

Progetto del corso Ingegneria di Internet e del
Web a.a. 2017-2018

Andrea Graziani - matricola 0189326

19 dicembre 2017

Indice

1	Descrizione dell'applicazione	2
1.1	Architettura	2
1.1.1	Gestione delle richieste	3
1.1.1.1	Il processo server	3
1.1.1.2	Il processo client	5
1.2	Implementazione del servizio di trasmissione affidabile	9
1.2.1	I messaggi di controllo	9
1.2.2	Gestione delle ritrasmissioni	10
1.2.2.1	Calcolo dell'intervallo di timeout	11
1.2.2.2	I Timer	13
1.2.2.3	Configurazione e gestione dei timer	15
1.3	Gestione della trasmissione	17
2	Guida all'utilizzo dell'applicazione	20
2.1	Descrizione dell'interfaccia	20
2.1.1	I comandi <code>client</code> , <code>server</code> e <code>options</code>	20
2.1.2	Il comando <code>list</code>	23
2.1.3	I comandi <code>get</code> e <code>put</code>	25
2.2	Alcuni esempi di utilizzo dell'applicazione	26
2.3	Descrizione piattaforma software usata per lo sviluppo	28
2.4	Guida alla compilazione e all'avvio dell'applicazione	28
2.5	Prestazioni	29

Capitolo 1

Descrizione dell'applicazione

L'applicazione di rete sviluppata, eseguibile su sistemi operativi basati su Linux, consente, attraverso un proprio protocollo applicativo, la comunicazione e la trasmissione *affidabile* di dati tra host utilizzando, come protocollo di trasporto, UDP (*User Datagram Protocol*).

Nel corso di questa relazione descriveremo le caratteristiche del protocollo applicativo utilizzato nonché le modalità attraverso cui viene garantita la trasmissione affidabile dei dati.

1.1 Architettura

L'architettura dell'applicazione di rete è di tipo **client-server** dove un host chiamato *server* è incaricato di soddisfare le richieste di servizio di molti altri host, detti *client*.

Sappiamo che la maggior parte dei server che utilizzano il protocollo di trasporto UDP, come ad esempio i server DNS¹, è *iterativa*²; in questo caso è sufficiente che il server attenda un messaggio di richiesta da parte di un client, quindi, dopo aver letto ed elaborato la richiesta ricevuta, spedisce un messaggio di risposta al client per poi attendere il successivo messaggio di richiesta.

Tuttavia questo approccio, per quanto di semplice implementazione, non è adatta quando l'elaborazione della richiesta del client richiede "*molto tempo*". Ad esempio, se arrivano due messaggi di richiesta entro 10 ms l'una dall'altra e il server impiega una media di 5 secondi per servire un client, il secondo client dovrà aspettare circa 10 secondi prima di ricevere una qualche risposta; qualora una richiesta fosse gestita immediatamente senza alcuna attesa, il secondo client attenderà, invece, circa 5 secondi. Da questo semplice esempio è facile capire che realizzando un'applicazione per il trasferimento di file, in cui spesso si richiede la trasmissione di molti pacchetti (centinaia o migliaia, a seconda delle dimensioni del file), i tempi di attesa possono diventare inaccettabili.

Di conseguenza, data l'impraticabilità di un approccio iterativo, si è scelto di realizzare un'applicazione di rete di tipo **concorrente** di cui riportiamo uno

¹Cfr. Andrew S. Tanenbaum & David J. Wetherall - *Reti di calcolatori 5/Ed*, Pearson, pp. 189

²Cfr. Richard Stevens & Bill Fenner & Andrew M. Rudoff - *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*, Addison Wesley, Cap. 22.7

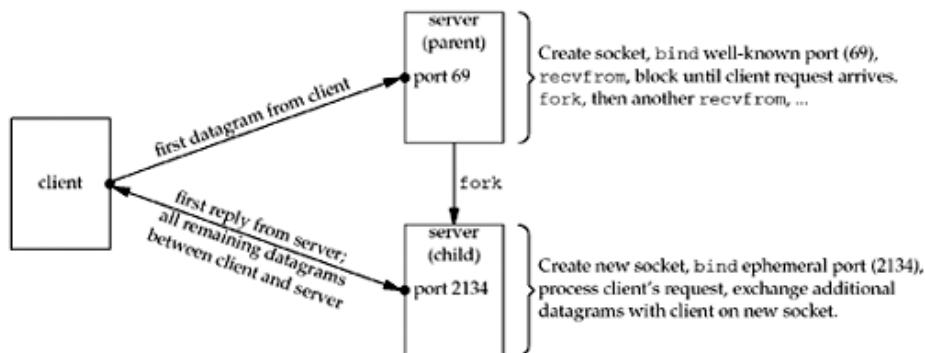


Figura 1.1: Schema del funzionamento di un server UDP concorrente.

schema in figura 1.1. In un server UDP concorrente il compito di gestire e soddisfare le richieste ricevute dai client, in modo tale da permettere al processo server di poter continuare a ricevere nuove richieste, viene affidata ad un processo (o *thread*) distinto da quello principale creandolo al momento oppure scegliendolo da un pool di processi precedentemente creati. Nei paragrafi seguenti analizzeremo nel dettaglio il funzionamento di un server UDP concorrente.

1.1.1 Gestione delle richieste

1.1.1.1 Il processo server

Dopo aver stabilito la natura concorrente dell'architettura dell'applicazione sviluppata, esaminiamo ora alcuni aspetti implementativi del lato server dell'applicazione incominciando dalla gestione delle richieste.

Come è possibile osservare dalla porzione di codice raffigurante la funzione `start_server` (implementata all'interno del file `server_manager.c`) riportata in *listing* 1.1, l'attività svolta dal processo server principale è suddivisibile nelle seguenti fasi:

Inizializzazione socket ascolto In questa fase, mediante l'esecuzione della funzione `__server_initialization` (anch'essa implementata all'interno del file `server_manager.c`), avviene la procedura di inizializzazione di un *socket di ascolto* su una porta *nota*; ogni client intenzionato a trasmettere una richiesta al server dovrà inviare un opportuno messaggio di richiesta attraverso il suddetto socket.

Caricamento metadati Questa fase prevede il caricamento di una serie di informazioni utili a soddisfare richieste di tipo `list`; questo aspetto verrà approfondito nel paragrafo 2.1.2

Fase di ascolto Dopo aver allocato un'apposita area di memoria dove allocare il messaggio ricevuto dal client, il server (mediante la chiamata di sistema `recvfrom` di tipo *bloccante*) rimane in attesa attendendo l'arrivo di un messaggio di richiesta sulla socket di ascolto precedentemente creata.

Creazione del thread Non appena viene ricevuto un nuovo messaggio di richiesta, viene creato un nuovo thread a cui affidare il compito di gestire e

soddisfare la richiesta ricevuta. Successivamente il processo server ritorna in fase di ascolto.

Listing 1.1: Implementazione della funzione `start_server`

```
1 void start_server() {
2
3     ThreadArgument *ThreadArgument_obj;
4     int received_bytes;
5
6     /* Listener Socket file descriptor: Initialization
       server */
7     int listener_socket_fd = __server_initialization(1);
8
9     /* Preload metadata about shared files stored in
       application server directory */
10    __metadata_preload();
11
12    ...
13
14    while (1) {
15
16        /* Preparing necessary data... */
17        /* ***** */
18        ThreadArgument_obj = calloc(1, sizeof(
            ThreadArgument));
19        if (ThreadArgument_obj == NULL)
20            exit_failure("calloc");
21
22        ThreadArgument_obj->DataNetwork_obj.
            address_len = sizeof(ThreadArgument_obj->
            DataNetwork_obj.address);
23
24        ...
25
26        /* Receiving request from client */
27        /* ***** */
28        while (1) {
29
30            errno = 0;
31            received_bytes = recvfrom(
                listener_socket_fd, &
                ThreadArgument_obj->
                RequestPacket_obj, sizeof(
                RequestPacket), 0,
32                (struct sockaddr *)
                    &
                    ThreadArgument_obj
                    ->DataNetwork_obj
                    .address, &
                    ThreadArgument_obj
                    ->DataNetwork_obj
                    .address_len);
33
```

```

34         ...
35     }
36     ...
37     /* Launch thread */
38     /* ***** */
39     thread_initialization(thread_job, (void **)
        ThreadArgument_obj);
40 }
41 }

```

Questo approccio, per quanto semplice, presenta tuttavia un inconveniente. Il server UDP deve poter scambiare più di un pacchetto con il client per poter soddisfare le richieste. Il problema è che l'unico numero di porta noto al client per poter comunicare con il server è, sfortunatamente, la stessa porta utilizzata da quest'ultimo per ricevere nuove richieste. Dopo che il client ha inviato un messaggio di richiesta attraverso la porta di ascolto, come fa dunque il server a distinguere tra pacchetti successivi provenienti da quel client e nuove richieste?

La soluzione è quella di affidare ai thread incaricati di soddisfare le richieste ricevute anche il compito di creare un'apposita socket di supporto eseguendo il binding su una porta effimera e utilizzando quel socket per inviare e ricevere i successivi messaggi con un client. Ciò ovviamente richiede che il client in questione guardi il numero di porta della prima risposta del server e invii i pacchetti successivi a quella porta. Tutto ciò consente al server principale di continuare a gestire altre richieste proveniente da altri client.

1.1.1.2 Il processo client

In accordo alle specifiche di progetto, ogni processo client, senza alcuna operazione di autenticazione preliminare, deve essere in grado di:

- Visualizzare tutti i file disponibili sul server;
- Scaricare file dal server;
- Caricare file sul server;

L'esecuzione delle suddette operazioni è possibile previo l'invio da parte del client di un apposito *messaggio di richiesta* al processo server.

Da un punto di vista implementativo, per poter inoltrare una qualsiasi richiesta, il processo client fa uso di due strutture dati fondamentali:

DataNetwork Essa rappresenta tecnicamente un contenitore per tutte le informazioni necessarie per comunicare con un host, ovvero indirizzo IP e numero di porta. Tale struttura, utilizzata sia dal processo client che da quello server e di cui riportiamo il codice in *listing 1.2*, è composta da:

1. Un campo ospitante il *file descriptor* di una socket.
2. Un campo contenente una struttura di tipo `sockaddr_in` all'interno della quale sono contenute le informazioni necessarie per comunicare con un host.
3. Un campo utilizzato per ospitare un quantità di tipo `socklen_t` che specifica la lunghezza della struttura `sockaddr_in` precedente.

Listing 1.2: Implementazione della struttura `DataNetwork`

```

1  typedef struct data_network {
2
3      /* Socket file descriptor */
4      int socket_fd;
5      /* Internet socket address */
6      struct sockaddr_in address;
7      /* Internet socket address length */
8      socklen_t address_len;
9
10 } DataNetwork;

```

`RequestPacket` Questa struttura modella un *messaggio di richiesta* ed è composta da:

1. Un campo di 35 byte utilizzato per ospitare una speciale struttura dati, denominata `Settings`, utilizzata per ospitare una serie di parametri trasmissivi tra cui la dimensione della finestra di spedizione, durata di timeout e probabilità di perdita dei messaggi. Approfondiremo in seguito i dettagli relativi alle impostazioni.
2. Un campo di 1 byte utilizzato per specificare il tipo di richiesta (*list*, *put* e *get*).
3. Un campo di dimensione eventualmente nulla inferiore ai 255 byte utilizzato per specificare un eventuale parametro o informazione aggiuntiva; per esempio, se il client trasmettesse un messaggio di tipo *get*, il suddetto campo conterrebbe il nome di un file.

Listing 1.3: Implementazione della struttura `RequestPacket`

```

1  typedef struct request_client_packet {
2
3      Settings client_setting;
4      char request_type;
5      char request_type_argument[NAME_MAX];
6
7  } RequestPacket;

```

Per inviare un messaggio di richiesta il processo client non fa altro che inizializzare opportunamente una struttura di tipo `RequestPacket` inviandola al server servendosi delle informazioni contenute all'interno della struttura di tipo `DataNetwork` tra cui l'indirizzo IP, specificato dall'utente, e il numero della porta di ascolto sul server nota a priori, per poi attendere una risposta da parte del server. Qualora non si ricevesse alcuna risposta, il processo client deve provvedere a inoltrare nuovamente la richiesta.

Nel paragrafo precedente abbiamo stabilito che il client, conoscendo esclusivamente il numero della porta di ascolto, non può sapere da quale porta verranno scambiati i successivi messaggi con il server; pertanto il client dovrà guardare il numero di porta della prima risposta del server e, aggiornando opportunamente i valori dei campi contenuti nella struttura `DataNetwork`, inviare i pacchetti successivi a quella porta. Questa operazione molto delicata viene eseguita attraverso la funzione `__receive_datagram` definita all'interno del file

`data_transmission.c` di cui ne riportiamo il codice in *listing 1.4*. Dopo la ricezione della prima risposta da parte del server, verrà aggiornato il contenuto della struttura `DataNetwork` consentendo al client non solo di poter scambiare altri messaggi ma anche di ignorare tutti quei pacchetti non provenienti dal processo server con cui interagisce; inoltre permette a quest'ultimo di ignorare tutti i pacchetti provenienti da altri client. Quest'ultima operazione è resa possibile mediante un semplice confronto dell'indirizzo IP e del numero di porta del messaggio ricevuto con le informazioni contenute all'interno della struttura `DataNetwork`.

Listing 1.4: Implementazione della funzione `__receive_datagram`

```

1  int __receive_datagram(DataNetwork *DataNetwork_obj, void *
    buffer, size_t buffer_size, char *check_replay) {
2
3      int bytes_received;
4
5      ...
6      struct sockaddr *preplay_addr = malloc(
            DataNetwork_obj->address_len);
7      if (preplay_addr == NULL)
8          exit_failure("malloc");
9
10     do {
11         /* Receiving data in 'check replay' mode */
12         /* ***** */
13         bytes_received = recvfrom(DataNetwork_obj->
            socket_fd, buffer, buffer_size, 0,
            preplay_addr, &DataNetwork_obj->
            address_len);
14
15         /* Checking error */
16         if (bytes_received == -1) {
17             if (errno == EINTR) {
18
19                 ...
20
21                 /* Free memory */
22                 free(preplay_addr);
23
24                 return -1;
25             } else
26                 exit_failure("recvfrom_wc");
27         }
28
29         if (check_replay == NULL || *check_replay ==
            CHECK_REPLAY) {
30
31             /* Check reply */
32             /* ***** */
33             if (memcmp((struct sockaddr*) &
                DataNetwork_obj->address,
                preplay_addr, DataNetwork_obj->
                address_len) != 0) {

```



```

34         ...
35         continue;
36     }
37
38     } else {
39
40         memcpy(&DataNetwork_obj->address,
41             preply_addr, sizeof(*preply_addr)
42             );
43
44         /* Check replay... */
45         *check_replay = CHECK_REPLAY;
46     }
47     } while (0);
48
49     /* Free memory */
50     free(preply_addr);
51
52     ...
53     return bytes_received;
54 }

```

1.2 Implementazione del servizio di trasmissione affidabile

In questo capitolo descriveremo come è stato implementato un servizio di trasmissione affidabile utilizzando un protocollo di trasporto come UDP che offra un servizio di comunicazione di tipo *best effort*³, ovvero non affidabile.

1.2.1 I messaggi di controllo

Incominciamo con alcune constatazioni:

- Dato che mittente e destinatario sono generalmente in esecuzione su sistemi periferici diversi, magari separati da migliaia di chilometri, l'unico modo che ha il mittente per sapere se un pacchetto sia stato ricevuto correttamente o meno consiste nel *feedback esplicito* del destinatario.
- Dovendo rispettare le specifiche di progetto, l'applicazione deve simulare a livello applicativo un protocollo a **ripetizione selettiva** (o semplicemente SR, *selective-repeat protocol*). Il protocollo SR ci consente di per evitare una serie di ritrasmissioni non necessarie in quanto il mittente deve ritrasmettere solo quei pacchetti su cui esistono sospetti di errore, ossia smarrimento o alterazione, rilevati in base alla mancata o ritardata ricezione del riscontro di un pacchetto.

Queste due considerazioni costringono pertanto il destinatario a mandare riscontri specifici per i pacchetti ricevuti in modo corretto; questi riscontri costituiscono i cosiddetti *messaggi di controllo*.

Da un punto di vista implementativo, per modellare i messaggi di controllo ci si è serviti di un'apposita struttura denominata **ControlDatagram** di cui riportiamo il codice in *listing* 1.5. I messaggi di controllo vengono utilizzati per:

1. Inviare al mittente un riscontro (detto anche *acknowledgement*, quest'ultimo abbreviato spesso con l'acronimo ACK) per i pacchetti ricevuti in modo corretto.
2. Gestire le procedure di instaurazione e chiusura delle connessioni.

Una struttura di tipo **ControlDatagram** è composta dai seguenti campi:

1. Un campo di un byte utilizzato per specificare il *tipo* di messaggio di controllo. I tipi utilizzati sono:

REQ È usato per instaurare una connessione con un altro host; benché molto diverso, esso corrisponde al segmento SYN usato nel protocollo di trasporto TCP⁴.

REQ_ACK È usato per rispondere ad una richiesta di connessione in seguito alla ricezione di un messaggio di tipo REQ; corrisponde al segmento SYNACK usato in TCP⁵.

³Cfr. Andrew S. Tanenbaum & David J. Wetherall - *Reti di calcolatori 5/Ed*, Pearson, pp. 179

⁴Cfr. *ivi*, pp. 239

⁵Cfr. *ibid.*

FIN Viene utilizzato per terminare una trasmissione e incominciare la procedura di chiusura della connessione; corrisponde al segmento FIN usato in TCP⁶.

FIN_ACK Viene utilizzato per confermare la terminazione di una trasmissione.

CLS Viene utilizzato per terminare una connessione.

ACK Viene utilizzato per indicare un messaggio di riscontro.

2. Un campo di un byte utilizzato per ospitare il *numero di sequenza* di un pacchetto; tale campo è utilizzato esclusivamente quando il messaggio di controllo in questione rappresenta un ACK, in caso contrario la dimensione del campo è nulla.

Listing 1.5: Implementazione della struttura `ControlDatagram`

```
1 typedef struct control_datagram {  
2     char type;  
3     char n;  
4 } ControlDatagram;
```

1.2.2 Gestione delle ritrasmissioni

Riuscire a rilevare e a gestire lo *smarrimento* dei pacchetti, un evento non raro sulle odierne reti di calcolatori, e decidere cosa fare quando ciò avviene rappresenta il problema più importante e complesso. La soluzione a questo problema è tuttavia molto semplice: se dopo la trasmissione di un certo pacchetto il mittente non ricevesse alcun riscontro da parte del destinatario dopo un certo lasso di tempo, quest'ultimo deve provvedere a ritrasmettere il pacchetto smarrito.

Implementare un meccanismo di ritrasmissione basato sul tempo richiede l'utilizzo di un *timer* capace di segnalare al mittente l'avvenuta scadenza di un dato lasso di tempo e che il mittente sia in grado di:

1. Inizializzare un timer ogni volta che invia un pacchetto.
2. Rispondere adeguatamente all'interrupt generato dal timer in caso di timeout procedendo quindi alla ritrasmissione del pacchetto smarrito e reinizializzare il timer.
3. Disabilitare il timer quando il mittente riceve un riscontro positivo da parte del destinatario.

Questo approccio introduce, tuttavia, la presenza di pacchetti duplicati nel canale di trasmissione. Dal momento che il destinatario non può sapere a priori se un pacchetto in arrivo contenga dati nuovi o rappresenti una ritrasmissione, è necessario utilizzare dei *numeri di sequenza*: al destinatario sarà sufficiente controllare questo numero per sapere se il pacchetto si tratta di una ritrasmissione o meno. Inoltre, dal momento che l'applicazione simula il comportamento di un protocollo a ripetizione selettiva, sarà necessario associare ad ogni pacchetto trasmesso un timer dedicato.

⁶Cfr. *ibid.*

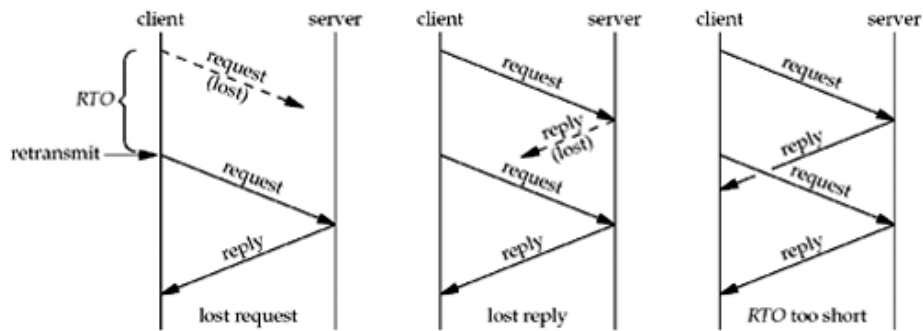


Figura 1.2: Possibili scenari che si possono verificare dopo un timeout

1.2.2.1 Calcolo dell'intervallo di timeout

Occupiamoci ora di una questione cruciale: la lunghezza dell'intervallo di timeout. Ovviamente l'intervallo di timeout dovrebbe essere più grande del tempo di andata e ritorno del segnale sulla connessione per evitare timeout prematuri evitando così ritrasmissioni inutili. Per calcolare l'intervallo di timeout si usa lo stesso algoritmo utilizzato nel protocollo TCP. L'applicazione misura una quantità di tempo detta *RTT* (*Round Trip Time*); quest'ultima rappresenta la quantità di tempo che intercorre tra l'istante di invio del segmento e quello di ricezione del relativo ACK. Tuttavia, in base alla congestione nei router e al diverso carico sui sistemi periferici durante la trasmissione, i campioni di *RTT* calcolati possono variare da pacchetto a pacchetto, perciò si preferisce calcolare una media dei valori di *RTT*: tale quantità, che chiameremo *EstimatedRTT*, si calcola con la seguente formula:

$$EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times RTT \quad (1.1)$$

Dove $\alpha = 1/8$. Si noti che *EstimatedRTT* è una media ponderata dei valori *RTT*: tale media, meglio conosciuta in statistica come **media mobile esponenziale ponderata** (EWMA, *exponential weighted moving average*), attribuisce maggiore importanza ai campioni recenti rispetto a quelli vecchi in modo tale da riflettere meglio la congestione attuale della rete. Tuttavia, oltre a calcolare la media dei valori di *RTT*, è anche importante possedere anche la misura della variabilità dei *RTT* calcolati. Tale quantità verrà chiamata *DevRTT* ed è calcolabile come segue:

$$DevRTT = (1 - \beta) \times DevRTT + \beta \times |RTT - EstimatedRTT| \quad (1.2)$$

Come già detto, l'intervallo di timeout non può essere inferiore a quello di *EstimatedRTT*, altrimenti verrebbero inviate ritrasmissioni non necessarie. Ma l'intervallo stesso non dovrebbe essere molto maggiore di *EstimatedRTT*, altrimenti avremo gravi ritardi sul trasferimento dei dati. È pertanto auspicabile impostare il timeout a *EstimatedRTT* più un certo margine che dovrebbe essere grande quando c'è molta fluttuazione nei valori di *RTT* e piccolo in caso contrario. Qui entra in gioco il valore di *DevRTT*. Tutti questi aspetti vengono presi in considerazione dalla seguente equazione:

$$RTO = EstimatedRTT + 4 \times DevRTT \quad (1.3)$$

Da un punto di vista implementativo, tutte le informazioni riguardante la durata dell'intervallo di timeout (indicata con l'acronimo RTO, *Recovery Time Objective*), la media dei valori di RTT e la loro varianza sono contenute all'interno di una speciale struttura denominata `TimeInfo` definita all'interno del file `time_management.h` di cui riportiamo il codice in *listing 1.6*.

Listing 1.6: Implementazione della struttura `time_info`

```

1 typedef struct time_info {
2
3     /* Current 'Recovery Time Objective' (RTO) to use */
4     struct timespec RTO;
5
6     /* Weighted average of RTT values */
7     struct timespec Estimated_RTT;
8     /* Variance */
9     struct timespec Dev_RTT;
10
11 } TimeInfo;
```

L'algoritmo descritto in precedenza per il calcolo degli intervalli di timeout è implementato all'interno della funzione `TimeInfo_calc_time` definita all'interno del file `time_management.c` di cui ne riportiamo il codice in *listing 1.7*.

Listing 1.7: Implementazione della funzione `TimeInfo_calc_time`

```

1 /*
2  * Updates RTT estimators and calculates new RTO.
3  *
4  * @param *input -> It represents a 'TimeInfo' object.
5  * @param *sending_time -> It represents a 'timespec' object
6  *
7  * @param *receiving_time -> It represents a 'timespec'
8  * object.
9  */
10 void TimeInfo_calc_time(TimeInfo *input, struct timespec *
11     sending_time, struct timespec *receiving_time) {
12
13     const float alpha = 0.125;
14     const float beta = 0.25;
15
16     struct timespec RTT;
17
18     /* Calculation */
19     /* ===== */
20
21     /* Calculate 'Round Trip Time' */
22     RTT.tv_sec = receiving_time->tv_sec - sending_time->
23         tv_sec;
24     RTT.tv_nsec = receiving_time->tv_nsec - sending_time->
25         tv_nsec;
26
27     /* Calculate weighted average of RTT values */
```

```

23     input->Estimated_RTT.tv_sec = (1.0 - alpha) * input
        ->Estimated_RTT.tv_sec + alpha * RTT.tv_sec;
24     input->Estimated_RTT.tv_nsec = (1.0 - alpha) * input
        ->Estimated_RTT.tv_nsec + alpha * RTT.tv_nsec;
25
26     /* Calculate variance */
27     input->Dev_RTT.tv_sec = (1.0 - beta) * input->
        Dev_RTT.tv_sec + beta * abs(RTT.tv_sec - input->
        Estimated_RTT.tv_sec);
28     input->Dev_RTT.tv_nsec = (1.0 - beta) * input->
        Dev_RTT.tv_nsec + beta * abs(RTT.tv_nsec - input
        ->Estimated_RTT.tv_nsec);
29
30     /* Calculate RTO */
31     input->RTO.tv_sec = input->Estimated_RTT.tv_sec +
        (4.0 * input->Dev_RTT.tv_sec);
32     input->RTO.tv_nsec = input->Estimated_RTT.tv_nsec +
        (4.0 * input->Dev_RTT.tv_nsec);
33
34     /* Converting nanosec to sec */
35     /* ===== */
36     __timespec_converting_from_ns_to_s(&input->RTO);
37     __timespec_converting_from_ns_to_s(&input->
        Estimated_RTT);
38     __timespec_converting_from_ns_to_s(&input->Dev_RTT);
39
40     /* Checking if RTO is within the permitted limits...
        */
41     /* ===== */
42     __timespec_checking_limit(&input->RTO);
43 }

```

Per concludere è opportuno precisare che le strutture `TimeInfo` vanno inizializzate mediante la funzione `TimeInfo_init` definita nel file `time_management.c` prima di essere utilizzate; questa operazione è necessaria sia per impostare un intervallo di timeout iniziale, che in accordo alle [RFC 6298]⁷ deve essere impostato ad un secondo, sia per impostare eventualmente un intervallo di timeout stabilito dall'utente

1.2.2.2 I Timer

Dovendo simulare il comportamento di un protocollo di trasmissione selettiva, ogni pacchetto trasmesso dal mittente deve essere associato ad un proprio timer logico. In particolare, i timer associato ad uno specifico pacchetto deve essere attivato all'atto della sua trasmissione. Dopo la trasmissione di un pacchetto possono verificarsi due eventi:

1. Il mittente riceve un riscontro da parte del destinatario; il timer viene pertanto disattivato.
2. Il mittente non riceve alcun riscontro da parte del destinatario fino allo scadere dell'intervallo di timeout; il timer genera quindi un interrupt

⁷Cfr. Andrew S. Tanenbaum & David J. Wetherall - *Reti di calcolatori 5/Ed*, Pearson, pp. 228

in seguito alla quale il mittente provvederà a ritrasmettere il pacchetto associato a quel timer.

Da un punto di vista implementativo, per allocare e assegnare un timer logico a ciascun pacchetto da trasmettere, si è usato un'apposita struttura denominata `DatagramTimer` definita all'interno del file `time_management.h` e di cui riportiamo il codice in *listing* 1.8. Questa struttura comprende:

- Un puntatore al pacchetto associato al timer.
- Un puntatore ad un oggetto di tipo `DataNetwork`.
- Un puntatore denominato `abort_trasmission_ptr` utilizzato per interrompere il processo di trasmissione qualora venga superato il limite massimo di ritrasmissioni consentite per quel pacchetto.
- Un campo utilizzato per specificare il tipo di pacchetto associato al timer e distinguere perciò i pacchetti ordinari contenuti dai messaggi di controllo.
- Un oggetto di tipo `timer_t` rappresentante un timer UNIX⁸.
- Il numero di ritrasmissione effettuate per un dato pacchetto.
- Un oggetto di tipo `itimerspec`⁹ usato per rappresentare la durata l'intervallo di timeout.
- Due oggetti di tipo `timespec` usati per indicare l'istante di invio del pacchetto e ricezione del relativo riscontro.

Listing 1.8: Implementazione della struttura `DatagramTimer`

```
1 typedef struct datagram_timer {  
2  
3     void *Datagram_ptr;  
4     DataNetwork *DataNetwork_ptr;  
5     char *abort_trasmission_ptr;  
6  
7     char datagram_type;  
8     timer_t timer;  
9     char times_retransmitted;  
10    struct itimerspec timervals;  
11    struct timespec sending_time;  
12    struct timespec receiving_time;  
13  
14 } DatagramTimer;
```

⁸Cfr. Michael Kerrisk - *The Linux Programming Interface*, No Starch Press, pp. 495-496

⁹Cfr. *ivi*, pp. 480

1.2.2.3 Configurazione e gestione dei timer

Ora che abbiamo definito le strutture di tipo `DatagramTimer`, descriviamo come possono essere utilizzate. Innanzitutto, per poter usare un timer e associarlo ad uno specifico pacchetto, è necessario eseguire una procedura di inizializzazione attraverso una funzione denominata `DatagramTimer_init` e implementata nel file `time_management.c` di cui riportiamo il codice in *listing 1.9*.

Operativamente, l'inizializzazione di un oggetto di tipo `DatagramTimer` si divide nelle seguenti fasi:

Associazione del pacchetto Popolando opportunamente i campi della struttura `DatagramTimer`, si specifica il pacchetto associato al timer in questione. In questa fase viene popolato anche il puntatore alla struttura di tipo `DataNetwork` necessario per ritrasmettere il pacchetto in caso di timeout.

Configurazione handler In questa fase viene dichiarata la procedura da seguire in caso di timeout; operativamente viene specificato un apposito handler attraverso l'uso della chiamata di sistema `sigaction`.

Configurazione della struttura sigevent Questa fase è necessaria per specificare come l'applicazione debba essere avvisata dell'avvenuto timeout; ciò viene effettuato attraverso la configurazione di una speciale struttura dati di tipo `sigevent`¹⁰. In particolare il metodo di notifica scelto è di tipo `SIGEV_THREAD_ID`: in accordo alle specifiche delle API POSIX, scegliendo il suddetto metodo di notifica, in seguito al verificarsi di un timeout, il timer invia un segnale a quel thread il cui identificatore coincide con il valore specificato in `sigev_un._tid`. Popolando opportunamente il campo `sigev_value.sival_ptr` con un puntatore ad una struttura di tipo `DatagramTimer`, l'handler incaricato di gestire l'avvenuto timeout potrà accedere al pacchetto associato al timer appena scaduto.

Creazione del timer In questa fase avviene semplicemente la creazione del timer attraverso la chiamata di sistema `timer_create()`¹¹ usata compatibilmente alle specifiche delle API POSIX.

Listing 1.9: Implementazione della funzione `DatagramTimer_init`

```
1 void DatagramTimer_init(DatagramTimer *DatagramTimer_obj,
2   void **Datagram_ptr, DataNetwork **DataNetwork_ptr, char
3   *abort_trasmission_ptr,
4   char datagram_type, void (*handler)(int,
5   siginfo_t*, void*)) {
6
7   struct sigevent sev;
8   struct sigaction sa;
9
10  /* Configuration 'DatagramTimer' */
11  /* ===== */
12  if (DataNetwork_ptr != NULL)
13      DatagramTimer_obj->DataNetwork_ptr = *
14      DataNetwork_ptr;
```

¹⁰Cfr. *ivi*, pp. 496

¹¹Cfr. *ivi*, pp. 495


```

11     if (Datagram_ptr != NULL)
12         DatagramTimer_obj->Datagram_ptr = *
            Datagram_ptr;
13
14     DatagramTimer_obj->abort_trasmission_ptr =
        abort_trasmission_ptr;
15     DatagramTimer_obj->datagram_type = datagram_type;
16
17     /* Establish handler for notification signal */
18     /* ===== */
19     sa.sa_flags = SA_SIGINFO;
20     sa.sa_sigaction = handler;
21     sigemptyset(&sa.sa_mask);
22     sigaddset(&sa.sa_mask, SIGRTMAX);
23     if (sigaction(SIGRTMAX, &sa, NULL) == -1)
24         exit_failure("sigaction");
25
26     /* Configuration structure to transport application-
        defined values with signals. */
27     /* ===== */
28     sev.sigev_notify = SIGEV_THREAD_ID;
29     sev._sigev_un._tid = syscall(SYS_gettid);
30     sev.sigev_signo = SIGRTMAX;
31     sev.sigev_value.sival_ptr = DatagramTimer_obj;
32
33     /* Create new per-process timer using CLOCK_ID. */
34     /* ===== */
35     if (timer_create(CLOCK_REALTIME, &sev, &
        DatagramTimer_obj->timer) == -1)
36         exit_failure("timer_create");
37 }

```

Una volta inizializzato l'oggetto `DatagramTimer` è possibile avviare un timer attraverso la chiamata di sistema `timer_settime()`¹²; la durata dell'intervallo di timeout può essere aggiornata attraverso i valori calcolati e contenuti all'interno della struttura `TimeInfo`. La disattivazione di un timer in seguito alla ricezione di un riscontro positivo avviene utilizzando la chiamata di sistema `timer_settime()` specificando un intervallo di timeout nullo¹³. Una volta completato il processo di trasferimento dei dati, tutti i timer precedentemente allocati possono essere rimossi mediante la chiamata di sistema `timer_delete()`¹⁴.

¹²Cfr. *ivi*, pp. 498

¹³Cfr. *ibid*

¹⁴Cfr. *ivi*, pp. 499

1.3 Gestione della trasmissione

Descriveremo ora le modalità attraverso cui un processo client e un processo server possano effettivamente scambiarsi dati. Il protocollo applicativo realizzato è principalmente descritto all'interno del `data_transmission.c` dove sono presenti diverse funzioni tra cui:

- `receiving_data_selective_repeat`
- `sending_data_selective_repeat`

La prima funzione descrive il comportamento adottato dal destinatario mentre la seconda quello del mittente. Analizzando le suddette funzioni è possibile comprendere che qualsiasi operazione di trasferimento dati richiede le seguenti operazioni:

Inizializzazione della trasmissione Durante questa fase devono essere allocate e inizializzate tutte le strutture dati necessarie per eseguire la trasmissione tra cui:

- Un *buffer di ricezione* (lato client) e un *buffer di invio* (lato server) entrambi di dimensione N . La relazione ottimale¹⁵ tra la dimensione dei buffer N e la dimensione dell'apertura della finestra di spedizione W è:

$$N = 2W \quad (1.4)$$

- Un *buffer di supporto* di dimensione N utilizzato dal server per ospitare oggetti di tipo `DatagramTimer`; quest'ultimi sono necessari per eseguire l'associazione tra i timer e i corrispondenti pacchetti contenuti nel buffer di invio. Tale associazione avviene nel modo seguente: sia A il buffer di invio e sia B il buffer di supporto contenente i timer; il pacchetto $A[i]$ sarà associato al timer $B[i]$ con $0 \leq i \leq N - 1$. In altri termini il pacchetto k -esimo contenuto nel buffer di invio verrà associato al k -esimo oggetto `DatagramTimer` contenuto nel buffer di supporto. Nel codice sorgente questo buffer è denominato `DatagramTimer_buffer`.
- Un oggetto di tipo `TimeInfo` che verrà utilizzato per ospitare le informazioni necessarie per eseguire il calcolo e l'aggiornamento della durata dell'intervallo di timeout in base alle condizioni della rete.
- Un oggetto di tipo `NetworkStatistics` che verrà utilizzato per elaborare una serie di informazioni utili all'analisi delle prestazioni del protocollo tra cui: il numero totale di pacchetti inviati; il numero di timeout che si sono verificati durante la trasmissione; l'istante di avvio e di chiusura della trasmissione e altro ancora.
- Una serie di variabili di controllo necessarie a gestire la connessione. Invitiamo ad analizzare il codice sorgente per una descrizione dettagliata.

¹⁵Cfr. Achille Pattavina - *Reti di telecomunicazione, Networking e Internet, Seconda edizione*, McGraw-Hill, pp. 114

Instaurazione della trasmissione Questa fase, similmente a quanto avviene in TCP, prevede che mittente invii per primo un messaggio di controllo di tipo **REQ**; alla ricezione di quest'ultimo, il destinatario risponde con un messaggio di controllo di tipo **REQ_ACK**. Ricordiamo che i primi due messaggi non trasportano payload, ossia non hanno dati a livello applicativo. In seguito alla ricezione del messaggio **REQ_ACK**, il mittente potrà finalmente inviare il primo pacchetto ordinario contenente dati di interesse. Dato che i due host si scambiano tre pacchetti, questa procedura che instaura una connessione viene detta **handshake a tre vie** (*three-way handshake*)¹⁶.

Trasmissione dati In questa fase avviene la trasmissione dati vera e propria. Per motivi di spazio analizzeremo nel dettaglio solo il comportamento del mittente (il comportamento del destinatario è perfettamente analogo a quello previsto dal protocollo SR) il quale svolge le seguenti operazioni:

Incapsulamento dati In questa fase il mittente legge dal file system i dati da inviare al mittente e li incapsula all'interno di un pacchetto ordinario rappresentato da una struttura di tipo **Datagram**. Quest'ultimo verrà ovviamente inserito nel buffer di invio e automaticamente associato ad un timer logico.

Invio In questa fase il mittente invia al destinatario il pacchetto precedentemente creato attivando il relativo timer. Ovviamente, trattandosi di un protocollo SR, il mittente potrà inviare tutti quei pacchetti il cui numero di sequenza ricade in un certo intervallo. Per essere precisi, dato un buffer di dimensione pari a N , sia L l'estremo inferiore della finestra di spedizione e sia W la dimensione dell'apertura di quest'ultima; il mittente potrà inviare al destinatario tutti quei pacchetti senza la necessità di riscontro il cui numero di sequenza n soddisfi la seguente relazione¹⁷:

$$L \leq n \leq (L + W - 1)_{mod N} \quad (1.5)$$

Compatibilmente con il protocollo SR, il mittente riesegue la fase di incapsulamento dati e di invio finché possibile, ovvero nel momento in cui il mittente avrà inviato un numero di pacchetti non riscontrati pari alla dimensione della finestra di spedizione.

Gestione dei riscontri Al momento della ricezione di un messaggio di controllo di tipo **ACK** associato ad un pacchetto inviato il mittente provvede a:

- Ruotare la finestra di spedizione compatibilmente con il protocollo SR.
- Disattivare il timer associato al pacchetto.
- Eseguire l'aggiornamento della durata dell'intervallo di timeout nelle modalità descritte in 1.2.2.1.

¹⁶Cfr. Andrew S. Tanenbaum & David J. Wetherall - *Reti di calcolatori 5/Ed*, Pearson, pp. 239

¹⁷Cfr. Achille Pattavina - *Reti di telecomunicazione, Networking e Internet, Seconda edizione*, McGraw-Hill, pp. 107

Ritrasmissione Qualora un timer associato a un pacchetto inviato ma non riscontrato generi un segnale di timeout, mediante l'esecuzione della funzione `time_out_handler` il mittente esegue la ritrasmissione del pacchetto *raddoppiando* l'intervallo di timeout associato al pacchetto smarrito.

Chiusura connessione e termine della trasmissione Se il mittente non ha più dati da inviare al destinatario e rileva che anche l'ultimo pacchetto inviato ha ricevuto un riscontro positivo, viene eseguita la fase di terminazione della trasmissione. Il protocollo prevede che il mittente invii dapprima un messaggio di controllo di tipo `FIN` alla cui ricezione il destinatario risponderà con un messaggio di controllo di tipo `FIN_ACK`; il mittente risponde quindi con un messaggio di controllo di tipo `CLS` chiudendo definitivamente la propria connessione. Il destinatario potrà chiudere la propria connessione solo dopo la ricezione del messaggio `CLS`, tuttavia, poiché quest'ultimo è l'unico messaggio a non essere ritrasmesso in caso di smarrimento, anche in caso di mancata ricezione il destinatario potrà comunque chiudere la propria connessione con successo (dal momento che ha ricevuto tutti i pacchetti di interesse applicativo) dopo un certo lasso di tempo.

Capitolo 2

Guida all'utilizzo dell'applicazione

2.1 Descrizione dell'interfaccia

Per facilitare l'utilizzo dell'applicazione da parte degli utenti, è stata realizzata un'apposita interfaccia di tipo testuale in grado di comunicare all'utente tutte le informazioni necessarie al suo utilizzo. Ricordiamo innanzitutto che l'applicazione può essere avviata semplicemente lanciando l'eseguibile senza parametri attraverso un qualsiasi emulatore di terminale.

In questo paragrafo descriveremo pertanto l'interfaccia dell'applicazione inclusa la sintassi e il significato di tutti i comandi disponibili analizzandone anche l'implementazione.

2.1.1 I comandi client, server e options

Una volta avviato il programma verrà visualizzata la schermata *Home* riportata in figura 2.1 attraverso la quale l'utente potrà:

- Avviare il processo server;
- Avviare il processo client;
- Personalizzare le impostazioni;
- Terminare l'esecuzione dell'applicazione;

Come sappiamo, il corretto funzionamento dell'applicazione richiede l'esecuzione contemporanea di un processo server e di uno, o più, processi client. Poiché si è scelto di integrare le funzionalità lato client e server all'interno di uno stesso eseguibile, si è presentata la necessità di implementare un'interfaccia che consenta agli utenti di poter avviare all'occorrenza un processo server o client; questo è lo scopo dell'interfaccia *Home*.

I comandi richiamabili dalla schermata *home* sono quattro:

client [Indirizzo IP processo server] Questo comando, seguito dall'indirizzo IP della macchina su cui è in esecuzione il processo server, è utilizza-

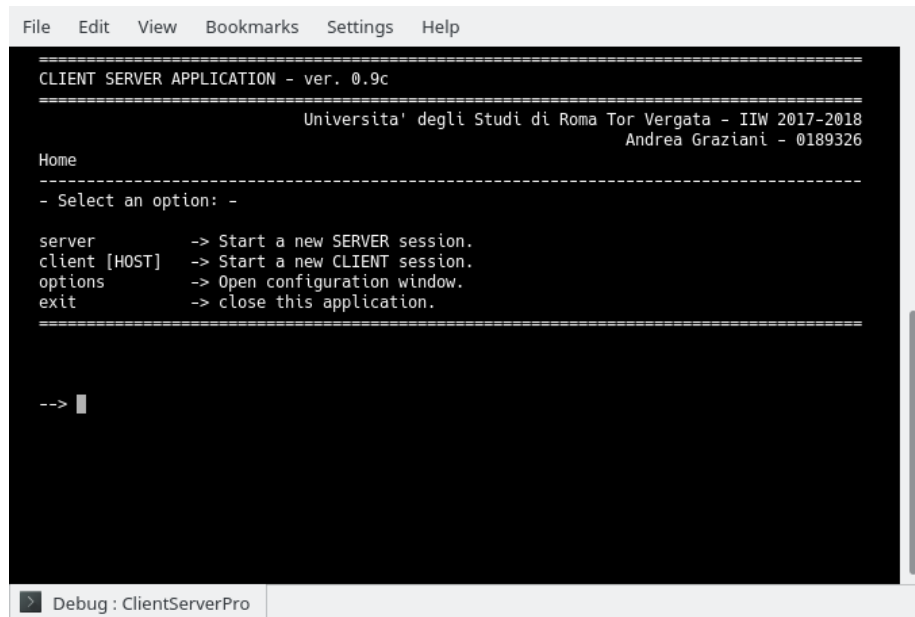


Figura 2.1: Schermata *Home*.

to per avviare, ovviamente, il processo client. Per evitare messaggi di errore, è importante specificare un indirizzo IP valido; da un punto di vista implementativo, il controllo della validità dell'indirizzo IP immesso dell'utente avviene mediante l'esecuzione della funzione `client_initialization_for_transmission` definita all'interno di `client_manager.c` la quale si occupa di allocare e inizializzare una struttura di tipo `DataNetwork` che verrà poi utilizzata per trasmettere messaggi al server.

server L'esecuzione di questo comando permette di avviare il processo server il quale una volta avviato rimarrà in attesa di messaggi di richiesta provenienti da uno o più altri processi client.

options Il comando permette agli utenti di poter accedere alla schermata *Options* riportata in figura 2.2 all'interno della quale, attraverso una serie di appositi comandi opportunamente descritti a video, è possibile personalizzare:

1. La probabilità di perdita dei messaggi scambiati con il processo server; questo parametro è settato a 0 per impostazione predefinita.
2. La dimensione della finestra di spedizione, settata a 20 per impostazione predefinita.
3. Abilitare/disabilitare il calcolo automatico della durata del timeout. Disabilitando il calcolo automatico, verrà adottato una durata fissa dell'intervallo di timeout; quest'ultimo valore può essere specificato dall'utente. Il calcolo automatico della durata di timeout è abilitato per impostazione predefinita.

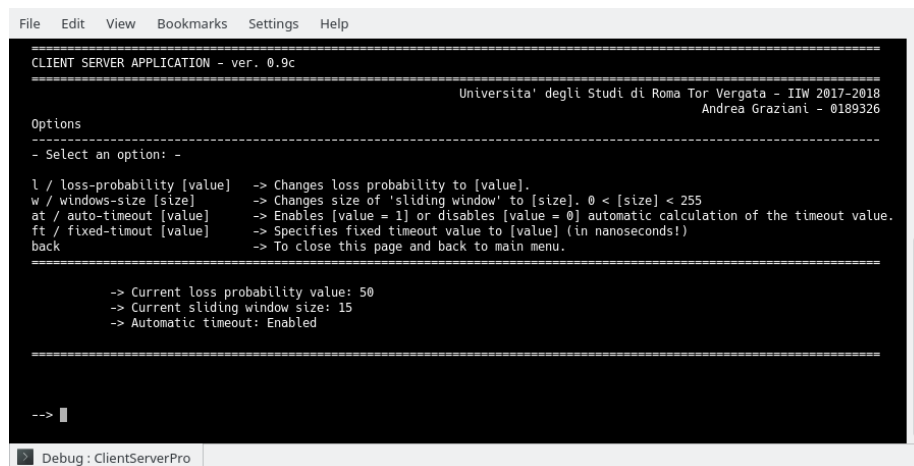


Figura 2.2: Schermata *Options*.

Da un punto di vista implementativo, prima dell'invio di un messaggio di richiesta al processo server da parte di un processo client, quest'ultimo legge le impostazioni direttamente dal file system attraverso un apposito file di impostazioni all'interno delle directory principale dell'applicazione. Qualora non venga trovato alcun file di impostazione (perché cancellato o mai creato), l'applicazione provvede a creare un nuovo file popolandolo con le impostazioni di default. Quando un processo client deve inoltrare un nuovo messaggio di richiesta al processo server, esso legge le informazioni sulle impostazioni dal file system popolandolo un'apposita struttura dati denominata `Settings.c` definita all'interno del file `Settings.h` e di cui riportiamo il codice in *listing 2.1*; dopo la fase di allocazione e inizializzazione della suddetta struttura, il processo client provvede alla sua serializzazione e copiatura all'interno dei messaggi di richiesta quindi copiata all'interno dei messaggi di richiesta affinché il processo server sia informato sulle impostazioni dell'utente. La logica di gestione delle impostazioni è descritta all'interno del file `Settings.c`.

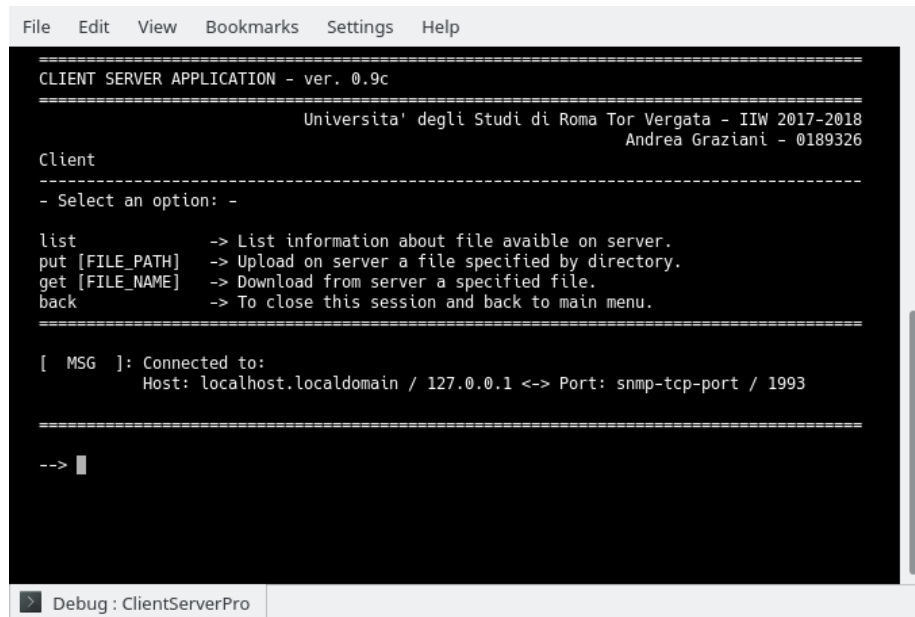
Listing 2.1: Implementazione della struttura `Settings`

```

1 typedef struct settings {
2
3     unsigned char sliding_window_size;
4     unsigned char loss_probability;
5     unsigned char auto_timeout;
6     unsigned long fixed_timeout;
7
8 } Settings;

```

`exit` Quest'ultimo comando permette all'utente di poter terminare l'esecuzione del programma.



```
File Edit View Bookmarks Settings Help

=====
CLIENT SERVER APPLICATION - ver. 0.9c
=====
Universita' degli Studi di Roma Tor Vergata - IIW 2017-2018
Andrea Graziani - 0189326

Client
-----
- Select an option: -

list      -> List information about file available on server.
put [FILE_PATH] -> Upload on server a file specified by directory.
get [FILE_NAME] -> Download from server a specified file.
back      -> To close this session and back to main menu.
=====

[ MSG ]: Connected to:
Host: localhost.localdomain / 127.0.0.1 <=> Port: snmp-tcp-port / 1993

=====

--> █

Debug : ClientServerPro
```

Figura 2.3: Schermata *Client*.

2.1.2 Il comando `list`

Per richiedere un elenco dei file disponibili sul server è possibile eseguire il comando `list` dalla schermata *Client* riportata in figura 2.3, quest'ultima richiamabile dalla schermata *Home* per mezzo del comando `client`. In caso di successo, verrà riprodotto, in formato tabellare, tutti i nomi, le dimensioni (esprese in byte) e le date di ultimo accesso e modifica dei file disponibili sul server; un esempio dell'output generato dall'esecuzione del comando `list` è riportata in figura 2.4.

Da un punto di vista implementativo, dopo la ricezione di un messaggio di richiesta di tipo `list`, il lavoro svolto dal processo server si divide nelle seguenti fasi:

Elaborazione directory Per elaborare correttamente la richiesta di tipo `list`, il processo server deve innanzitutto stabilire la directory all'interno risiedono i file. Durante questa operazione, attraverso le funzioni definite all'interno del file `directory_manager.c`, bisogna innanzitutto stabilire la directory *home* dell'utente che ha avviato il processo server all'interno della quale verranno posizionate tutte le directory necessarie per contenere tutti i file dell'applicazione (file condivisi, meta-dati, file di configurazione ecc.). Una volta nota la directory principale dell'applicazione verrà effettuata la ricerca di quella contenente i file. Il processo server provvede automaticamente alla creazione di tutte le directory necessarie in caso di primo avvio del programma o cancellazione delle directory suddette.

Lettura e formattazione dati Una volta stabilita la directory corretta il processo server provvede alla lettura di tutti i file *regolari* presenti in essa. Attraverso la funzione `get_file_information` definita all'interno del file

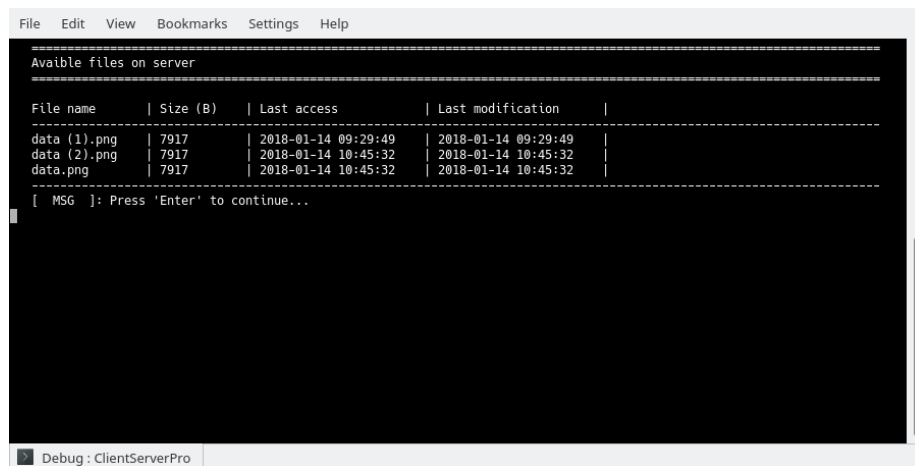


Figura 2.4: Output generato dall'esecuzione del comando `list`.

`FilePropertyList_struct.c`, il processo server crea, per ogni file elaborato, una struttura di tipo `FileProperty` all'interno della quale vengono inserite tutte le informazioni di rilievo sui file (nome, dimensione, ecc.). Tutti gli oggetti `FileProperty` creati verranno inseriti in una lista denominata `FilePropertyList`. Queste strutture sono entrambe definite all'interno del file `FilePropertyList_struct.h` di cui riportiamo il codice in *listing 2.2* e *2.3*.

Listing 2.2: Implementazione della struttura `FileProperty`

```
1 typedef struct file_property {
2
3     struct stat file_stat;
4     char *file_name;
5
6 } FileProperty;
```

Listing 2.3: Implementazione della struttura `FileProperty`

```
1 typedef struct file_property_list {
2
3     FileProperty *data;
4
5     /* pointer to next node */
6     struct file_property_list *next;
7
8 } FilePropertyList;
```

Serializzazione dati e invio Per poter inviare al client la struttura dati `FilePropertyList` precedentemente creata, quest'ultima deve essere innanzitutto serializzata; dopo la trasmissione, il processo client provvederà a de-serializzare i dati ricevuti e quindi a ricostruire la struttura dati originale per poi elaborarla e visualizzarla all'utente. La responsabilità di dover gestire l'operazione di serializzazione e de-serializzazione è affidata alle funzioni

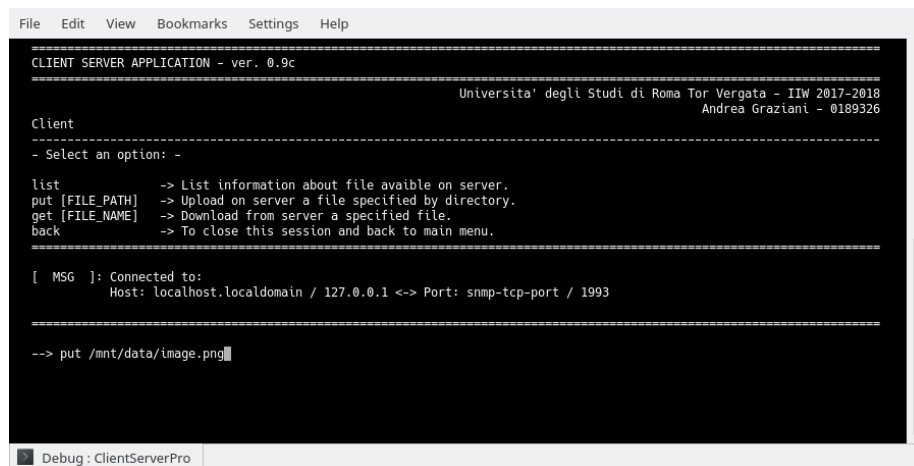
`FilePropertyList_serializer` e `FilePropertyList_deserializer` entrambe definite e implementate all'interno del file `FilePropertyList_struct.c`.

Durante l'esecuzione del processo server, potrebbe capitare di dover elaborare molte richieste di tipo `list` obbligando perciò tutti i thread responsabili di gestire le suddette richieste ad eseguire separatamente le stesse operazioni precedentemente descritte; ciò implica un enorme spreco di memoria e potenza computazionale. Pertanto si è preferito implementare un'architettura che preveda l'elaborazione della struttura `FilePropertyList` una volta soltanto aggiornandola qualora si verificassero operazioni di tipo `put`. Secondo tale implementazione, il processo server provvede alla creazione di un file temporaneo contenente la struttura `FilePropertyList` già popolata e serializzata in modo tale che tutti i thread responsabili di gestire richieste di tipo `list` possano inviare direttamente i dati contenuti nel file senza eseguire altre operazioni. La creazione e l'aggiornamento del suddetto file avviene eseguendo la funzione `__metadata_preload()`; questa funzione viene richiamata sia all'avvio del processo server sia in seguito all'elaborazione di un'operazione di tipo `put`.

2.1.3 I comandi `get` e `put`

La trasmissione di dati tra processo client e processo server, avviene attraverso i seguenti comandi richiamabili dalla schermata *Client*:

- get** [nome file] Questo comando permette al client di poter eseguire il download di un file disponibile sul server. Affinché l'operazione abbia successo, è importante specificare un nome valido di un file effettivamente disponibile sulla macchina server; in caso contrario il processo server invierà al client un messaggio di errore con la relativa descrizione.
- put** [percorso file] Questo comando, seguito da un percorso ad un file valido, permette di eseguire l'upload del suddetto sul server. Se il parametro immesso dell'utente non è valido viene visualizzato un messaggio di errore. Qualora sulla macchina server esistesse già un file avente lo stesso nome del file di cui si intende eseguire l'upload, il processo server rinomina opportunamente il file caricato per evitare conflitti (ad esempio `nome_file`, `nome_file_1`, `nome_file_2` ecc.). Al termine dell'operazione, il processo server si occuperà di aggiornare i meta-dati dei file disponibili richiamando la funzione `__metadata_preload`.



```
File Edit View Bookmarks Settings Help

CLIENT SERVER APPLICATION - ver. 0.9c

Universita' degli Studi di Roma Tor Vergata - IIW 2017-2018
Andrea Graziani - 0189326

Client
-----
- Select an option: -

list      -> List information about file available on server.
put [FILE_PATH] -> Upload on server a file specified by directory.
get [FILE_NAME] -> Download from server a specified file.
back      -> To close this session and back to main menu.
-----

[ MSG ]: Connected to:
Host: localhost.localdomain / 127.0.0.1 <=> Port: snmp-tcp-port / 1993

--> put /mnt/data/image.png
```

Figura 2.5: Esempio di utilizzo del comando `put`

2.2 Alcuni esempi di utilizzo dell'applicazione

Giunti a questo punto è doveroso presentare alcuni esempi di utilizzo dell'applicazione.

Supponiamo di avere in esecuzione il processo server dell'applicazione all'interno della macchina locale. Come abbiamo precedentemente specificato, affinché un client possa interagire con il server, l'utente deve specificare l'indirizzo IP della macchina dove è in esecuzione il processo server. In questo caso digiteremo dalla schermata *Home* il comando `client localhost`. Qualora digitassimo un indirizzo IP non valido verrà visualizzato un apposito messaggio di errore.

Supponiamo ora di dover eseguire un'operazione di tipo `put`; in questo scenario supponiamo perciò di voler eseguire l'upload di un file denominato `image.png` localizzato in `/mnt/data`.

Come riportato in figura 2.5, dalla schermata *Client* dobbiamo semplicemente digitare il comando `put /mnt/data/image.png` e dare invio; l'esito dell'operazione ci verrà prontamente comunicato mediante un apposito messaggio. È molto importante prestare attenzione all'argomento del comando `put`; la digitazione di un directory o di un file inesistente comporta la visualizzazione di un messaggio di errore. Eventualmente possiamo verificare l'avvenuto trasferimento del file attraverso l'esecuzione del comando `list`.

Supponiamo ora che un altro client esegua un'operazione di tipo `put` sullo stesso server caricando un file avente lo stesso nome del file caricato da noi; possiamo simulare quest'eventualità digitando ancora una volta il comando `put /mnt/data/image.png`. Potete facilmente verificare quanto avvenuto osservando la schermata riportata in figura 2.6 ottenuta attraverso l'esecuzione del comando `list`; come descritto in 2.1.3, l'applicazione rinomina opportunamente i file per evitare ogni conflitto.

Supponiamo ora di dover eseguire un'operazione di tipo `get` scaricando dal server il file `image.png`; quello che dobbiamo fare è digitare il comando `get image.png` dalla schermata *Client* come riportato in figura 2.7. Ricordiamo che

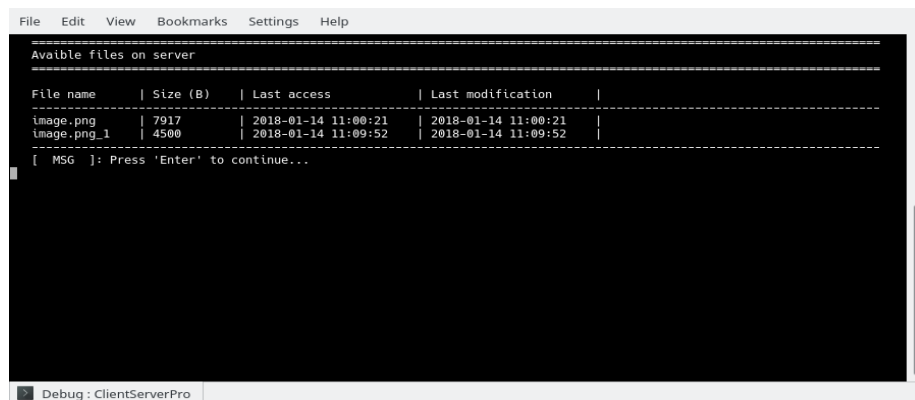


Figura 2.6: Comportamento dell'applicazione in seguito al caricamento di uno o più file con lo stesso nome

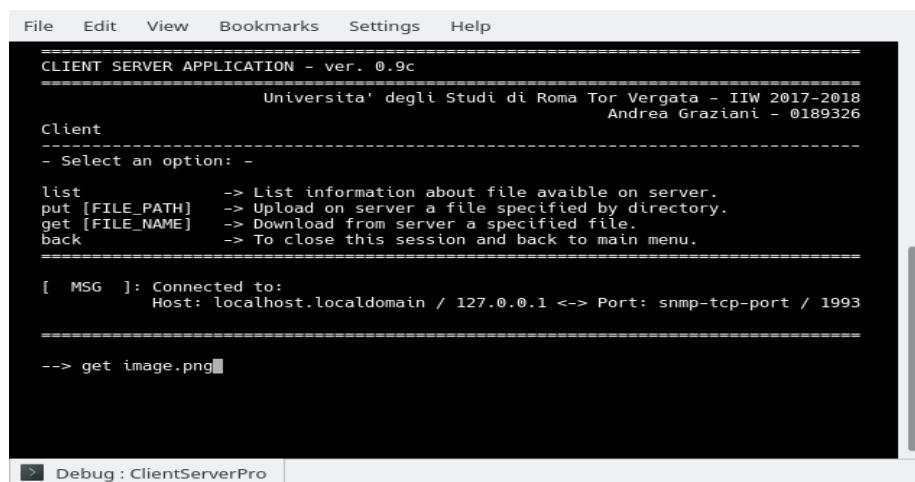


Figura 2.7: Esempio di utilizzo del comando `get`

il parametro del comando `get` deve coincidere con il nome di un file presente sul server; se si specifica un nome non valido, l'applicazione restituisce un messaggio di errore. L'esito della suddetta operazione ci verrà prontamente comunicato mediante un apposito messaggio. Ricordiamo che il file scaricato verrà salvato all'interno della directory `~/ClientServerApplication/client`.

2.3 Descrizione piattaforma software usata per lo sviluppo

Riportiamo nella tabella 2.1 i dati della piattaforma utilizzata per lo sviluppo e il test dell'applicazione; è opportuno precisare che tutti i test sono stati eseguiti in locale, ovvero mediante l'esecuzione di un processo server e di un processo client all'interno della stessa macchina.

Tabella 2.1: Dati piattaforma software usata per lo sviluppo

IDE	Eclipse IDE for C/C++ Developers
Versione IDE	Oxygen.2 Release (4.7.2), build-id 20171218-0600
Sistema operativo	Arch Linux
Versione kernel	v.4.14.13-1-ARCH
Architettura	x86_64

2.4 Guida alla compilazione e all'avvio dell'applicazione

Per eseguire correttamente la compilazione del codice sorgente dell'applicazione occorre:

1. Estrarre i file contenuti nell'archivio `ClientServerProject.zip`;
2. Eseguire il comando `make all` all'interno della cartella `makefile`; l'esecuzione di questo comando genera un eseguibile denominato `ClientServerProject` all'interno della suddetta cartella.

Dopo la compilazione, per avviare il programma è sufficiente lanciare l'eseguibile `ClientServerProject` da un qualsiasi emulatore di terminale.

2.5 Prestazioni

Riportiamo nella tabella 2.2 alcuni dati ottenuti sperimentalmente utili per valutare le prestazioni dell'applicazione.

Tabella 2.2: Analisi sperimentale delle prestazioni dell'applicazione trasferendo un file di dimensione pari a 13,2 MB.

Comando	Dimensione finestra spedizione	Probabilità di perdita	Dati Elaborati	Pacchetti Ritrasmessi	Totali ¹	Intervallo Timeout	Tempo (s) (ns)		Esito trasmissione
put	15	35%	9236	8265	17504	Dinamico	11	575087004	Riuscita
put	25	35%	9236	8126	17364	Dinamico	21	164630756	Riuscita
put	35	35%	9236	8210	17449	Dinamico	8	107812143	Riuscita
put	15	50%	9236	13821	23060	Dinamico	56	833894105	Riuscita
put	25	50%	9236	14157	23396	Dinamico	70	803986995	Riuscita
put	35	50%	874	1385	2260	Dinamico	46	358877914	Fallita
put	35	50%	9236	13909	23148	Dinamico	47	23890293	Riuscita
put	35	60%	9236	19603	28842	Dinamico	276	162591968	Riuscita
put	35	35%	9236	8410	17650	1 ms	7	109219969	Riuscita
put	35	35%	9236	8170	17409	10 ms	61	418463705	Riuscita
list	35	15%	2	1	6	Dinamico	0	1810316	Riuscita
list	25	50%	2	3	8	Dinamico	0	5559479	Riuscita
get	25	50%	9236	14040	23279	Dinamico	77	356101859	Riuscita
get	35	35%	9236	8500	17739	Dinamico	17	269738706	Riuscita
get	35	0%	9236	0	9239	Dinamico	0	150107979	Riuscita

¹Include sia pacchetti dati ordinari sia pacchetti di controllo (ACK, REQ, FIN ecc.)

Elenco delle figure

1.1	Schema del funzionamento di un server UDP concorrente.	3
1.2	Possibili scenari che si possono verificare dopo un timeout	11
2.1	Schermata <i>Home</i>	21
2.2	Schermata <i>Options</i>	22
2.3	Schermata <i>Client</i>	23
2.4	Output generato dall'esecuzione del comando <code>list</code>	24
2.5	Esempio di utilizzo del comando <code>put</code>	26
2.6	Comportamento dell'applicazione in seguito al caricamento di uno o più file con lo stesso nome	27
2.7	Esempio di utilizzo del comando <code>get</code>	27

Elenco delle tabelle

2.1	Dati piattaforma software usata per lo sviluppo	28
2.2	Analisi sperimentale delle prestazioni dell'applicazione trasferen- do un file di dimensione pari a 13,2 <i>MB</i>	29

Listings

1.1	Implementazione della funzione <code>start_server</code>	4
1.2	Implementazione della struttura <code>DataNetwork</code>	6
1.3	Implementazione della struttura <code>RequestPacket</code>	6
1.4	Implementazione della funzione <code>__receive_datagram</code>	7
1.5	Implementazione della struttura <code>ControlDatagram</code>	10
1.6	Implementazione della struttura <code>time_info</code>	12
1.7	Implementazione della funzione <code>TimeInfo_calc_time</code>	12
1.8	Implementazione della struttura <code>DatagramTimer</code>	14
1.9	Implementazione della funzione <code>DatagramTimer_init</code>	15
2.1	Implementazione della struttura <code>Settings</code>	22
2.2	Implementazione della struttura <code>FileProperty</code>	24
2.3	Implementazione della struttura <code>FileProperty</code>	24