

Relazione progetto B: Trasferimento file su UDP A.A. 2018/2019

Briscese Filippo Maria, Dalain Samy, Ilardi Davide

1 Descrizione dell'architettura, scelte progettuali e limitazioni riscontrate:

Essendoci stato richiesto per la nostra applicazione di garantire la trasmissione affidabile dei messaggi e dei file su un servizio di comunicazione non affidabile, quale è UDP, è stato necessario implementare sia per il client che per il server a livello applicativo il protocollo di scambio di messaggi Selective Repeat con una finestra di spedizione di dimensione fissata (opportunamente modificabile). La prima questione che abbiamo affrontato è stata la ridefinizione, secondo una nostra convenzione, del campo dati del pacchetto UDP in maniera tale da poterlo adattare alle nostre esigenze implementative (che verranno descritte nel seguito). Si è in primis scelto di limitare il numero di bytes del campo dati ad un massimo di 1028 i quali vengono opportunamente divisi in ulteriori campi di lunghezza fissata: il campo "seqNum", costituito dai primi 4 bytes del campo dati, contiene il numero di sequenza del pacchetto, i successivi 3 bytes, per i soli messaggi di comando, sono riservati al campo "cmd" il quale contiene la stringa indicante il comando da mandare dal mittente al ricevente (i comandi utilizzabili sono "put", "ls" e "get"), mentre i restanti bytes contengono i dati effettivi. Il pacchetto UDP, così come è stato brevemente descritto, è riportato in Figura 1.

Per le scelte progettuali descritte si è quindi deciso di distinguere i pacchetti di comando dai pacchetti di dati: i pacchetti di comando hanno il campo "cmd" riservato alla stringa indicante il comando richiesto, mentre i pacchetti di dato hanno solamente il campo "seqNum" ed il campo "Dati".

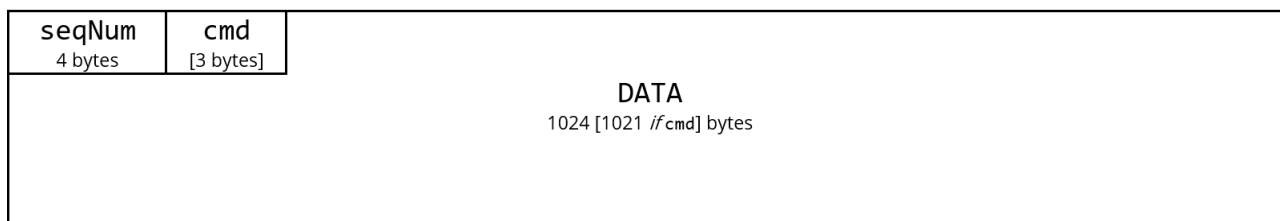


Figure 1: Campo dati UDP modificato utilizzando le nostre convenzioni

Una volta ridefinito il formato del campo dati si è proceduto quindi a realizzare un semplice Client UDP e Server UDP concorrenti per lo scambio di messaggi e di file. Realizzare un Server UDP concorrente non richiede, come accadeva con TCP, di effettuare semplicemente una fork() di un nuovo processo figlio che si occupi di gestire le richieste di un nuovo Client. Mentre in TCP ogni connessione di un Client è unica e quindi lo è anche la corrispondente socket, in UDP vi sono due differenti tipologie di Server concorrenti. La prima tipologia di Server è quella che dapprima legge un'unica richiesta di un Client e quindi manda una risposta: il Server che legge la richiesta del Client effettua la fork() di un figlio e gli "delega" la gestione di tale richiesta. La richiesta è passata al figlio nella sua memoria tramite la chiamata della fork() e questo procederà quindi a rispondere direttamente al Client richiedente. Il secondo tipo di Server scambia più di un pacchetto con un Client. In questo caso il problema risiede nel fatto che l'unica porta conosciuta dal Client è quella impostata inizialmente per il Server. Il Client manda il primo pacchetto di richiesta a tale porta ed il Server per distinguere il susseguirsi di pacchetti arrivatogli da tale Client e nuove richieste effettuatogli crea una nuova socket per ogni Client, effettua una bind() di una porta fittizia per quella socket e utilizza tale socket per tutte le sue richieste: il Client guarda alla porta indicata dalla prima risposta del Server e manda il successivo insieme dei pacchetti di richiesta a tale porta.

Per il nostro progetto è stata realizzata la seconda tipologia di Server UDP concorrente.

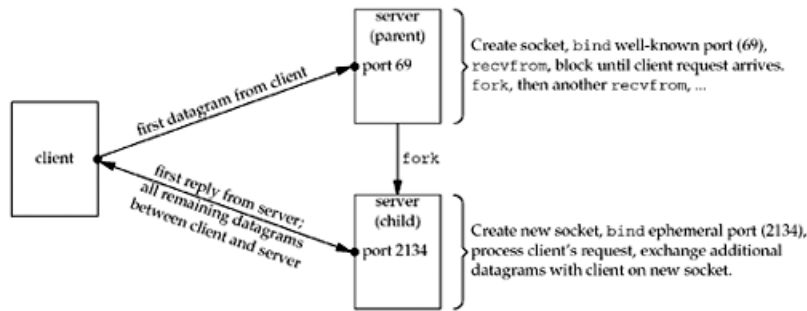


Figure 2: Schema di funzionamento di un Server UDP concorrente

Una volta creata la "base" per il nostro progetto, per procedere alla realizzazione del protocollo Selective Repeat abbiamo dovuto effettuare alcune scelte progettuali che riguardano principalmente la creazione e gestione della finestra di spedizione, la gestione del timer di rispedizione ed il controllo del ricevimento degli ACK da parte del mittente del pacchetto.

La finestra di spedizione del mittente è stata realizzata mediante una coda circolare di pacchetti di dimensione pari a quella della finestra (valore N fissato). La scelta di realizzare la finestra di spedizione del mittente con una coda è dovuta in primis al fatto che, così come è richiesto per il progetto da realizzare, i pacchetti vanno inoltrati in ordine verso il ricevente, quindi una coda con politica FIFO risulta particolarmente adatta a tale fine. Vi sono inoltre altri vantaggi per quanto riguarda le performance che ci hanno indotto a tale scelta. Un vantaggio riguarda la verifica, una volta concluso l'invio di tutti i pacchetti, che la finestra di spedizione sia vuota: per tale operazione questa implementazione richiede un ridotto costo computazionale pari a $O(1)$ mentre l'implementazione con un array di pacchetti avrebbe richiesto la scansione dell'intero array per effettuare tale verifica, quindi un costo computazionale di $O(N)$. Nell'implementazione della finestra di spedizione del mittente con un semplice array di pacchetti inoltre saremmo stati costretti ad introdurre delle variabili booleane per ogni pacchetto per verificare se si era incorsi in un timeout e per vedere se si era ricevuto un ACK: tale operazione era molto onerosa e richiedeva un costo computazionale nell'ordine di $O(N)$.

Prima di introdurre la coda circolare avevamo provato ad utilizzare una coda semplice e un array di appoggio. L'array d'appoggio introdotto serviva per il controllo dei riscontri in quanto l'accesso casuale tramite indice era più veloce che risalire i puntatori nella coda per trovare il pacchetto del quale si voleva modificare l'informazione sul riscontro, ma le operazioni di dequeue sulla coda (che corrispondono allo spostamento della finestra) implicavano lo spostamento dell'informazione contenuta nell'array di appoggio e per questa operazione il costo computazionale è pari ad $O(N)$. Per migliorare l'efficienza si è pensato quindi di utilizzare un array circolare che ha eliminato la necessità di spostare l'informazione all'interno dell'array. Successivamente si è deciso di non utilizzare una ulteriore struttura di appoggio oltre alla coda, ma di usare soltanto una coda circolare che mantiene indici di testa e coda, implementata con un array per salvare le informazioni (i pacchetti) al posto di avere per ogni elemento un puntatore al suo successivo. In questo modo si ha una coda circolare che sfrutta anche l'accesso tramite indice ai suoi elementi.

Si è inizialmente deciso di far verificare e gestire le trasmissioni ad un unico thread e la verifica del timeout (ed eventuale ritrasmissione) per i singoli pacchetti direttamente a dei threads, uno per ogni pacchetto inserito nella finestra di spedizione, per ridurre i controlli e i compiti da far effettuare al thread principale.

Questo approccio richiedeva necessariamente l'utilizzo di risorse condivise e quindi di semafori per la gestione della concorrenza. Inoltre, per l'esecuzione parallela e sincrona del thread che gestiva il timer, andavano introdotti un mutex ed una condition per utilizzare il `condwait()`. Visto che la gestione del parallelismo, della condivisione e della schedulazione aggiungeva complessità si è deciso di cambiare implementazione. La nuova implementazione da noi realizzata riduce la complessità mediante l'uso della funzione `select()` con timeout nullo: viene effettuato il controllo del descrittore di socket in lettura e, qualora pronto, viene effettuata la lettura del dato ricevuto o, in caso contrario, viene effettuata la trasmissione del prossimo pacchetto in finestra o di un pacchetto da ritrasmettere (la ritrasmissione è privilegiata rispetto alla trasmissione).

La finestra di spedizione del destinatario è stata inizialmente realizzata con un array di pacchetti di dimensione pari ad N in quanto, seguendo i principi del protocollo Selective Repeat, i pacchetti possono arrivare in disordine, quindi ci era risultato comodo a tal fine mantenere i pacchetti nelle "celle" di un array (ed utilizzare l'indice per ordinarli) in attesa di essere mandati al livello superiore piuttosto di utilizzare una coda semplice come per il mittente.

Quindi, come descritto in precedenza, le finestre di spedizione inizialmente erano realizzate in modo differente: rispettivamente con una coda semplice per il mittente ed un array di pacchetti per il destinatario. Si è in seguito

proceduto ad uniformare le due finestre di spedizione relizzandole con una coda circolare di pacchetti in cui viene mantenuto un "indice di testa" ed un "indice di coda" al fine di muoversi lungo l'array di riferimento della coda. Nella definizione del timer fisso si è preferito utilizzare un tipo di timer con crescita esponenziale all'aumentare del numero di ritrasmissioni piuttosto che un timer con una scadenza univoca e fissata. Il parametro T definito dall'utente in compilazione viene utilizzato come timer di partenza che dunque varierà il proprio valore secondo la legge esponenziale riportata qui di seguito:

$$T = T_{prev} * 2^r$$

Si è scelto di utilizzare questa tipologia di legge per evitare che l'inserimento di un valore del timer T troppo piccolo (inferiore al RTT) potesse portare all'impossibilità di completare la trasmissione dei pacchetti al destinatario e la relativa trasmissione dell'ACK al mittente prima di una loro ritrasmissione. Quindi, in presenza di eventi di perdita il timer raddoppia il proprio valore e nell'intervallo che intercorre tra una perdita e la successiva questo mantiene un valore fisso pari a T . Lo svantaggio di questa scelta, come verrà descritto nel seguito, è dato dal decremento esponenziale delle prestazioni al crescere del numero di perdite ma si ha il vantaggio di avere una rapida "uscita dalla condizione di stallo" causata da continue ritrasmissioni nel caso in cui l'utente inserisca valori molto bassi del valore iniziale del parametro T .

Per il caso del "Timer adattativo" si è scelto di applicare la stessa logica realizzativa del "Timer di ritrasmissione" del protocollo TCP. Quindi si ha che:

$$EstimatedRTT = (1 - a) * EstimatedRTT_{prev} + a * SampleRTT$$

con $a = 1/8$

$$DEV = (1 - b) * DEV_{prev} + b * |(SampleRTT - EstimatedRTT)|$$

con $b = 1/4$

da cui si ricava il valore del timer:

$$TimeoutInterval = EstimatedRTT + 4 * DEV$$

2 Descrizione dell'implementazione

Come descritto nella sezione precedente, dopo aver realizzato un semplice Client e Server per lo scambio di messaggi, nello stesso modo in cui ci era stato presentato a lezione, si è proceduto ad aggiungere via via nuove "funzionalità" richieste dalla traccia. Per prima cosa dovevamo "scompattare" un file (dopo averlo aperto e dopo aver verificato la sua esistenza) in pacchetti in maniera tale che il Client potesse inserirli, una volta implementata, nella finestra di spedizione. Per effettuare tale operazione sono state introdotte due funzioni: `fileDivide()` e `readFileBlock()`. La prima tra le due funzioni calcola dapprima la dimensione del file posizionando l'indicatore di posizione alla fine di esso tramite la chiamata di `fseek()` ed `ftell()`, viene quindi ottenuto il numero di pacchetti complessivi come la parte intera superiore del rapporto tra la dimensione del file e la dimensione del campo dati, fissata a 1024 bytes. La seconda tra le due funzioni legge invece blocchi di 1024 bytes dal file che gli viene passato, ne inserisce il contenuto all'interno di un buffer e ritorna il numero di bytes letti.

Una volta gestita la lettura del file e la divisione in pacchetti abbiamo definito il tipo di dato `Packet` come una struct contenente il numero di sequenza, il dato (da passare alla `sendto()`) definito come un puntatore a `char`, un intero indicante la dimensione del dato, un intero (`pktState`) indicante lo stato del pacchetto, una struttura `timeval` per mantenere il tempo di invio ed una per mantenere il timer di invio (calcolato esclusivamente nel caso di timer adattativo). La variabile `pktState` è stata introdotta al fine di identificare lo stato del pacchetto considerato: se il suo valore è pari a -1 il pacchetto è nello stato di "Timeout", se il valore è pari a 0 il pacchetto è non è stato ancora inviato, se il valore è 1 il pacchetto è stato ricevuto e se pari a 2 il pacchetto è "in trasmissione".

La scelta di utilizzare la struttura `Packet` per i pacchetti ci ha aiutato a mantenere le informazioni utili al trasferimento dei blocchi del file ed implementare il protocollo Selective Repeat (nelle finestre di spedizione si sono inseriti dei pacchetti).

A questo punto siamo partiti all'implementazione del protocollo Selective Repeat. Come riportato nella sezione 1, si è scelto di implementare la finestra di spedizione del mittente come una coda circolare. A livello implementativo l'introduzione della coda ci ha indotto a realizzare le operazioni di "push" e "pull" (che sono state tradotte nelle funzioni chiamate rispettivamente `enqueue()` e `dequeue()`) ed ad introdurre ulteriori funzionalità tramite altre funzioni atte ad inizializzare la coda (`initQueue()`), stamparne il contenuto (`printQueue()`) e verificare se essa sia vuota (`isEmpty()`) o piena (`isFull()`).

La funzione `enqueue()` è stata utilizzata inizialmente per la prima operazione di riempimento della finestra: tale operazione (inserita nel file sorgente `clientfunc.c`) è stata realizzata in un ciclo `while` che effettua iterazioni fino a quando il numero di pacchetti in coda non è pari alla dimensione della finestra e fino a quando una variabile indicante il numero di pacchetti che costituiscono il file da inserire non si azzerava. All'interno di tale ciclo `while` viene inizializzato un pacchetto (per effettuare questa operazione abbiamo introdotto una funzione apposita chiamata `initializePacket()`), effettuata l'operazione di push nella coda, incrementato il numero di sequenza relativo al prossimo pacchetto di una unità rispetto al precedente (il valore del primo numero di sequenza è generato randomicamente) e decrementato il valore relativo al numero di pacchetti del file da inserire in finestra.

Una volta occupatoci dell'implementazione del pacchetto e della finestra di spedizione, si è proceduto con l'implementazione del comando `put` per il Client (identico al caso `get` per il Server) e si è cercato di trovare il modo adatto di permettere a quest'ultimo di inviare pacchetti e ricevere nel frattempo i riscontri di quelli che sono stati già spediti. A questo scopo è stato utile sfruttare la funzione `select` con timeout nullo per controllare lo stato del descrittore del socket in lettura. Nel caso in cui il descrittore è pronto, si procede a leggere il riscontro del pacchetto:

```

1 while (nPktToAck > 0){
2     timeout2.tv_sec = 0;
3     timeout2.tv_usec = 0;
4     FD_ZERO(&set);
5     FD_SET(sockfd, &set);
6     while(1){
7         n = select(sockfd + 1, &set, NULL, NULL, &timeout2);
8         if (n < 0){
9             perror("Errore in select");
10            return -1;
11        }
12        // Se ho qualcosa da leggere sul socket lo leggo
13        else if (n > 0){
14            seqnum = getACKseqnum(sockfd, ack, sizeof(ack));
15            printf("<< Ricevuto il riscontro del pacchetto con numero di sequenza %d >>\n", seqnum);
16            ...

```

Se il riscontro era già stato ricevuto (può accadere se il timer scade prima di ricevere il riscontro) allora si continua nella ricezione di nuovi riscontri.

```

1 // Gestisce il caso in cui il timer è troppo basso e il riscontro è stato ricevuto
2 if ((seqnum - (int)queue.pkt[queue.head].seq_num) < 0) {
3     printf("Riscontro del pacchetto numero %d già ricevuto!\n", seqnum);
4     continue;
5 }
6 int position = (queue.head + (seqnum - queue.pkt[queue.head].seq_num)) % queue.size;
7 if (queue.pkt[position].pkt_state == 1){
8     printf("Riscontro del pacchetto numero %d già ricevuto!\n\n", seqnum);
9     continue;
10 }
11 ...

```

Inoltre, se il timer è stato scelto adattativo e se il pacchetto relativo al riscontro non ha fatto timeout, si prende il tempo di ricezione e si aggiorna il timer da impostare per i prossimi pacchetti (l'aggiornamento del timer segue la stessa logica di TCP).

```

1 ...
2 if (queue.pkt[position].pkt_state != -1 && timeout_user == 0){
3     printf("Tempo di invio: %ld\n", queue.pkt[position].send_time.tv_usec);
4     timersub(&(current_time), &(queue.pkt[position].send_time), &sample_RTT);
5     timer = generate_timeout(&sample_RTT, &estimated_RTT, &dev, timeout_user);
6 }
7 queue.pkt[position].pkt_state = 1;
8 nPktToAck--;
9 }
10 // Se non ho più nulla da leggere allora invio
11 else if (n == 0) break;

```

Si controlla anche se bisogna eventualmente spostare la finestra. Lo spostamento della finestra di `n` posizioni equivale ad `n` operazioni di `dequeue` ed al più `n` operazioni di `enqueue`.

```

1 // Spostamento della finestra
2 i = 0;
3 while(queue.pkt[queue.head].pkt_state == 1){

```

```

4     i++;
5     dequeue(&queue);
6     if (nPkt > 0){
7         initializePacket(&pkt, pktseq, fp);
8         enqueue(&queue, pkt);
9         pktseq++;
10        nPkt--;
11    }
12 }

```

Nel caso in cui il descrittore non è pronto in lettura, allora si controlla prima se ci sono pacchetti in timeout tra quelli inviati (con pktState = 2) controllando se l'ultimo tempo di ricezione di un pacchetto (relativo all'ultima volta che il descrittore è stato pronto in lettura) sia maggiore della somma del tempo di invio e il timer per ogni pacchetto. Se si verifica un timeout allora viene ritrasmesso il pacchetto corrispondente e si ritorna nella select per il controllo dello stato descrittore in lettura.

```

1 // Prima controllo i timeout
2 if (gettimeofday(&(current_time), NULL) == -1){
3     perror("Errore: gettimeofday");
4     exit(1);
5 }
6 i = queue.head;
7 int check = 0;
8 while(1){
9     if (queue.pkt[i].pkt_state != 1 && queue.pkt[i].pkt_state != 0){
10        timersub(&current_time, &(queue.pkt[i].send_time), &elapsed_time);
11        elapsed_time.tv_usec = elapsed_time.tv_usec + 1e6 * elapsed_time.tv_sec;
12        elapsed_time.tv_sec = 0;
13        if (timercmp(&elapsed_time, &(queue.pkt[i].timer), >) != 0){
14            printf("\n!!! Pacchetto con numero di sequenza %d TIMEOUT !!!\n", queue.pkt[i].seq_num);
15            ;
16            queue.pkt[i].pkt_state = -1;
17            timeradd(&(queue.pkt[i].timer), &(queue.pkt[i].timer), &(queue.pkt[i].timer));
18            // Ritrasmissione
19            memcpy(buff, &(queue.pkt[i].seq_num), 4);
20            memcpy(buff + 4, queue.pkt[i].data, queue.pkt[i].dataSize);
21            printf("\n\nPacchetto seqnum = %d in timeout rinviato con timer di %ld microsec\n\n",
22                queue.pkt[i].seq_num, queue.pkt[i].timer.tv_usec);
23            if (gettimeofday(&(queue.pkt[i].send_time), NULL) == -1){
24                perror("Errore: gettimeofday");
25                exit(1);
26            }
27            sim_sendto(sockfd, buff, queue.pkt[queue.cursor].dataSize + 4, (struct sockaddr *) &
28                servAddr, servAddrLen, p_loss);
29            check = 1;
30            break;
31        }
32    }
33    if (i == queue.cursor) break;
34    i = (i + 1) % queue.size;
35 }
36 if (check == 1) continue;

```

Qualora invece non risultasse nessun timeout, si procede con la trasmissione del prossimo pacchetto in finestra. Per tenere traccia del "prossimo" pacchetto da inviare si è aggiunto un attributo nella struttura della coda circolare (long int cursor) che ne mantiene l'indice (esso viene quindi incrementato ogni volta che si spedisce un pacchetto in finestra).

```

1 if (!isEmpty(&queue)){
2     if (queue.pkt[queue.cursor].pkt_state == 0){
3         queue.pkt[queue.cursor].timer = timer; // Imposto il timer
4         memcpy(buff, &(queue.pkt[queue.cursor].seq_num), 4);
5         memcpy(buff + 4, queue.pkt[queue.cursor].data, queue.pkt[queue.cursor].dataSize);
6         printf("\n\nPacchetto seqnum = %d inviato con timer di %ld microsec\n\n", queue.pkt[queue
7             .cursor].seq_num, queue.pkt[queue.cursor].timer.tv_usec);
8         queue.pkt[queue.cursor].pkt_state = 2; // Pacchetto in trasmissione
9         printf("##### Timer avviato per il pacchetto %d #####\n", queue.pkt[queue.cursor].seq_num
10            );
11    }
12    // Prendo il tempo di invio
13    if (gettimeofday(&(queue.pkt[queue.cursor].send_time), NULL) == -1){

```

```

12         perror("Errore: gettimeofday");
13         exit(1);
14     }
15     // Invio
16     sim_sendto(sockfd, buff, queue.pkt[queue.cursor].dataSize + 4, (struct sockaddr *) &
servAddr, servAddrLen, p_loss);
17     queue.cursor = (queue.cursor + 1) % queue.size;
18 }
19 }
20 }

```

Si è utilizzata a riga 16 la funzione `sim_sendto()` che è una `sendto()` in cui è inglobata la probabilità di perdita del pacchetto: se la probabilità di perdita del pacchetto (P) è maggiore della quantità fissata dall'utente allora viene stampata una stringa che indica la perdita del pacchetto, altrimenti viene inviato il pacchetto tramite la chiamata di `sendto()`.

Per la valutazione della perdita o meno del pacchetto è stata realizzata la funzione `is_pkt_lost()` che parte generando un numero random tra 0 e 99: la probabilità che tale numero random sia strettamente minore del valore che fissa la probabilità di perdita del pacchetto (ossia P) è esattamente pari a P (se ad esempio $P = 40$, infatti un evento del tipo "ho pescato un numero tra 0 e 99 minore di 40" si verifica con probabilità del 40 %) , quindi se tale evento si verifica la funzione torna 1, altrimenti torna 0.

Alla fine, dopo aver inviato tutto il file, si invia il messaggio di fine file ("ENDOFFILE") ed utilizzando il timeout con la funzione `select` si ritrasmette il messaggio in caso non si riceve il riscontro di fine file a fine timer.

Per far in modo che il ricevente identificasse la fine della trasmissione del file si era pensato anche di trasmettere un primo pacchetto che contenesse l'informazione per la dimensione del file da inviare (ovvero il numero di pacchetti che il ricevente si deve aspettare). Entrambi i metodi hanno funzionato, sebbene ci sia una considerazione da fare: per la soluzione del messaggio di fine file, se le informazioni di un generico pacchetto inviato tramite `sendto()` sul socket iniziassero con una stringa "ENDOFFILE" (improbabile) allora il trasferimento del file fallirà. Anche se improbabile, l'altra soluzione eliminerebbe qualsiasi possibilità che si verifichi questo evento.

Dalla parte del ricevente (caso put del server e caso get del client) invece la finestra di ricezione è implementata con la stessa struttura della finestra di spedizione, ma si è aggiunta un'operazione `insert()` da usare al posto di `enqueue()` che inserisce un pacchetto in una posizione a scelta all'interno dell'array della coda circolare per far fronte all'arrivo disordinato di pacchetti. Il ricevente aspetta i pacchetti e quando ne arriva uno trasmette il riscontro e controlla se il pacchetto non sia stato già ricevuto, in caso non lo sia allora lo salva tramite `insert()`.

```

1 // ricevo un pacchetto (seqN|data)
2 if ((nBytes = recvfrom(sockfd, recv_buff, MAXLINE + 4, 0, (struct sockaddr *)&servAddr, &
servAddrLen)) < 0) {
3     perror("errore in recvfrom");
4     //exit(1);
5     return -1;
6 }
7 printf("\n\t#####\n\t# Bytes ricevuti dal server: %d\n\t
#####\n\n", nBytes);
8 if (strncmp(recv_buff, "ENDOFFILE", 9) == 0) break; // se ricevo ENDOFFILE esco da while
9
10 // mando ack
11 memcpy(ack, recv_buff, sizeof(int));
12 sim_sendto(sockfd, ack, sizeof(ack), (struct sockaddr *)&servAddr, servAddrLen, p_loss);
13
14 // metto in finestra di ricezione
15 memcpy(&(pkt.seq_num), ack, sizeof(int));
16 printf("<< Mandato ack del pacchetto con numero di sequenza %d >>\n", pkt.seq_num);
17 memcpy(pkt.data, recv_buff + sizeof(int), nBytes - 4);
18 pkt.dataSize = nBytes - 4;
19 if(pkt.seq_num - start_seqN >= 0){
20     insert(&queue, pkt, pkt.seq_num - start_seqN);
21     printf("Pacchetto numero %d inserito in finestra in posizione %ld\n", pkt.seq_num, (queue.
head + pkt.seq_num - start_seqN) % queue.size);
22 }
23 else continue;

```

Ogni volta che il ricevente salva un pacchetto in finestra scrive su file il contenuto informativo del campo data dei pacchetti partendo dall'inizio della finestra fino a che non trova una cella vuota, dopodichè si rimette in ricezione dei riscontri.

```

1 i = queue.head;

```

```

2 while(queue.pkt[i].seq_num != -1){
3     printf("Scrivo su file i dati del pacchetto %d\n", queue.pkt[i].seq_num);
4     fwrite(queue.pkt[i].data, 1, queue.pkt[i].dataSize, fp);
5     queue.pkt[i].seq_num = -1;
6     dequeue(&queue);
7     start_seqN++;
8     i = (i + 1) % queue.size;
9 }

```

Quando il contenuto informativo del buffer in ricezione è la stringa speciale "ENDOFFILE" il ricevente chiude il file e smette di ricevere pacchetti, e per terminare invia il riscontro di fine file. Questo passaggio viene eseguito in una select con timeout di 2 secondi perchè anche il riscontro si può perdere, perciò il client ritrasmetterà il messaggio di fine file (ha un timer di circa 1 secondo). Se il ricevente non riceve più nulla per un tempo maggiore di 2 secondi allora il client deve aver ricevuto il riscontro di fine file.

```

1
2 fprintf(stdout,"Invio ACK per fine file\n");
3 memcpy(ack, "ACK", 3);
4 n = 1;
5 while(n){
6     timeout2.tv_sec = 2; // aspetta 2 secondi per capire se il client ha ricevuto ack di fine
        file
7     timeout2.tv_usec = 0;
8     FD_ZERO(&set);
9     FD_SET(sockfd, &set);
10    sim_sendto(sockfd, ack, 3, (struct sockaddr *)&servAddr, servAddrLen, p_loss);
11    n = select(sockfd + 1, &set, NULL, NULL, &timeout2); // se si verifica il timeout n = 0
12    if (n < 0){
13        perror("Errore in select");
14        return -1;
15    }
16    if (n > 0){ // n > 0: c'e qualcosa da leggere sul socket
17        if (recvfrom(sockfd, recv_buff, 9, 0, (struct sockaddr *)&servAddr, &servAddrLen) < 0) {
18            perror("errore in recvfrom");
19            exit(1);
20        }
21    }
22 }

```

Per identificare i casi di disconnessione si è utilizzata la setsockopt per impostare un timeout in ricezione.

Successivamente si è proceduto all'implementazione del comando list. Nella realizzazione di tale comando vengono eseguiti i seguenti passi:

1. il client manda un pacchetto al server con campo cmd = "ls"
2. il server inserisce in una stringa di testo i nomi dei file presenti nella cartella FILES all'interno di SERVER e la manda al client
3. il client stampa sul terminale il contenuto della stringa ricevuta.

Infine, l'implementazione del server concorrente è stata effettuata seguendo la logica accennata nella sezione 1. Il server concorrente è in grado di gestire un numero limitato di connessioni contemporanee, definito mediante il parametro MAX_N_CHILD in utils.h. Il processo principale, lanciato con ./server, è in continuo ascolto sulla sua porta e alla ricezione di un messaggio genera un processo figlio a cui delegare la gestione della richiesta del client: il figlio dunque si aprirà una nuova socket propria, diversa dal padre, per poter comunicare con il client "affidatogli" dal padre. La generazione dei figli è limitata usando un semaforo (di tipo sys/sem.h) che viene controllato dal padre prima di mettersi in ascolto e incrementato e decrementato rispettivamente alla morte e nascita di un figlio.

3 Tests e considerazione sui risultati ottenuti

Dopo aver implementato tutto ciò che ci era stato richiesto ci siamo occupati di testare che il sistema. La fase di test è stata realizzata mediante il software Open Source "Wireshark", il quale viene utilizzato per l'analisi del traffico di pacchetti in rete. Al fine di visualizzare i pacchetti inviati dalla macchina su cui viene fatto girare il software verso se stessa è stata utilizzata l'interfaccia "Loopback". I tests effettuati riguardano l'analisi del traffico di rete utilizzando diverse tipologie di file (file di testo, immagini, eseguibili, ...) da trasferire: sono stati inviati comandi di "put", "ls" e "get" e si è monitorato il corretto trasferimento dei pacchetti, il tempo di trasferimento di questi, il verificarsi degli eventi di timeout e la corretta ricezione degli ACK. Si è dunque deciso di realizzare alcuni

grafici per valutare le prestazioni del protocollo al variare della dimensione della finestra (N), della probabilità di perdita dei messaggi (P) e del valore T iniziale fissato per il timer. In figura 3 è riportato il grafico che mette in relazione il throughput (pkt/s) e la dimensione della finestra di spedizione.

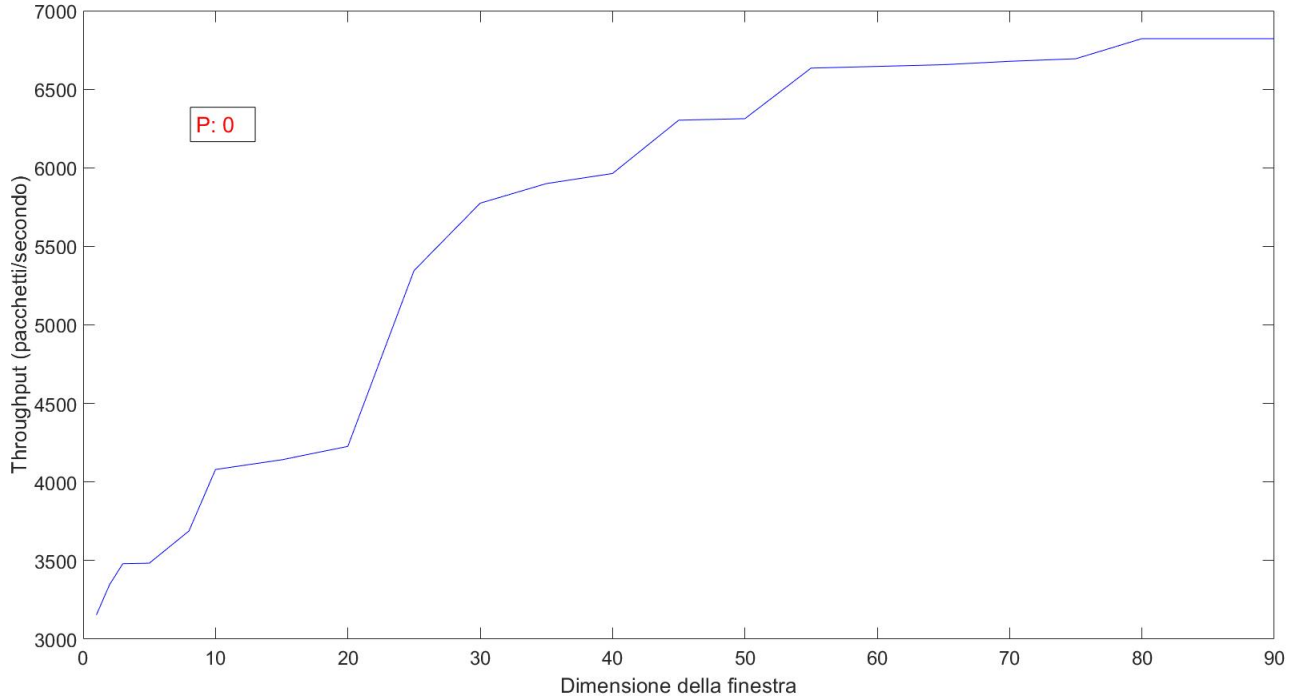


Figure 3: Grafico Dimensione della finestra - Throughput (P = 0)

Per realizzare tale grafico si sono effettuati 10 trasferimenti dello stesso file F in cui si è posta la probabilità di perdita P a 0 (al fine di non influenzare il valore del Throughput a causa della perdita di pacchetti), utilizzato sempre lo stesso tipo di timer (fisso) e fatto variare esclusivamente la dimensione della finestra di spedizione. I valori ottenuti per il tempo totale di trasferimento (relativi ad ogni trasferimento) del file F sono stati registrati e ne è stata fatta la media e quindi da questa calcolato il valore del Throughput relativo alla dimensione della finestra N fissata. La procedura precedentemente descritta è stata effettuata per tutti i valori di N riportati nel grafico (per ogni valore N si sono effettuati i 10 trasferimenti), quindi è stato passato il file Excel con tutti i dati a Matlab, fatta la media sui valori con lo stesso parametro N e realizzato il plot del risultato ottenuto. Da questo grafico risulta evidente che, al crescere della dimensione della finestra, nelle condizioni precedentemente descritte, il throughput cresce fino ad assestarsi attorno al valore 7000 pkts/s dalla dimensione della finestra pari ad 80 in poi. Per dare una giustificazione del risultato ottenuto indichiamo con K un pacchetto contenuto in posizione 1 nella finestra e con J l'ultimo pacchetto contenuto nella finestra di spedizione in cui è contenuto K. Indichiamo quindi con τ_J l'istante di tempo in cui è stato trasmesso il pacchetto J e con τ_K^A l'istante di tempo in cui il mittente riceve l'ACK relativo al pacchetto K:

- se $\tau_K^A > \tau_J$ aumentare le dimensioni della finestra aumenta l'efficienza poichè posso inviare un numero maggiore di pacchetti nell'attesa di ricevere l'ACK di K. Arrivati ad una certa dimensione della finestra si tenderà a cadere nel caso $\tau_K^A \leq \tau_J$ data la crescita del numero di pacchetti da inviare.

- Se $\tau_K^A \leq \tau_J$ aumentare le dimensioni della finestra non altera l'efficienza poichè è possibile inviare il pacchetto che segue J prima di τ_J dato che l'arrivo dell'ACK di K permette l'avanzamento della finestra.

In figura 4 è riportato il grafico che mette in relazione il throughput (pkt/s) e la probabilità di perdita (simulata) scelta dall'utente.

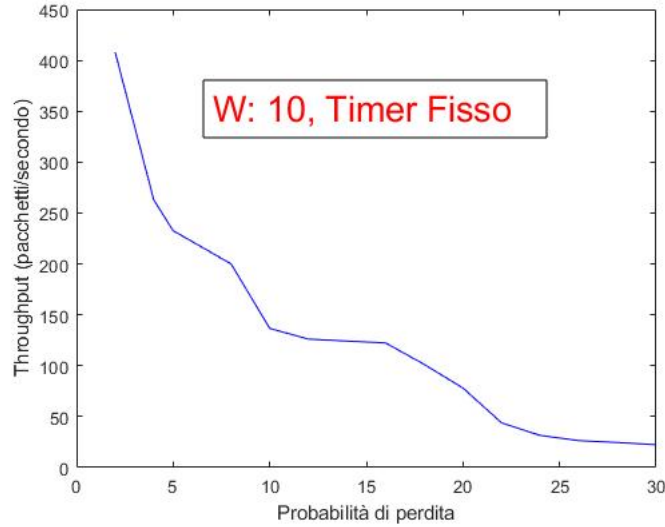


Figure 4: Grafico Probabilità di perdita - Throughput

Per realizzare tale grafico si sono effettuate varie trasmissioni in cui si è posta la probabilità di perdita P variabile, si è utilizzato sempre lo stesso tipo di timer (fisso) e mantenuta costante la dimensione della finestra ad $N = 10$. Come nel test descritto in precedenza, sono stati inizialmente trasferiti per un numero fisso di volte e con lo stesso valore della finestra i pacchetti relativi ad un certo file F , dunque sono stati registrati i valori totali di trasferimento ottenuti, ne è stata fatta la media e da questa calcolato il valore del throughput relativo alla probabilità di perdita P fissata. La procedura precedentemente descritta è stata effettuata per tutti i valori di P riportati nel grafico, quindi è stato passato il file Excel con tutti i dati a Matlab e realizzato il plot del risultato ottenuto. Il grafico risultante è una funzione decrescente che parte da un valore pari a circa 429 pacchetti/secondo (il primo valore scelto per realizzare il grafico è $P = 2$), arriva a circa 46 pacchetti/secondo per $P = 23$ e da quel valore in poi decresce più lentamente fino a scendere a circa 22 pacchetti/secondo per $P = 30$. Questo andamento ottenuto sperimentalmente è in sintonia con quello che si potrebbe facilmente intuire: se la probabilità di perdita cresce deve crescere anche il tempo di medio di trasmissione del file F ed essendo questa quantità inversamente proporzionale al Throughput questo deve decrescere al crescere di quest'ultima. Ovvero, indicando con C il numero totale di pacchetti trasmessi (relativi al file F), con T_t il tempo totale di trasmissione di F e con P la probabilità di perdita, è valida la seguente relazione:

$$Throughput = \frac{C}{T_t(P)}$$

Quindi, come riporta il grafico, per valori elevati di P (nel nostro caso $P > 30$) il Throughput risulta quasi azzerarsi a causa dell'aumento elevato del tempo di trasmissione del file F .

In figura 5 è riportato il grafico che mette in relazione il throughput (pkt/s) ed il timer T iniziale nel caso Timer Fisso (con $P=5$), in figura 6 è riportato lo stesso grafico per il caso adattativo ed insieme ad esso l'andamento che si ha in assenza di perdite.

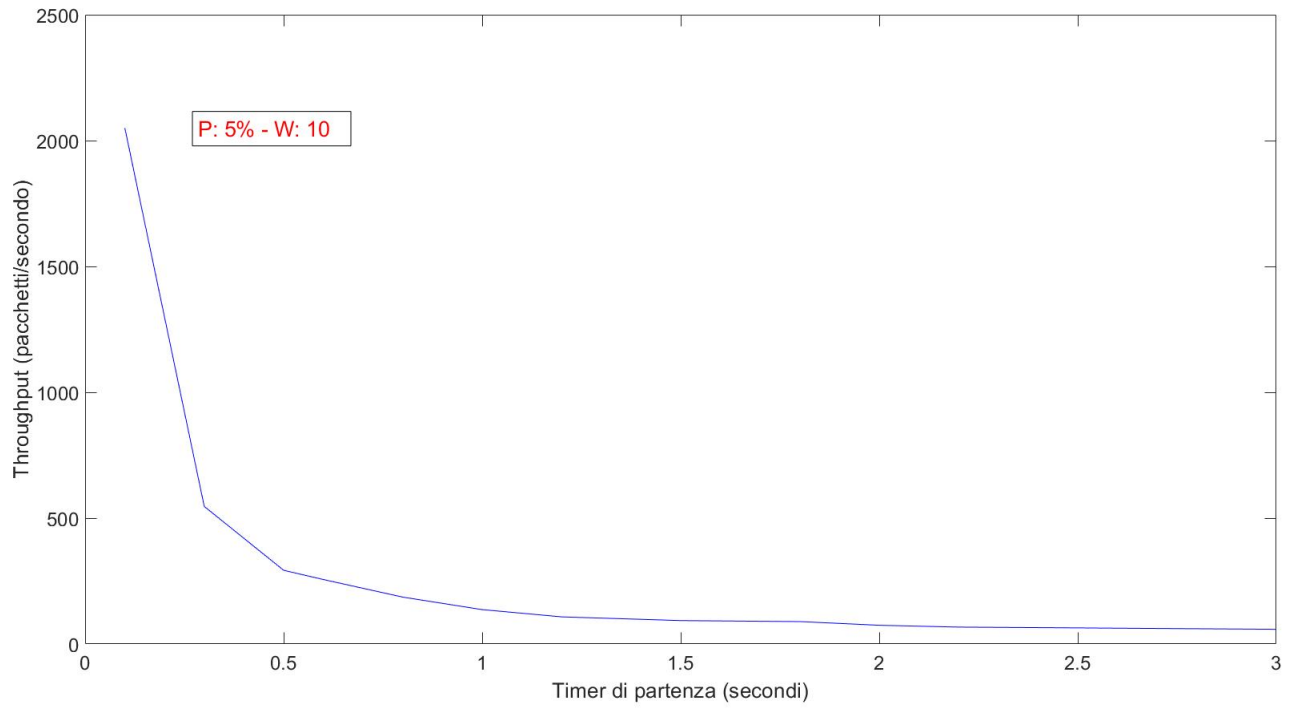


Figure 5: Throughput - Timer di partenza ($P = 5$, caso Timer Fisso)

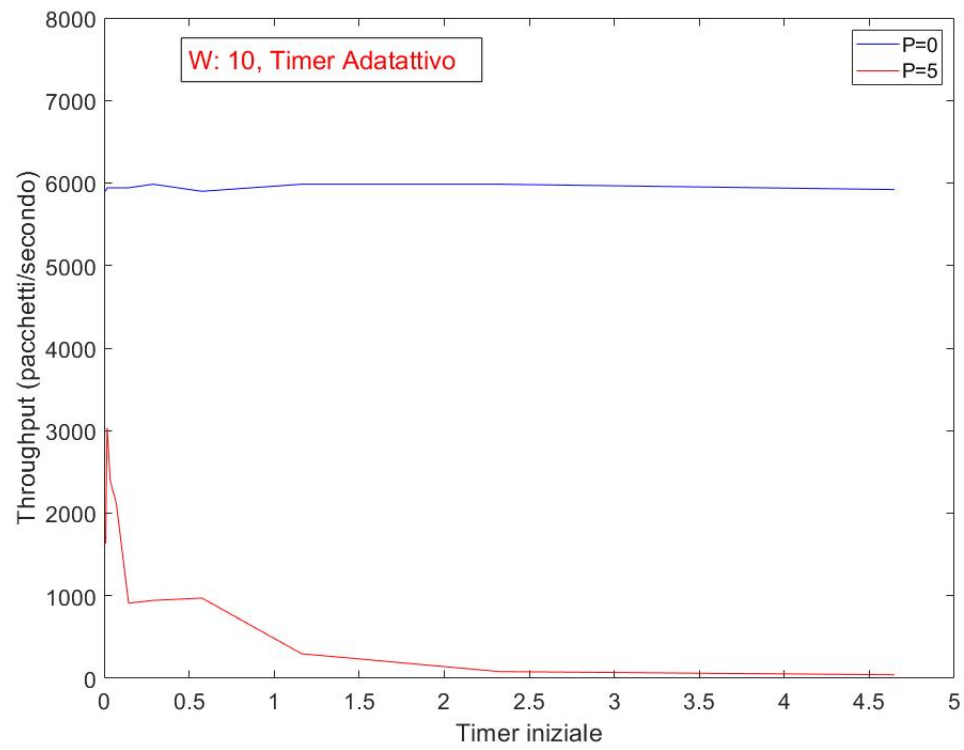


Figure 6: Throughput - Timer di partenza ($P = 5$, caso Timer Adattativo)

La logica adottata per la creazione dei grafici è la stessa dei casi precedenti ad eccezione del fatto che viene fatto variare questa volta il Timer iniziale T . Nella nostra implementazione dopo l'invio del primo pacchetto il valore del timer viene modificato ed in particolare raddoppiato nel caso si verifichino eventi di perdita (sia per il timer fisso che per quello adattativo) oppure adattato alle condizioni della rete (per il solo timer adattativo) qualora venga ricevuto un riscontro di un pacchetto inviato.

Il grafico in figura 5 mostra come al crescere del timer T iniziale si abbia, per il caso di Timer Fisso, una riduzione elevata del Throughput: vengono raggiunti valori prossimi a 2000 pkt/sec per $T = 0.1$ sec e prossimi a 50 pkt/sec per $T = 3$ sec. Il risultato ottenuto è quello che ci aspettavamo: se il timer iniziale viene fatto crescere si ha che più si verificano eventi di perdita e più potenzialmente un pacchetto perso può dover "attendere" un tempo maggiore per essere ritrasmesso e quindi, riducendosi il numero di pacchetti al secondo trasmessi, il Throughput decresce.

Per giustificare il risultato ottenuto, indicando con t_i il tempo totale di trasmissione e ricezione dell'ACK di un pacchetto i , con v_t la velocità di trasmissione e con r il numero di ritrasmissioni di tale pacchetto, si ha il seguente legame:

$$t_i = f(dim(pkt), v_t, \dots, T * 2^r)$$

con $dim(pkt)$ la dimensione (da noi fissata) dei pacchetti. Tra le dipendenze di t_i è stato inserito il fattore $T * 2^r$ poichè il timer raddoppia il suo valore ad ogni ritrasmissione: essendo il Throughput inversamente proporzionale alla sommatoria dei t_i relativi ai pacchetti del file F questo deve dunque decrescere all'aumentare del fattore $T * 2^r$, come accade in figura 5.

Il grafico in figura 6 mostra che, al crescere del parametro T (ottenuto scegliendo un opportuno valore iniziale di RTT ed impostando dev alla sua metà), il Throughput decresce (per $P > 0$) e tende asintoticamente a zero. Questo andamento è dovuto al fatto che, a differenza del caso $P = 0$ in cui non si hanno perdite e dunque non avviene alcun evento di Timeout, il valore del parametro T per $P > 0$, influenzando le stime iniziali di RTT e dev, influenza i valori dei timer successivi fino a quando le stime di RTT e dev si assestano al valore reale. La perdita di un pacchetto tra quelli iniziali genererà, al crescere di T , un tempo via via maggiore di attesa per la sua ritrasmissione incrementando il tempo totale di trasmissione del file F . Essendo il Throughput inversamente proporzionale al tempo totale di trasmissione, si ha che questo ridurrà di conseguenza il proprio valore all'aumentare del parametro T , tendendo a zero per T elevati. Si ha inoltre che, all'aumentare della probabilità di perdita P e del numero di pacchetti, a causa della maggior perdita media si avrà una conseguente maggiore influenza del parametro T sul singolo trasferimento del file F .

4 Manuale d'uso

Per compilare il client e il server usare il comando `make` su un terminale aperto nella cartella principale del progetto, oppure per compilare solo il client o il server usare rispettivamente `make client` o `make server`.

Per lanciare il client aprire un terminale dalla cartella `CLIENT` e usare `./client`. Avviato il programma è possibile digitare i seguenti comandi:

- `ls`: ricevi l'elenco dei file presenti sul server
- `put (file)`: carica sul server 'file', dove (file) è il nome di un file all'interno della cartella `FILES`
- `get (file)`: scarica dal server 'file'
- `h`: stampa i comandi utilizzabili
- `q`: terminare programma

Per lanciare il server aprire un terminale dalla cartella `SERVER` e usare `./server`.

Il programma verrà avviato con i seguenti parametri di default:

- dimensione della finestra (N) = 8
- probabilità di perdita (p) = 20
- timer (T) = timer fisso pari a 1000 microsec

Oppure è possibile scegliere i parametri usando `./server < N > < p > < T >`.

Si noti che per scegliere di usare il timer "adattativo" è necessario porre il valore del parametro T a 0, qualunque altro valore di T inserito conduce all'impostazione del valore del timer fisso in microsecondi.

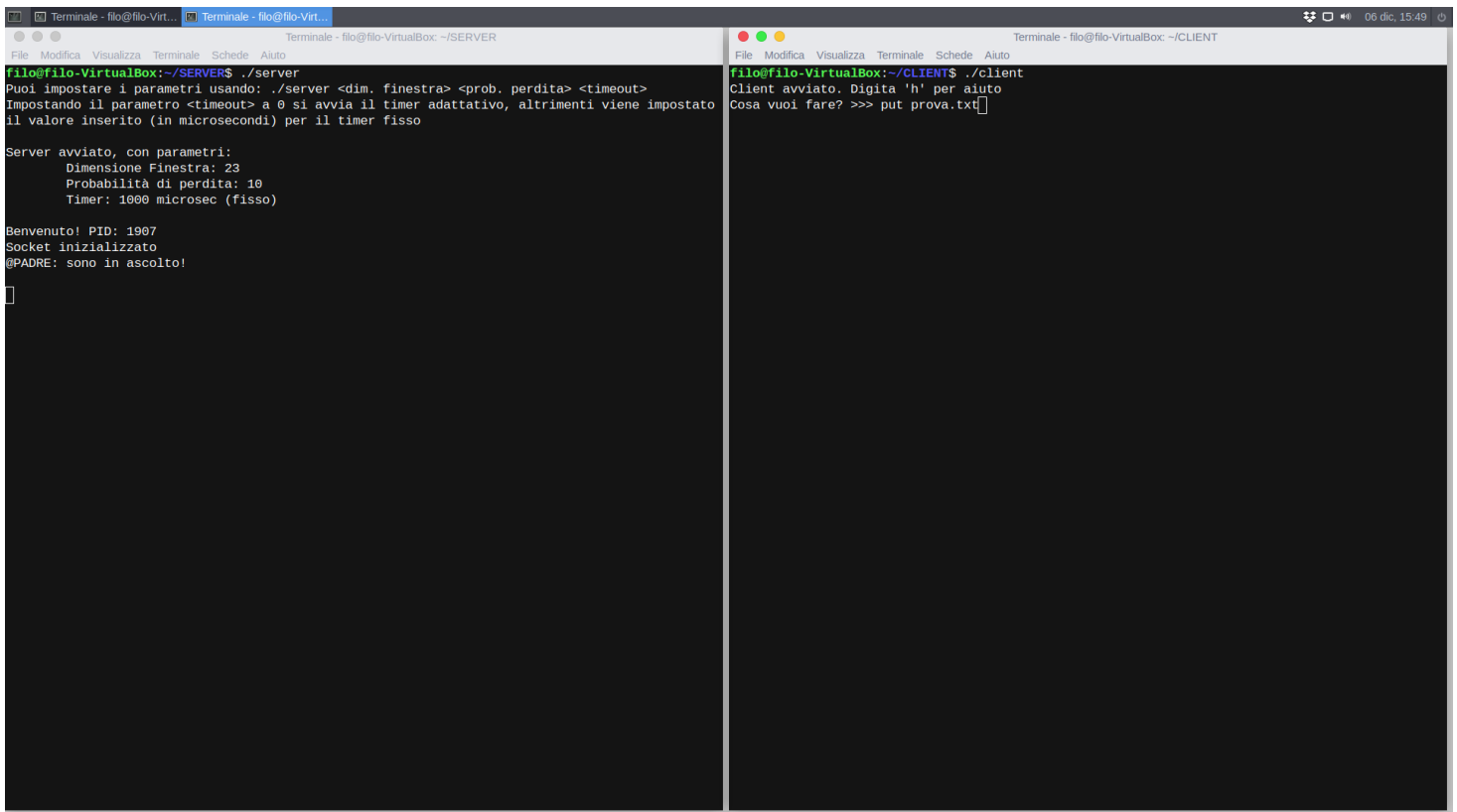
5 Esempi di funzionamento

In figura 7 e 8 viene effettuata una "put" del file di testo "testoprova.txt" da parte del client. Vengono stampati a schermo nelle finestre del client e del server tutti i dati utili alla verifica del corretto trasferimento del file. Quindi alla fine del processo viene segnalato l'invio e la ricezione dell'ultimo ACK per il messaggio ENDOFFILE.

In figura 9 e 10 viene effettuata una "get" dell'immagine "OPlocandina.jpeg" da parte del client e viene riportata, come in precedenza, l'avvio e la fine dell'operazione.

In figura 11 e 12 viene effettuata una "ls" per visionare la lista dei files disponibili e quindi viene stampata tale lista in output.

In figura 13 è riportata una finestra del server in cui vengono settati i parametri di trasferimento da parte dell'utente. Viene quindi mostrato il messaggio di avvio del server in seguito ai parametri inseriti.



```
Terminale - filo@filo-Virt... Terminale - filo@filo-Virt...
Terminale - filo@filo-VirtualBox: ~/SERVER
filo@filo-VirtualBox:~/SERVER$ ./server
Puoi impostare i parametri usando: ./server <dim. finestra> <prob. perdita> <timeout>
Impostando il parametro <timeout> a 0 si avvia il timer adattativo, altrimenti viene impostato
il valore inserito (in microsecondi) per il timer fisso

Server avviato, con parametri:
  Dimensione Finestra: 23
  Probabilità di perdita: 10
  Timer: 1000 microsec (fisso)

Benvenuto! PID: 1907
Socket inizializzato
@PADRE: sono in ascolto!

Terminale - filo@filo-VirtualBox: ~/CLIENT
filo@filo-VirtualBox:~/CLIENT$ ./client
Client avviato. Digita 'h' per aiuto
Cosa vuoi fare? >>> put prova.txt
```

Figure 7: put prova.txt - avvio

```
Terminale - filo@filo-Virt... Terminale - filo@filo-Virt...
Terminale - filo@filo-VirtualBox: ~/SERVER
File Modifica Visualizza Terminale Schede Aiuto

<= | -1 || 910 || 911 || 912 || 913 || 914 || 915 || 916 | <=

#####
# Bytes ricevuti dal client: 1028
#####

<< Mandato ack del pacchetto con numero di sequenza 909 >>
Sto per inserire data
Pacchetto numero 909 inserito in finestra in posizione 0
Scrivo su file i dati del pacchetto 909
Scrivo su file i dati del pacchetto 910
Scrivo su file i dati del pacchetto 911
Scrivo su file i dati del pacchetto 912
Scrivo su file i dati del pacchetto 913
Scrivo su file i dati del pacchetto 914
Scrivo su file i dati del pacchetto 915
Scrivo su file i dati del pacchetto 916

FINESTRA DOPO LO SPOSTAMENTO
Coda vuota

#####
# Bytes ricevuti dal client: 1028
#####

!!! PACCHETTO PERSO !!!
<< Mandato ack del pacchetto con numero di sequenza 913 >>

#####
# Bytes ricevuti dal client: 1028
#####

<< Mandato ack del pacchetto con numero di sequenza 913 >>

#####
# Bytes ricevuti dal client: 9
#####

Invio ACK per fine file
fine gest_msg(), comando PUT, ritorno
fine gestisci_messaggio() ritorno
@FIGLIO 1 [PID: 2177; PID padre: 1987], ho svolto il mio compito, termino.

Terminale - filo@filo-VirtualBox: ~/CLIENT
File Modifica Visualizza Terminale Schede Aiuto

| 2 || 1 || 1 || 1 || 2 || 1 || 1 || 1 |

!!! Pacchetto con numero di sequenza 909 TIMEOUT !!!

Pacchetto seqnum = 909 in timeout rinviato con timer di 2000 microsec

<< Ricevuto il riscontro del pacchetto con numero di sequenza 909 >>

<= | 909 || 910 || 911 || 912 || 913 || 914 || 915 || 916 | <=

| 1 || 1 || 1 || 1 || 2 || 1 || 1 || 1 |

Finestra spostata di 4 posizioni

<= | 913 || 914 || 915 || 916 | <=

| 2 || 1 || 1 || 1 |

!!! Pacchetto con numero di sequenza 913 TIMEOUT !!!

Pacchetto seqnum = 913 in timeout rinviato con timer di 2000 microsec

!!! Pacchetto con numero di sequenza 913 TIMEOUT !!!

Pacchetto seqnum = 913 in timeout rinviato con timer di 4000 microsec

<< Ricevuto il riscontro del pacchetto con numero di sequenza 913 >>

<= | 913 || 914 || 915 || 916 | <=

| 1 || 1 || 1 || 1 |

Finestra spostata di 4 posizioni
Coda vuota

Fine Selective repeat

Trasmetto il messaggio ENDOFFILE
Ricevuto ACK di fine file!

>>>
```

Figure 8: put prova.txt - fine

```
Terminale - filo@filo-Virt... Terminale - filo@filo-Virt...
Terminale - filo@filo-VirtualBox: ~/SERVER
File Modifica Visualizza Terminale Schede Aiuto

filo@filo-VirtualBox:~/SERVER$ ./server
Puoi impostare i parametri usando: ./server <dim. finestra> <prob. perdita> <timeout>
Impostando il parametro <timeout> a 0 si avvia il timer adattativo, altrimenti viene impostato
il valore inserito (in microsecondi) per il timer fisso

Server avviato, con parametri:
  Dimensione Finestra: 23
  Probabilità di perdita: 10
  Timer: 1000 microsec (fisso)

Benvenuto! PID: 2251
Socket inizializzato
@PADRE: sono in ascolto!

Terminale - filo@filo-VirtualBox: ~/CLIENT
File Modifica Visualizza Terminale Schede Aiuto

filo@filo-VirtualBox:~/CLIENT$ ./client
client avviato. Digita 'h' per aiuto
Cosa vuoi fare? >>> get OPlocandina.jpeg
```

Figure 9: get OPlocandina.jpeg - avvio

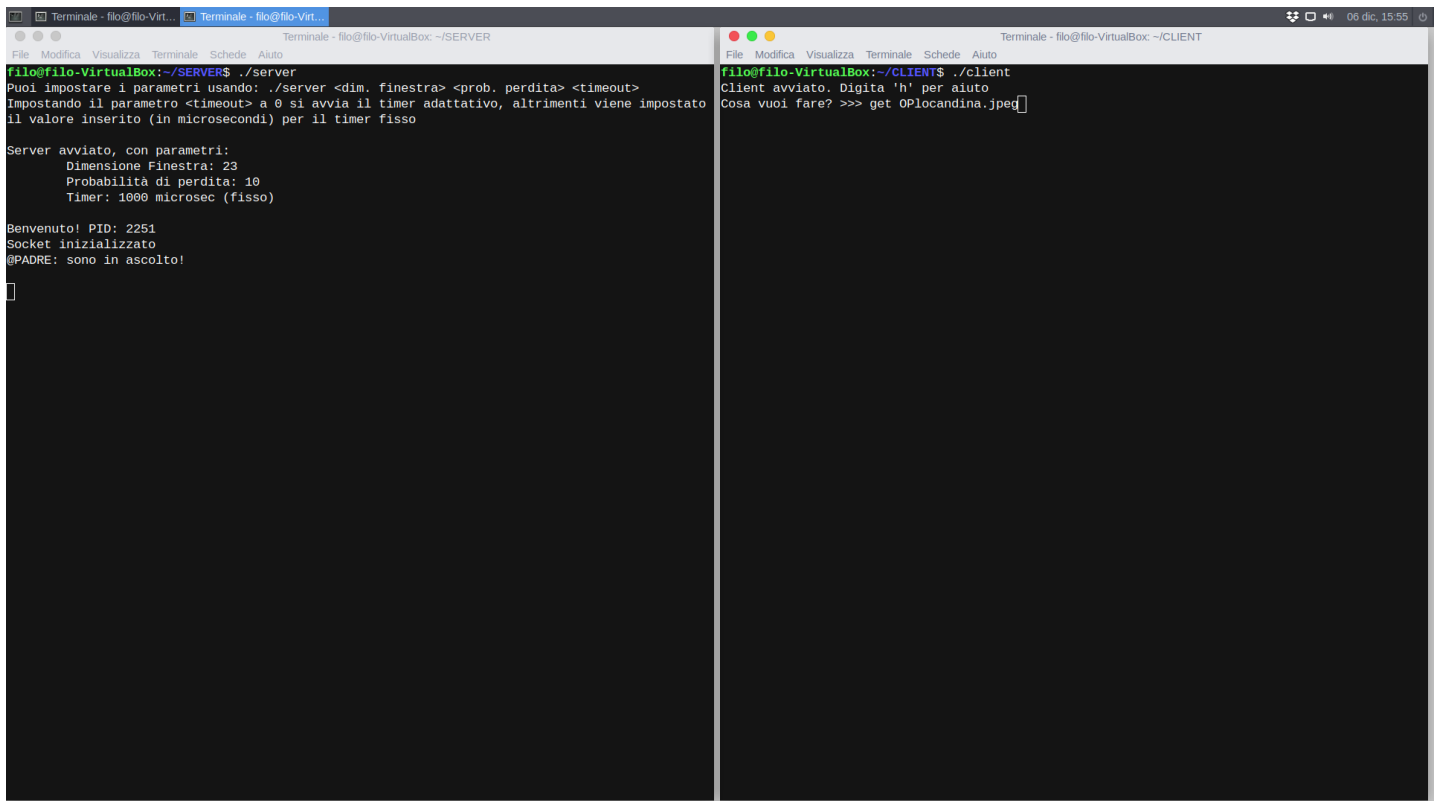


Figure 10: get OPlocandina.jpeg - fine

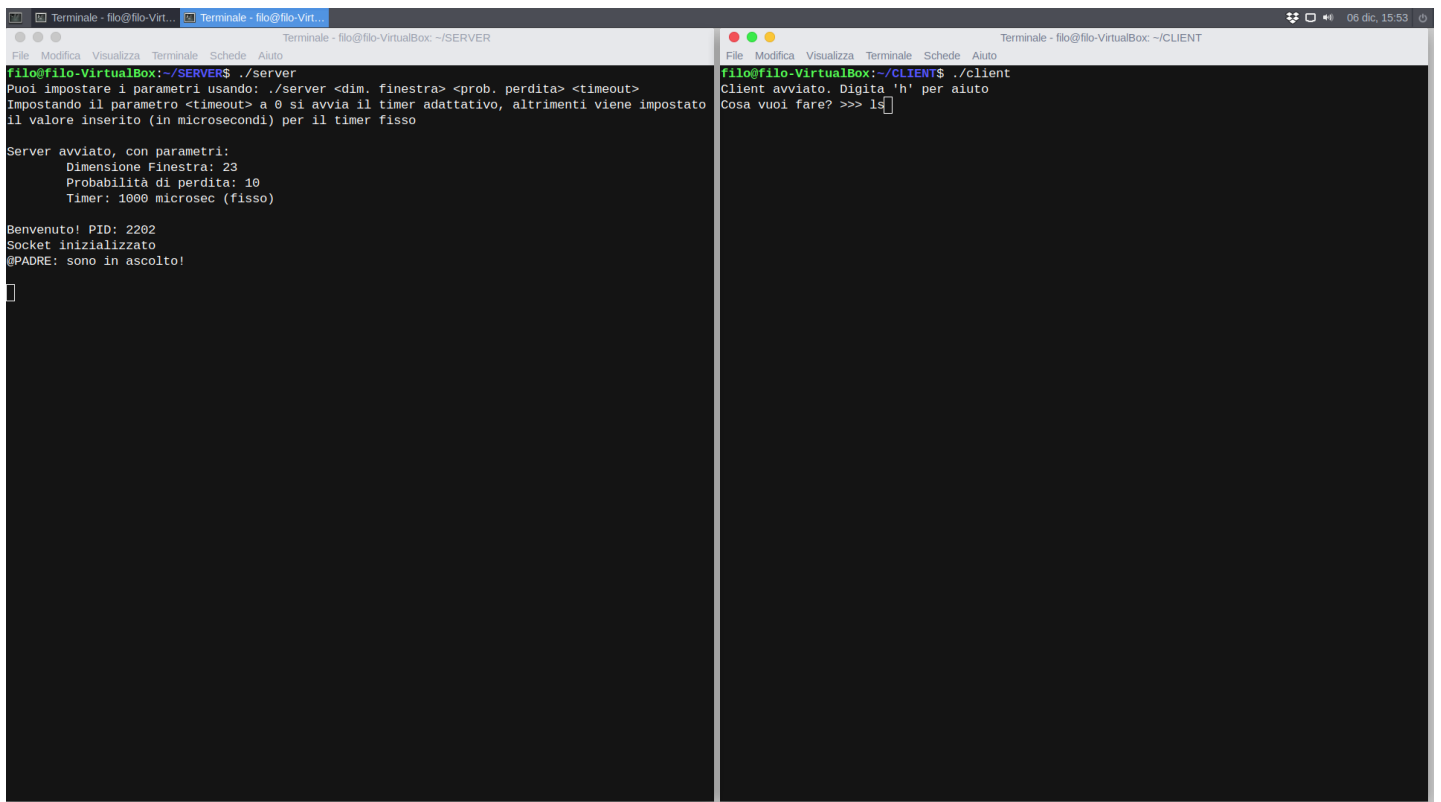
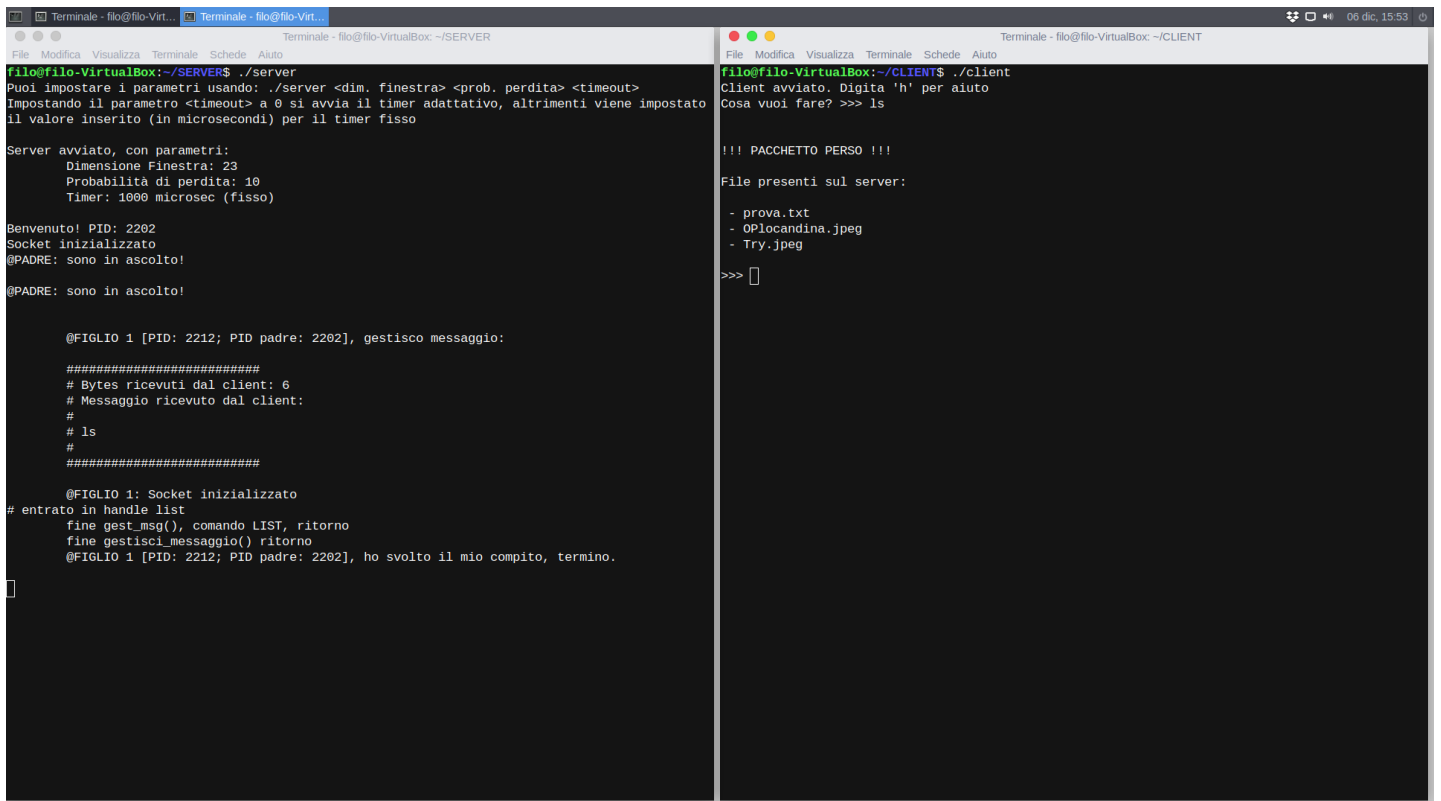


Figure 11: ls - avvio



```
Terminale - filo@filo-Virt... Terminale - filo@filo-Virt...
Terminale - filo@filo-VirtualBox: ~/SERVER Terminale - filo@filo-VirtualBox: ~/CLIENT
File Modifica Visualizza Terminale Schede Aiuto File Modifica Visualizza Terminale Schede Aiuto
filo@filo-VirtualBox:~/SERVER$ ./server filo@filo-VirtualBox:~/CLIENT$ ./client
Puoi impostare i parametri usando: ./server <dim. finestra> <prob. perdita> <timeout> Client avviato. Digita 'h' per aiuto
Impostando il parametro <timeout> a 0 si avvia il timer adattativo, altrimenti viene impostato Cosa vuoi fare? >>> ls
il valore inserito (in microsecondi) per il timer fisso

Server avviato, con parametri:

    Dimensione Finestra: 23
    Probabilità di perdita: 10
    Timer: 1000 microsec (fisso)

Benvenuto! PID: 2202
Socket inizializzato
@PADRE: sono in ascolto!

@PADRE: sono in ascolto!


    @FIGLIO 1 [PID: 2212; PID padre: 2202], gestisco messaggio:

    #####
    # Bytes ricevuti dal client: 6
    # Messaggio ricevuto dal client:
    #
    # ls
    #
    #####

    @FIGLIO 1: Socket inizializzato
# entrato in handle list
    fine gest_msg(), comando LIST, ritorno
    fine gestisci_messaggio() ritorno
    @FIGLIO 1 [PID: 2212; PID padre: 2202], ho svolto il mio compito, termino.

[]
```

Figure 12: ls - fine



```
Terminale - filo@filo-VirtualBox: ~/SERVER
File Modifica Visualizza Terminale Schede Aiuto
filo@filo-VirtualBox:~/SERVER$ ./server 1000 13 750
Puoi impostare i parametri usando: ./server <dim. finestra> <prob. perdita> <timeout>
Impostando il parametro <timeout> a 0 si avvia il timer adattativo, altrimenti viene impostato
il valore inserito (in microsecondi) per il timer fisso

Server avviato, con parametri:

    Dimensione Finestra: 1000
    Probabilità di perdita: 13
    Timer: 750 microsec (fisso)

Benvenuto! PID: 2190
Socket inizializzato
@PADRE: sono in ascolto!
```

Figure 13: server con parametri