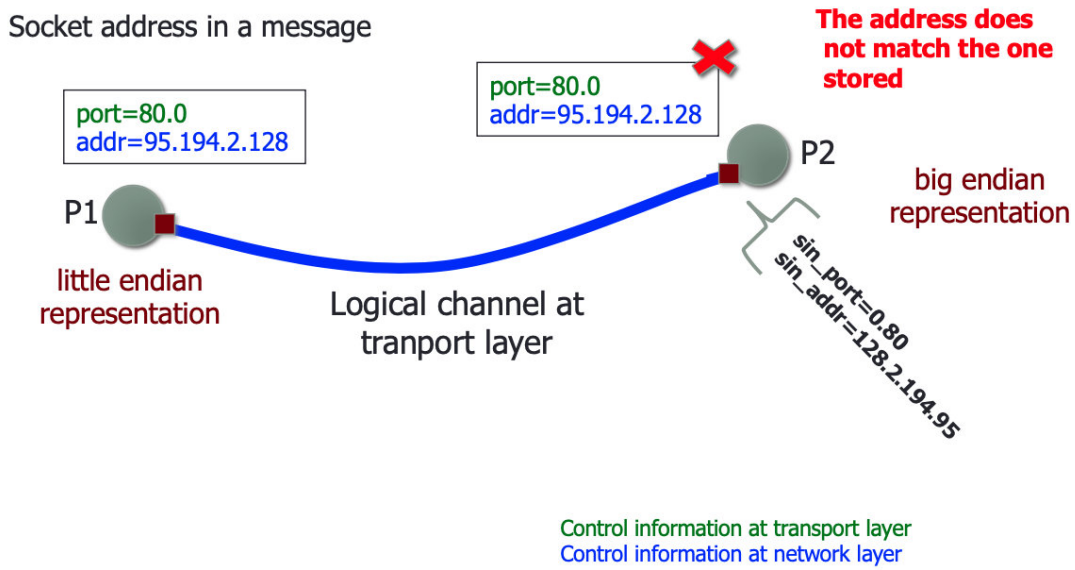
# Rappresentazione Eterogenea

```
union {
  u_int32_t addr;
  unsigned char c[4]
} un;

int main(int argc, char *argv[]){
  un.addr = 0x8002c25f;
  printf("%d\n", un.c[0]);
}
```

# Representation conversion




# Paradigma client server con sockets

Dobbiamo aggiungere giusto qualche linea di codice rispetto a quello visto con il client server:

```
void main(){
  int s = socket(...);
  ...
  struct message m1, m2;

  <il messaggio m1 viene preparato>

  send(s, SERVER, &m1);
  receive(s, CLIENT, &m2);

  <il messaggio m2 viene processato>
}
```

> CLIENT

```
void main(){
  int s = socket(...);
  ...
    struct message m1, m2;

  while(1){
    receive(s, SERVER, &m1);
    <protocollo applicazione:
        m1 è processato per produrre m2>

    send(s, m1.source, &m2);
  }
}
```

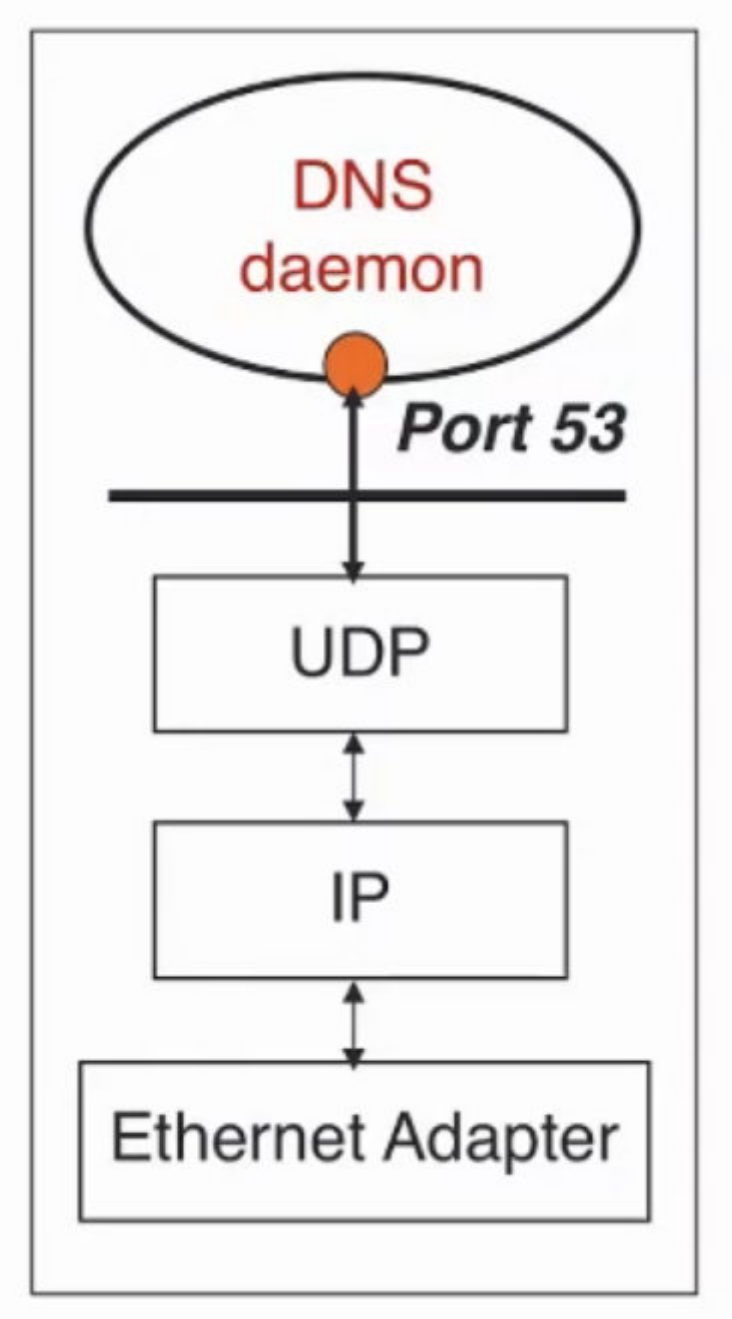
> SERVER

**E' importante** che i due processi comunicanti si scambino dei messaggi aventi **la stessa struttura**, se poi i messaggi hanno variabili chiamate in maniere diverse, non è un problema.

# Datagram oriented communication model

## Un server per la comunicazione datagram

Un **DNS** è un esempio di protocollo applicativo che impiega, a livello di trasporto, il protocollo **UDP**; nel caso specifico l'applicazione si basa sul modello orientato ai datagram.




Il DNS utilizza sia UDP che TCP a livello di trasporto, ma verrà analizzato meglio nelle prossime lezioni. Un'altra informazione è quella del numero di porta: **53**; la porta 53 è infatti associata al **servizio DNS**, dei numeri di porta "well known".

**Riassumendo:** vogliamo usare il protocollo UDP e vogliamo scrivere il server che dia vita al **DNS deamon**.

## Socket()

Non vedremo il protocollo, ma solo come scrivere la parte comune ad altri server che utilizzano lo stesso paradigma client-server.

La prima operazione da effettuare è l'invocazione della funzione socket():

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

int fd;

if((fd = socket(PF_INET, SOCK_DGRAM, 0)) < 0){
  perror("socket");
  exit(1)
}
```

Utilizziamo il parametro **SOCK_DGRAM** proprio perchè vogliamo usare il modello di comunicazione orientato ai datagram. Con PF_INET stiamo implicitamente che il protocollo utilizzato a livello di trasporto è il protocollo UDP, avendo idicato "0" come parametro (protocollo di default; nel caso di TCP/IP è UDP).

Chiediamo quindi al SO di creare un socket, ovvero un endpoint di comunicazione che ci consentirà di comunicare con quel tipo di servizio, che ci restituisce un intero (**fd**); nel caso il valore fosse < 0, allora l'invocazione della socket() ha prodotto un errore.

# bind()

```c
int fd;
struct sockaddr_in srv;
/* creazione della socket */

srv.sin_family = AF_INET;

srv.sin_port = htons(53);

srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0){
  perror("bind");
  exit(1);
}
```

🏁 0:14

La creazione della socket è solo una delle operazioni preliminari; dopo la creazione della socket (commento), dobbiamo assegnare a quella socket un **indirizzo di trasporto**; questo perchè deve poter essere visibile anche **dall'esterno**.

Per fare ciò usiamo la struttura vista nella lezione precedente **sockaddr_in**. Questa struttura è una struttura caratterizzata da 3 campi significativi: un campo per specificare la famiglia degli indirizzi e due campi per specificare il numero di porta ed indirizzo ip.

Il campo relativo alla famglia degli indirizzi avrà valore **AF_INET**; questa costante indica che vogliamo utilizzare gli indirizzi della famiglia internet.

Per quanto riguarda il numero  di porta vogliamo usare la porta 53; questo numero sarà assegnato alla socket una delle funzioni di conversione viste in precedenza: in particolare ragioniamo con numeri rappresentabili con 16 bit (quindi usare gli short) e quindi impieghiamo la **funziona di conversione htons, ovvero host to network short**.

Per l'indirizzo, siccome stiamo definendo un indirizzo di trasporto che vogliamo assegnare alla socket presente sul server, dovremmo scegliere uno degli indirizzi ip della macchina (nel caso di un home computer è uno). L'alternativa è quella di usare una costante (come nel codice riportato sopra), che dice che vogliamo usare un qualsiasi indirizzo ip disponibile sulla macchina che esegue il processo; in questo modo il server può ricevere pacchetti da una qualsiasi interfaccia di rete (e quindi ip) presente (nella macchina).

### bind()

siamo finalmente pronti ad assegnare questo indirizzo alla socket creata in precedenza; questa operazione è effettuata invocando la funzione **bind()**: concettualmente ha due parametri:

1. L'intero che identifica localmente la socket
2. L'indirizzo di trasporto che vogliamo associare alla socket.

Infatti, il secondo parametro che vogliamo associare alla socket è **srv**, dichiarato in precedenza con la struct.

Il terzo parametro, relativamente poco importante, specifica **la dimensione della struttura**; questo perchè tutte le funzioni della socket sono state pensate con strutture di tipo diverso. Se guardiamo meglio il codice, notiamo che è presente un cast con `(struct sockaddr*) &srv`; possiamo pensare alla struttura **sockaddr** come una "**superclasse**" (anche se siamo in c), e non è altro che un generico indirizzo di trasporto.

Il vantaggio di usare **INADDR_ANY**, che come abbiamo visto indica di poter usare un qualsiasi indirizzo della macchina, è che possiamo scrivere il programma senza dover essere a conoscenza dell'ip della macchina su cui il server web verrà eseguito, quindi possiamo rendere indipendente il codice dalla macchina che lo esegue.

### bind() in poche parole

Quando invochiamo la socket() creiamo un endpoint **non inizializzato**; la socket è gestibile da un processo eprchè ha un identificativo numerico. Per contattare quell'endpoint abbiamo bisogno che l'endpoint abbia un **indirizzo significativo da remoto**, dobbiamo quindi assegnargli un **indirizzo di trasporto**, che è una coppia di informazioni: numero di porta ed inidirizzo ip della macchina (questi valori sono contenuti all'interno di una struct c).

Questa operazione viene eseguita tramite la **bind()**.

## recvfrom()

Dopo aver creato la socket ed aver assegnato ad essa un indirizzo, il server è pronto a ricevere messaggi. Questa operazione è effettuata tramite una primitiva simile alla receive vista nella lezione precedente.

Questa primitiva prevede come primo parametro il **descrittore della socket**, perché vogliamo specificare da che tipo di socket vogliamo ricevere.

Il secondo parametro è un **buffer**, in questo caso un array da 512 bytes, nel quale verranno riversati i dati che la recvfrom recupererà dalla rete.

Il terzo buffer specifica la dimensione del buffer, e quindi il numero massimo di bytes che possiamo estrarre dalla rete; se non specifichiamo il numero di byte massimo che può leggere, rischiamo di scrivere su aree di memoria non allocate, ottenendo un segmentation fault.

Successivamente abbiamo un campo utilizzato per caratterizzare il comportamento della recvfrom attraverso dei **flag**.

Il penultimo campo è l'indirizzo di trasporto del mittente (client) del messaggio che estraiamo (con la recvfrom stessa); viene effettuato un casting con una struct generica di tipo **sockaddr**.

L'ultimo parametro serve per gestire la flessibilità della funzinoe, viene passata anche la dimensione della struttura (del penultimo campo).
Questi due ultimi parametri sono molto importanti, perchè ci permettono di capire chi è il mittente, e quindi poter **filtrare** la risposta ad un insieme di ip, e quindi client, ristretto.

```
int fd;
struct sockaddr_in srv;
struct sockaddr_in cli:
char buf[512];
int cli_len = sizeof(cli);
int nbytes;

// creazione della socket
// binding della socket ad un indirizzo

nbytes = recvfrom(fd, buf, sizeof(buf), 0 /*flags*/, (struct sockaddr*) &cli,
&cli_len);

if(nbytes < 0){
  perror("recvfrom");
  exit(1);
}
```

### Flags

I flags usati nella funzione recvfrom sono i seguenti:

- **MSG_PEEK** che ci consente di estrarre i dati dal buffer di sistema senza rimuoverli; potranno quindi essere riestratti.
- **MSG_WAITALL** con questo flag aspettiamo che **tutti i frammenti** arrivino prima di estrarre il messaggio
- **MSG_DONTWAIT** che ci consente di non avere un comportamento **bloccante**; quindi la recvfrom **non sospenderà il processo** in attesa di un messaggio.

# Lato client per la comunicazione datagram

Ora vediamo come deve essere scritto il codice del client:

```
int fd;
struct sockaddr_in srv;
char buf[512];

//creazione socket
// sendto: invia dati all'indirizzo ip 128.2.35.50 e porta 53

srv.sin_family = AF_INET;
srv.sin_port = htons(53);
srv.sin_addr.s_addr = inet_addr("128-2-35-50");
```

```
nbytes = sendto(fd, buf, sizeof(buf), 0 /*flags*/, (struct sockaddr*) &srv,
sizeof(srv));

if(nbytes < 0){
  perror("sendto");
  exit(1);
}
```

> Per disaccoppiare il codice dall'istanza specifica, ovvero poter inviare messaggi a diversi server, e non solo a quello specificato nel codice, potremmo passare l'indirizzo come parametro o leggero da stdin.

Lato clienti **potremmo** usare la band() per associare alla socket un indirizzo di trasporto, ma non è necessario. E' necessario da lato server perchè l'endpoint **deve essere noto ai client**, proprio per il primo contatto, ma non è importante conoscere **a priori** l'indirizzo del client, proprio perchè può essere estratto dopo la prima comunicazione tra client e server.

Infatti, nella fase preliminare (prime righe dove viene esplicitato porta ed indirizzo), l'indirizzo di trasporto esplicitato è **quello del server**, che verrà usato dal client per inviare il messaggio.

🏁 0:42

**Quindi**, la funzione **sendto** trasferisce il contenuto del buffer specificato alla socket identificata precedentemente (server), attraverso la socket locale, il cui descrittore è fd.

la funzione **inet_addr** ci permette di trasformare l'indirizzo ip in un **intero a 32 bit** previsto dal campo **s_addr**, che è un **unsigned long**.

# Gestione degli indirizzi IP

## Convertire stringhe in numeri

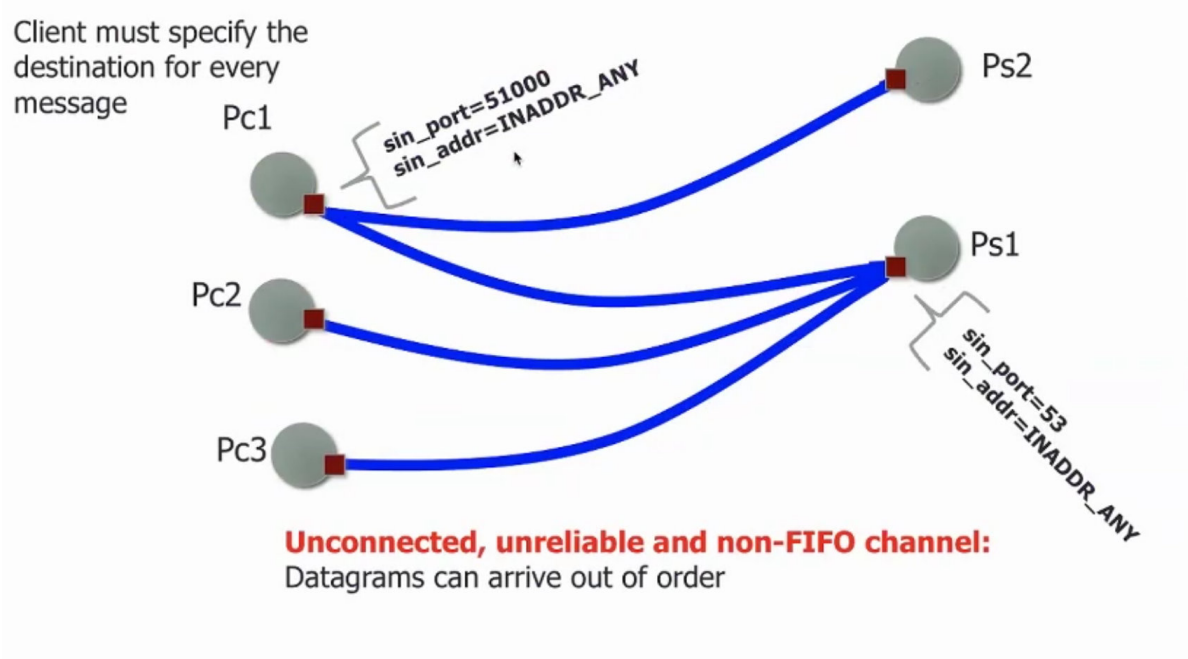
```
struct sockaddr_in srv;
srv.sin_addr.s_addr = inet_addr("12823550");
if(srv.sin_addr.s_addr == -1){
  fprintf(stderr, "inet_addr failed!\n");
  exit(1);
}

-----------------------------------------------

struct in_addr inp;
if(inet_aton("128.2.35.50", &inp) > 0)
  ...
  //invalid address
```

## Convertire numeri in stringhe

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0){
  //error
}
```

# One-to-may communication

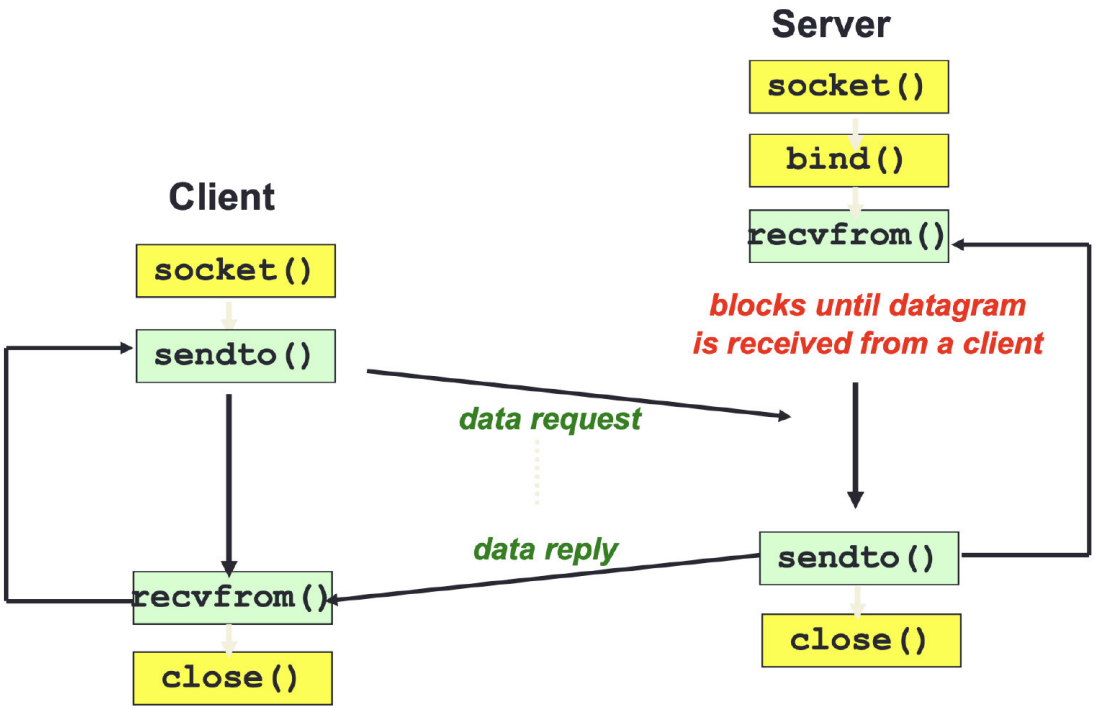
La socket creata dal lato client usata per comunicare con il server, può essere teoricamente usata con un server qualsiasi. Questo perchè questo tipo di socket è detta **"socket libera"**.




Questo tipo di comunicazione (protocllo UDP) non è **FIFO**, quindi non è detot che i pacchetti arrivino con lo stesso ordine di spedizione.

Le socket, essendo non connesse, sono utili nel momento in cui si vuole realizzare un tipo di collegamenteo **uno a molti**; questo non è possibile con le socket basate su **TCP**.

## Summary




# Esempio

Vogliamo realizzare un'applicazione dove il server tiene traccia del numero di contatti che riceve dal client; manterrà quindi una variabile il cui valore sarà incrementato ogni volta che il processo server sarà contattato dal client.

Ad ogni richiesta promossa dal client il server risponderà con il valore corrente del contatore.

Il client invia quindi invia un messaggio, e quando verrà ricevuto dal server, esso lo conterà come contatto. Successivamente all'invio del messaggio il client attenderà la risposta del server e poi mostrerà la risposta sullo stdout.

# Protocollo applicativo

Dobbiamo capire come è composto il messaggio di richiesta, e come è fatto il messaggio di risposta.

Nel messaggio di richiesta *non deve essere contenuto nulla*, proprio perché al server non interessa nessun tipo di messaggio specifico.

Il messaggio di risposta, d'altra parte, deve contenere il semplice numero dei contatti fino a quel momento.

# Programmi

La differenza con il codice scritto finora, è che invece di scrivere un singolo main ne verranno scritti due:

- Uno nel file server.c
- Uno nel file client.c

Dobbiamo inoltre scegliere un numero di porta: **5193**; questo sarà il numero di porta sul quale il server dovrà porsi in ascolto, e sarà la porta che il client dovrà contattare.

## Server.c

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>

#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]){
  char buf[256];
  int serverSocket;
  int clientAddrLen;
  int visist = 0;

  struct sockaddr_in serverAddr;
  struct sockaddr_in clientAddr;

  serverSocket = socket(PF_INET, SOCK_DGRAM, 0);

  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons(5193);
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

> Fino a questo punto l'unica cosa diversa dai programmi precedenti è il numero di porta.

### Funzione memset

La funzione memset è una funzione di inizializzazione della memoria a basso livello, e nel caso specifico stiamo dicendo che voglaimo inizializzare a zero, e quindi rimuovere qualsiasi contenuto informativo scritto in precedenza, dall'area di memoria a partire da `&serverAddr` a `sizeof(serverAddr)`.

---

Dopo queste operazioni preliminari, invochiamo la **bind**:

```
  bind(serverSocket, (struct sockaddr*) &serverAddr, sizeof(serverAddr));
  clientAddrLen = sizeof(clientAddr);

  while(1){
    memset(buf, 0, sizeof(buf));

    recvfrom(serverSocket, buf, sizeof(buf), 0, (struct sockaddr*) &clientaddr,
&clientAddrLen);

    visits++;
    sprintf(buf, "questo server è stato contattato %d volte", visits);

    sendto(serverSocket, buf, sizeof(buf), 0, (struct sockaddr*) &clientAddr,
clientAddrLen);
  }
}
```

> per rendere la comunicazione più efficiente potremmo inviare solo l'intero count.

Prima di ricevere anche in questo caso viene pulita l'area di memoria del buffer.

Se il client non ha inviato nulla, la recvfrom **blocca il server**.

### sprintf()

Invece di inviare solo l'intero `visits`, spediamo una stringa. La funzione **sprintf()** ci consente di inviare ad un'area di memoria specificata dal primo parametro (buf nel nostro caso). Scriviamo la stringa nel buffer che verrà poi inviato come risposta al client.

## Client.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>

#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]){
  char buf[256];
  int serverSocket;

  struct sockaddr_in remoteAddr;
  int remoteAddrLen;
```

```
  clientSocket = socket(PF_INET, SOCK_DGRAM, 0);

  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons(5193);
  serverAddr.sin_addr.s_addr = inet_addr(argv[1]); // passiamo l'indirizzo del
server come argomento così da poter usare il client con diversi server.

  remoteaddrLen = sizeof(remoteAddr);

  memset(buf, 0, sizeof(buf));
  sendto(clientSocket, buf, sizeof(buf), 0, (strct sockaddr*) &remoteAddr,
remoteAddrLen);

  memset(buf, 0, sizeof(buf));
  recvfrom(clientSocket, buf, sizeof(buf), 0, (struct sockaddr*) &remoteAddr,
&remoteAddrLen);

  printf("%s\n", buf);

  close(clientSocket);
}
```

> la memset va fatta ogni volta che il buffer *potrebbe* essere sporcato.

---

fine lezione 5