# Laboratorio di Reti di Calcolatori

Socket UDP.

Paolo D'Arco

**Abstract**

Scopo della lezione è descrivere le funzioni che l'interfaccia dei socket offre per far interagire client e server attraverso il protocollo UDP, mostrare alcuni esempi concreti, e analizzare i problemi che possono sorgere nel corso dell'esecuzione.

## 1 Protocollo UDP

UDP è un protocollo di trasporto semplice. Contrariamente a TCP che è orientato alla connessione ed è affidabile, UDP *non* stabilisce connessioni tra client e server e *non* offre garanzie di affidabilità. Inoltre, mentre TCP trasmette flussi di byte (inizio e fine come ricordate vengono concordati dal client e dal server), il protocollo UDP invia pacchetti di taglia fissa, detti *datagram*. Esistono condizioni in cui UDP può essere utile. Infatti

- L'assenza di connessione (protocolli di handshake in tre passi e di chiusura in quattro passi) rendono UDP più veloce di TCP per trasferimenti di pochi byte.
- Gli errori di trasmissione non sono la normalità ma eventi sporadici. Per cui se i dati non sono critici, anche un protocollo che funziona *quasi sempre bene* può essere utile.

Esempi di applicazioni che usano come protocollo di trasporto UDP sono il DNS, NFS e il protocollo SNMP (che analizzerete a fondo nel corso di teoria).

L'interazione tipica tra client e server attraverso UDP è mostrata in Figura 1: Il server crea un socket, invoca *bind*() e attende tramite la funzione *recvfrom*() di ricevere dati. Il client crea il proprio socket, e invia dati tramite la funzione *sendto*(). L'interazione continua attraverso chiamate da ambo le parti a *recvfrom*() e *sendto*(). Alla fine, sia il server che il client chiudono i rispettivi socket.

Le funzioni *socket*(), *bind*() e *close*() sono esattamente le stesse descritte nella trattazione dei socket TCP. Le funzioni *recvfrom*() e *sendto*() sono invece definite come segue.

```
#include <sys/socket.h>

int recvfrom(int sd, void *buf, int nbytes, int flags, struct sockaddr *from, socklen_t *len);

int sendto(int sd, const void *buf, int nbytes, int flags, const struct sockaddr *to, socklen_t len);

valore di ritorno: -1 se errore
                   numero di byte letti/scritti, altrimenti
```

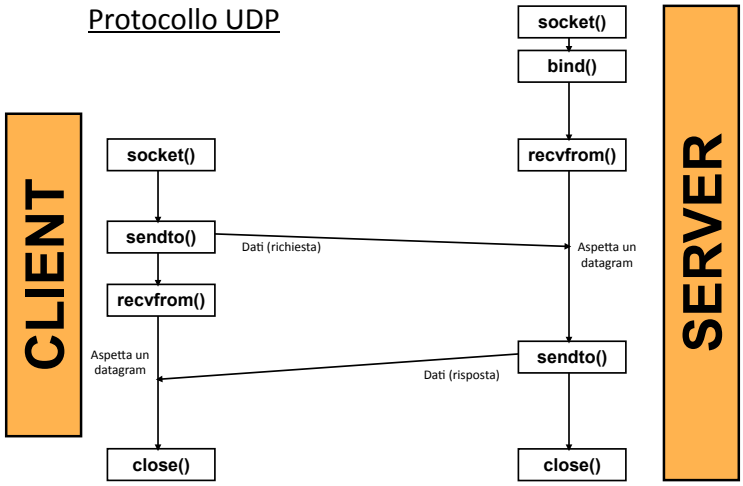



Figure 1: Interazione Client-Server attraverso UDP

Gli argomenti delle due funzioni sono:

- *sd*, *buf*, *nbytes* e *flags* che indicano, rispettivamente, un socket descriptor, il buffer in cui leggere o contenente i byte da scrivere, la lunghezza in byte del buffer/ dati da scrivere, e una serie di opzioni (per ora flags sarà uguale a 0).
- *from/ to* e *len* che sono, invece, strutture socket e la loro lunghezza. In *recvfrom*() possono essere passati due puntatori NULL, nel caso in cui il ricevente non sia interessato a conoscere l'identità del mittente. Se vengono passati puntatori a strutture, la funzione le riempie con i parametri del socket del client.

Gli ultimi due parametri sono simili a quelli della funzione *accept*() incontrata studiando i socket TCP. Allo stesso modo, si noti che, in *recvfrom*(), *len* è un puntatore ad una variabile intera che contiene la lunghezza della struttura. Si tratta di un parametro valore-risultato (i.e., viene riempita dalla funzione al termine della computazione). In *sendto*, invece, *len* è semplicemente la lunghezza della struttura.

## 2 Esempi: echo client ed echo server con UDP

Riscriviamo echo client ed echo server utilizzando il protocollo UDP. Per interagire con il server, il client utilizza una funzione, client_echo_udp(), definita successivamente (linea 1.). All'interno del corpo principale del programma dichiara variabili e strutture necessarie nel seguito (linee 3. 4.), controlla che il programma sia stato invocato correttamente dall'utente (linea 5.), memorizza

indirizzo IP e porta del server e crea un socket[1] di tipo SOCK_DGRAM (linee 6. - 11). Infine invoca la funzione client_echo_udp() che si occupa di gestire l'interazione con il server.

```
#include        "basic.h"

1. void client_echo_udp(FILE *fp, int sockfd,
     const struct sockaddr *p_servaddr, socklen_t servlen );

2. int main(int argc, char **argv) {
3.  int                     sockfd;
4.  struct sockaddr_in      servaddr;

5.  if (argc != 3){
   printf("usage: udpclient <IPaddress> <PORT>\n"); exit(1);
  }
6.  bzero(&servaddr, sizeof(servaddr));
7.  servaddr.sin_len = (uint8_t) 16;
8.  servaddr.sin_family = AF_INET;
9.  servaddr.sin_port = htons(atoi(argv[2]));
10. inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

11.  if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ){
       printf("socket error\n"); exit(1);
}
12.  client_echo_udp(stdin, sockfd, (struct sockaddr *) &servaddr,
13. sizeof(servaddr));

14.  exit(0);
}
```

La funzione client_echo_udp() prende in input un file pointer *fp* (per ricevere input da tastiera), un socket descriptor *sockfd* (per comunicare con il server), un puntatore ad una struttura sockaddr **p_servaddr*, struttura che contiene i parametri del server, e la lunghezza di questo tipo di struttura *servlen*. Ricordo che la struttura sockaddr è di *tipo generale*. Poichè stiamo usando strutture sockaddr_in, si serviamo dell'operatore cast () per convertire la struttura nel tipo giusto.

La funzione dichiara variabili e strutture che saranno utili nel seguito (linee 1. - 5.). Alloca spazio per un'altra struttura di tipo sockaddr (linea 6.), usata successivamente in *recvline*(). Quindi, comincia a leggere da tastiera (linea 7.). Invia la riga ricevuta (linea 8.) e legge la risposta del server (linee 9. - 11). Infine, il client controlla che la risposta ricevuta sia stata inviata *realmente dal server* e da nessun altro (linee 12. - 15.) Se questo è il caso, completa la riga ricevuta con 0 finale, affinchè rappresenti una stringa in C, e la stampa a video usando la funzione *fputs*() (linee 16. - 17.).

[1]Nota che in genere non ci occupiamo di inizializzare la lunghezza della struttura (linea 7.). In questo esempio, è stato necessario perchè il sistema operativo Leopard del mio Macbook, inizializza questo campo. Poichè la funzione client_echo_udp() nel seguito effettua un confronto bit a bit tra questa struttura, inizializzata dal client, e quella ricevuta dal server, senza l'inizializzazione del campo len il confronto fallisce. Con altri sistemi operativi, non succede. Se usate Linux, provate a compilare ed eseguire l'esempio senza la linea 7.

```
void client_echo_udp(FILE *fp, int sockfd,
const struct sockaddr *p_servaddr, socklen_t servlen){
1.  int   n;
2.  char   sendline[MAXLINE], recvline[MAXLINE + 1], buff[MAXLINE];
3.  socklen_t       len;
4.  struct sockaddr  *p_replyaddr;
5.  struct sockaddr_in *sa;

6.   p_replyaddr = malloc(servlen);

7.  while (fgets(sendline, MAXLINE, fp) != NULL) {
8.   if (sendto(sockfd, sendline, strlen(sendline), 0, p_servaddr, servlen)<0){
      printf("sendto error\n"); exit(1);
     }
9.   len = servlen;
10. if( (n = recvfrom(sockfd, recvline, MAXLINE, 0, p_replyaddr, &len)) < 0 ){
11.   printf("recvfrom error\n"); exit(1);
     }

12.  if( (len != servlen) || memcmp(p_servaddr, p_replyaddr, len) != 0 ) {
13.  sa = (struct sockaddr_in *) p_replyaddr;
14.  printf("risposta da %s ignorata\n",
             inet_ntop(AF_INET, &sa->sin_addr, buff, sizeof(buff)));
15.  continue;
      }
16. recvline[n] = 0;
17. fputs(recvline, stdout);
    }
}
```

Il controllo sull'identità del server (linee 12. - 15.) evidenzia una caratteristica tipica di UDP. Non essendo stata stabilita una connessione tra client e server, chiunque può inviare al client un datagram al posto del server. Nel seguito mostreremo con un esempio come creare una situazione del genere.

La struttura del server è molto simile a quella del server TCP. Crea un socket di tipo SOCK_DGRAM, invoca la funzione *bind*(), e la funzione server_echo_udp(), che si occupa dell'interazione reale con il client.

```
#include        "basic.h"

1.void server_echo_udp(int sockfd, struct sockaddr *p_cliaddr, socklen_t clilen);

2. int main(int argc, char **argv) {
3. int    sockfd;
4. struct sockaddr_in      servaddr, cliaddr;

5. if( argc != 2){
          printf("Usage: echosrv <PORT> \n"); exit(1);
}
6. if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ){
               printf("socket error\n"); exit(1);
}
7.  bzero(&servaddr, sizeof(servaddr));
8.  servaddr.sin_family      = AF_INET;
9.  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
10. servaddr.sin_port        = htons(atoi(argv[1]));

11.  if( bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ){
               printf("bind error\n"); exit(1);
}
12.  server_echo_udp(sockfd, (struct sockaddr *) &cliaddr, sizeof(cliaddr));
}
```

La funzione server_echo_udp prende in input un socket descriptor *sockfd*, un puntatore ad una struttura sockaddr **p_cliaddr* (conterrà i parametri del client ), e la lunghezza di questo tipo di struttura. La funzione semplicemente riceve (linea 6.) e invia il messaggio ricevuto (linea 8.).

```
void server_echo_udp(int sockfd, struct sockaddr *p_cliaddr, socklen_t clilen)
{
1. int             n;
2. socklen_t       len;
3. char                    mesg[MAXLINE];

4. for ( ; ; ) {
5.  len = clilen;
6.  if ( (n = recvfrom(sockfd, mesg, MAXLINE, 0, p_cliaddr, &len) ) < 0){
7.          printf("recvfrom error \n"); exit(1);
     }
8.  if( sendto(sockfd, mesg, n, 0, p_cliaddr, len) != n ){
9.          printf("sendto error\n"); exit(1);
      }
  }
}
```

## 3 Inviare messaggi al client

Come dicevamo in precedenza, chiunque può inviare un datagram al client che è in attesa di una risposta dal server. Il programma che segue, spedisci_dg, fa esattamente questo. Crea un socket, costruisce una struttura *servaddr* con indirizzo IP e numero di porta ricevuti tramite linea di comando (saranno esattamente indirizzo IP e porta del client), costruisce una stringa contenente un messaggio generico ed invia la stringa al processo specificato nella struttura *servaddr*.

```
#include        "basic.h"

int main(int argc, char **argv) {
 int                     sockfd, len;
 struct sockaddr_in      servaddr;
 struct sockaddr         *p_servaddr;
 char      sendline[MAXLINE], recvline[MAXLINE];

 if (argc != 3){
   printf("usage: udpclient <IPaddress> <PORT>\n"); exit(1);
   }
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(atoi(argv[2]));
  inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

   if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ){
               printf("socket error\n"); exit(1);
   }
   strncpy(sendline,"Datagram!",MAXLINE);

   p_servaddr = (struct sockaddr *) &servaddr;

   if (sendto(sockfd, sendline, strlen(sendline), 0,
                                        p_servaddr, sizeof(servaddr)) < 0) {
   printf("Errore nella spedizione del datagram\n"); exit(1);
   }
   printf("Datagram spedito.\n");
   exit(0);
}
```

Possiamo facilmente verificare cosa accade, operando come segue. Modifichiamo il server in modo da far si che attenda un po' di secondi prima di rispedire al client il messaggio ricevuto. A tal fine, basta aggiungere dopo la linea 7. nel codice del server l'istruzione sleep(15); Una volta ricompilato, mandiamolo in esecuzione. Quindi, mandiamo in esecuzione anche il client su un secondo terminale e, su un terzo terminale, usiamo il comando *netstat -na -pudp* per conoscere indirizzo IP e numero di porta del client. Individuata la porta del client, digitiamo un messaggio sul terminale del client, e subito dopo, sul terzo terminale, mandiamo in esecuzione spedisci_dg. Provate!

## 4 Datagram Persi

Client e server dell'esempio precedente, così come strutturati, non offrono alcuna garanzia di affidabilità. Se il datagram inviato dal client viene perso, perchè eliminato da un router, il client resterà bloccato in *recvfrom*() in attesa del datagram di risposta del server. Una situazione simile ovviamente potrebbe verificarsi dal lato del server.

Un modo semplice per risolvere questo problema prevede l'uso di un *timeout*. Il client o il server attendono per un tempo massimo l'arrivo del datagram: dopodichè riprendono l'esecuzione. Si noti tuttavia che l'uso di un timeout in alcuni casi non è sufficiente. Per esempio, allo scadere del timeout, il client non riesce a capire se non ha ricevuto risposta perchè il datagram del server è andato perso, o non ha ricevuto risposta perchè il suo datagram non ha mai raggiunto il server! In alcuni contesti applicativi, la differenza può essere importante (si pensi ad un trasferimento di denaro dal conto A al conto B).

Un altro problema che occorre gestire lavorando con socket UDP è il seguente: si supponga di mandare in esecuzione il client mentre il server non è in esecuzione. Il client invia un datagram e attende dal server un datagram di risposta. Come abbiamo già posto in evidenza in precedenza, scrivere su un socket, e in questo caso inviare un datagram, significa scrivere il datagram in un buffer locale (dal quale successivamente il sistema operativo lo preleverà e lo invierà al server lungo la rete). La funzione *sendto*(), pertanto, ritorna non appena il datagram è stato accettato dal buffer. Il ritorno della funzione *non dice nulla* circa l'avvenuta trasmissione del datagram. Nel caso in esame il datagram viene spedito, ma poichè il server non è in esecuzione, interviene il protocollo ICMP che invia un messaggio di errore al sistema operativo del client. Si noti che l'errore in fondo è stato causato dalla *sendto*() che però è ritornata con successo non appena il datagram è stato riposto nel buffer. Pertanto l'errore è *asincrono* rispetto all'esecuzione del client. Purtroppo il protocollo UDP *non notifica* questo errore al client, che resterà bloccato in attesa del datagram di risposta del server.

Una seconda situazione di errore che mostra un'altra carenza dei socket UDP è la seguente: si supponga che un client invii tre datagram a tre server diversi su *un singolo* socket UDP e attenda le rispettive risposte. Due dei tre server ricevono il datagram e rispondono. Il terzo server non è in esecuzione. Pertanto interviene il protocollo ICMP, che invia al sistema operativo del client un messaggio di errore contenente l'indirizzo IP e il numero di porta del server che non è raggiungibile. Ancora una volta, per come sono stati implementati i socket UDP, il sistema operativo *non ha un modo* per fornire al client questa informazione.

Tuttavia, i problemi appena descritti, possono essere risolti usando *socket UDP connessi*.

## 5 Socket UDP connessi.

Sebbene UDP sia un protocollo senza connessione, è possibile invocare *connect*() su un socket UDP. Tuttavia, si faccia attenzione: invocare *connect*() su un socket UDP *non* significa in alcun modo creare una connessione, e.g., non vengono attivate le procedure di handshake nè di chiusura della connessione. Semplicemente, il kernel *memorizza* l'indirizzo e la porta con cui si vuole comunicare. Distingueremo pertanto tra due tipi di socket UDP:

- Socket UDP non connessi. Sono i socket creati di default.
- Socket UDP connessi. Sono socket UDP su cui è stata invocata la funzione *connect*().

I socket UDP connessi hanno tre caratteristiche:

- Nell' invio di datagram *non è* più possibile specificare l'indirizzo IP e la porta del destinatario. Inoltre, per inviare datagram, non useremo più la funzione *sendto*(), ma *write*() o *send*(). Qualsiasi cosa scritta su un socket UDP connesso, viene *automaticamente* inviato all'indirizzo specificato tramite la funzione *connect*().

- Similmente, per ricevere datagram, useremo *read*(), *recv*() e *recvmsg*(). Datagram spediti al socket UDP connesso da *destinazioni diverse* da quella stabilita, vengono automaticamente eliminati. Essenzialmente l'uso di *connect*() riduce la comunicazione ad uno scambio di datagram di tipo *uno a uno*.

- Gli errori asincroni (che, come abbiamo discusso in precedenza, costituiscono un problema per i socket UDP non connessi) vengono restituiti al processo.

In generale, un client/server che comunica tramite socket UDP dovrebbe invocare *connect*() *solo se* il processo usa il socket per comunicare con esattamente un altro processo. Di solito è il client UDP che invoca *connect*(), ma ci sono anche casi in cui è il server, che comunica per un lungo periodo di tempo con un singolo client, ad invocare *connect*().

Un'altra caratteristica peculiare dell'uso di *connect*() con socket UDP è che può essere invocata *più di una volta* sullo stesso socket. Per i socket TCP, *connect*() può essere invocata *una ed una sola volta* su un socket, dopo la sua creazione per connettere il socket al server. Gli usi di questa possibilità sono essenzialmente due:

- Specificare, ad un certo punto dell'esecuzione, un nuovo indirizzo IP ed una nuova porta.

- Disconnettere il socket UDP connesso, per farne di nuovo un uso generale.

La disconnessione di un socket UDP può essere effettuata invocando la funzione e specificando, come parametro per il campo *family*, la costante AF_UNSPEC. Si noti che, in alcuni casi, con questo parametro *connect*() può restituire l'errore EAFNOSUPPORT, ma non è un problema.

La funzione client_echo_udp(), usando *connect*(), diviene:

```
void client_echo_udp(FILE *fp, int sockfd,
const struct sockaddr *p_servaddr, socklen_t servlen){
1.  int   n;
2.  char   sendline[MAXLINE], recvline[MAXLINE + 1];

3. if ( connect(sockfd, (struct sockaddr *) pservaddr, servlen) < 0 ) {
       printf("connect error \n"); exit(1);
     }
4.  while (fgets(sendline, MAXLINE, fp) != NULL) {
         write(sockfd, sendline, strlen(sendline) );

5.  while (n=read(sockfd, recvline, MAXLINE) < 0 ){
        printf("read error\n"); exit(1);
     }
6. recvline[n] = 0;
7. fputs(recvline, stdout);
  }
}
```

Mandando in esecuzione nuovamente echo client, con la funzione client_echo_udp() che invoca *connect*(), è facile verificare che, se il server non è in esecuzione, il client riceve la notifica dell'errore. Nota tuttavia che la notifica dell'errore non avviene a seguito della invocazione della funzione *connect*() ma *nel momento in cui invia il datagram*. E' solo allora che ICMP invia al kernel dell'host del client il messaggio di errore, che viene passato dal kernel al client.

Un ultimo aspetto da evidenziare, relativamente all'uso del protocollo UDP, è la *mancanza di controllo del flusso*. Il progettista di applicazioni che decide di usare UDP, deve avere coscienza di ciò. L'esempio che segue rende bene l'idea. Il server riceve un flusso di datagram e li conta. Non appena viene interrotto (l'utente digita CRTL C da tastiera), il server stampa a video il numero di datagram ricevuti (tecnicamente, il server usa un gestore del segnale SIGINT, associato alla pressione della sequenza CTRL C, per stampar a video il numero di datagram contati). D'altra parte, il client invia semplicemente datagram, senza aspettare risposta.

```
#include        "basic.h"

#define NDG     2000    /* #datagrams to send */
#define DGLEN   1400    /* length of each datagram */

void client_echo_udp_count(FILE *fp, int sockfd,
         const struct sockaddr *pservaddr, socklen_t servlen);

int main(int argc, char **argv) {
        int                     sockfd;
        struct sockaddr_in      servaddr;

        if (argc != 3){
                printf("usage: countclient <IPaddress> <Port> \n");
                exit(1);
}
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(atoi(argv[2]));
        inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

        if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ){
                printf("socket error\n"); exit(1);
}
        client_echo_udp_count(stdin, sockfd,
         (struct sockaddr *) &servaddr, sizeof(servaddr));
        exit(0);
}

void client_echo_udp_count(FILE *fp, int sockfd,
           const struct sockaddr *pservaddr, socklen_t servlen) {
        int     i;
        char    sendline[MAXLINE];

        for (i = 0; i < NDG; i++) {
                sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
        }
}
```

```
#include        "basic.h"

void server_echo_udp_count(int sockfd,
                        struct sockaddr *p_cliaddr, socklen_t clilen);
static void gestisci_interrupt(int signo);
int count = 0;

int main(int argc, char **argv) {
        int                     sockfd;
        struct sockaddr_in      servaddr, cliaddr;

        if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ){
                printf("socket error \n"); exit(1);
}
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family      = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port        = htons(atoi(argv[1]));

        if( bind(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ){
                printf("bind error \n"); exit(1);
}
        signal(SIGINT, gestisci_interrupt);
        server_echo_udp_count(sockfd,
                           (struct sockaddr *) &cliaddr, sizeof(cliaddr));
}

void server_echo_udp_count(int sockfd,
                            struct sockaddr *pcliaddr, socklen_t clilen) {
        int             n;
        socklen_t       len;
        char            mesg[MAXLINE];

        n = 240 * 1024;
        setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

        for ( ; ; ) {
                len = clilen;
                recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
                count++;
                sleep(1); /* rallentiamo il server */
        }
}

static void gestisci_interrupt(int signo) {
        printf("\nDatagrams ricevuti: %d\n", count);
        exit(0);
}
```

Cosa accade se eseguiamo client e server? Procediamo come segue: invochiamo il comando *netstat -s -pudp*. Poi, mandiamo in esecuzione il server e, su un altro terminale, il client. Dopo un paio di secondi interrompiamo il server con CTRL C. Invochiamo nuovamente il comando *netstat -s -pudp*. L'output sarà simile a:

```
macbook-di-paolo-darco:FIN pd$ netstat -s -pudp
udp:
31335 datagrams received
0 with incomplete header
0 with bad data length field
0 with bad checksum
0 dropped due to no socket
9179 broadcast/multicast datagrams dropped due to no socket
1164 dropped due to full socket buffers
0 not for hashed pcb
20992 delivered
4337 datagrams output
macbook-di-paolo-darco:FIN pd$ ./echoudpsrv-count 9999
^C
Datagrams ricevuti: 1262
macbook-di-paolo-darco:FIN pd$ netstat -s -pudp
udp:
33402 datagrams received
0 with incomplete header
0 with bad data length field
0 with bad checksum
0 dropped due to no socket
9199 broadcast/multicast datagrams dropped due to no socket
1902 dropped due to full socket buffers
0 not for hashed pcb
22301 delivered
6338 datagrams output
macbook-di-paolo-darco:FIN pd$
```

Il client ha inviato 2000 datagram. Il server ne ha ricevuti sono 1262. La differenza tra la prima riga delle due invocazioni del comando *netstat -s -pudp*, ci dice che, tra le due chiamate, sono stati inviati $33402 - 31335 = 2067$ datagram (oltre al nostro client che ha inviato 2000 datagram, qualche altro client nel sistema ha inviato qualche datagram). Di questi $1902 - 1164 = 738$ sono stati eliminati perchè i buffer di ricezione erano pieni. Poichè $1262 + 738 = 2000$, è stato proprio il buffer del nostro server a non riuscire a reggere la *velocità* di spedizione di datagram del client.

# 6 Conclusioni.

Nella lezione di oggi, abbiamo:

- Capito come utilizzare socket UDP per applicazioni client-server, analizzando un' interazione tipica e le funzioni di base per lavorare con socket UDP. In particolare, abbiamo evidenziato

che UDP è un protocollo molto semplice, che non stabilisce connessioni, e non offre garanzie di affidabilità, ma che può essere preferibile a TCP in casi in cui il client ed il server si scambino pochi dati di natura non critica.

- Discusso i problemi legati all'uso di UDP (e.g., errori asincroni), gli strumenti disponibili per farvi fronte (e.g., socket UDP connessi) e le contromisure che possono essere adottate in diversi casi.

# 7 Esercizi.

**Esercizio 1.** Nell'esempio echo client / echo server descritto, l'intervento di un altro processo (spedisci_dg nel nostro caso) che invia un datagram al client, porta alla *desincronizzazione* di client e server. Si modifichi il codice di echo client, in modo tale da "buttar via" messaggi ricevuti da altri processi e di attendere il messaggio di risposta del server, prima di inviare al server stesso un nuovo messaggio.

**Esercizio 2.** Client-Server mini-calcolatrice. Si implementi la seguente applicazione client-server: il client legge da tastiera due interi ed un operatore, che può essere "+", "-", "*", o "*mod*" e li invia al server. Il server riceve i due numeri e l'operatore, effettua l'operazione, ed invia al client il risultato. Il client stampa a video il valore ricevuto.