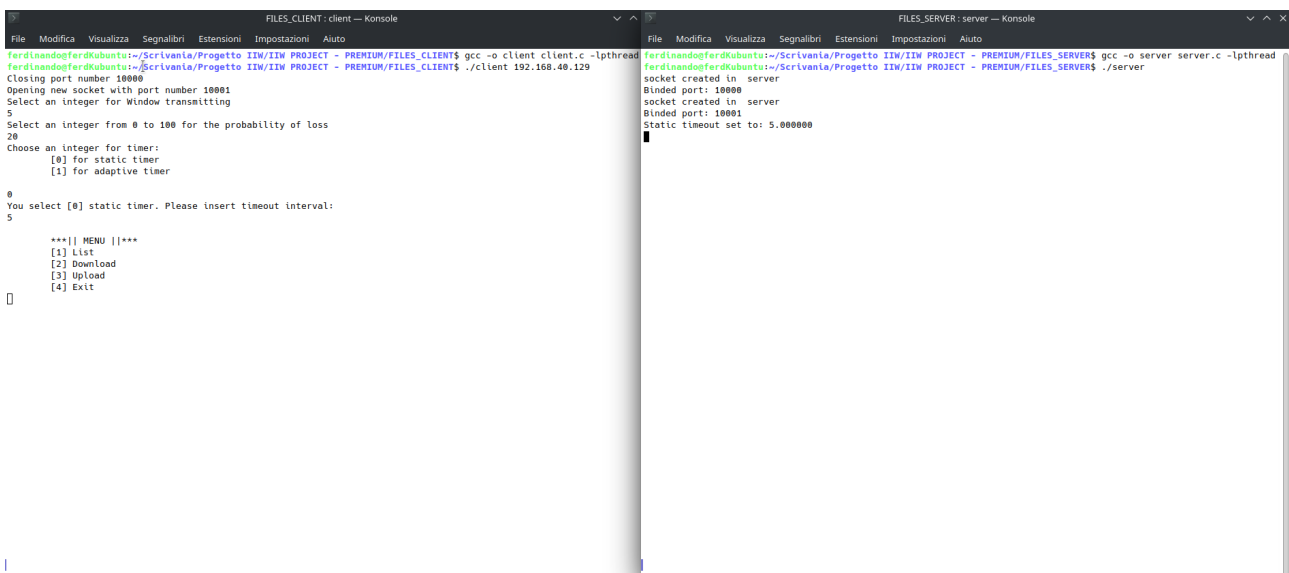


The image shows two terminal windows side-by-side. The left window, titled 'FILES_CLIENT: client - Konsole', shows the compilation of 'client.c' and its execution. The right window, titled 'FILES_SERVER: server - Konsole', shows the compilation of 'server.c' and its execution. Both windows show the user 'ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT\$' and 'ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER\$' respectively.

```
FILES_CLIENT: client - Konsole
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT$ gcc -o client client.c -lpthread
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT$ ./client 192.168.48.129
Closing port number 10000
Opening new socket with port number 10001
Select an integer for Window transmitting
5

FILES_SERVER: server - Konsole
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ gcc -o server server.c -lpthread
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ ./server
socket created in server
Binded port: 10000
socket created in server
Binded port: 10001
```

● Inserimento Parametri



The image shows two terminal windows side-by-side. The left window, titled 'FILES_CLIENT: client - Konsole', shows the execution of the client program with various parameters. The right window, titled 'FILES_SERVER: server - Konsole', shows the execution of the server program with various parameters. Both windows show the user 'ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT\$' and 'ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER\$' respectively.

```
FILES_CLIENT: client - Konsole
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT$ gcc -o client client.c -lpthread
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT$ ./client 192.168.48.129
Closing port number 10000
Opening new socket with port number 10001
Select an integer for Window transmitting
5
Select an integer from 0 to 100 for the probability of loss
20
Choose an integer for timer:
[0] for static timer
[1] for adaptive timer
0
You select [0] static timer. Please insert timeout interval:
5

***|| MENU ||***
[1] List
[2] Download
[3] Upload
[4] Exit

FILES_SERVER: server - Konsole
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ gcc -o server server.c -lpthread
ferdinando@ferdiKubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ ./server
socket created in server
Binded port: 10000
socket created in server
Binded port: 10001
Static timeout set to: 5.000000
```

● Comando List

```
FILES_CLIENT: client — Konsole
File Modifica Visualizza Segnalibri Estensioni Impostazioni Aiuto
ferdi@andorferd@ubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_CLIENT$ gcc -o client client.c -lpthread
Closing port number 10000
Opening new socket with port number 10001
Select an integer for Window transmitting
5
Select an integer from 0 to 100 for the probability of loss
20
Choose an integer for timer:
[0] for static timer
[1] for adaptive timer

0
You select [0] static timer. Please insert timeout interval:
5

***|| MENU ||***
[1] List
[2] Download
[3] Upload
[4] Exit

1
Contents of the list:
- index.png
- sample_lm.pdf
- lucky.jpeg
- smallpdf.pdf
- vaccini.txt
- sample.pdf
- liw.txt
- server.c
- seneca.txt
- maometto.m4a
- dante.txt
- index.jpeg
- server

***|| MENU ||***
[1] List
[2] Download
[3] Upload
[4] Exit

FILES_SERVER: server — Konsole
File Modifica Visualizza Segnalibri Estensioni Impostazioni Aiuto
ferdi@andorferd@ubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ gcc -o server server.c -lpthread
socket created in server
Blinded port: 10000
socket created in server
Blinded port: 10001
Static timeout set to: 5.000000
FILES ON SERVER:
- index.png
- sample_lm.pdf
- lucky.jpeg
- smallpdf.pdf
- vaccini.txt
- sample.pdf
- liw.txt
- server.c
- seneca.txt
- maometto.m4a
- dante.txt
- index.jpeg
- server
```

● Comando Download

```
FILES_CLIENT: client — Konsole
File Modifica Visualizza Segnalibri Estensioni Impostazioni Aiuto
0
You select [0] static timer. Please insert timeout interval:
5

***|| MENU ||***
[1] List
[2] Download
[3] Upload
[4] Exit

2
Contents of the list:
- index.png
- sample_lm.pdf
- lucky.jpeg
- smallpdf.pdf
- vaccini.txt
- sample.pdf
- liw.txt
- server.c
- seneca.txt
- maometto.m4a
- dante.txt
- index.jpeg
- server

Enter file name to download :
seneca.txt
NAME OF TEXT FILE RECEIVED : seneca.txt
Size of the file: 1842
Number of packet: 2

-----
Packet 1
IS_IN[1]== 0
WRITE 1024 BYTES
Packet received
ACK number: 1 SENT
-----

Packet 2
IS_IN[2]== 0
Packet received
WRITE 818 BYTES
ACK number: 2 SENT
END = 1

FILES_SERVER: server — Konsole
File Modifica Visualizza Segnalibri Estensioni Impostazioni Aiuto
ferdi@andorferd@ubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ ./server
socket created in server
Blinded port: 10000
socket created in server
Blinded port: 10001
Static timeout set to: 5.000000
FILES ON SERVER:
- index.png
- sample_lm.pdf
- lucky.jpeg
- smallpdf.pdf
- vaccini.txt
- sample.pdf
- liw.txt
- server.c
- seneca.txt
- maometto.m4a
- dante.txt
- index.jpeg
- server

NAME OF TEXT FILE TO SEND : seneca.txt
Reading file contents.
Size of the file : 1842
Packets to send: 2

-----
New entry for pkt 1 insert
READ 1024 BYTES
PACKET SENT N°:1
.....PACCHETTO 1      ACK = 1 .....
-----Start_Window = 1      End_Window = 2-----
Child Thread for packet 1 EXIT Normally.

New entry for pkt 2 insert
READ 818 BYTES
PACKET SENT N°:2
.....PACCHETTO 2      ACK = 2 .....
-----Start_Window = 1      End_Window = 2-----
Child Thread for packet 2 EXIT Normally.
```

● Comando Upload

```
FILES_CLIENT: client — Konsole
File Modifica Visualizza Segnalibri Estensioni Impostazioni Aiuto
[3] Upload
[4] Exit

3
AVAILABLE FILES ON CLIENT:
- client.c
- index.png
- sample_lm.pdf
- lucky.jpeg
- smallpdf.pdf
- sample.pdf
- liw.txt
- seneca.txt
- client
- maometto.mp4
- dante.txt
- index.jpeg

Enter file name to Upload:
seneca.txt
Reading file contents.
Size of the file : 1842
Packets to send: 2

-----
New entry for pkt 2 insert
READ 818 BYTES
PACKET SENT N°:2
.....PACCHETTO 2      ACK = 2 .....
-----Start_Window = 1      End_Window = 2-----
Child Thread for packet 2 EXIT Normally.

New entry for pkt 1 insert
READ 1024 BYTES
PACKET SENT N°:1
.....PACCHETTO 1      ACK = 1 .....
-----Start_Window = 1      End_Window = 2-----
Child Thread for packet 1 EXIT Normally.

***|| MENU ||***
[1] List
[2] Download
[3] Upload
[4] Exit

FILES_SERVER: server — Konsole
File Modifica Visualizza Segnalibri Estensioni Impostazioni Aiuto
ferdi@andorferd@ubuntu:~/Scrivania/Progetto IIM/IIM PROJECT - PREMIUM/FILES_SERVER$ ./server
socket created in server
Blinded port: 10000
socket created in server
Blinded port: 10001
Static timeout set to: 5.000000
NAME OF TEXT FILE RECEIVED : seneca.txt
Size of the file: 1842
Number of packet: 2

-----
Packet 2
WRITE 818 BYTES
Packet received
ACK number: 2 SENT
-----

Packet 1
WRITE 1024 BYTES
Packet received
ACK number: 1 SENT
```


● Comando Exit

```
FILES_CLIENT: bash - Konsole
[4] Exit
3
AVAILABLE FILES ON CLIENT:
- client.c
- index.png
- sample_lam.pdf
- lucky.jpeg
- smallpdf.pdf
- sample.pdf
- ilw.txt
- seneca.txt
- client
- maometto.mp4
- dante.txt
- index.jpeg

Enter file name to Upload:
seneca.txt
Reading file contents.
Size of the file : 1842
Packets to send: 2

-----
New entry for pkt 2 insert
READ 818 BYTES
PACKET SENT N°:2
-----PACKET 2-----ACK = 2-----
-----Start_Window = 1 End_Window = 2-----
Child Thread for packet 2 EXIT Normally.

-----
New entry for pkt 1 insert
READ 1024 BYTES
PACKET SENT N°:1
-----PACKET 1-----ACK = 1-----
-----Start_Window = 1 End_Window = 2-----
Child Thread for packet 1 EXIT Normally.

-----
***|| MENU ||***
[1] List
[2] Download
[3] Upload
[4] Exit
4
|ferdiLnando@ferdiKubuntu:~/Scrivania/Progetto IIV/IIV PROJECT - PREMIUM/FILES_CLIENT$

FILES_SERVER: server - Konsole
|ferdiLnando@ferdiKubuntu:~/Scrivania/Progetto IIV/IIV PROJECT - PREMIUM/FILES_SERVER$ ./server
socket created in server
Binded port: 10000
socket created in server
Binded port: 10001
Static timeout set to: 5.000000
NAME OF TEXT FILE RECEIVED : seneca.txt
Size of the file: 1842
Number of packet: 2

-----
Packet 2
WRITE 818 BYTES
Packet received
ACK number: 2 SENT
-----
Packet 1
WRITE 1024 BYTES
Packet received
ACK number: 1 SENT
Closing socket with port number 10001
Child 4987 exit with code 0
|
```

● Istruzioni per l'utilizzo

Lista comandi creazione Server/Client:

- **gcc -o client client.c -lpthread:** permette di compilare il client.
- **gcc -o server server.c -lpthread:** permette di compilare il server.

Lista comandi Client:

- **./client <ip_address>:** lancia il client connettendolo ad un **server già aperto in precedenza**
- In seguito viene chiesto all'utente di inserire i seguenti parametri:
 1. **N (int):** dimensione della window utilizzata dal selective repeat che è condivisa con il server.
 2. **p (int):** probabilità di perdita dei pacchetti e degli ack.
 3. **[0] o [1]** per scegliere il tipo di timer da utilizzare (0 per statico ed 1 per adattivo)
 - In caso venga scelto [0] viene chiesto all'utente di scegliere un (int) per determinare il timeout statico
- Successivamente appare un menù che permette all'utente di eseguire varie funzioni:
 - a. **[1]:** visualizzare la lista dei file presenti nel server.
 - b. **[2]:** effettuare il download di file presenti nel server.
 - successivamente verrà chiesto all'utente di selezionare il file che desidera scaricare.
 - c. **[3]:** effettuare l'upload di file presenti nel client verso il server.
 - successivamente verrà chiesto all'utente di selezionare il file che desidera caricare.
 - d. **[4]:** chiude il client.

Lista comandi Server:

- **./server:** lancia il server che risponderà con la porta in cui si è stabilita la connessione con il client.
 - se risponderà "Binded" allora la connessione è stata stabilita.
 - se risponderà "Not Binded" allora si è verificato un errore nella connessione.

