

Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica
Sistemi operativi e reti
A.A. 2018-2019

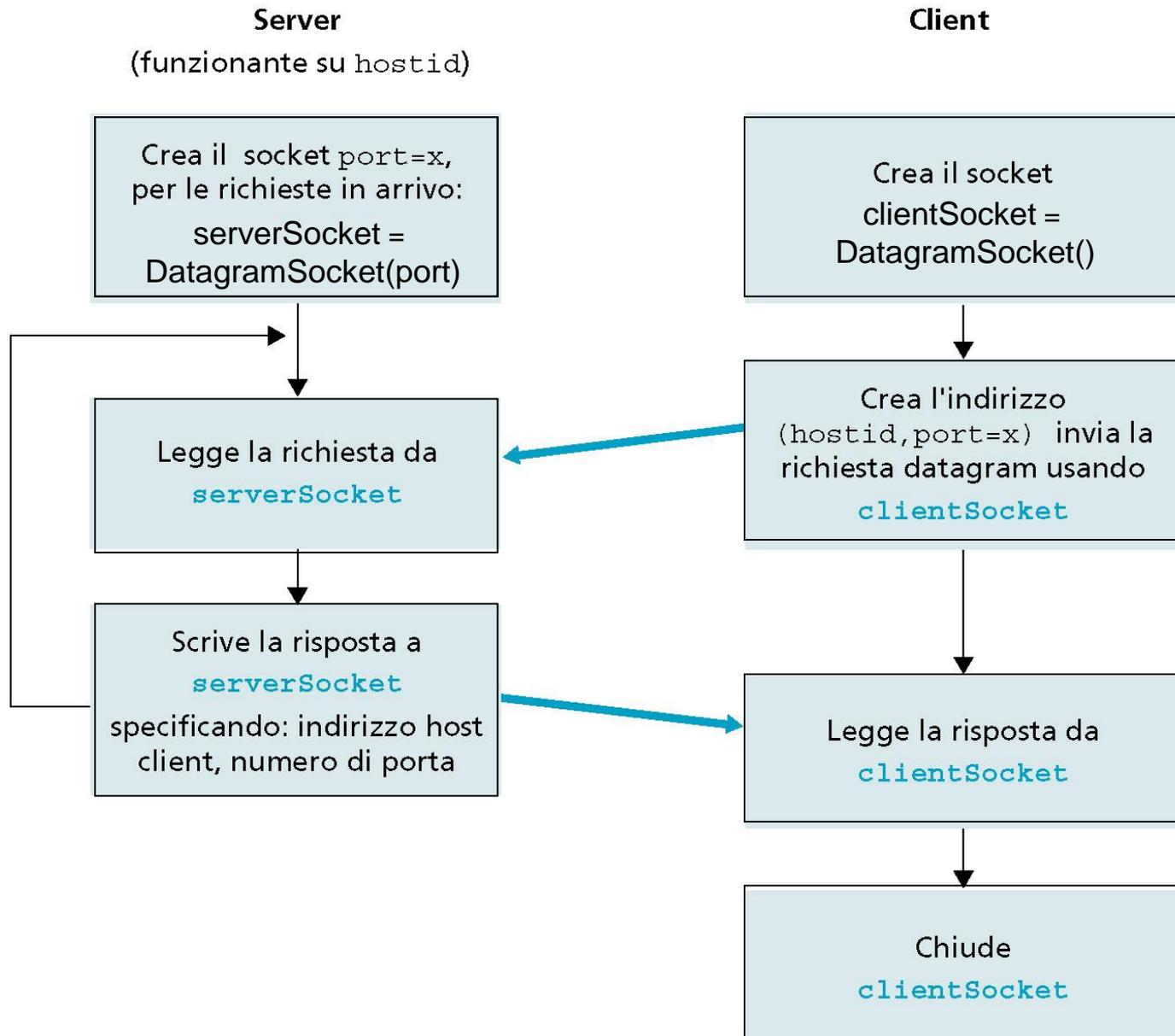
Pietro Frasca

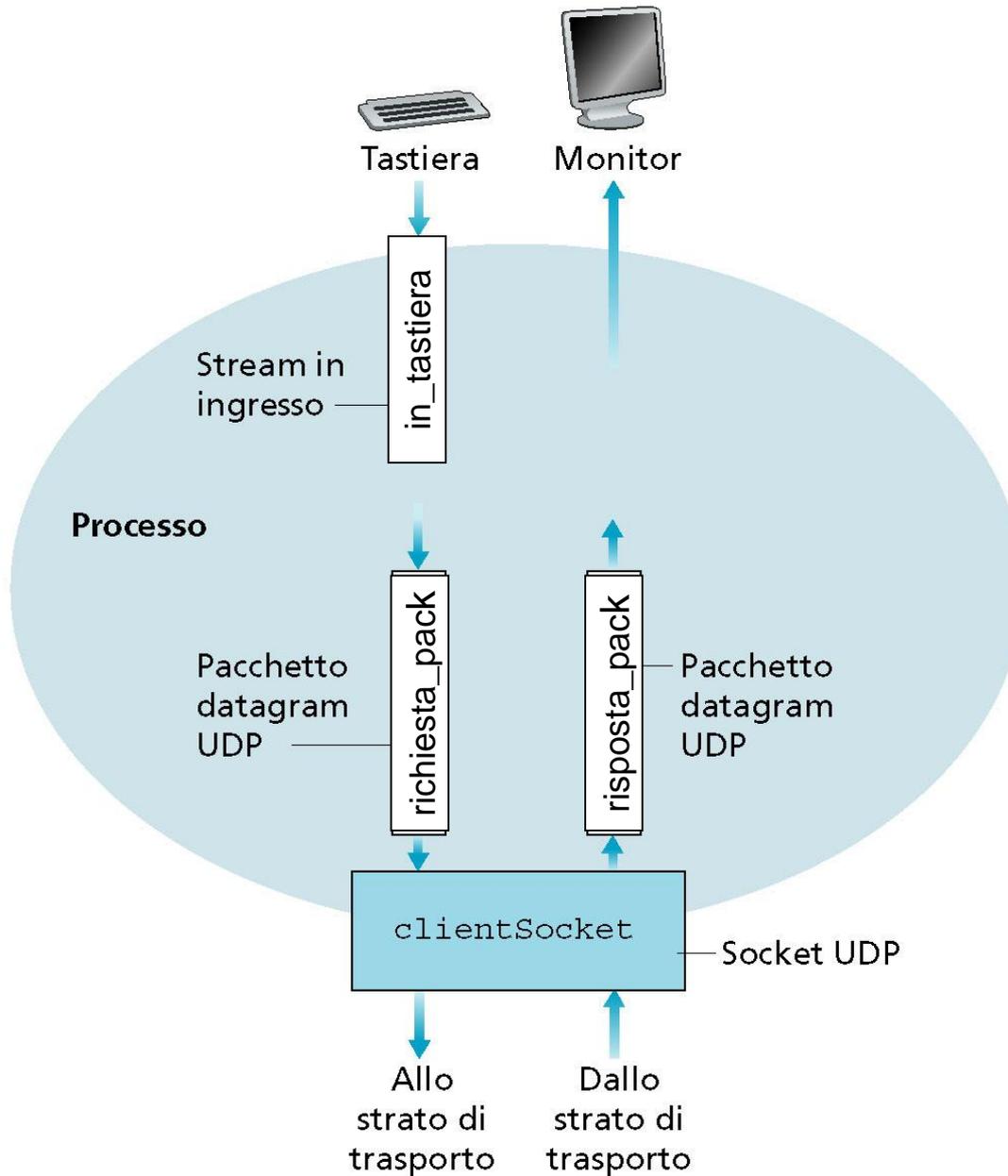
Parte II: Reti di calcolatori
Lezione 13 (37)

Martedì 16-04-2019

Programmazione delle socket con UDP

- Anche l'UDP consente a due (o più) processi remoti di comunicare.
- Poiché l'UDP non crea un canale permanente, un processo destinatario identifica **l'indirizzo IP** e il **numero di porta** del mittente del processo mittente estraendoli dai dati ricevuti.
- Questa identificazione deve essere fatta per ogni segmento che il destinatario riceve.
- La figura seguente mostra le principali attività della connessione UDP.





```

1.  import java.io.*;
2.  import java.net.*;
3.  class UDPClient {
4.      public static void main(String args[]) throws Exception{
5.          String nomeServer = "localhost";
6.          int portaServer = 1234;
7.          String richiesta = ".,risposta;
8.          BufferedReader in_tastiera = new BufferedReader (new
                InputStreamReader(System.in) ) ;
9.          DatagramSocket clientSocket = new DatagramSocket();
10.         InetAddress indirizzoIPServer =
                InetAddress.getByName(nomeServer); //client DNS
11.         byte[] richiesta_b = new byte[1024];
12.         byte[] risposta_b = new byte[1024];
13.         while (! richiesta.equals("")) {
14.             System.out.print("Scrivi una frase: ");
15.             richiesta = in_tastiera.readLine();
16.             richiesta_b = richiesta.getBytes();
17.             DatagramPacket richiesta_pack = new
                DatagramPacket(richiesta_b,richiesta_b.length,
                indirizzoIPServer,1234);

```

```
18.     clientSocket.send(richiesta_pack);
19.     DatagramPacket risposta_pack = new
        DatagramPacket(risposta_b,risposta_b.length);
20.     clientSocket.receive(risposta_pack);
21.     risposta = new String(risposta_pack.getData());
22.     System.out.println("Risposta dal Server:" + risposta);
23.     }
24.     clientSocket.close();
25.     }
26. }
```

- L'UDP usa un tipo diverso di socket rispetto al TCP. La socket `clientSocket`, è un oggetto della classe `DatagramSocket` invece che di `Socket`.
- Lo stream `in_tastiera` è come nell'esempio del TCP.
- Adesso esaminiamo le linee del codice che differiscono in modo significativo da quelle del `TCPClient.java`.
- `DatagramSocket clientSocket = new DatagramSocket();`

crea l'oggetto `clientSocket` della classe `DatagramSocket`. Il client non esegue la connessione con il server eseguendo questa istruzione. Infatti, il costruttore di `DatagramSocket()` non ha parametri per specificare l'hostname e il numero di porta del server. Semplicemente, crea una porta per il processo client (con numero superiore a 1024) ma non crea un canale logico fra i due processi.

```
InetAddress indirizzoIPServer =  
    InetAddress.getByNome(nomeServer);
```

Per comunicare con un processo di destinazione, è necessario ottenere l'indirizzo IP dell'host su cui gira il processo stesso.

- Il metodo **getByName()**, implementa il lato client del DNS e ritorna l'indirizzo IP del server nell'oggetto **indirizzoIPServer** di tipo **InetAddress**.
- **byte[] richiesta_b = new byte[1024];**
byte[] risposta_b = new byte[1024];

Gli array di byte **richiesta_b** e **risposta_b** contengono i dati che il client invia e riceve, rispettivamente.

▪

- `richiesta_b = richiesta.getBytes();`

La linea sopra esegue la conversione di tipo da string ad array di byte

- `DatagramPacket richiesta_pack =
 new DatagramPacket(richiesta_b,
 richiesta_b.length, indirizzoIPServer,
 1234);`

Questa linea costruisce il pacchetto, `richiesta_pack`, che il client invia al server mediante la suo socket.

Il pacchetto contiene:

- i dati che sono contenuti nell'array `richiesta_b`;
- La dimensione dell'array `richiesta_b`;
- l'indirizzo IP del server;
- il numero della porta del processo server (in questo esempio 1234).

- `clientSocket.send(richiesta_pack);`

Il metodo `send()` dell'oggetto `clientSocket` il pacchetto appena costruito.

- `DatagramPacket risposta_pack = new
DatagramPacket(risposta_b, risposta_b.length);`

il client crea un oggetto `risposta_pack` di tipo `DatagramPacket`, per contenere i dati che riceverà dal server.

- `clientSocket.receive(risposta_pack);`

Il client **resta bloccato**, fino alla ricezione di un pacchetto; quando il client riceve un segmento, si risveglia e il contenuto del segmento sarà salvato in `risposta_pack`.

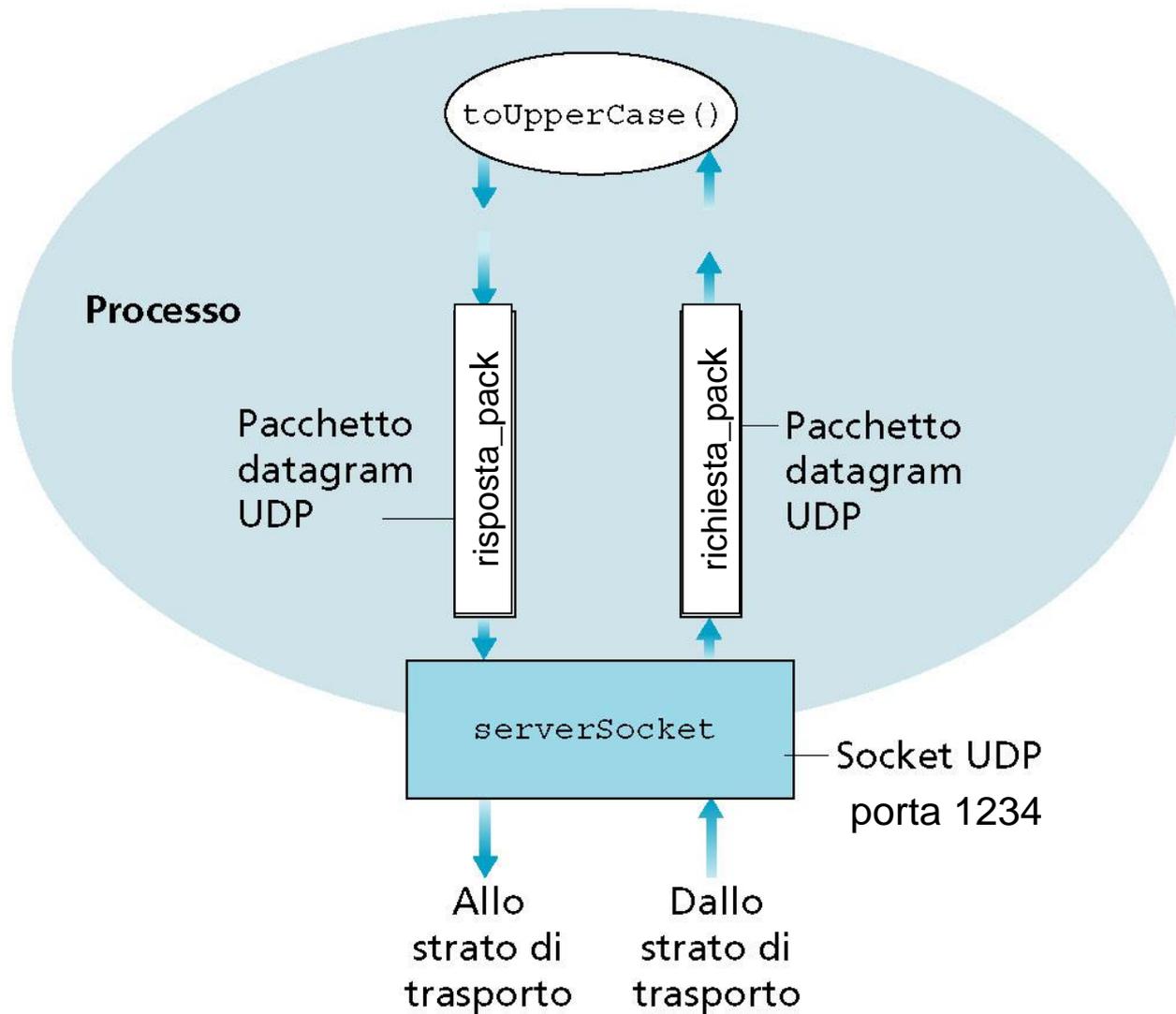
- `String risposta = new String(risposta_pack.getData());`

estrae i dati da `risposta_pack` e compie una conversione di tipo, convertendo un array di byte nella stringa `risposta`.

- `System.out.println("Risposta dal server:" + risposta);`

stampa la stringa `risposta` sul monitor del client.

- `Clientsocket.close();` Chiude la socket.



```
1. import java.io.*;
2. import java.net.*;
3. class UDPServer {
4.     public static void main(String args[]) throws Exception{
5.         int portaServer = 1234;
6.         DatagramSocket serverSocket = new
           DatagramSocket(portaServer);
7.         System.out.println("Server UDP in ascolto sulla porta"
           + portaServer);
8.         byte[] richiesta_b = new byte[1024];
9.         byte[] risposta_b = new byte[1024];
10.        while(true){
11.            DatagramPacket richiesta_pack = new
               DatagramPacket(richiesta_b, richiesta_b.length);
12.            serverSocket.receive(richiesta_pack); //metodo bloccante
13.            String richiesta = new String(richiesta_pack.getData());
14.            InetAddress indirizzoIPClient =
               richiesta_pack.getAddress();
15.            int portaClient = richiesta_pack.getPort();
16.            String risposta = richiesta.toUpperCase();
17.            risposta_b = risposta.getBytes();
18.            DatagramPacket risposta_pack = new
               DatagramPacket(risposta_b, risposta_b.length,
               indirizzoIPClient, portaClient);
19.            serverSocket.send(risposta_pack);
20.        }
21.    }
22.}
```

- Il programma `UDPServer.java` costruisce una socket, come mostrato in figura.
- La socket `serverSocket` è un oggetto della classe `DatagramSocket`, come la socket del lato client dell'applicazione.
- Vediamo ora le linee del codice che differiscono da `TCPServer.java`.
- **`DatagramSocket serverSocket = new DatagramSocket(1234);`**
- Crea l'oggetto `serverSocket` con porta 1234. Tutti i dati inviati e ricevuti passeranno attraverso questa socket. Poiché l'UDP è senza connessione, non serve una nuova socket per ascoltare nuove richieste di connessione, come fatto in `TCPServer.java`. Se più client accedono a questa applicazione, essi dovranno tutti spedire i loro pacchetti su questa porta singola, `serverSocket`.

- `String richiesta = new
String(richiesta_pack.getData));
InetAddress indirizzoIPClient =
richiesta_pack.getAddress();
int portaClient = richiesta_pack.getPort();`

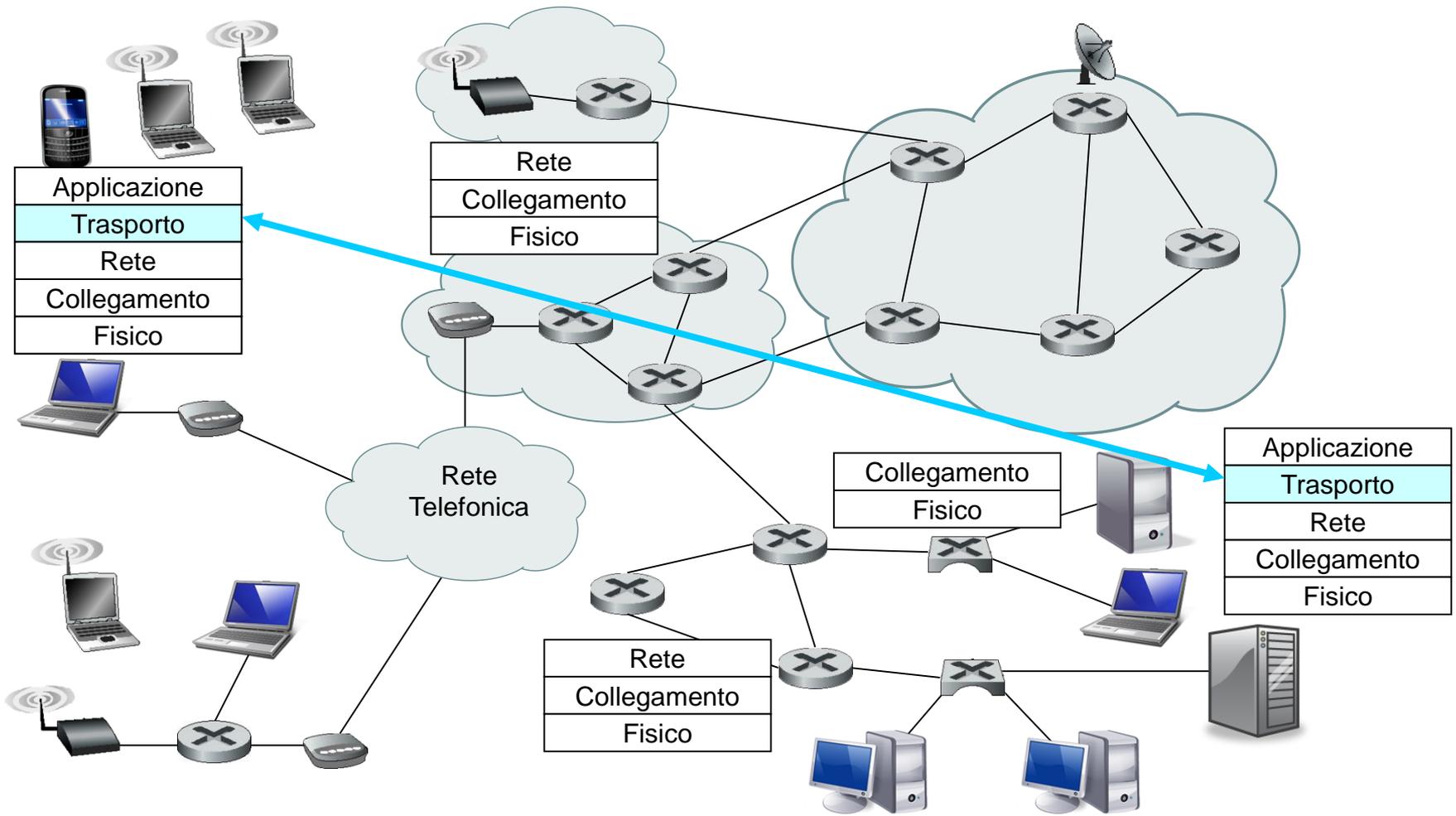
le tre linee estraggono i campi dal pacchetto:

- la prima linea estrae i dati dal pacchetto e li assegna alla string richiesta.
- la seconda linea estrae l'indirizzo IP del client;
- la terza linea estrae il *numero di porta del client*.

Per il server è necessario identificare il client, mediante l'indirizzo IP e il numero di porta, per poter rinviare la frase con lettere maiuscole.

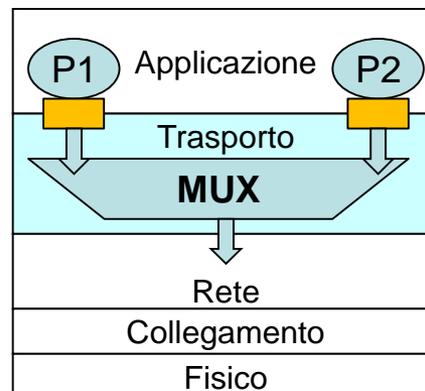
Lo strato di trasporto

- Un protocollo dello strato di trasporto fornisce una **comunicazione logica fra i processi applicativi** che sono in esecuzione su host remoti.
- Per comunicazione logica si intende che dal punto di vista dell'applicazione, è come se gli host remoti fossero direttamente connessi.
- In riferimento alla pila protocollare del TCP/IP, i pacchetti dello strato di trasporto (strato 4) sono chiamati **segmenti**.
- I protocolli dello strato di trasporto che esamineremo sono il **TCP** (*Transmission Control Protocol*) e l'**UDP** (*User Datagram Protocol*).

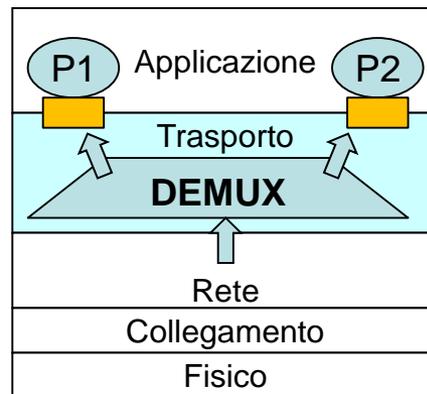


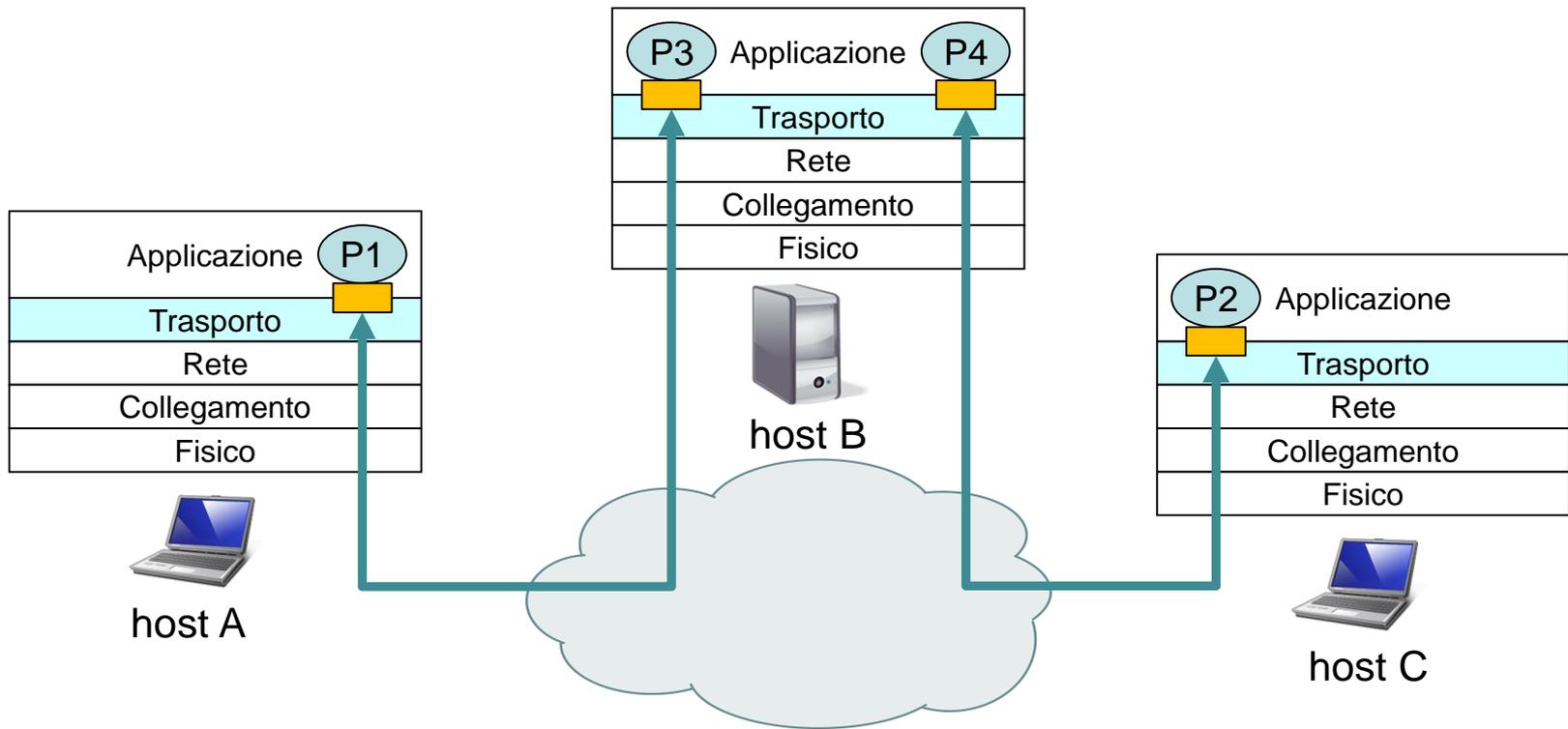
Multiplexing e demultiplexing

- Poiché, allo stesso tempo, in un host possono essere attivi più processi di rete e ciascuno può utilizzare una o più socket, ogni socket è identificata in modo **univoco**.
- TCP e UDP identificano una socket con parametri diversi.
- La funzione di **multiplexing** è eseguita dal protocollo di trasporto quando i processi di rete inviano dati e consiste nelle seguenti operazioni:
 - prelevare i dati dalle socket dei processi mittenti;
 - aggiungere campi di intestazione ai dati di ciascun processo. Tra questi campi ci sono il campo **numero di porta sorgente** e il campo **numero di porta di destinazione**.
 - passare i segmenti allo strato di rete.



- La funzione di **demultiplexing** è eseguita dal protocollo di trasporto quando i processi applicativi ricevono dati dalla rete e consiste nelle seguenti operazioni:
 - prelevare i segmenti dallo strato di rete;
 - assegnare i dati, contenuti in un segmento, alla corretta socket, esaminando il campo numero di porta **destinazione**, usato nella fase di multiplexing dall'host mittente.





Multiplexing e demultiplexing dello strato di trasporto

- Nella figura, lo strato di trasporto nell'host B deve demultiplexare i segmenti che arrivano dallo strato di rete, per entrambi i processi P3 e P2, consegnando i dati dei segmenti in arrivo alla socket del corrispondente processo.
- Lo strato di trasporto nell'host B deve anche eseguire la funzione di multiplexing, raggruppando i dati uscenti da queste socket, costruire segmenti, e passare questi segmenti allo strato di rete.

- Il numero di porta è un campo a 16 bit, e va da 0 a 65535.
- I numeri di porta compresi tra 0 e 1023 sono chiamati **numeri di porta ben conosciuti (well-known port numbers)**; sono assegnati dall'Internet Assigned Numbers Authority (IANA).



I campi numeri di porta sorgente e destinazione in un segmento dello strato di trasporto

Multiplazione e demultiplazione senza connessione (UDP)

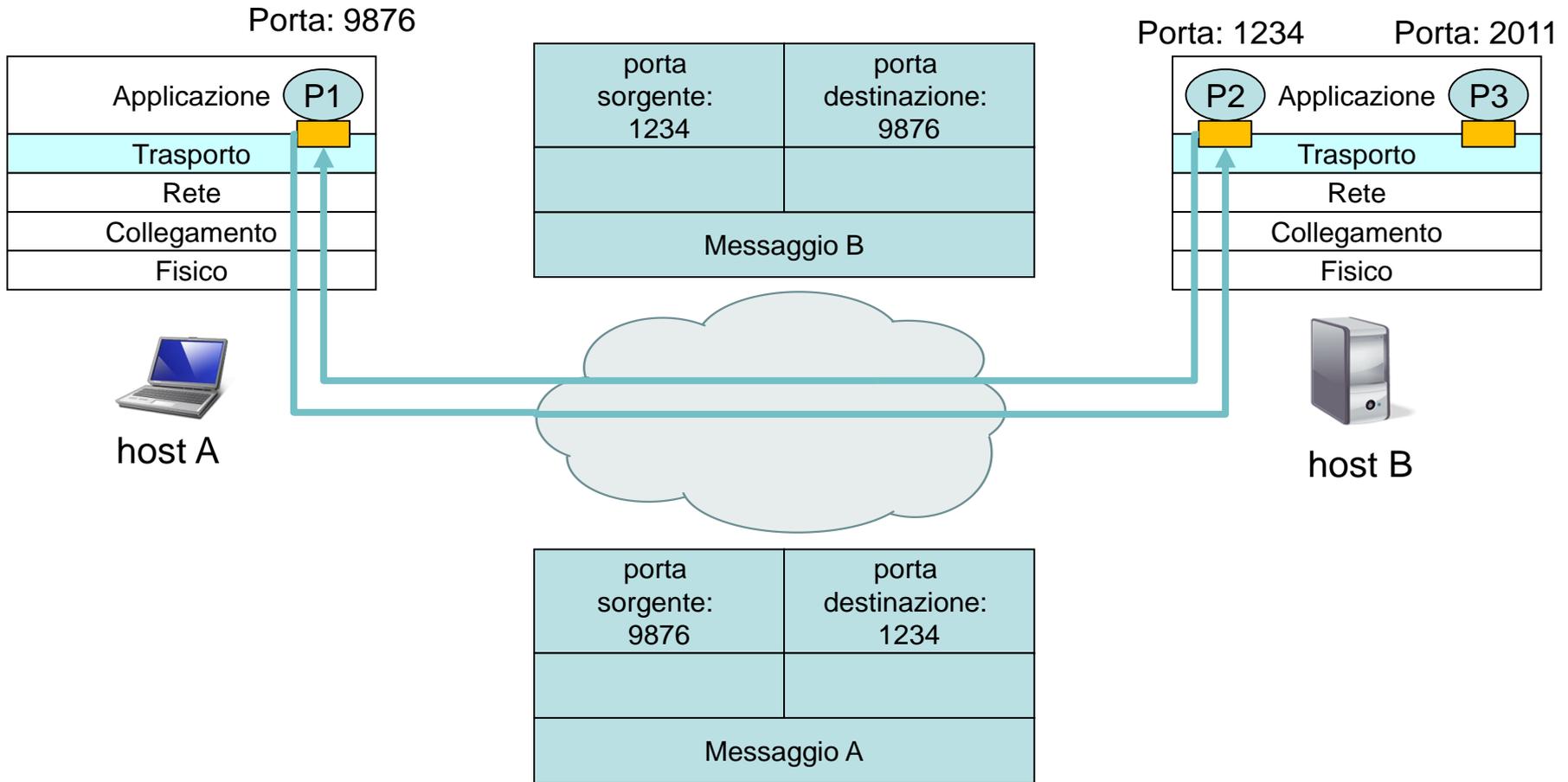
- Ricordiamo che java consente di creare una socket UDP creando un oggetto della classe DatagramSocket:

DatagramSocket mioSocket = new DatagramSocket();

- In questo caso, non passando al costruttore di **DatagramSocket()** un numero di porta, UDP assegna automaticamente un numero di porta alla socket, nel intervallo da 1024 a 65535, che in quel momento non è assegnato ad altre porte UDP nell'host.
- In alternativa, si potrebbe creare una socket assegnando un numero di porta specifico (nel nostro esempio, 1234) alla socket UDP.

DatagramSocket mioSocket = new DatagramSocket(1234);

- Generalmente, nel lato client dell'applicazione si assegna automaticamente il numero di porta (detto numero di porta effimero), mentre nel lato server si assegna un numero di porta specifico.
- Facciamo un esempio. Supponiamo che un processo con una socket con numero di porta 9876 che gira sull'host A voglia inviare un messaggio a un processo con socket UDP con numero di porta 1234 nel host B. L'UDP nel host A crea un segmento che contiene:
 - **il numero di porta sorgente (9876)**
 - **il numero di porta di destinazione (1234)**
 - **altri due campi (che vedremo dopo).**
 - **i dati dell'applicazione**
- L'UDP passa quindi il segmento allo strato di rete il quale incapsula il segmento in un **datagram IP**.



Multiplexing e demultiplexing con UDP

- Se il segmento arriva all'host B, l'UDP in B esamina il numero di porta di destinazione nel segmento (1234) e consegna il segmento al suo socket identificato dalla porta 1234.
- Sull'host B potrebbero essere attivi più processi di rete, ciascuno con il suo socket UDP e il numero di porta associato.
- Un **socket UDP** è univocamente determinato da due parametri consistenti in un **indirizzo IP** e un **numero di porta**.
- Il numero di **porta sorgente** serve per costruire l'indirizzo di ritorno: quando B vuole mandare indietro un segmento ad A il numero di porta di destinazione nel segmento da B ad A assumerà il valore del numero di porta sorgente del segmento da A a B. L'indirizzo completo di ritorno è formato dall'indirizzo IP di A e dal numero di porta sorgente.
- Per esempio, riprendiamo il codice del server UDP scritto in java. Il server, per conoscere il numero di porta sorgente lo recupera dal segmento che riceve dal client usando il metodo `getPort()` di `DatagramPacket` :

```
int portaClient = richiesta_pack.getPort().
```

esso manda quindi un nuovo segmento al client, con il numero di porta sorgente estratto che serve da numero di porta di destinazione in questo nuovo segmento.

Multiplazione e demultiplazione nel TCP

- A differenza di UDP, una socket TCP è identificata da 4 parametri:
 - **indirizzo IP del mittente**
 - **numero di porta mittente**
 - **indirizzo IP di destinazione**
 - **numero di porta di destinazione.**
- I segmenti TCP che arrivano al destinatario, che hanno indirizzi IP del mittente diversi o numeri di porta mittente diversi saranno diretti verso diverse socket.
- Per chiarire questo punto riprendiamo l'esempio del server TCP che abbiamo realizzato.
- L'applicazione server TCP crea una "**socket di ascolto**" sulla porta numero 1234 e resta in attesa fino a quando giunge una richiesta di connessione da un client.

- Il client TCP genera **un segmento di richiesta** di instaurazione di connessione con:

Socket clientSocket = new Socket ("nomeServer", 1234) ;

- Una richiesta di instaurazione di connessione consiste in uno speciale segmento TCP con un numero di porta di destinazione (nell'esempio 1234) e un **bit** di instaurazione di connessione posto a 1 nell'intestazione TCP. Il segmento include anche un numero di porta mittente, scelto dal client TCP (l'indirizzo IP del client sarà contenuto nell'intestazione dell'IP).
- La linea di cui sopra crea anche un socket TCP attraverso il quale il client può inviare e ricevere i dati.
- Quando il TCP nel server riceve il segmento di richiesta di connessione con porta di destinazione 1234 da un client, esso informa il processo server che crea un socket di connessione:

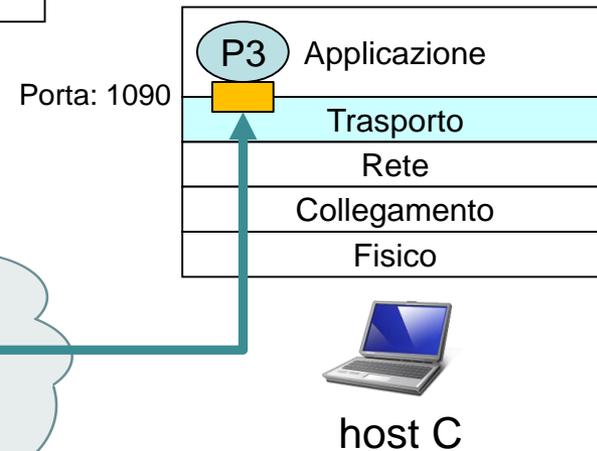
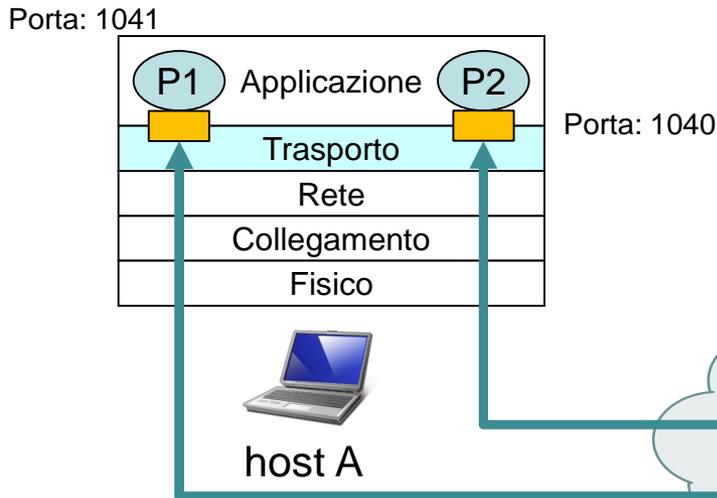
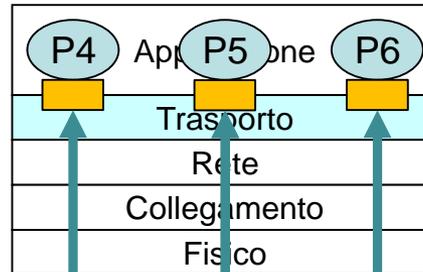
Socket connection `Socket = ascoltoSocket.accept();`

- Inoltre, il TCP lato server associa i seguenti quattro valori del segmento di richiesta di connessione alla socket di connessione appena creata:
 - **il numero di porta del host mittente**
 - **l'indirizzo IP del host mittente**
 - **il numero di porta di destinazione (del server)**
 - **il proprio indirizzo IP (del server).**
- tutti i segmenti che arriveranno, aventi la porta sorgente, l'indirizzo IP sorgente, la porta destinazione e l'indirizzo IP destinazione uguale a questi quattro valori saranno indirizzati a questa socket.
- Una volta instaurata la connessione TCP, il client e il server possono scambiarsi dati.

IP sorg.: A	IP dest.: B
porta sorg:1041	porta dest.: 80

IP sorg.: A	IP dest.: B
porta sorg:1040	porta dest.: 21

IP sorg.: C	IP dest.: B
porta sorg:1090	porta dest.: 110



Multiplexing e demultiplexing con TCP

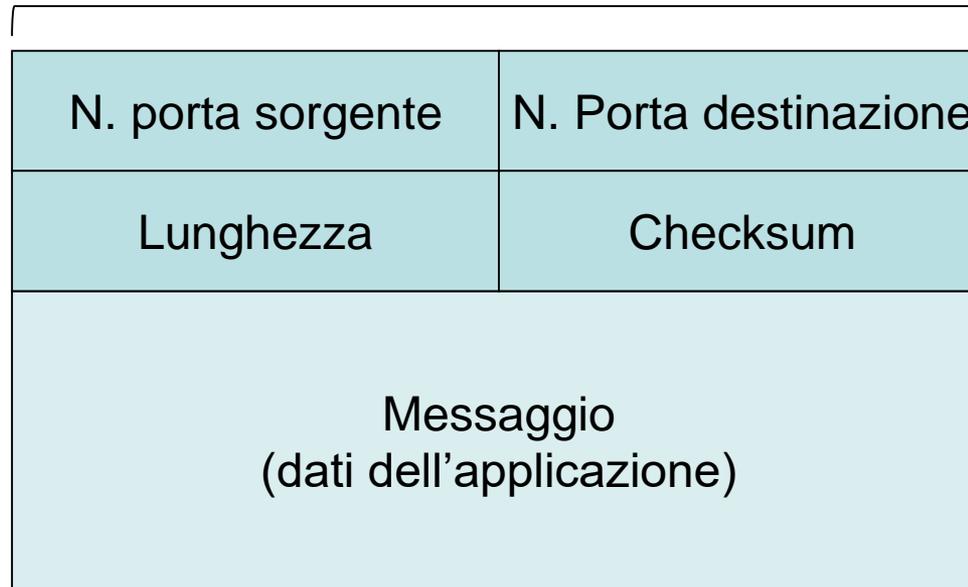
UDP

- L'UDP è un protocollo di trasporto che svolge solo le funzioni di multiplexing/demultiplexing e una semplice verifica degli errori.

Struttura del segmento UDP

- L'intestazione dell'UDP è di solo 8 byte, ed è formata di quattro campi, ciascuno di due byte:
 - Numero di **porta sorgente**
 - Numero di **porta destinazione**
 - Il campo **lunghezza** specifica la lunghezza del segmento UDP (intestazione più dati) in byte.
 - La **checksum** è usata per controllare se si sono verificati errori nel segmento nella trasmissione.
 - Il campo **dati** contiene i dati dell'applicazione (messaggio).

32 bit



Struttura del segmento UDP.

Checksum di UDP

- La checksum di UDP è usata per determinare se i bit nel segmento UDP hanno subito errori nella trasmissione.
- L'UDP nel lato mittente calcola il **complemento a 1** della **somma di tutte le parole a 16 bit** del segmento e di alcuni campi dell'intestazione IP, e l'eventuale riporto finale, del bit più significativo, viene sommato al bit meno significativo (primo bit). Il risultato è inserito nel **campo checksum** del segmento.
- Per esempio, supponiamo di avere le seguenti tre parole di 16 bit:

0110011001100110

0101010101010101

1000111100001111

- La somma delle prime due di queste parole di 16 bit è

```

0110011001100110
0101010101010101
-----
1011101110111011

```

Aggiungendo la terza parola la somma dà

```

1011101110111011
1000111100001111
-----
0100101011001010

```

checksum

1 Complemento a 1

0100101011001011 \longrightarrow 1011010100110100

- Il complemento a 1 si ottiene invertendo tutti gli **0** in **1** e viceversa gli **1** in **0**. Quindi il complemento a 1 della somma 0100101011001011 è 1011010100110100, che diventa la checksum.

- Al destinatario arrivano tutte le parole a 16 bit, inclusa la checksum. Se nel pacchetto arriva senza errori, la somma calcolata al ricevente deve essere **1111111111111111**. Se uno o più bit vale zero, significa che il pacchetto contiene degli errori.