

```
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
```

```
1  /*
2  *      proxyreceiver.c : usando datagram UDP simula una connessione TCP
3  *      attraverso un programma ritardatore che scarta i pacchetti in modo
4  *      casuale, simulando una rete reale
5  */
6
7  /* file di header */
8  #include <stdio.h>
9  #include <unistd.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <errno.h>
13 #include <sys/select.h>
14 #include <netinet/in.h>
15 #include <arpa/inet.h>
16
17 /* file di header del progetto */
18 #include "utils.h"
19
20 int main(int argc, char *argv[]){
21
22     /* variabile globale che contiene il numero di elementi nella lista
23      * viene tenuta aggiornata dalle funzioni di manipolazione delle
24      * liste */
25     extern int nlist;
26
27     /* variabile booleana che controlla se è arrivato il datagram
28      * di terminazione del protocollo */
29     int arrivata_terminazione = 0;
30
31     int udp_sock, tcp_sock, nread, local_port, remote_port;
32
33     struct sockaddr_in from, to;
34
35     lista buf_l;
36     packet buf;
37     char remote_ip[40];
38     char ritardatore_ip[40];
39     uint16_t porte_ritardatore[3];
40     in_addr_t ip_ritardatore;
41
42     /* una lista ordinata che contiene i pacchetti da spedire al
43      receiver */
44     lista to_send;
45
46     uint32_t id_to_wait = 1;
47
48     fd_set rfd;
49     struct timeval tv;
50     int ret;
51
52     struct in_addr indirizzo_ritardatore, indirizzo_receiver;
53
54     /* recupero parametri da riga di comando
55      * nessun parametro è obbligatorio, infatti se un parametro non
56      * viene fornito, viene utilizzato il valore di default */
57
58     /* IP del ritardatore, default localhost */
59     if(argc > 1)
60         strcpy(ritardatore_ip, argv[1]);
61     else
62         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 endianness 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_socket(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(&rfd);
119          FD_SET(udp_sock, &rfd);
120          tv.tv_sec = CLOSETIMEOUT;
121          tv.tv_usec = 0;
122          retval = select(udp_sock+1, &rfd, 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("\npacchetti 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             /* azzerò 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

```

```
1  /*
2  *      proxysender.c : usando datagram UDP simula una connessione TCP
3  *      attraverso un programma ritardatore che scarta i pacchetti in modo
4  *      casuale, simulando una rete reale
5  */
6
7  /* File di header */
8  #include <stdio.h>
9  #include <string.h>
10 #include <stdlib.h>
11 #include <sys/select.h>
12 #include <unistd.h>
13 #include <errno.h>
14 #include <arpa/inet.h>
15 #include <sys/time.h>
16
17 /* File di header del progetto */
18 #include "utils.h"
19
20 int main(int argc, char *argv[]){
21
22     /* numero di elementi nella lista,
23      * variabile globale aggiornata dalle funzioni delle liste */
24     extern int nlist;
25
26     int udp_sock = 0;
27     int tcp_sock = 0;
28     int nread = 0;
29     int local_port_tcp = 0;
30     int local_port_udp = 0;
31     int rit_port[3];
32     uint16_t porte_rit[3];
33     int rit_turno = 0;
34
35     struct sockaddr_in to, from;
36
37     packet buf_p;
38     lista  buf_l;
39
40     char remote_ip[16];
41     in_addr_t ip_ritardatore;
42
43     uint32_t progressive_id = 0;
44
45     /* per la select */
46     fd_set rfd;
47     int retsel;
48     int fdmax;
49
50     /* contatori per le statistiche */
51     int quanti_timeout = 0;
52     int quanti_pacchetti_tcp = 0;
53     int quanti_datagram_inviati = 0;
54     int quanti_icmp = 0;
55     int quanti_ack = 0;
56     int overhead = 0;
57
58     struct in_addr indirizzo_ritardatore;
59
60     /* per gestire il timeout dei pacchetti */
61     struct timeval timeout;
62 }
```

```

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_socket(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(&rfd);
167         if(tcp_sock != -1)
168             FD_SET(tcp_sock, &rfd);
169         FD_SET(udp_sock, &rfd);
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, &rfd, 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, &rfd, 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, &rfdsets)){
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, &rfdsets)){
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 programma */
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_pacchetti_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 }
```

```

1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <netdb.h>
8  #include <netinet/in.h>
9  #include <arpa/inet.h>
10 #include <errno.h>
11 #include <sys/time.h>
12
13 #include "utils.h"
14
15 /* QUERY DNS */
16 /* riceve una stringa contenente il nome di cui cercare l'IP */
17 struct in_addr DNSquery (const char* hostname){
18     struct hostent *he;
19     struct in_addr **addr_list;
20
21     if ((he = gethostbyname(hostname)) == NULL) {
22         printf("Nome DNS non trovato\n");
23         exit(1);
24     }
25
26     addr_list = (struct in_addr **)he->h_addr_list;
27
28     return *addr_list[0];
29 }
30
31 /* ----- Sottrarre TIMEVAL -----
32 * http://www.gnu.org/software/libc/manual/html\_node/Elapsed-Time.html
33 */
34
35 /* CALCOLA X - Y
36 * ritorna 1 se il risultato è negativo */
37
38 /* Subtract the 'struct timeval' values X and Y,
39 storing the result in RESULT.
40 Return 1 if the difference is negative, otherwise 0. */
41
42 int timeval_subtract (struct timeval *result,
43                      struct timeval *x,
44                      struct timeval *y) {
45     /* Perform the carry for the later subtraction by updating y. */
46     if (x->tv_usec < y->tv_usec) {
47         int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
48         y->tv_usec -= 1000000 * nsec;
49         y->tv_sec += nsec;
50     }
51     if (x->tv_usec - y->tv_usec > 1000000) {
52         int nsec = (x->tv_usec - y->tv_usec) / 1000000;
53         y->tv_usec += 1000000 * nsec;
54         y->tv_sec -= nsec;
55     }
56
57     /* Compute the time remaining to wait.
58        tv_usec is certainly positive. */
59     result->tv_sec = x->tv_sec - y->tv_sec;
60     result->tv_usec = x->tv_usec - y->tv_usec;
61
62     /* 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 sockfd;
233     sockfd = socket(AF_INET, type, 0);
234     if (sockfd == -1) {
235         printf ("socket() fallita, Err: %d %s\n", errno, strerror(errno));
236         exit(1);
237     }
238     return sockfd;
239 }
240
241 /* questa funzione imposta il socket con l'opzione REUSEADDR */
242 void sock_opt_reuseaddr(int sockfd){
243     /* avoid EADDRINUSE error on bind() */
244     int OptVal = 1;
245     int ris;
246     ris = setsockopt(sockfd,
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 sockfd, struct sockaddr_in* Local){
277     int ris;
278     ris = bind(sockfd, (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 sockfd, 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
```

```
1  /* la dimensione massima dei pacchetti */
2  #define MAXSIZE 65000
3
4  /* la dimensione in byte dell'header dei datagram UDP */
5  #define HEADERSIZE 5
6
7  /* la dimensione massima del body dei pacchetti */
8  #define BODYSIZE (MAXSIZE) - (HEADERSIZE)
9
10 /* il timeout della select del proxysender */
11 #define TIMEOUT 0 /* secondi */
12 #define MTIMEOUT 800000 /* microsecondi */
13
14 /* timeout di attesa della chiusura UDP da parte del proxyreceiver */
15 #define CLOSETIMEOUT 5
16
17 /* struttura del pacchetto UDP */
18 typedef struct packet{
19     /* l'ID del pacchetto, intero senza segno a 32 bit */
20     uint32_t id;
21     /* il carattere che identifica il tipo di pacchetto:
22      * 'B' per i normali pacchetti BODY, anche gli ACK
23      * 'I' per gli ICMP
24      * 'E' per gli errori interni (aggiunto da noi) */
25     char tipo;
26     /* una stringa per il corpo del pacchetto */
27     char body[BODYSIZE];
28 } __attribute__((packed)) packet;
29
30 /* la struttura dell'ICMP */
31 typedef struct ICMP{
32     /* intero senza segno a 32 bit che rappresenta l'id dell'ICMP */
33     uint32_t id;
34     /* un carattere che rappresenta il tipo 'I' dell'ICMP */
35     char tipo;
36     /* l'ID del pacchetto scartato */
37     uint32_t idpck;
38 } __attribute__((packed)) ICMP;
39
40 /* la struttura degli elementi delle liste dinamiche utilizzate dai
41  * proxy. Questa struttura è usata anche dalle sentinelle, che però
42  * utilizzano solo il campo next */
43 typedef struct lista{
44     /* il puntatore all'elemento successivo */
45     struct lista* next;
46     /* il pacchetto */
47     packet p;
48     /* il timestamp abbinato al pacchetto */
49     struct timeval sentime;
50     /* la dimensione del pacchetto */
51     int size;
52 } __attribute__((packed)) lista;
53
54 /* prototipi delle funzioni di utility definite in "utils.c" */
55
56 struct in_addr DNSquery (const char* hostname);
57
58 int timeval_subtract (struct timeval *result,
59                      struct timeval *x,
60                      struct timeval *y);
61
62 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 sockfd);
76 void name_socket(struct sockaddr_in *opt,
77                  uint32_t ip_address,
78                  uint16_t port
79                  );
80 void sock_bind(int sockfd, 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 sockfd, 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
```