# *Documentazione del Progetto: "trasporto affidabile multi-percorso"*

*per Reti di Calcolatori a.a. 2011/2012*


## *Componenti del gruppo*

- Mattia D'Ambrosio matricola: 0000589696
- Riccardo Serafini matricola: 0000593281


## *Finalità e scenario generale del progetto*

Si vuole simulare uno scambio di dati tra due programmi, attraverso una rete inaffidabile che scarta e ritarda i pacchetti transitanti.
Questa funzione viene ricoperta da un programma *ritardatore,* attraverso il quale passano tutti i pacchetti.
Il ritardatore comunica con due host diversi, in particolare con due programmi proxy, *proxysender* e *proxyreceiver* che inoltrano il flusso TCP attraverso dei datagram UDP.

Ci siamo occupati di implementare *proxysender, proxyreceiver* ed un protocollo che garantisca l'arrivo ordinato dei pacchetti al *receiver*.
Il progetto è stato realizzato secondo le specifiche fornite dal professore Vittorio Ghini disponibili all'indirizzo:
http://www.cs.unibo.it/~ghini/didattica/reti_lpr/progetti/aa2011-12/SpecificheProgettoLabReti_aa11-12.pdf

## *Protocollo implementato*

Tra *proxysender* e *proxyreceiver* viene implementato un protocollo UDP che permette di trasportare un flusso TCP tra i due host (*sender* e *receiver*) mantenendolo invariato.
Il *ritardatore* scarta i pacchetti e li ritarda in modo casuale, a volte mandando una notifica, a volte senza avvisare il processo mittente. Il protocollo UDP deve rimediare a queste perdite, cioé deve essere in grado di capire se un pacchetto non é arrivato a destinazione ed eventualmente inviarlo di nuovo.
I datagram UDP, inoltre, possono arrivare per loro natura in ordine diverso da quello di invio, quindi il protocollo deve anche prevedere il riordinamento dei pacchetti, in modo da inviarli nella giusta sequenza al *receiver*.

I datagram UDP hanno 5 byte di header in cui sono contenute delle informazioni relative al pacchetto:

- 4 byte contenenti un intero senza segno in endianess di rete che rappresenta **l'ID** univoco del pacchetto
- Un byte contenente un carattere che indica il **tipo** del pacchetto: 'B' rappresenta un pacchetto di dati, 'I' rappresenta un pacchetto ICMP, uno speciale pacchetto generato dal ritardatore che notifica la perdita di un datagram.

Dopo l'header é presente una stringa di dati di dimensione indefinita.

**Apertura del protocollo**

Non abbiamo implementato una vera e propria apertura del protocollo UDP, semplicemente questo si avvia nel momento in cui viene trasferito il primo datagram valido con ID = 1.

**Trasferimento dei dati**

Man mano che il *proxysender* legge i dati TCP dal sender, incapsula le stringhe di dimensione fissata che legge nell'header appena descritto. Il primo pacchetto ha id 1, e questo viene incrementato di uno ad ogni pacchetto letto, in modo che ad ogni datagram corrisponda un ID univoco.
Una volta formato, il *proxysender* invia il datagram ad una delle tre porte del *ritardatore* e lo aggiunge in coda alla lista dei pacchetti che devono essere confermati. Per garantire l'invio di tutti i pacchetti nonostante le perdite, infatti, e' necessario che il *proxyreceiver* confermi la ricezione di ogni datagram. La porta a cui il *proxysender* invia il datagram cambia ad ogni ciclo, per distribuire uniformemente il carico sulla rete.

Il *proxyreceiver* legge i datagram e li aggiunge ad una lista ordinata in base all'ID dei pacchetti. In testa alla lista c'é il pacchetto con ID minore, in coda alla lista quello con ID maggiore.
Quando viene ricevuto il datagram, inoltre, il *proxyreceiver* invia immediatamente un pacchetto di acknowledgement al *ritardatore*.
L'ACK e' un pacchetto di 6 byte: i primi 4 sono l'ID del pacchetto che deve essere confermato, il quinto byte e' il carattere 'B' e il sesto byte e' il primo carattere della stringa di dati del pacchetto da confermare.
L'ACK viene rispedito indietro dalla stessa porta del *ritardatore* da cui e' arrivato, perché ci sono maggiori possibilitá che questa non sia in BURST.
Il *proxyreceiver* mantiene una variabile con l'ID che sta attendendo per poter inviare nuovi dati al receiver. Se riceve un datagram con quell'ID, scorre la lista dalla testa ed invia tutti i pacchetti che trova con ID consecutivo. Appena trova un pacchetto con ID non successivo del precedente, si interrompe e memorizza tale ID nella variabile.
Se al *proxyreceiver* arrivano pacchetti duplicati o già inviati, questi vengono semplicemente scartati, così come se al *proxysender* arriva un ACK per un pacchetto già confermato.
Quando al *proxysender* arrivano gli ACK dei pacchetti, questi vengono semplicemente rimossi dalla lista. Se un pacchetto non viene confermato entro un tempo TIMEOUT, questo viene inviato nuovamente dal *proxysender* al *ritardatore* e re-inserito in coda alla lista.
Quando arriva un ICMP al *proxysender*, questo preleva il pacchetto corrispondente dalla lista, lo invia di nuovo e lo riaggiunge in fondo alla lista.
Quando arriva un ICMP al *proxyreceiver* viene inviato indietro un nuovo ACK per quel pacchetto.
Questa procedura si ripete in modo iterativo per un tempo indefinito, fino a quando il *sender* chiude la connessione TCP con il *proxysender* e la lista dei pacchetti da confermare nel *proxysender* si svuota completamente.
A questo punto si avvia il protocollo di chiusura UDP.

**Protocollo di chiusura**
Questo protocollo si avvia quando tutti i pacchetti sono stati trasferiti, e serve per informare il *proxyreceiver* che puo' chiudere la connessione TCP con il *receiver* e terminare l'esecuzione.
Senza questo scambio finale, infatti, il *proxyreceiver* non avrebbe modo di sapere se tutti i pacchetti sono stati inviati correttamente. Il *proxysender* avvia un handshake di chiusura inviando un datagram di 6 byte, con ID=0, tipo 'B' e con un carattere '1' dopo l'header. Il *proxyreceiver* risponde a questo datagram con un ACK, come al solito. Se questi pacchetti vengono scartati dal ritardatore, vengono inviati nuovamente come accade durante l'invio dei dati, finché al *proxysender* non arriva l'ACK.
A questo punto il *proxysender* invia un secondo pacchetto di chiusura, con ID=0, tipo 'B' e con un carattere '2' dopo l'header. Il *proxysender* ora sa che il *proxyreceiver* sa che deve chiudere l'esecuzione, quindi può terminare lui stesso l'esecuzione.
Questo ultimo pacchetto non deve essere confermato. Se il *proxyreceiver* lo riceve si chiude immediatamente, non prima di aver terminato l'invio di tutti i dati al *receiver*, altrimenti attende CLOSETIMEOUT secondi e termina la sua esecuzione, chiudendo la connessione TCP con il *receiver*.

## *Documentazione tecnica*
Il linguaggio di programmazione utilizzato per realizzare questo progetto, è "ANSI C", compilato con i flag "*-ansi -pedantic -Wall -Wunused*".
Abbiamo implementato *proxysender* e *proxyreceiver* in 4 files:

- **"proxysender.c"** contiene il main di esecuzione del proxysender
- **"proxyreceiver.c"** contiene il main di esecuzione del proxyreceiver
- **"utils.c"** contiene le procedure utilizzate da tutti e due i programmi
- **"utils.h"** contiene i prototipi delle funzioni, le strutture e le costanti del progetto

Abbiamo deciso di gestire le connessioni senza usare i thread, ma solo con l'uso di I/O Multiplexing ( select() ).

**Diagramma di flusso del proxyrsender**

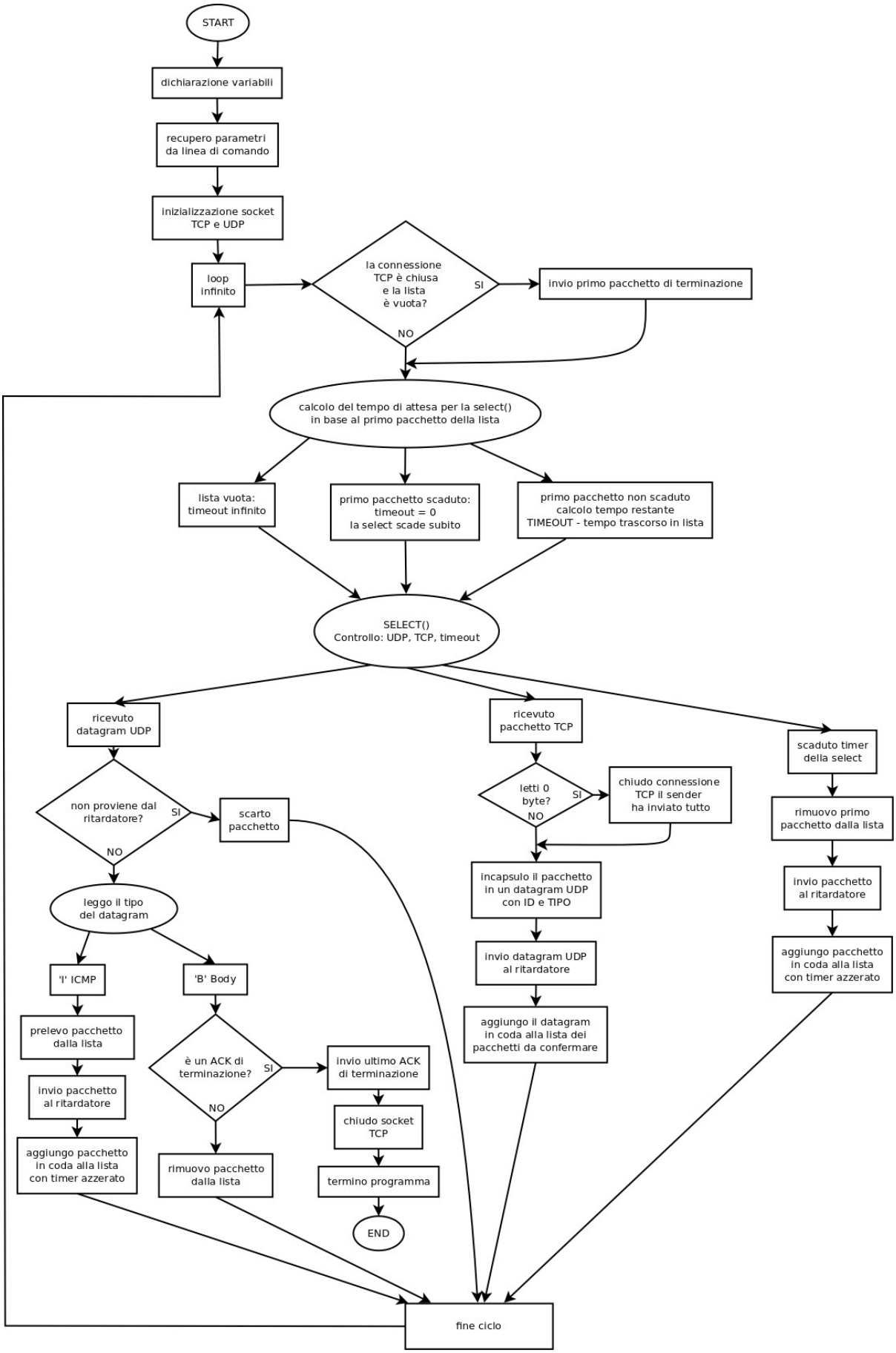
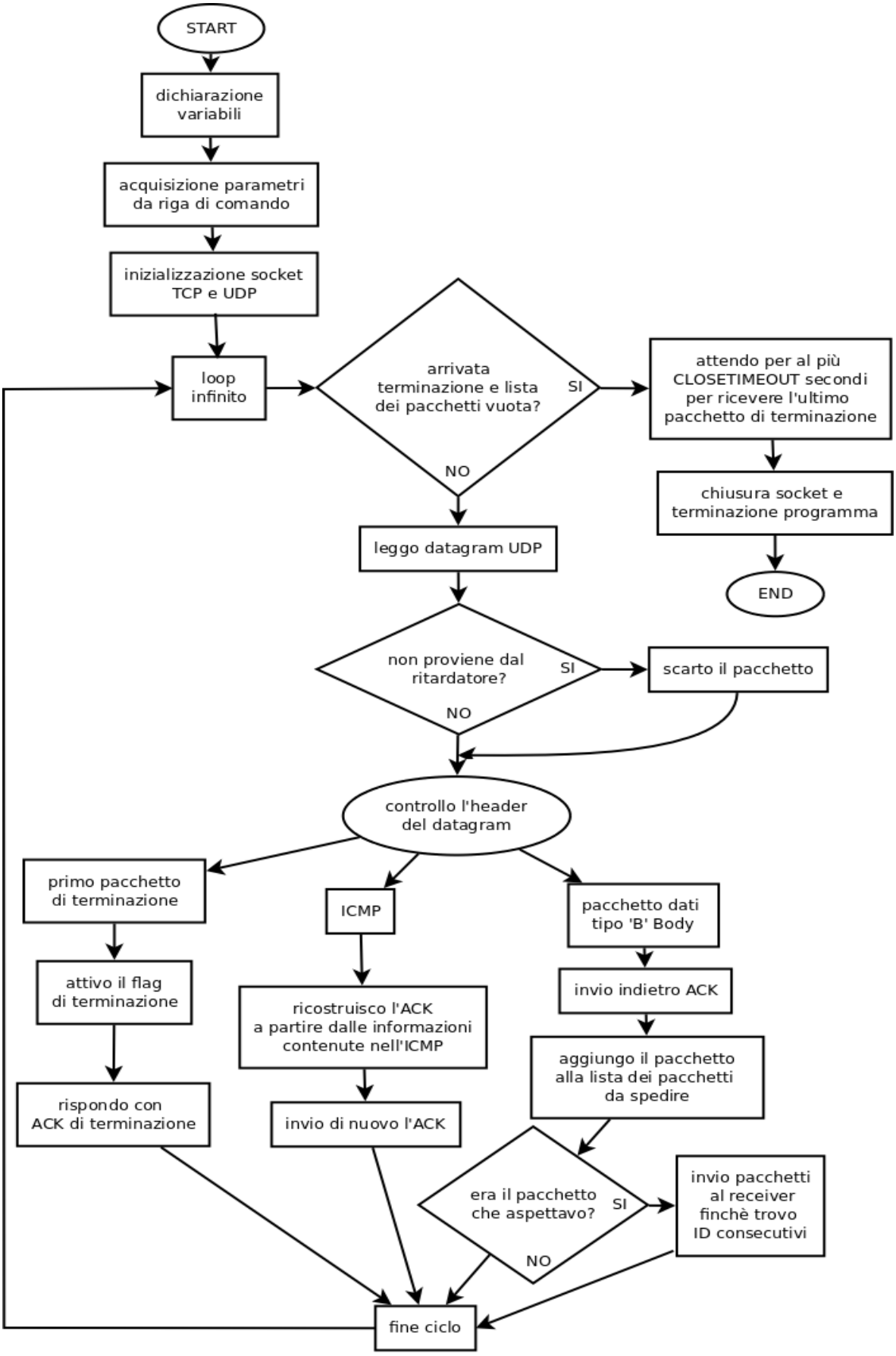


**Diagramma di flusso del proxyreceiver**



## *Utility*

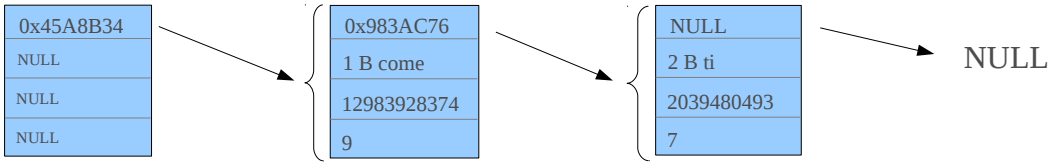
Nei files "**utils.c**" e **"utils.h"** abbiamo racchiuso le dichiarazioni di strutture, costanti e funzioni utilizzate da tutti e due i programmi proxy, in particolare:

**Strutture, liste dinamiche**

Abbiamo dichiarato la struttura **lista** per implementare delle liste dinamiche con *malloc()* e *free()*.
La struttura è così definita:

```
typedef struct lista{
    struct lista* next;          /* il puntatore all'elemento successivo */
    packet p;                    /* il pacchetto */
    struct timeval sentime;      /* il timestamp abbinato al pacchetto */
    int size;                    /* la dimensione del pacchetto */
} __attribute__((packed)) lista;
```

Si tratta di una lista con puntatore all'elemento successivo non circolare con sentinella, la sentinella è rappresentata da una struttura **lista** di cui si utilizza solo il campo **next**.




La lista è implementata come una coda, con inserimento in coda (push) ed eliminazione in testa (pop), ma è possibile anche inserire un elemento in ordine a seconda dell'ID e rimuovere un elemento con un determinato ID.

Le procedure realizzate sono le seguenti:

```
void aggiungi( lista* sentinella, packet p, int size );
void aggiungi_in_ordine( lista* sentinella, packet p, int size );
lista pop( lista* sentinella);
lista rimuovi( lista* sentinella, uint32_t id);
```

Queste funzioni aggiornano la lista allocando e deallcoando memoria con *malloc(),* e *free()* e aggiornando i puntatori nel modo corretto.

Abbiamo inoltre dichiarato altre due strutture: **packet** e **ICMP**, due contenitori per ingabbiare i buffer in lettura e scrittura ed estrarre le informazioni dell'header.
Sono dichiarate nel seguente modo:

```
typedef struct packet{
      /* l'ID del paccheto, intero senza segno a 32 bit */
      uint32_t id;
      /* il carattere che identifica il tipo di pacchetto */
      char tipo;
      /* una stringa per il corpo del pacchetto */
      char body[BODYSIZE];
} __attribute__((packed)) packet;

typedef struct ICMP{
      /* intero senza segno a 32 bit che rappresenta l'id dell'ICMP */
      uint32_t id;
      /* un carattere che rappresenta il tipo 'I' dell'ICMP */
      char tipo;
      /* l'ID del pacchetto scartato */
      uint32_t idpck;
} __attribute__((packed)) ICMP;
```

**Costanti**
Nel file "**utils.h**" abbiamo racchiuso le costanti del programma. I valori di questi costanti sono stati definiti dopo numerevoli tentativi, per cercare di individurare i valori ottimali in base a come si modificavano le prestazioni. Abbiamo definito queste costanti:

```
/* la dimensione massima dei pacchetti */
#define MAXSIZE 65000

/* la dimensione in byte dell'header dei datagram UDP */
#define HEADERSIZE 5

/* la dimensione massima del body dei pacchetti */
#define BODYSIZE (MAXSIZE) - (HEADERSIZE)

/* il timeout della select del proxysender */
#define TIMEOUT 0   /* secondi */
#define MSTIMEOUT 800000 /* microsecondi */

/* timeout di attesa della chiusura UDP da parte del proxyreceiver */
#define CLOSETIMEOUT 5
```

**Funzioni per i socket e I/O di rete**
Abbiamo definito in "**utils.c**" delle funzioni per creare socket TCP e UDP, fare bind, mettersi in ascolto, collegarsi e trasferire dati attraverso la rete.
Ad ogni chiamata di systemcall viene controllato il valore di ritorno ed in caso di errore il programma viene terminato stampando a video il tipo di errore.

Per inviare i dati, in particolare, abbiamo implementato la funzione **writen**, che scrive su un socket tutti i dati che si vogliono trasmettere, ripetendo la **send** se questa viene interrotta dal sistema operativo, con errore "EINTR".

## *Prestazioni del progetto*
Per testare la velocità di esecuzione del progetto abbiamo effettuato diverse prove, su tre macchine del laboratorio, lanciando il Ritardatore con lo scarto di pacchetti di default del 15%.

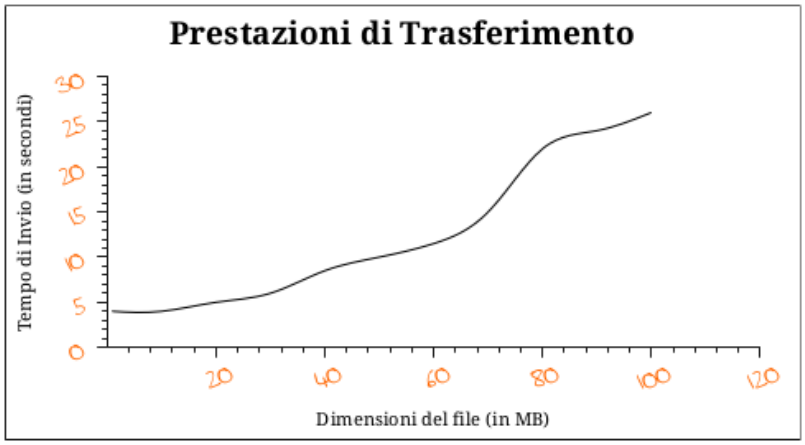
**Affidabilitá**
Il flusso di dati TCP inviato dal sender, deve essere identico a quello ricevuto dal receiver, i pacchetti devono arrivare tutti e nello stesso ordine.
Per verificare questo requisito abbiamo trasferito dei file e confrontato il fingerprint di questi con le utility *md5sum*, *sha1sum*, *diff* e *cmp*.
In tutte le prove effettuate, il file trasferito si é dimostrato identico a quello inviato.

**Latenza**
La latenza di base presente tra due macchine del laboratorio, misurata con il comando *ping*, ha un valore medio di 0.475 ms (andata e ritorno dell'ICMP). Considerato che i pacchetti inviati dal sender al receiver effettuano 4 hop, di cui due attraverso Ethernet e due tra processi della stessa macchina, possiamo approssimare il limite inferiore della latenza del progetto a 0.5 ms.
Attraverso l´utility *netcat*, abbiamo utilizzato il progetto come una chat monodirezionale, inviando piccole stringhe di testo. Dal momento in cui si preme ENTER al momento in cui la stringa compare sul terminale del receiver, puó passare un tempo variabile da meno di un secondo a 2 secondi.
I pacchetti che non vengono scartati dal Ritardatore, infatti, arrivano quasi immediatamente, mentre quelli che vengono scartati, vengono ritrasmessi dopo 0.8 secondi ed arrivano in media dopo un paio di secondi.

**Uso di CPU**
Per misurare l'utilizzo della CPU durante l'esecuzione di un trasferimento, abbiamo utilizzato il programma *htop*, che visualizza le statistiche di tutti i processi del sistema.
Durante il trasferimento di un file di 100 MB, il carico della CPU, misurato nelle macchine del laboratorio è circa del 6% per il proxysender e del 2% per il proxyreceiver.
L' utilizzo della CPU naturalmente dipende dalla quantitá di pacchetti in circolazione, un dato che si può empiricamente dedurre dal numero di pacchetti presenti nelle liste dinamiche del proxysender e del proxyreceiver.

**Occupazione di Memoria**
Come per l'utilizzo di CPU, abbiamo utilizzato il programma *htop* per verificare l'utilizzo di memoria. Anche questo dipende dalla quantità di pacchetti presenti nelle liste.
Questo valore può essere calcolato semplicemente moltiplicando la dimensione dei pacchetti trasferiti per il numero di pacchetti memorizzati nella lista.
Nel caso del trasferimento di file i pacchetti trasferiti hanno tutti una dimensione simile.
Il proxysender legge N byte alla volta dal sender, e i pacchetti vengono quindi dimensionati secondo questa costante.
Dopo diverse prove, abbiamo impostato questo valore a 65000 byte. Anche se é molto maggiore all' MTU della connessione Ethernet (1500 byte) e quindi si genera frammentazione, abbiamo riscontrato prestazioni decisamente migliori utilizzando questo valore.
In questo caso quindi 100 pacchetti in lista occuperanno circa 6 MB.

**Velocitá di trasferimento**
Abbiamo provato ad inviare files di diverse dimensioni, misurando il tempo impiegato per trasferirli con il programma *time* e con le statistiche stampate dal *proxysender*.
Come ci aspettavamo, il tempo di trasferimento cresce abbastanza linearmente con la dimensione del file inviato.
Per files piccoli peró, la differenza si nota meno perché il protocollo di chiusura UDP impiega circa sempre lo stesso tempo, indifferentemente dalla dimensione del file trasferito.
Abbiamo rappresentato l'andamento della velocità di trasferimento nel grafico sottostante:




| asse X | asse Y |
|---|---|
| Dimensioni del file (in MB) | Tempo di Invio (secondi) |
| 100 | 26 |
| 90 | 24 |
| 80 | 22 |
| 70 | 15 |
| 60 | 11.5 |
| 50 | 10 |
| 40 | 8.5 |
| 30 | 6 |
| 20 | 5 |
| 10 | 4 |
| 1 | 4 |

## *Conclusioni*
Date le considerazioni espresse in questa documentazione, riteniamo di aver implementato con successo il progetto ***trasporto affidabile multi-percorso*** rispettando a pieno le specifiche dettate dal professore.

```
1  #Parametri richiesti dalle specifiche
2  GCCFLAGS= -ansi -pedantic -Wall -Wunused
3
4  #COLORI
5  GREEN=\033[32m
6  NORMAL=\033[00m
7
8  all: utils.o proxysender.exe proxyreceiver.exe
9
10 #CREAZIONE FILES .exe
11 proxysender.exe: proxysender.o
12                  @gcc -o proxysender.exe proxysender.o utils.o
13                  @echo "\n $(GREEN)[PROXY SENDER OK]$(NORMAL)"
14
15 proxyreceiver.exe: proxyreceiver.o
16                    @gcc -o proxyreceiver.exe proxyreceiver.o utils.o
17                    @echo "\n $(GREEN)[PROXY RECEIVER OK]$(NORMAL)"
18
19 #CREAZIONE FILE OGGETTO
20 proxyreceiver.o: proxyreceiver.c
21             @gcc $(GCCFLAGS) -o proxyreceiver.o -c proxyreceiver.c
22
23 proxysender.o: proxysender.c
24                @gcc $(GCCFLAGS) -o proxysender.o -c proxysender.c
25
26 utils.o: utils.c
27         @gcc $(GCCFLAGS) -o utils.o -c utils.c
28         @echo "\n $(GREEN)[UTILS OK]$(NORMAL)"
29
30 # ELIMINAZIONE DEI FILE
31 clean:
32     @rm -f core* *.stackdump
33     @rm -f *.exe
34     @rm -f *.o
35     @echo "CLEAN...      $(GREEN)[OK]$(NORMAL)"
36
```

```c
/*
 *     proxyreceiver.c : usando datagram UDP simula una connessione TCP
 *     attraverso un programma ritardatore che scarta i pacchetti in modo
 *     casuale, simulando una rete reale
 */

/* file di header */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* file di header del progetto */
#include "utils.h"

int main(int argc, char *argv[]){

    /* variabile globale che contiene il numero di elementi nella lista
     * viene tenuta aggiornata dalle funzioni di manipolazione delle
     * liste */
    extern int nlist;

    /* variabile booleana che controlla se è arrivato il datagram
     * di terminazione del protocollo */
    int arrivata_terminazione = 0;

    int udp_sock, tcp_sock, nread, local_port, remote_port;

    struct sockaddr_in from, to;

    lista  buf_l;
    packet buf;
    char remote_ip[40];
    char ritardatore_ip[40];
    uint16_t porte_ritardatore[3];
    in_addr_t ip_ritardatore;

    /* una lista ordinata che contiene i pacchetti da spedire al
       receiver */
    lista to_send;

    uint32_t id_to_wait = 1;

    fd_set rfds;
    struct timeval tv;
    int retval;

    struct in_addr indirizzo_ritardatore, indirizzo_receiver;

    /* recupero parametri da riga di comando
      * nessun parametro è obbligatorio, infatti se un parametro non
     * viene fornito, viene utilizzato il valore di default */

    /* IP del ritardatore, default localhost */
    if(argc > 1)
        strcpy(ritardatore_ip, argv[1]);
    else
        strcpy(ritardatore_ip, "127.0.0.1");
```

```
63      /* IP del receiver, default localhost */
64      if(argc > 2)
65          strcpy(remote_ip, argv[2]);
66      else
67          strcpy(remote_ip, "127.0.0.1");
68      /* porta UDP del proxyreceiver, default 63000 */
69      if(argc > 3)
70          local_port = atoi(argv[3]);
71      else
72          local_port = 63000;
73      /* porta TCP del receiver, default 64000 */
74      if(argc > 4)
75          remote_port = atoi(argv[4]);
76      else
77          remote_port = 64000;
78      /* prima porta del ritardatore latoreceiver, serve solo per
79       * filtrare i pacchetti estranei */
80      if(argc > 5)
81          porte_ritardatore[0] = htons(atoi(argv[5]));
82      else
83          porte_ritardatore[0] = htons(62000);
84
85      porte_ritardatore[1] = htons(ntohs(porte_ritardatore[0])+1);
86      porte_ritardatore[2] = htons(ntohs(porte_ritardatore[1])+1);
87
88      /* copia dell'ip del ritardatore in endianess di rete per
89       * migliori performance durante il filtraggio */
90      /* stampa dei parametri utilizzati */
91      indirizzo_ritardatore = DNSquery(ritardatore_ip);
92      ip_ritardatore = indirizzo_ritardatore.s_addr;
93      indirizzo_receiver = DNSquery(remote_ip);
94
95      printf("IP ritardatore: %s\n", inet_ntoa(indirizzo_ritardatore));
96      printf("IP receiver: %s\n", inet_ntoa(indirizzo_receiver));
97      printf("porta proxyreceiver: %d\n", local_port);
98      printf("porta receiver: %d\n", remote_port);
99      printf("prima porta ritardatore: %d\n", ntohs(porte_ritardatore[0]));
100
101     /* inizializzazione del socket UDP */
102     udp_sock = UDP_sock(local_port);
103     /* inizializzazione del socket TCP, viene ritornato il socket di
104      * connessione, non quello generico */
105     tcp_sock = TCP_connection_send(indirizzo_receiver, remote_port);
106
107     name_socket(&from, htonl(INADDR_ANY), 0);
108
109     /* il programma consiste in un loop infinito interrotto da
110      * determinati datagram UDP */
111     while(1) {
112         /* se è stato ricevuto il datagram di terminazione attendo
113          * 6 secondi per ulteriori pacchetti, nel caso in cui il
114          * proxysender non avesse ricevuto l'ACK di terminazione.
115          * Se durante CLOSETIMEOUT secondi ricevo l'ACK finale termino,
116          * oppure se passano CLOSETIMEOUT secondi senza ricevere nulla*/
117         if(arrivata_terminazione && nlist == 0){
118             FD_ZERO(&rfds);
119             FD_SET(udp_sock, &rfds);
120             tv.tv_sec = CLOSETIMEOUT;
121             tv.tv_usec = 0;
122             retval = select(udp_sock+1, &rfds, NULL, NULL, &tv);
123             if (retval == -1)
124                perror("select()");
```

```
125                 else if (retval == 0){
126                     close(udp_sock);
127                     close(tcp_sock);
128                     printf("\nTimeout scaduto: terminazione.\n");
129                     exit(EXIT_SUCCESS);
130                 }
131             }
132 
133             /* DEBUG: stampo quanti pacchetti ci sono nella lista dei
134              * pacchetti da inviare al receiver */
135             printf("\rpacchetti rimanenti: %d  ", nlist);
136             fflush(stdout);
137 
138             /* leggo un datagram UDP dal Ritardatore */
139             nread = readn(udp_sock, (char*)&buf, MAXSIZE, &from);
140 
141             /* se il pacchetto non proviene dal ritardatore viene scartato*/
142             if( nread > HEADERSIZE &&
143                 from.sin_addr.s_addr == ip_ritardatore &&
144                (from.sin_port == porte_ritardatore[0] ||
145                 from.sin_port == porte_ritardatore[1] ||
146                 from.sin_port == porte_ritardatore[2])){
147 
148                 /* se si tratta di un pacchetto "Body" */
149                 if(buf.tipo == 'B'){
150                     /* se ha id=0 è un pacchetto di terminazione */
151                     if(ntohl(buf.id) == 0 &&
152                        buf.body[0] == '1' &&
153                        !arrivata_terminazione){
154                         arrivata_terminazione = 1;
155                         printf("\nArrivata terminazione\n");
156                     }
157                     if(ntohl(buf.id) == 0 &&
158                        buf.body[0] == '2' &&
159                        arrivata_terminazione){
160                         /* se è il secondo pacchetto di terminazione, il
161                          * protocollo di chiusura è completo perchè il
162                          * proxysender è stato informato della terminazione
163                          * quindi viene chiuso il programma e i socket*/
164                         printf("\nArrivato ACK, terminazione.\n");
165                         close(udp_sock);
166                         close(tcp_sock);
167                         exit(EXIT_SUCCESS);
168                     }
169 
170                     /* Imposto la destinazione dell'ACK
171                      * inviandolo sullo stesso canale dal quale è arrivato
172                      * l'udp perchè probabilmente non è in BURST */
173                     name_socket(&to, ip_ritardatore, ntohs(from.sin_port));
174 
175                     /* invio ACK del pacchetto ricevuto */
176                     writen(udp_sock, (char*)&buf, HEADERSIZE+1, &to);
177 
178                     /* solo se il pacchetto ricevuto ha un ID >= del primo ID
179                      * della lista, e quindi deve essere ancora inviato,
180                      * lo aggiungo alla lista dei pacchetti da inviare */
181                     if(ntohl(buf.id) >= id_to_wait){
182                         buf.id = ntohl(buf.id);
183                         aggiungi_in_ordine(&to_send, buf, nread-HEADERSIZE);
184                     }
185 
186                     /* se il datagram appena arrivato ha permesso l'invio di
```

```
187                      * altri pacchetti in ordine al receiver, li rimuovo dalla
188                      * lista e li invio in TCP */
189                     while(to_send.next != NULL &&
190                             to_send.next->p.id==id_to_wait){
191                         /* azzero i buffer */
192                         memset(&buf_l, 0, sizeof(lista));
193                         memset(&buf_l.p, 0, sizeof(packet));
194                         /* rimuovo il pacchetto in testa alla lista */
195                         buf_l = pop(&to_send);
196                         /* invio il pacchetto al receiver senza header*/
197                         writen( tcp_sock,
198                                 buf_l.p.body,
199                                 buf_l.size,
200                                 NULL
201                                );
202                         id_to_wait++;
203                         /* DEBUG: stampo il numero di pacchetti nella lista */
204                         printf("\rpacchetti rimanenti: %d  ", nlist);
205                         fflush(stdout);
206                     }
207                 }
208                 /* se il datagram UDP ricevuto è di tipo ICMP
209                  * significa che l'ACK appena inviato è stato scartato,
210                  * quindi viene ricostruito a partire dall'id contenuto nell'
211                  * ICMP e inviato nuovamente */
212                 if(buf.tipo == 'I') {
213                     buf.id = ((ICMP*)&buf)->idpck;
214                     buf.tipo = 'B';
215                     writen(udp_sock, (char*)&buf, HEADERSIZE+1, &to);
216                 }
217             } else {
218                 printf("ricevuto pacchetto UDP estraneo, scarto.\n");
219             }
220         }
221         /* l'esecuzione non raggiunge mai questo punto del codice */
222 }
223
```

```c
/*
 *     proxysender.c : usando datagram UDP simula una connessione TCP
 *     attraverso un programma ritardatore che scarta i pacchetti in modo
 *     casuale, simulando una rete reale
 */

/* File di header */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/select.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <sys/time.h>

/* File di header del progetto */
#include "utils.h"

int main(int argc, char *argv[]){

    /* numero di elementi nella lista,
     * variabile globale aggiornata dalle funzioni delle liste */
    extern int nlist;

    int udp_sock = 0;
    int tcp_sock = 0;
    int nread = 0;
    int local_port_tcp = 0;
    int local_port_udp = 0;
    int rit_port[3];
    uint16_t porte_rit[3];
    int rit_turno = 0;

    struct sockaddr_in to, from;

    packet buf_p;
    lista  buf_l;

    char remote_ip[16];
    in_addr_t ip_ritardatore;

    uint32_t progressive_id = 0;

    /* per la select */
    fd_set rfds;
    int retsel;
    int fdmax;

    /* contatori per le statistiche */
    int quanti_timeout = 0;
    int quanti_pacchetti_tcp = 0;
    int quanti_datagram_inviati = 0;
    int quanti_icmp = 0;
    int quanti_ack = 0;
    int overhead = 0;

    struct in_addr indirizzo_ritardatore;

    /* per gestire il timeout dei pacchetti */
    struct timeval timeout;

```

```
63      /* timestamp di inizio e fine dell' esecuzione */
64      struct timeval inizio, fine;
65
66      /* elemento sentinella della lista dei pacchetti che
67       * devono ricevere ACK */
68      lista to_ack;
69      /* la lista viene inizializzata come vuota */
70      to_ack.next = NULL;
71
72      /* recupero dei parametri da linea di comando.
73       * nessun parametro è obbligatorio, infatti se un parametro non
74       * viene fornito, viene utilizzato il valore di default */
75
76      /* IP del Ritardatore, default localhost */
77      if(argc > 1)
78          strcpy(remote_ip, argv[1]);
79      else
80          strcpy(remote_ip, "127.0.0.1");
81
82      /* porta in ascolto del proxysender, default 59000 */
83      if(argc > 2)
84          local_port_tcp = atoi(argv[2]);
85      else
86          local_port_tcp = 59000;
87
88      /* porta in uscita del proxysender, default 60000 */
89      if(argc > 3)
90          local_port_udp = atoi(argv[3]);
91      else
92          local_port_udp = 60000;
93
94      /* prima porta del ritardatore */
95      if(argc > 4)
96          rit_port[0] = atoi(argv[4]);
97      else
98          rit_port[0] = 61000;
99
100         rit_port[1] = rit_port[0]+1;
101         rit_port[2] = rit_port[1]+1;
102
103     /* Variabili copiate nell'endianess di rete per migliorare le
104      * prestazioni quando vengono filtrati i pacchetti in ingresso */
105     indirizzo_ritardatore = DNSquery(remote_ip);
106     ip_ritardatore = indirizzo_ritardatore.s_addr;
107     porte_rit[0] = htons(rit_port[0]);
108     porte_rit[1] = htons(rit_port[1]);
109     porte_rit[2] = htons(rit_port[2]);
110
111     printf("IP ritardatore: %s\n", inet_ntoa(indirizzo_ritardatore));
112     printf("porta TCP: %d\n", local_port_tcp);
113     printf("porta UDP: %d\n", local_port_udp);
114     printf("porta ritardatore: %d\n", rit_port[0]);
115
116   /*********************************************************/
117
118     /* inizializzazione del socket UDP (utils.c) */
119     udp_sock = UDP_sock(local_port_udp);
120
121     printf("In attesa di connessione da parte del sender...\n");
122
123     /* inizializzazione del socket TCP (utils.c)
124      * il valore restituito è il socket di connessione, non quello
```

```
125         * iniziale generico */
126        tcp_sock = TCP_connection_recv(local_port_tcp);
127
128        /* salvo il tempo di inizio dell'esecuzione */
129        if(gettimeofday(&(inizio), NULL)){
130            printf("gettimeofday() fallita, Err: %d \"%s\"\n",
131                        errno,
132                        strerror(errno)
133                );
134            exit(EXIT_FAILURE);
135        }
136
137        /* calcolo NFDS: indica il range dei files descriptors a cui siamo
138         * interessati per la select() */
139        fdmax = (udp_sock > tcp_sock)? udp_sock+1 : tcp_sock+1;
140
141        /* il programma consiste in un ciclo infinito */
142        while(1){
143
144            /* se la connessione TCP è stata chiusa e sono stati inviati tutti
145             * i pacchetti invio segnale di terminazione al proxyreceiver */
146            if( tcp_sock == -1 && nlist == 0){
147                buf_p.id = 0;
148                buf_p.tipo = 'B';
149                buf_p.body[0] = '1';
150                quanti_datagram_inviati++;
151                writen(udp_sock, (char*)&buf_p, HEADERSIZE+1, &to);
152                aggiungi(&to_ack, buf_p, HEADERSIZE + 1);
153                printf("\nTutti i pacchetti inviati.\n");
154            }
155
156            /* debug: stampa quanti pacchetti ci sono nella lista */
157            printf("\rpacchetti rimanenti: %d  ", nlist);
158            fflush(stdout);
159
160            /* incremento del turno della porta del ritardatore */
161            rit_turno = (rit_turno + 1) % 3;
162            name_socket(&to, ip_ritardatore, rit_port[rit_turno]);
163
164            /* inizializzo le strutture per la select()
165             * il socket tcp viene aggiunto solo se è ancora attivo */
166            FD_ZERO(&rfds);
167            if(tcp_sock != -1)
168                FD_SET(tcp_sock, &rfds);
169            FD_SET(udp_sock, &rfds);
170
171            /* azzeramento dei buffer */
172            memset(&buf_l, 0, sizeof(lista));
173            memset(&buf_p, 0, sizeof(packet));
174
175            /* se la lista è vuota, la select attende all'infinito */
176            if(to_ack.next == NULL)
177                retsel = select(fdmax, &rfds, NULL, NULL, NULL);
178            else {
179                /* calcolo del timeout con cui chiamare la select:
180                 * dipende dal tempo di inserimento del primo pacchetto. */
181                timeout = to_ack.next->sentime;
182                if(controlla_scadenza(&timeout)){
183                    timeout.tv_sec = 0;
184                    timeout.tv_usec = 0;
185                }
186                retsel = select(fdmax, &rfds, NULL, NULL, &timeout);
```

```
187             }
188             /* se la select fallisce viene restituito errore */
189             if (retsel == -1){
190                 printf("select() fallita, Err: %d \"%s\"\n",
191                             errno,
192                             strerror(errno)
193                         );
194                  exit(EXIT_FAILURE);
195             }
196             /* altrimenti, se la select è stata interrotta da un socket */
197             else if (retsel) {
198 
199                 /* se è stato il socket TCP a svegliare la select significa
200                  * che ci sono dati disponibili da parte del sender*/
201                 if(FD_ISSET(tcp_sock, &rfds)){
202                     quanti_pacchetti_tcp++;
203                     /* lettura dei dati dal sender */
204                     nread = readn(tcp_sock,
205                                   (char*)buf_p.body,
206                                   BODYSIZE,
207                                   NULL
208                                  );
209                     /* NEL CASO DEL TCP SIAMO SICURI CHE IL MESSAGGIO
210                      * PROVENGA DAL SENDER */
211                     /* se non viene letto nulla, significa che il sender ha
212                      * chiuso la connessione quindi ha inviato tutti i dati */
213                     if(nread == 0){
214                         printf("\nLa connessione TCP e' stata chiusa\n");
215                         /* chiusura della connessione TCP */
216                         close(tcp_sock);
217                         /* il descrittore del socket viene impostato a -1 per
218                          * avviare la chiusura del protocollo UDP */
219                         tcp_sock = -1;
220                     } else {
221                         /* l'ID dei datagram UDP è progressivo, parte da 1*/
222                         progressive_id++;
223                         /* l'ID viene convertito nell'endianess di rete */
224                         buf_p.id = htonl(progressive_id);
225                         buf_p.tipo = 'B';
226 
227                         /* invio del datagram UDP contenente l'header
228                          * generato e il body ricevuto dal sender */
229                         quanti_datagram_inviati++;
230                         writen(udp_sock,
231                                (char*)&buf_p,
232                                HEADERSIZE + nread,
233                                &to);
234                         /* l'ID viene riconvertito nell'endianess della
235                          * macchina e il pacchetto viene inserito nella lista
236                          * dei datagram da confermare con ACK.
237                          * L'inserimento avviene in coda */
238                         buf_p.id = ntohl(buf_p.id);
239                         aggiungi(&to_ack, buf_p, nread+HEADERSIZE);
240                     }
241                 }
242                 /* la select si è interrotta perchè ci sono dati disponibili
243                  * da parte del proxyreceiver */
244                 if(FD_ISSET(udp_sock, &rfds)){
245 
246                     /* lettura dei dati UDP */
247                     nread = readn( udp_sock, (char*)&buf_p, MAXSIZE, &from);
248 
```

```
249                 /* se il pacchetto non proviene dal ritardatore,
250                  * viene scartato */
251                 if( nread > HEADERSIZE &&
252                     from.sin_addr.s_addr == ip_ritardatore &&
253                      (  from.sin_port == porte_rit[0]
254                      || from.sin_port == porte_rit[1]
255                      || from.sin_port == porte_rit[2])){
256 
257                     /* se viene ricevuto un ACK viene rimosso il
258                      * rispettivo pacchetto dalla lista */
259                     if(buf_p.tipo == 'B'){
260                         quanti_ack++;
261                         buf_p.id = ntohl(buf_p.id);
262                         buf_l = rimuovi(&to_ack, buf_p.id);
263                         /* se viene ricevuto un ACK del segnale di
264                          * terminazione posso chiudere il programmma */
265                         if(buf_p.id == 0 && tcp_sock == -1 && nlist == 0){
266                             printf("\nACK chiusura, terminazione.\n");
267                             /* Invio ACK finale al proxyreceiver */
268                             buf_p.body[0] = '2';
269                             writen(udp_sock,
270                                    (char*)&buf_p,
271                                    HEADERSIZE+1,
272                                    &to);
273                             close(udp_sock);
274                             /* prendo il tempo di fine esecuzione salvo
275                              * il tempo di inizio dell'esecuzione */
276                             if(gettimeofday(&(fine), NULL)){
277                                 printf("gettimeofday(), Err: %d %s\n",
278                                             errno,
279                                             strerror(errno)
280                                       );
281                                 exit(EXIT_FAILURE);
282                             }
283                             /* sottraggo i tempi */
284                             timeval_subtract(&fine, &fine, &inizio);
285                             /* stampo le statistiche */
286                             overhead = (quanti_datagram_inviati - quanti_pa
    cchetti_tcp) * 100 / quanti_pacchetti_tcp;
287                             printf("---- statistiche ----\n");
288                             printf("timeout: %d\n", quanti_timeout);
289                             printf("pacchetti tcp ricevuti: %d\n",
290                                     quanti_pacchetti_tcp);
291                             printf("datagram udp inviati: %d\n",
292                                     quanti_datagram_inviati);
293                             printf("icmp: %d\n", quanti_icmp);
294                             printf("ack: %d\n", quanti_ack);
295                             printf("overhead: %d%%\n", overhead);
296                             printf("tempo di esecuzione: %d sec\n",
297                                     (int)fine.tv_sec);
298                             exit(EXIT_SUCCESS);
299                         }
300                     }
301                     /* se viene ricevuto ICMP il pacchetto corrispondente
302                      * deve essere rimosso dalla lista, inviato di nuovo e
303                      * inserito in coda alla lista */
304                     if(buf_p.tipo == 'I'){
305                         quanti_icmp++;
306                         buf_l = rimuovi(&to_ack, ((ICMP*)&buf_p)->idpck);
307                         /* se il pacchetto non è nella lista ignoro ICMP */
308                         if(buf_l.p.tipo != 'E'){
309                             buf_l.p.id = htonl(buf_l.p.id);
```

```
310                                 quanti_datagram_inviati++;
311                                 writen(udp_sock,
312                                        (char*)&buf_l.p,
313                                        buf_l.size,
314                                        &to);
315                                 buf_l.p.id = ntohl(buf_l.p.id);
316                                 aggiungi(&to_ack, buf_l.p, buf_l.size);
317                             }
318                         }
319                     } else {
320                         printf("ricevuto datagram UDP estraneo, scarto.\n");
321                     }
322                 }
323             } else {
324                 /* se la select viene terminata dopo il timeout specificato
325                  * bisogna inviare nuovamente il primo pacchetto della lista
326                  * e reinserirlo in coda */
327                 quanti_timeout++;
328                 buf_l = pop(&(to_ack));
329                 buf_l.p.id = htonl(buf_l.p.id);
330                 quanti_datagram_inviati++;
331                 writen( udp_sock, (char*)&buf_l.p, buf_l.size, &to);
332                 buf_l.p.id = ntohl(buf_l.p.id);
333                 aggiungi(&to_ack, buf_l.p, buf_l.size);
334         }
335         }
336         /* L'esecuzione non arriva mai a questo punto */
337 }
338
```

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/time.h>

#include "utils.h"

/* QUERY DNS */
/* riceve una stringa contenente il nome di cui cercare l'IP */
struct in_addr DNSquery (const char* hostname){
    struct hostent *he;
    struct in_addr **addr_list;

    if ((he = gethostbyname(hostname)) == NULL) {
        printf("Nome DNS non trovato\n");
        exit(1);
    }

    addr_list = (struct in_addr **)he->h_addr_list;

    return *addr_list[0];
}

/* ---------------------- Sottrarre TIMEVAL -------------------------
 * http://www.gnu.org/software/libc/manual/html_node/Elapsed-Time.html
 */

/* CALCOLA X - Y
 * ritorna 1 se il risultato è negativo */

/* Subtract the `struct timeval' values X and Y,
storing the result in RESULT.
Return 1 if the difference is negative, otherwise 0. */

int timeval_subtract (struct timeval *result,
                      struct timeval *x,
                      struct timeval *y) {
/* Perform the carry for the later subtraction by updating y. */
if (x->tv_usec < y->tv_usec) {
 int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
 y->tv_usec -= 1000000 * nsec;
 y->tv_sec += nsec;
}
if (x->tv_usec - y->tv_usec > 1000000) {
 int nsec = (x->tv_usec - y->tv_usec) / 1000000;
 y->tv_usec += 1000000 * nsec;
 y->tv_sec -= nsec;
}

/* Compute the time remaining to wait.
  tv_usec is certainly positive. */
result->tv_sec = x->tv_sec - y->tv_sec;
result->tv_usec = x->tv_usec - y->tv_usec;

/* Return 1 if result is negative. */
```

```
63  return x->tv_sec < y->tv_sec;
64  }
65
66  /* quanti elementi nella lista */
67  int nlist = 0;
68
69  /* Controlla se un pacchetto del proxysender ha il timer scaduto */
70  int controlla_scadenza(struct timeval *p){
71      struct timeval attuale, trascorso, timeout;
72      int ret;
73      timeout.tv_sec  = TIMEOUT;
74      timeout.tv_usec = MSTIMEOUT;
75      /* prendo il tempo attuale */
76      if(gettimeofday(&(attuale), NULL)){
77          printf("gettimeofday() fallita, Err: %d \"%s\"\n",
78                      errno,
79                      strerror(errno)
80                );
81          exit(EXIT_FAILURE);
82      }
83      /* calcolo il tempo trascorso */
84      timeval_subtract(&trascorso, &attuale, p);
85      /* controllo se il tempo rimanente supera il timer */
86      ret = timeval_subtract(p, &timeout, &trascorso);
87      return ret;
88  }
89
90  /* -------- LISTE DINAMICHE con malloc() --------------*/
91  /* funzione di debug che stampa il contenuto di una lista passata */
92  void stampalista(lista* sentinella){
93      lista* cur = sentinella->next;
94      printf("[");
95      while(cur != NULL){
96          printf(" (%d - %d - %s) ",cur->p.id, cur->size, cur->p.body);
97          cur = cur->next;
98      }
99      printf("]\n");
100
101 }
102
103 /* aggiunge un pacchetto in coda alla lista, prende una sentinella */
104 void aggiungi( lista* sentinella, packet p, int size){
105     /* printf("[%d|%c|%s]\n", p.id, p.tipo, p.body); */
106     lista* new;
107     lista* cur = sentinella;
108     while(cur->next != NULL){
109         cur = cur->next;
110     }
111     /* allocazione memoria per il nuovo elemento con la malloc */
112     new = malloc((size_t)sizeof(lista));
113     if(new == NULL){
114         printf("malloc() failed\n");
115         exit(1);
116     }
117     /* nel pacchetto inserito viene aggiunto il timestamp attuale */
118     if(gettimeofday(&(new->sentime), NULL)) {
119         printf ("gettimeofday() Err: %d \"%s\"\n",errno,strerror(errno));
120         exit(1);
121     }
122     /* inserisco il pacchetto nella lista */
123     memcpy(&(new->p), &p, sizeof(packet));
124     /* aggiornamento dei puntatori per rendere la lista coerente */
```

```
125     new->size = size;
126     new->next = NULL;
127     cur->next = new;
128     /* aggiorno il numero di pacchetti attraverso la variabile globale*/
129     nlist++;
130 }
131 
132 /* come la funzione aggiungi, ma aggiunge il pacchetto in ordine
133  * nella lista in base all' ID */
134 void aggiungi_in_ordine( lista* sentinella, packet p, int size){
135     /* printf("[%d|%c|%s]\n", p.id, p.tipo, p.body); */
136     lista* new;
137     lista* cur = sentinella;
138     /* scorro la lista fino ad individuare la posizione in cui inserire
139      * il pacchetto */
140     while(cur->next != NULL && cur->next->p.id < p.id){
141         cur = cur->next;
142     }
143     /* se il pacchetto esiste già lo scarto */
144     if(cur->next != NULL && cur->next->p.id == p.id)
145         return;
146     /* alloco il nuovo elemento con malloc */
147     new = malloc((size_t)sizeof(lista));
148     if(new == NULL){
149         printf("malloc() failed\n");
150         exit(1);
151     }
152     /* il timestamp non viene inserito, perchè questa funzione è usata
153      * solo dal proxyreceiver, che non utilizza il tempo dei pacchetti
154      * in questo modo si incrementano le performance effettuando meno
155      * system call */
156     /*
157     if(gettimeofday(&(new->sentime), NULL)) {
158         printf ("gettimeofday() failed, Err: %d \"%s\"\n",
159                 errno,
160                 strerror(errno)
161                );
162         exit(1);
163     }
164     */
165     /* inserisco il pacchetto nella lista */
166     memcpy(&(new->p), &p, sizeof(packet));
167     /* aggiorno i puntatori in modo da mantenere la lista coerente */
168     new->size = size;
169     new->next = cur->next;
170     cur->next = new;
171     /* aggiorno il numero di pacchetti attraverso la variabile globale*/
172     nlist++;
173 }
174 
175 /* elimina e restituisce il pacchetto in testa alla lista passata come
176  * parametro (la sentinella) */
177 /* NOTA: questa funzione non viene mai chiamata con la lista vuota */
178 lista pop(lista* sentinella){
179     lista* todel;
180     lista ret;
181     memset(&ret, 0, sizeof(lista));
182     /* puntatore all'elemento da eliminare */
183     todel = sentinella->next;
184     /* aggiorno i puntatori */
185     sentinella->next = todel->next;
186     /* elemento da ritornare */
```

```
187     memcpy(&ret, todel, sizeof(lista));
188     /* libero la memoria */
189     free(todel);
190     /* aggiornamento del numero dei pacchetti nella lista */
191     nlist--;
192     return ret;
193 }
194 
195 /* elimina un pacchetto dalla lista con un determinato id passato come
196  * parametro, se il pacchetto non esiste viene restituito un pacchetto
197  * speciale "E" di errore */
198 lista rimuovi(lista* sentinella, uint32_t id){
199     lista* cur = sentinella;
200     lista* todel;
201     lista ret;
202     /* scorro fino alla fine della lista */
203     while(cur->next != NULL){
204         /* se l'elemento corrente è quello da eliminare */
205         if(cur->next->p.id == id){
206             todel = cur->next;
207             /* aggiorno i puntatori */
208             cur->next = cur->next->next;
209             /* copio il valore da ritornare */
210             memcpy(&ret, todel, sizeof(lista));
211             /* libero la memoria */
212             free(todel);
213             /* decremento il numero di pacchetti nella lista */
214             nlist--;
215             return ret;
216         }
217         /* scorro al prossimo elemento */
218         cur = cur->next;
219     }
220     /* se il pacchetto non è stato trovato restituisco un pacchetto
221      * speciale "E" di errore */
222     ret.p.tipo = 'E';
223     return ret;
224 }
225 /* -------------- SOCKET, funzioni ricorrenti -----------------*/
226 
227 /* questa funzione crea un socket TCP o UDP controllando gli errori */
228 int get_socket(int type){
229     /* SOCK_STREAM = TCP
230      * SOCK_DGRAM  = UDP
231      */
232    int socketfd;
233    socketfd = socket(AF_INET, type, 0);
234     if (socketfd == -1) {
235         printf ("socket() fallita, Err: %d %s\n", errno,strerror(errno));
236         exit(1);
237     }
238     return socketfd;
239 }
240 
241 /* questa funzione imposta il socket con l'opzione REUSEADDR */
242 void sock_opt_reuseaddr(int socketfd){
243     /* avoid EADDRINUSE error on bind() */
244     int OptVal = 1;
245     int ris;
246     ris = setsockopt(socketfd,
247                      SOL_SOCKET,
248                      SO_REUSEADDR,
```

```
249                      (char *)&OptVal,
250                      sizeof(OptVal)
251                     );
252     if (ris == -1) {
253         printf ("setsockopt() SO_REUSEADDR fallita, Err: %d \"%s\"\n",
254                   errno,
255                   strerror(errno)
256                 );
257         exit(1);
258     }
259 }
260 
261 /* popola la struttura sockaddr_in con indirizzo e porta */
262 void name_socket(struct sockaddr_in *opt,
263                  uint32_t ip_address,
264                  uint16_t port){
265     /* name the socket */
266     memset(opt, 0, sizeof(*opt));
267     opt->sin_family      =   AF_INET;
268     /* or htonl(INADDR_ANY)
269      * or inet_addr(const char *)
270      */
271     opt->sin_addr.s_addr =   ip_address;
272     opt->sin_port        =   htons(port);
273 }
274 
275 /* effettua la bind sul socket controllando gli errori */
276 void sock_bind(int socketfd, struct sockaddr_in* Local){
277     int ris;
278     ris = bind(socketfd, (struct sockaddr*) Local, sizeof(*Local));
279     if (ris == -1)  {
280         printf ("bind() fallita, Err: %d %s\n",errno,strerror(errno));
281         exit(1);
282     }
283 }
284 
285 /* esegue la connessione del socket TCP controllando gli errori */
286 void sock_connect(int sock, struct sockaddr_in *opt){
287     /* connection request */
288     int ris;
289     ris = connect(sock, (struct sockaddr*) opt, sizeof(*opt));
290     if (ris == -1)  {
291         printf ("connect() fallita, Err: %d %s\n",errno,strerror(errno));
292         exit(1);
293     }
294 }
295 
296 /* mette un socket TCP in ascolto su una determinata porta
297  * controllando gli errori */
298 void sock_listen(int sock){
299     int ris;
300     ris = listen(sock, 10 );
301     if (ris == -1) {
302         printf ("listen() fallita, Err: %d %s\n",errno,strerror(errno));
303         exit(1);
304     }
305 }
306 
307 /* accetta una connessione TCP e restituisce il socket che rappresenta
308  * la connessione appena instaurata, il socket di partenza non viene
309  * piu considerato, vengono controllati gli errori */
310 int sock_accept(int socketfd, struct sockaddr_in *opt){
```

```
311     int len, newsocketfd;
312     memset (opt, 0, sizeof(*opt) );
313     /* wait for connection request */
314     len=sizeof(*opt);
315     newsocketfd = accept(socketfd,
316                          (struct sockaddr*) opt,
317                          (socklen_t *)&len
318                         );
319     if (newsocketfd == -1)  {
320         printf ("accept() fallita, Err: %d %s\n",errno,strerror(errno));
321         exit(1);
322     }
323     return newsocketfd;
324 }
325
326 /* inizializza una connessione TCP effettuando tutte le system call
327  * necessarie, attraverso le funzioni definite sopra */
328 int TCP_connection_send(struct in_addr remote_ip, int remote_port){
329
330     int tcp_sock;
331     struct sockaddr_in local, server;
332
333     tcp_sock = get_socket(SOCK_STREAM);
334
335     sock_opt_reuseaddr(tcp_sock);
336
337     name_socket(&local, htonl(INADDR_ANY), 0);
338
339     sock_bind(tcp_sock, &local);
340
341     name_socket(&server, remote_ip.s_addr, remote_port);
342
343     sock_connect(tcp_sock, &server);
344
345     printf ("Connessione avvenuta\n");
346
347     return tcp_sock;
348 }
349
350 /*
351  * wait for TCP connection on the provided port
352  * and return the TCP socket when a client connects
353  */
354 int TCP_connection_recv(int local_port) {
355
356     int sock, newsocketfd;
357     struct sockaddr_in Local, Cli;
358
359     sock = get_socket(SOCK_STREAM);
360
361     sock_opt_reuseaddr(sock);
362
363     name_socket(&Local, htonl(INADDR_ANY), local_port);
364
365     sock_bind(sock,&Local);
366
367     sock_listen(sock);
368
369     newsocketfd = sock_accept(sock, &Cli);
370
371     printf("Connessione da %s : %d\n",
372         inet_ntoa(Cli.sin_addr),
```

```
373         ntohs(Cli.sin_port)
374     );
375 
376     return newsocketfd;
377 }
378 
379 /* inizializza un socket UDP restituendo il socket creato
380  * utilizza le funzioni definite sopra */
381 int UDP_sock(int local_port){
382     int sock;
383     struct sockaddr_in local;
384 
385     sock = get_socket(SOCK_DGRAM);
386     sock_opt_reuseaddr(sock);
387     name_socket(&local, htonl(INADDR_ANY), local_port);
388     sock_bind(sock,&local);
389 
390     return sock;
391 }
392 
393 /* legge dati da un socket TCP o UDP controllando gli errori
394  * se la syscall viene interrotta dal kernel con EINTR viene ripetuta */
395 ssize_t readn (int fd, char *buf, size_t n, struct sockaddr_in *from){
396     socklen_t len;
397     size_t nleft;
398     ssize_t nread;
399     nleft = n;
400     if(from == NULL)
401         len = 0;
402     else
403         len = sizeof(struct sockaddr_in);
404     while (1) {
405         nread = recvfrom( fd,
406                           buf+n-nleft,
407                           nleft,
408                           0,
409                           (struct sockaddr*) from,
410                           &len
411                         );
412         nleft -= nread;
413         if ( nread < 0) {
414             if (errno != EINTR){
415                 /*return(-1);*/   /* restituisco errore */
416                 printf ("recvfrom() fallita, Err: %d \"%s\"\n",
417                         errno,
418                         strerror(errno)
419                        );
420                 exit(1);
421             }
422         } else {
423             return( n - nleft);
424         }
425 
426     }
427     return( n - nleft);  /* return >= 0 */
428 }
429 
430 /* scrive esattamente n byte del buffer passato come parametro
431  * sul socket TCP o UDP passato, controlla gli errori e se la
432  * system call viene interrotta dal kernel con EINTR la ripete */
433 void writen (int fd, char *buf, size_t n, struct sockaddr_in *to){
434     size_t nleft;
```

```
435     ssize_t nwritten;
436     char *ptr;
437     ptr = buf;
438     nleft = n;
439     while (nleft > 0){
440         if ( (nwritten = sendto(
441                                 fd,
442                                 ptr,
443                                 nleft,
444                                 MSG_NOSIGNAL,
445                                 (struct sockaddr*)to,
446                                 (socklen_t )sizeof(struct sockaddr_in)
447                                  )) < 0) {
448             if (errno == EINTR)
449                 nwritten = 0;   /* and call write() again*/
450             else {
451                 /*return(-1);*/       /* error */
452                 printf ("sendto() fallita, Err: %d \"%s\"\n",
453                         errno,
454                         strerror(errno)
455                        );
456                 exit(1);
457             }
458         }
459         nleft -= nwritten;
460         ptr += nwritten;
461     }
462 }
463
```

```c
/* la dimensione massima dei pacchetti */
#define MAXSIZE 65000

/* la dimensione in byte dell'header dei datagram UDP */
#define HEADERSIZE 5

/* la dimensione massima del body dei pacchetti */
#define BODYSIZE (MAXSIZE) - (HEADERSIZE)

/* il timeout della select del proxysender */
#define TIMEOUT 0   /* secondi */
#define MSTIMEOUT 800000 /* microsecondi */

/* timeout di attesa della chiusura UDP da parte del proxyreceiver */
#define CLOSETIMEOUT 5

/* struttura del pacchetto UDP */
typedef struct packet{
    /* l'ID del paccheto, intero senza segno a 32 bit */
    uint32_t id;
    /* il carattere che identifica il tipo di pacchetto:
     * 'B' per i normali pacchetti BODY, anche gli ACK
     * 'I' per gli ICMP
     * 'E' per gli errori interni (aggiunto da noi) */
    char tipo;
    /* una stringa per il corpo del pacchetto */
    char body[BODYSIZE];
} __attribute__((packed)) packet;

/* la struttura dell'ICMP */
typedef struct ICMP{
    /* intero senza segno a 32 bit che rappresenta l'id dell'ICMP */
    uint32_t id;
    /* un carattere che rappresenta il tipo 'I' dell'ICMP */
    char tipo;
    /* l'ID del pacchetto scartato */
    uint32_t idpck;
} __attribute__((packed)) ICMP;

/* la struttura degli elementi delle liste dinamiche utilizzate dai
 * proxy. Questa struttura è usata anche dalle sentinelle, che però
 * utilizzano solo il campo next */
typedef struct lista{
    /* il puntatore all'elemento successivo */
    struct lista* next;
    /* il pacchetto */
    packet p;
    /* il timestamp abbinato al pacchetto */
    struct timeval sentime;
    /* la dimensione del pacchetto */
    int size;
} __attribute__((packed)) lista;

/* prototipi delle funzioni di utility definite in "utils.c" */

struct in_addr DNSquery (const char* hostname);

int timeval_subtract (struct timeval *result,
                      struct timeval *x,
                      struct timeval *y);

int controlla_scadenza(struct timeval *p);
```

```
63
64 /* variabile globale che contiene il numero di pacchetti contenuti
65  * in una lista */
66 int nlist;
67
68 void stampalista(lista* sentinella);
69 void aggiungi( lista* sentinella, packet p, int size );
70 void aggiungi_in_ordine( lista* sentinella, packet p, int size );
71 lista pop(lista* sentinella);
72 lista rimuovi(lista* sentinella, uint32_t id);
73
74 int get_socket(int type);
75 void sock_opt_reuseaddr(int socketfd);
76 void name_socket(struct sockaddr_in *opt,
77                  uint32_t ip_address,
78                  uint16_t port
79                 );
80 void sock_bind(int socketfd, struct sockaddr_in* Local);
81 void sock_connect(int sock, struct sockaddr_in *opt);
82 void sock_listen(int sock);
83 int sock_accept(int socketfd, struct sockaddr_in *opt);
84 int TCP_connection_send(struct in_addr remote_ip, int remote_port);
85 int TCP_connection_recv(int local_port);
86 int UDP_sock(int local_port);
87
88 ssize_t readn (int fd, char *buf, size_t n, struct sockaddr_in *from);
89 void writen (int fd, char *buf, size_t n, struct sockaddr_in *to);
90
```