

Reti informatiche cod. 545II [9 CFU]

Corso di Laurea in Ingegneria Informatica

Laboratorio e Programmazione di Rete

A. A. 2021/2022

Francesco Pistolesi, PhD

Dipartimento di Ingegneria dell'Informazione

`francesco.pistolesi@unipi.it`

Programma di oggi

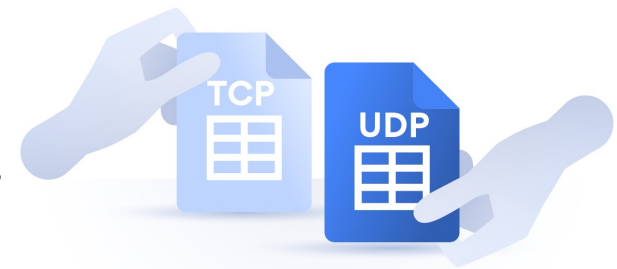
- Socket UDP
- Protocolli text e binary

Socket UDP

Socket UDP

Un socket **UDP** è **connectionless**, non usa operazioni preliminari per instaurare una connessione

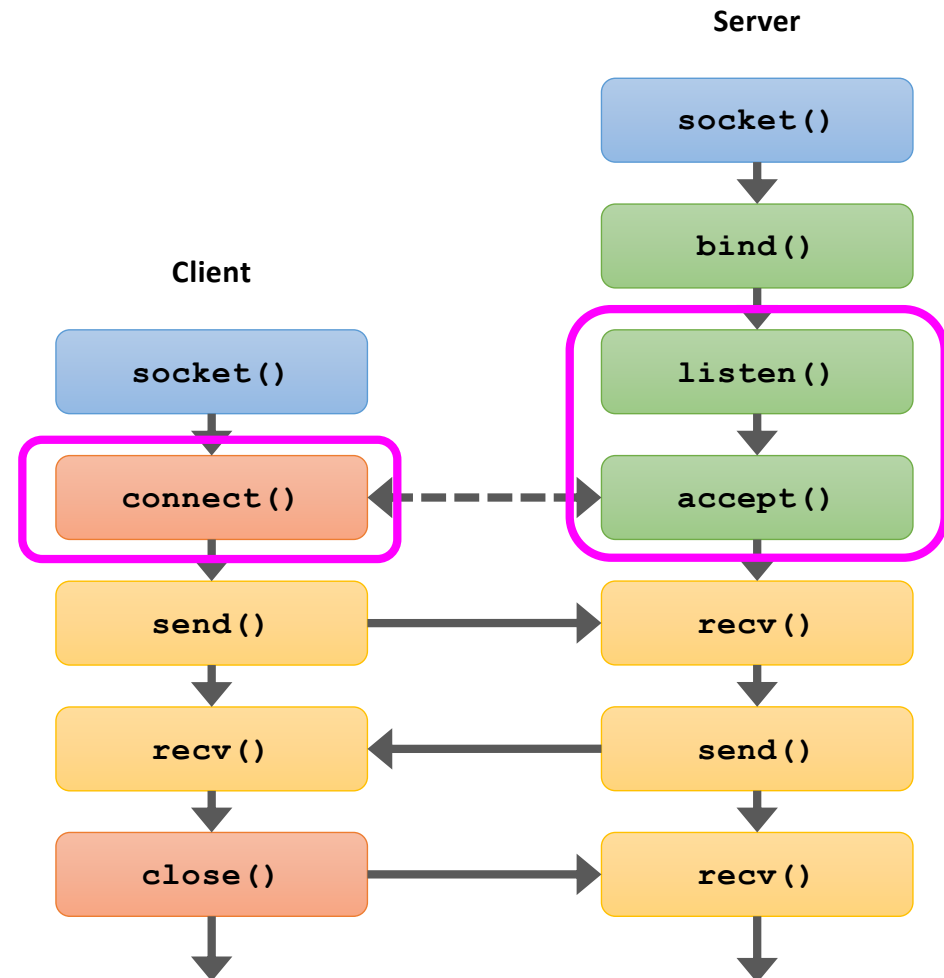
Più veloce di TCP: non fa nessun recupero,
nessun riordino, nessun controllo di flusso.
I pacchetti possono andare persi e/o corrompersi



Socket TCP

TCP instaura una **connessione**

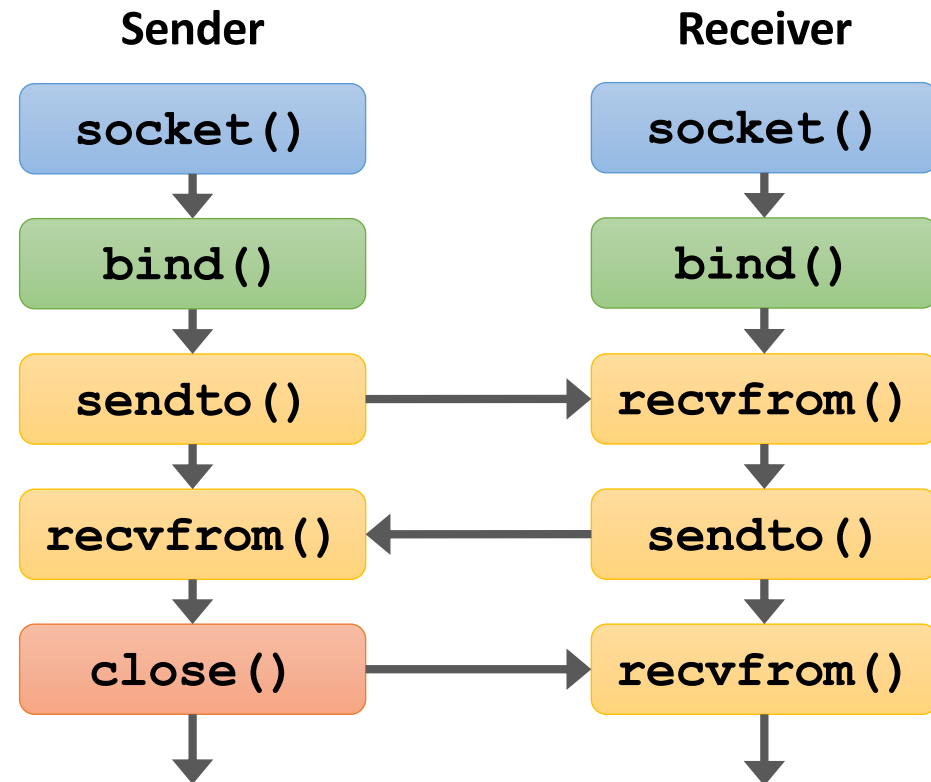
- Usa **operazioni preliminari** per creare un **canale** virtuale
- È affidabile: i pacchetti inviati **arrivano tutti, invariati**
- Comporta **latenza** per il riordino, ritrasmissioni, controllo di flusso...



Socket UDP

UDP non crea una connessione

- Le primitive **sendto()** e **recvfrom()** devono ogni volta specificare l'indirizzo del socket remoto con cui vogliono comunicare



Primitiva **sendto** ()

- **Invia un messaggio** attraverso un socket all'indirizzo specificato

```
ssize_t sendto(int sockfd, const void* buf, size_t len,  
                int flags, const struct sockaddr* dest_addr,  
                socklen_t addrlen);
```

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer contenente il messaggio da inviare
- **len**: dimensione in byte del messaggio
- **flags**: per settare le opzioni (lasciamolo a 0)
- **dest_addr**: puntatore alla struttura contenente l'indirizzo del destinatario
- **addrlen**: lunghezza di **dest_addr**
- Restituisce il **numero di byte inviati** (o -1 in caso di errore)
- **È bloccante**: il programma si ferma finché non ha scritto tutto il messaggio

Primitiva **recvfrom()**

- **Riceve un messaggio** attraverso un socket


```
ssize_t recvfrom(int sockfd, const void* buf, size_t len,  
                  int flags, struct sockaddr* src_addr,  
                  socklen_t addrlen);
```

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer contenente il messaggio da ricevere
- **len**: dimensione in byte del messaggio
- **flags**: per settare le opzioni
- **src_addr**: puntatore a una struttura vuota per salvare l'indirizzo del mittente
- **addrlen**: lunghezza di **dest_addr**
- Restituisce il **numero di byte ricevuti**, -1 in caso di errore, oppure 0 se il socket remoto si è chiuso
- **È bloccante**: il programma si ferma finché non ha letto *qualcosa*

Codice del server

```
int main () {
    int ret, sd, len;
    char buf[BUFLen];
    struct sockaddr_in my_addr, cl_addr;
    int addrlen = sizeof(cl_addr);
    /* Creazione socket UDP */
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    /* Creazione indirizzo */
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4242);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));

    while(1) {
        len = recvfrom(sd, buf, BUFLen, 0,
                      (struct sockaddr*)&cl_addr, &addrlen);
        //fai cose ...
    }
}
```




Codice del client

```
int main () {
    int ret, sd, len;
    char buf[BUFLEN];
    struct sockaddr_in sv_addr; // Struttura per il server

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_DGRAM, 0);

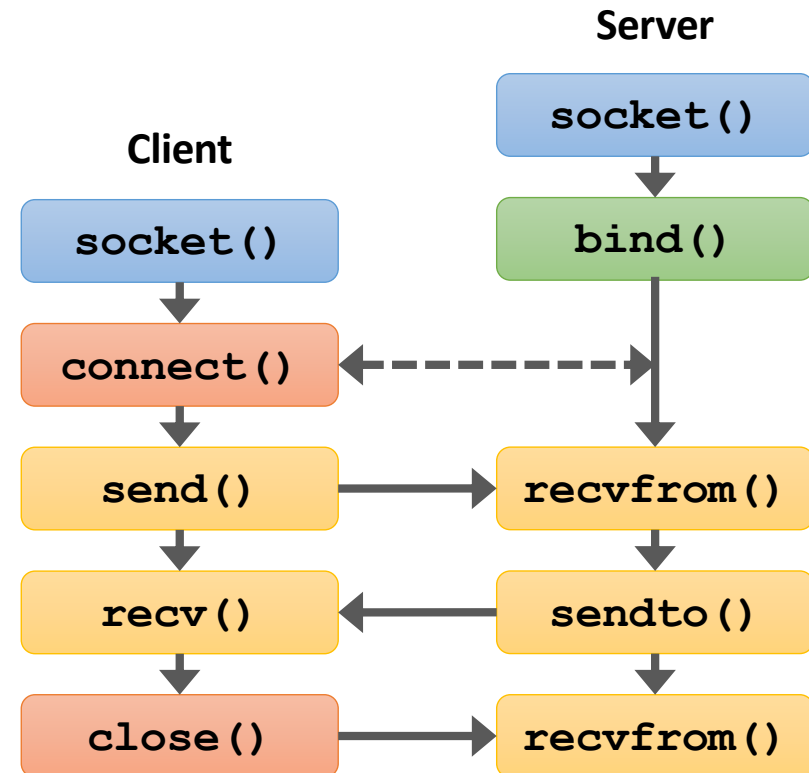
    /* Creazione indirizzo del server */
    memset(&sv_addr, 0, sizeof(sv_addr)); // Pulizia
    sv_addr.sin_family = AF_INET;
    sv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &sv_addr.sin_addr);

    while(1) {
        len = sendto(sd, buf, BUFLen, 0,
                    (struct sockaddr*)&sv_addr, sizeof(sv_addr));
        // fai cose...
    }
}
```



Socket UDP "connesso"

- Un client può usare **connect ()** per associare il **socket UDP** a un indirizzo remoto (quello del server)
- Il socket del client riceverà/invierà pacchetti **solo da/a quell'indirizzo**
- Ogni volta che invia (riceve), il client può **non specificare** l'indirizzo del destinatario (mittente), lasciando quindi a NULL il parametro **dst_addr (src_addr)** di **sendto()** (**recvfrom()**), mettendo a 0 **addrlen**
- In alternativa, come mostrato in figura, il client può anche **send ()** e **recv ()**
- **Non** è una connessione, nel livello di trasporto c'è ancora UDP!



Protocolli Text and Binary

Text Protocols vs Binary Protocols

Molti protocolli a livello applicativo inviano messaggi in formato testo (**text protocols**) mentre altri inviano le strutture dati (**binary protocols**)

Per il testo si usa solitamente
la codifica ASCII



Siparietto...

I am trying to pass whole structure from client to server or vice-versa. Let us assume my structure as follows

38

```
struct temp {  
    int a;  
    char b;  
}
```



37

I am using **sendto** and sending the address of the structure variable and receiving it on the other side using the **recvfrom** function. But I am not able to get the original data sent on the receiving end. In sendto function I am saving the received data into variable of type struct temp.

```
n = sendto(sock, &pkt, sizeof(struct temp), 0, &server, length);  
n = recvfrom(sock, &pkt, sizeof(struct temp), 0, (struct sockaddr *)&from, &fromlen
```

Where pkt is the variable of type struct temp.

Eventhough I am receiving 8bytes of data but if I try to print it is simply showing garbage values. Any help for a fix on it ?

Il guru risponde...



This is a very bad idea. Binary data should always be sent in a way that:

59



- Handles different **endianness**
- Handles different **padding**
- Handles differences in the **byte-sizes of intrinsic types**

Don't ever write a whole struct in a binary way, not to a file, not to a socket.

Come fare

Don't ever write a whole struct in a binary way, not to a file, not to a socket.

Always write each field separately, and read them the same way.

You need to have functions like

```
unsigned char * serialize_int(unsigned char *buffer, int value)
{
    /* Write big-endian int value into buffer; assumes 32-bit int and 8-bit char. */
    buffer[0] = value >> 24;
    buffer[1] = value >> 16;
    buffer[2] = value >> 8;
    buffer[3] = value;
    return buffer + 4;
}

unsigned char * serialize_char(unsigned char *buffer, char value)
{
    buffer[0] = value;
    return buffer + 1;
}

unsigned char * serialize_temp(unsigned char *buffer, struct temp *value)
{
    buffer = serialize_int(buffer, value->a);
    buffer = serialize_char(buffer, value->b);
    return buffer;
}
```

*Altrimenti usare
la htonl()*

Esempio

Codifica del numero 1234 in formato binary e text

binary

1234

byte 0

00000000

byte 1

00000000

byte 2

00000100

byte 3

11010010

text

"1234"

byte 0

00110001

byte 1

00110010

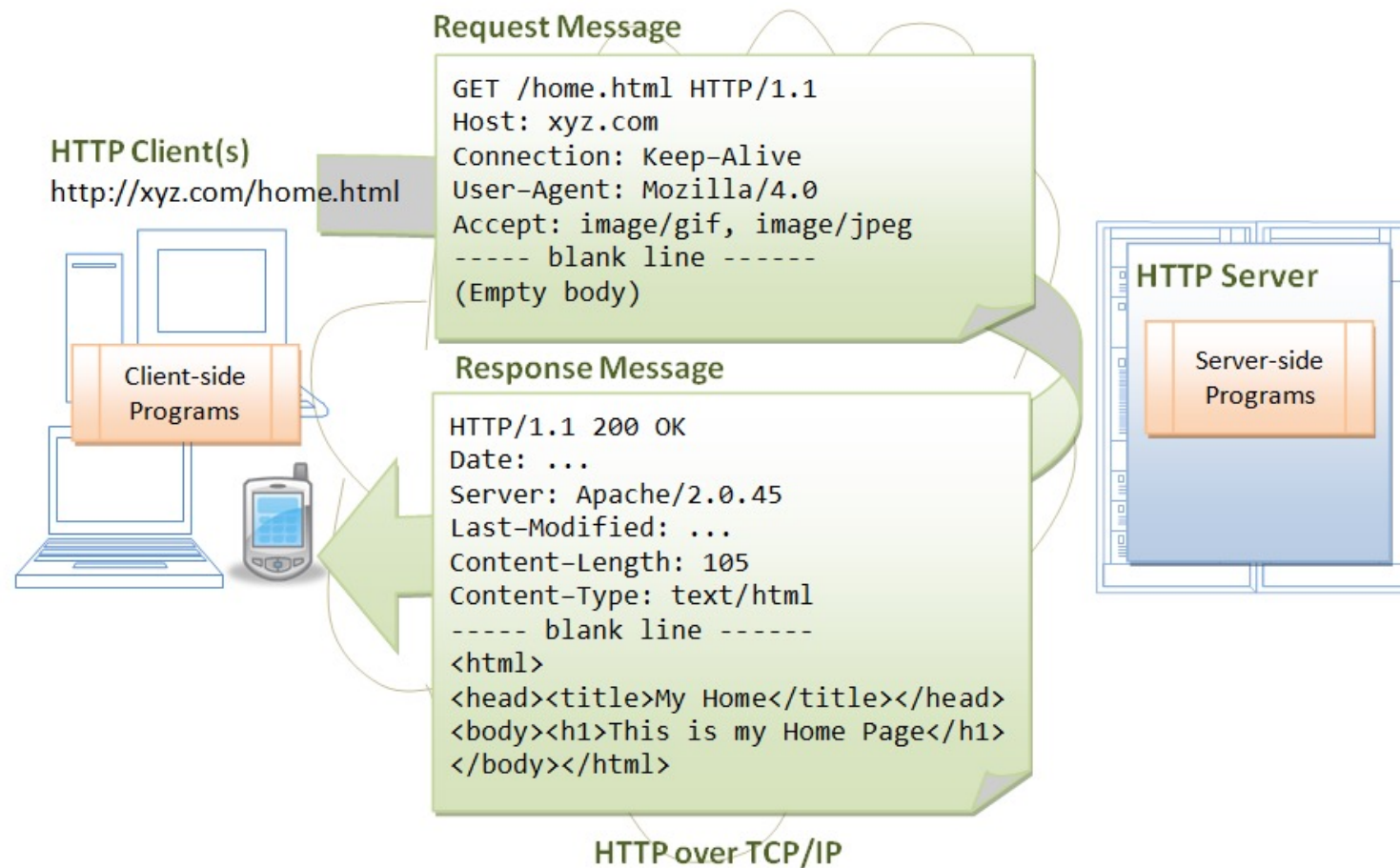
byte 2

00110011

byte 3

00110100

Text Protocols: esempio



Text protocols: da struttura a stringa

La struttura può essere convertita in testo:

```
char buffer[1024];

// Dichiarazione
struct temp{
    int a;
    char b;
};

// Istanziamento
struct temp t;

// Conversione a stringa
sprintf(buffer, "%d %c", t.a, t.b);
```

Text protocols: da stringa a struttura

- Il ricevente deve fare il *parsing* il messaggio:

```
...  
  
// Istanziamento  
struct temp t;  
  
// Ricezione  
...  
  
// Parsing e memorizzazione nei campi  
sscanf(buffer, "%d %c", &t.a, &t.b);
```

Binary Protocol

- Definire messaggi con **struttura fissata**, con campi che rappresentano l'informazione da scambiare
- Ogni campo ha una **lunghezza** e un **tipo che possa essere trasferito**
- Alcuni protocolli di tipo binario possono contenere campi di lunghezza variabile, in quel caso comunque il protocollo definisce una **lunghezza massima dei messaggi**

Binary Protocol: invio

```
...
struct temp{
    uint32_t a;
    uint8_t b;
};
struct temp t;
...
// Convertire in network order prima dell'invio
t.a = htonl(t.a);

// Spedire i campi sul socket 'new_sd'
ret = send(new_sd, (void*)&t.a, sizeof(uint32_t), 0);
...
ret = send(new_sd, (void*)&t.b, sizeof(uint8_t), 0);
...
```

Binary Protocol: ricezione

```
...
struct temp t;
...
ret = recv(new_sd, (void *)&t.a, sizeof(uint32_t), 0);
if (ret < sizeof(uint32_t)){
    // Gestione errore
}
// Convertire in host order il campo 'a'
t.a = ntohl(t.a);

ret = recv(new_sd, (void *)&t.b, sizeof(uint8_t), 0);
if (ret < sizeof(uint8_t)){
    // Gestione errore
}
...
```

Esercizio 1

- Implementare un semplice server UDP che attende richieste dai client e fa echo di ciò che riceve
- Implementare il client sia con socket UDP che con socket UDP connesso.
- Assumere una dimensione massima del messaggio di 20 byte

Esercizio 2

- Implementare un server UDP che periodicamente invia l'ora ai client che si registrano
 - Il server, periodicamente, controlla se c'è una richiesta di registrazione, se c'è, allora registra il client, poi invia a tutti i client registrati un pacchetto UDP contenente la data e ora
- Implementare il client che invia la richiesta, attende la risposta e la stampa ogni volta che arriva
- Non usare `fork()` o `select()`