Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2017-18

Pietro Frasca

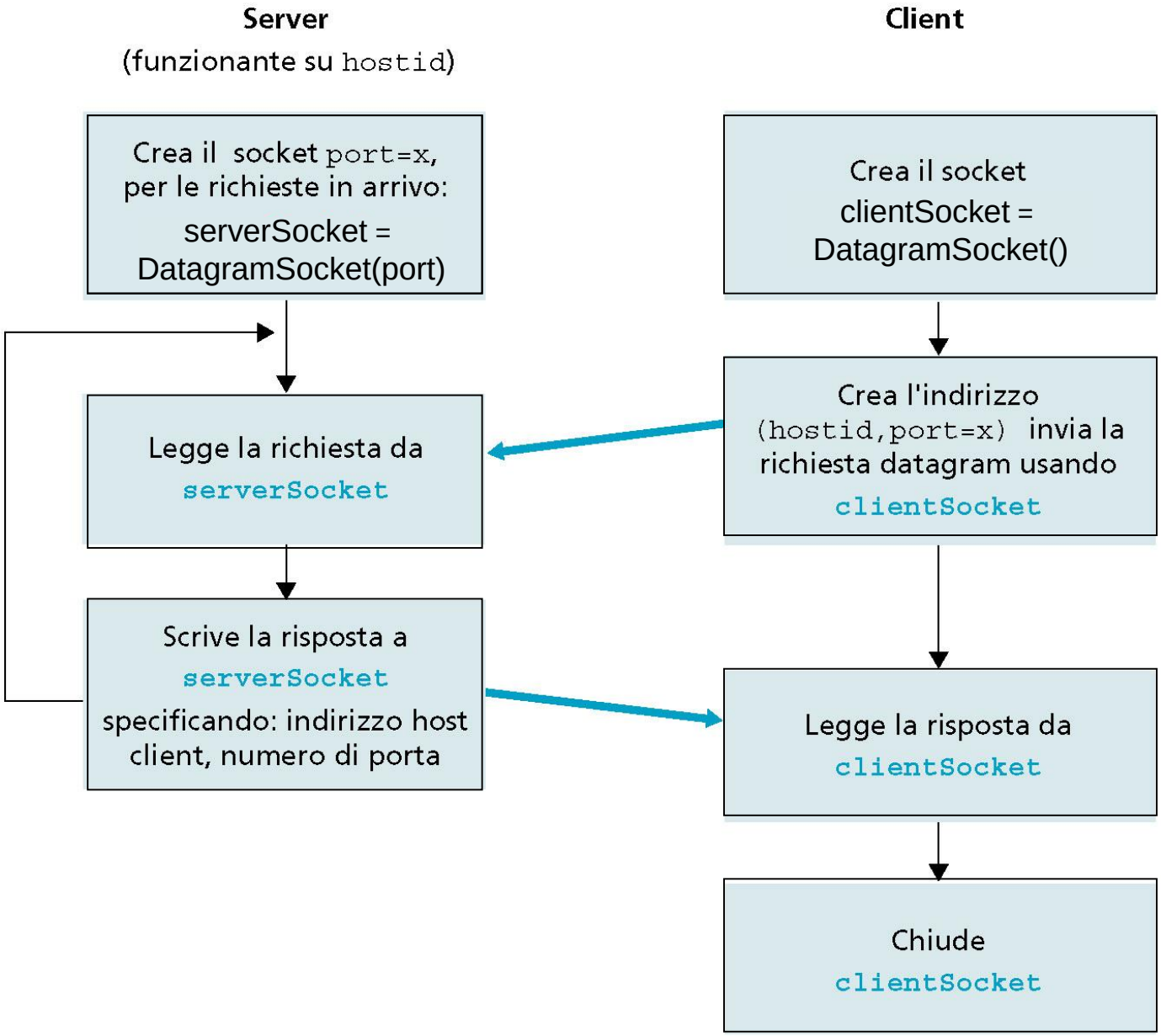
Parte II: Reti di calcolatori

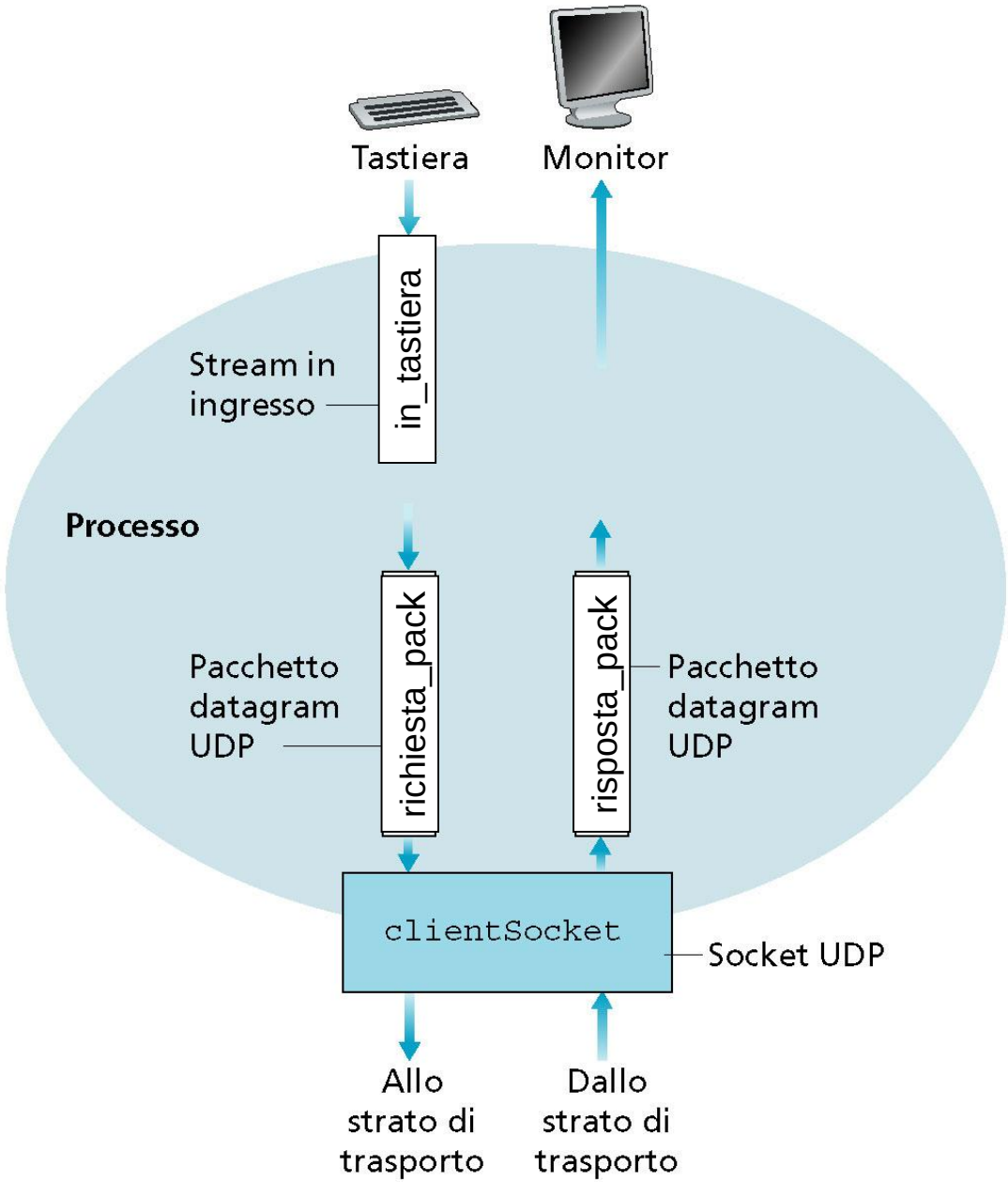
Lezione 13 (37)

Giovedì 26-04-2018

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.


Server
(funzionante su hostid)
Client
Crea il socket port=x, per le richieste in arrivo:
serverSocket = DatagramSocket(port)
Crea il socket
clientSocket = DatagramSocket()
Crea l'indirizzo (hostid,port=x) invia la richiesta datagram usando
clientSocket
Legge la richiesta da
serverSocket
Scrive la risposta a
serverSocket
specificando: indirizzo host client, numero di porta
Legge la risposta da
clientSocket
Chiude
clientSocket


Tastiera
Monitor
in_tastiera
Stream in ingresso
Processo
richiesta_pack
risposta_pack
Pacchetto datagram UDP
Pacchetto datagram UDP
clientSocket
Socket UDP
Allo strato di trasporto
Dallo strato di trasporto

```
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. Il 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.getByName(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 vuole inviare al server mediante il 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`** prende il pacchetto appena costruito e lo spedisce.

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

  il client crea un oggetto `risposta_pack` di tipo `DatagramPacket`, per contenere il pacchetto 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.

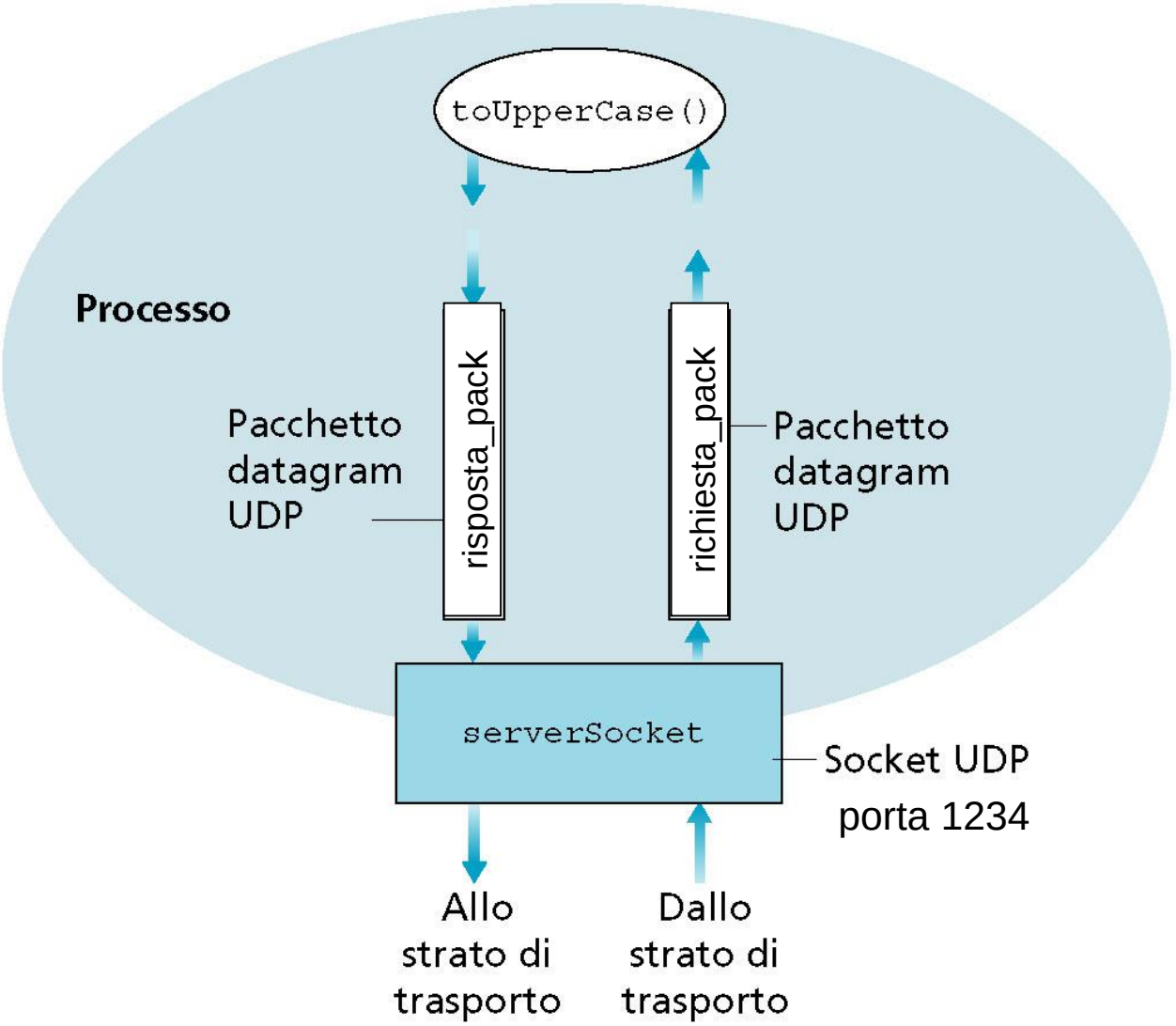
- `String risposta = new String(risposta_pack.getData());`

  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 il socket.


toUpperCase()
Processo
risposta_pack
richiesta_pack
Pacchetto datagram UDP
Pacchetto datagram UDP
serverSocket
Socket UDP porta 1234
Allo strato di trasporto
Dallo strato di trasporto

```
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 un socket, come mostrato in figura.
- Il socket serverSocket è un oggetto della classe DatagramSocket, come il 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 questo socket. Poiché l'UDP è senza connessione, non serve un nuovo 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.

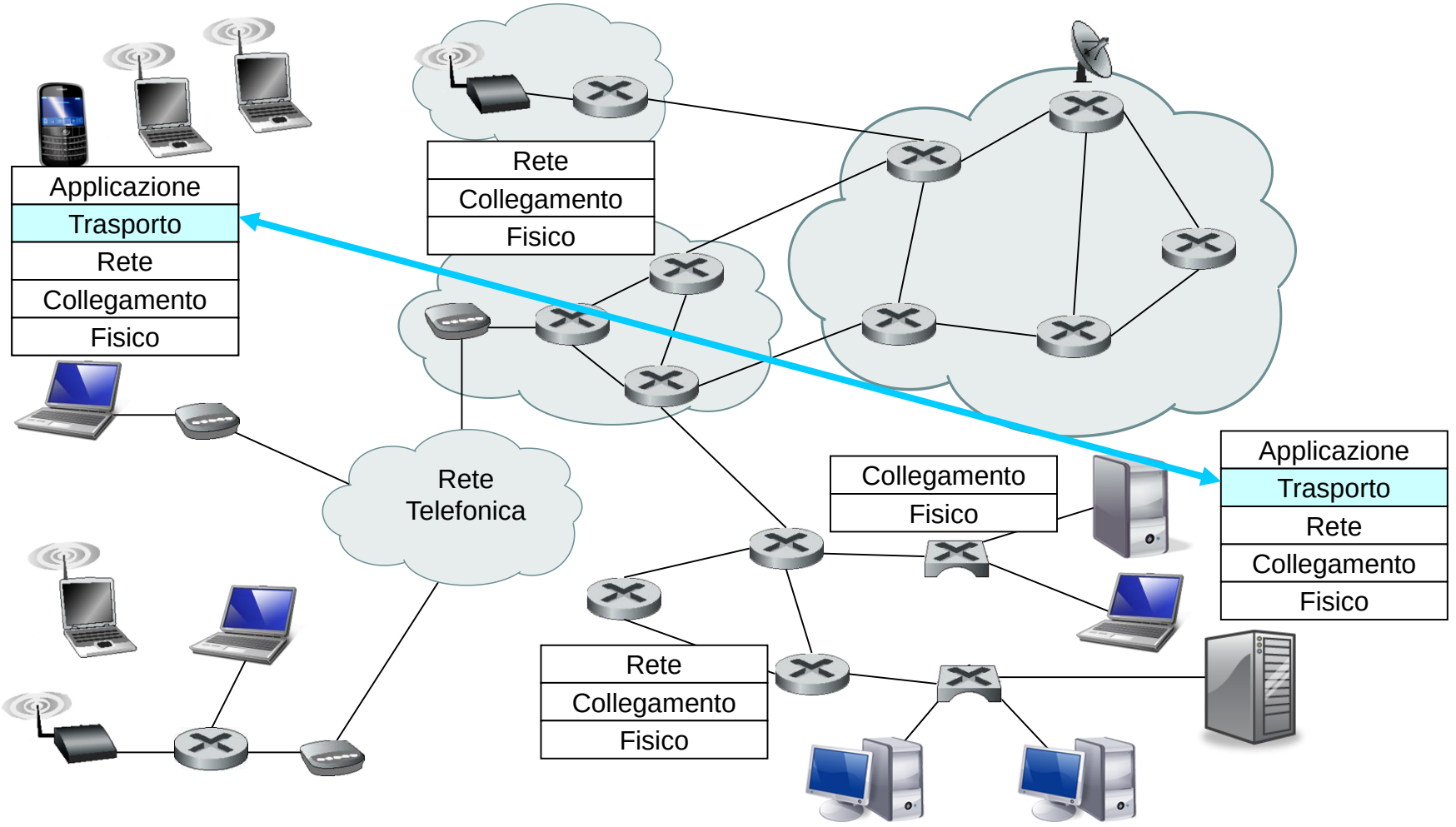
- ```
  String richiesta = new
      String(richiesta_pack.getData));
  InetAddress indirizzoIPclient =
      richiesta_pack.getAddress();
  int portaClient = richiesta_pack.getPort();
  ```

  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.
  ```

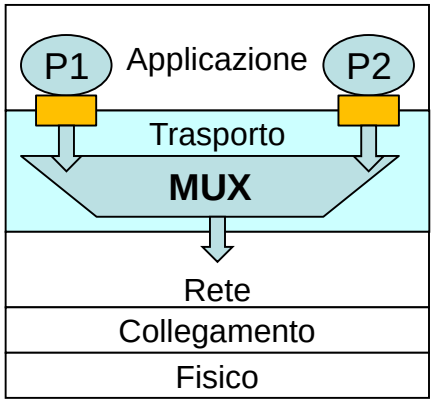
Lo strato di trasporto

- 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).*

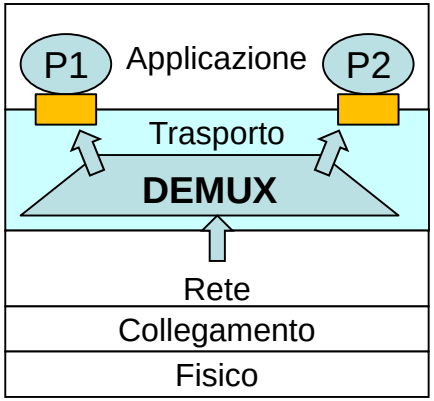

Applicazione
Trasporto
Rete
Collegamento
Fisico
Rete
Collegamento
Fisico
Rete
Telefonica
Collegamento
Fisico
Applicazione
Trasporto
Rete
Collegamento
Fisico
Rete
Collegamento
Fisico

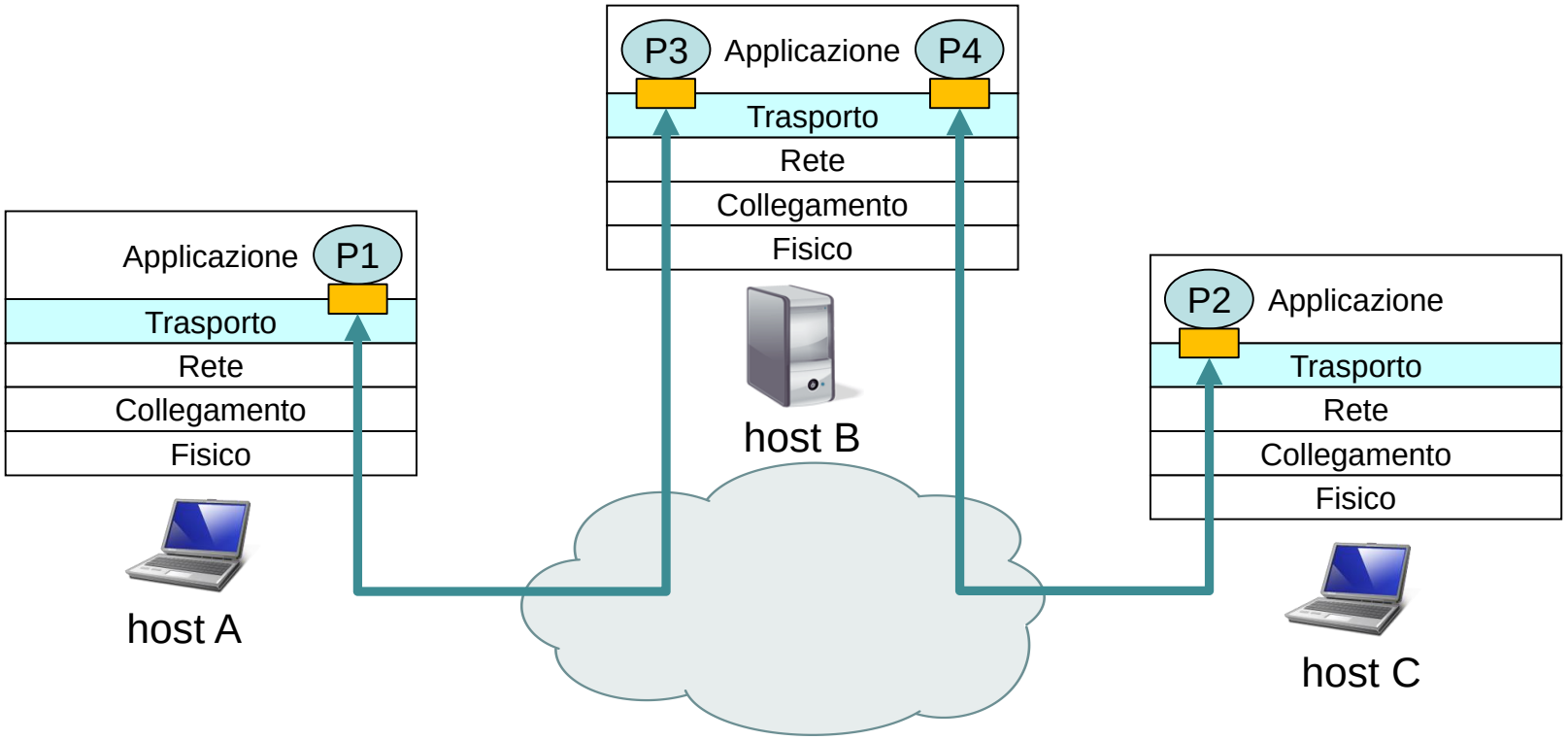
Multiplexing e demultiplexing

- Poiché, allo stesso tempo, in un host possono essere attivi più processi di rete e ciascuno può utilizzare uno 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.

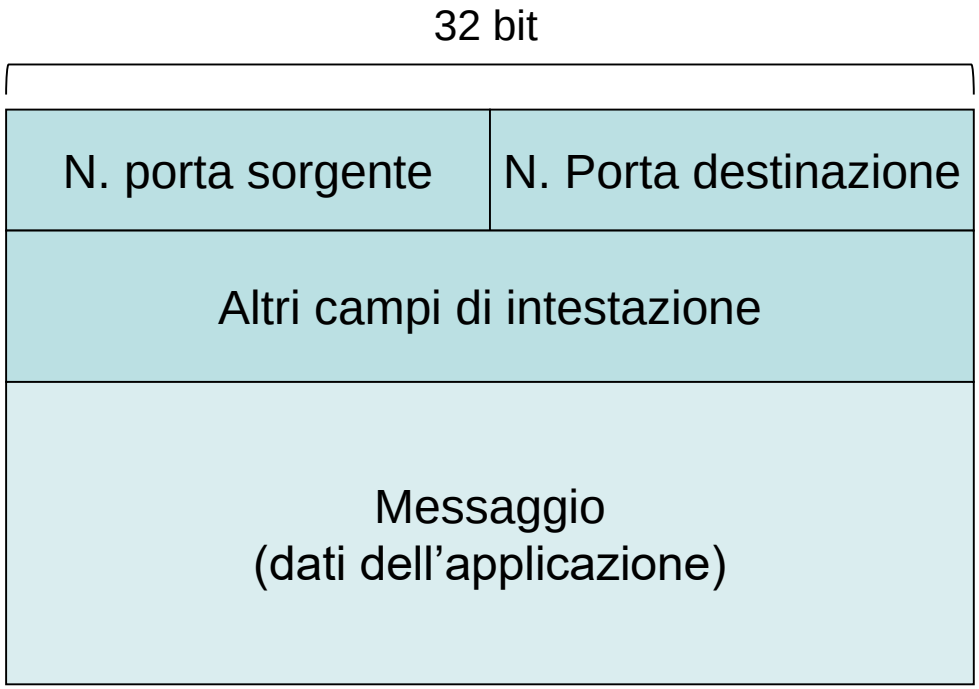




Multiplexing e demultiplexing dello strato di trasporto

- Nella figura, lo strato di trasporto nell'host B deve demultiplare 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 un socket UDP creando un oggetto della classe DatagramSocket:

  DatagramSocket mioSocket = new DatagramSocket();

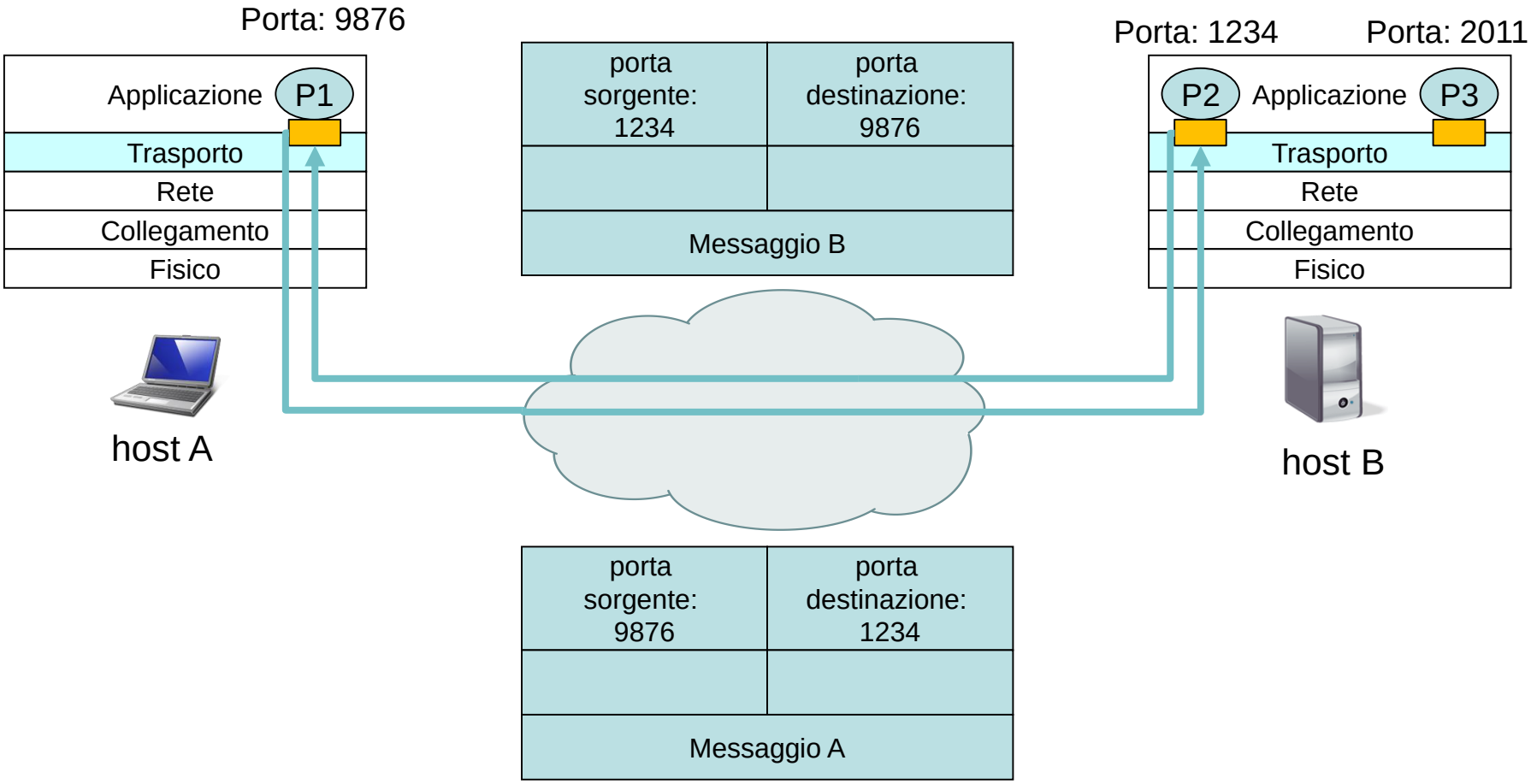
- In questo caso, non passando al costruttore di **DatagramSocket()** un numero di porta, UDP assegna automaticamente un numero di porta al socket, nel intervallo da 1024 a 65535, che in quel momento non è assegnato ad altre porte UDP nell'host.
- In alternativa, si potrebbe creare un socket assegnando un numero di porta specifico (nel nostro esempio, 1234) al 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, un socket TCP è identificato 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 diversi socket.
- Per chiarire questo punto riprendiamo l'esempio del server TCP che abbiamo realizzato.
- L'applicazione server TCP crea un "***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:

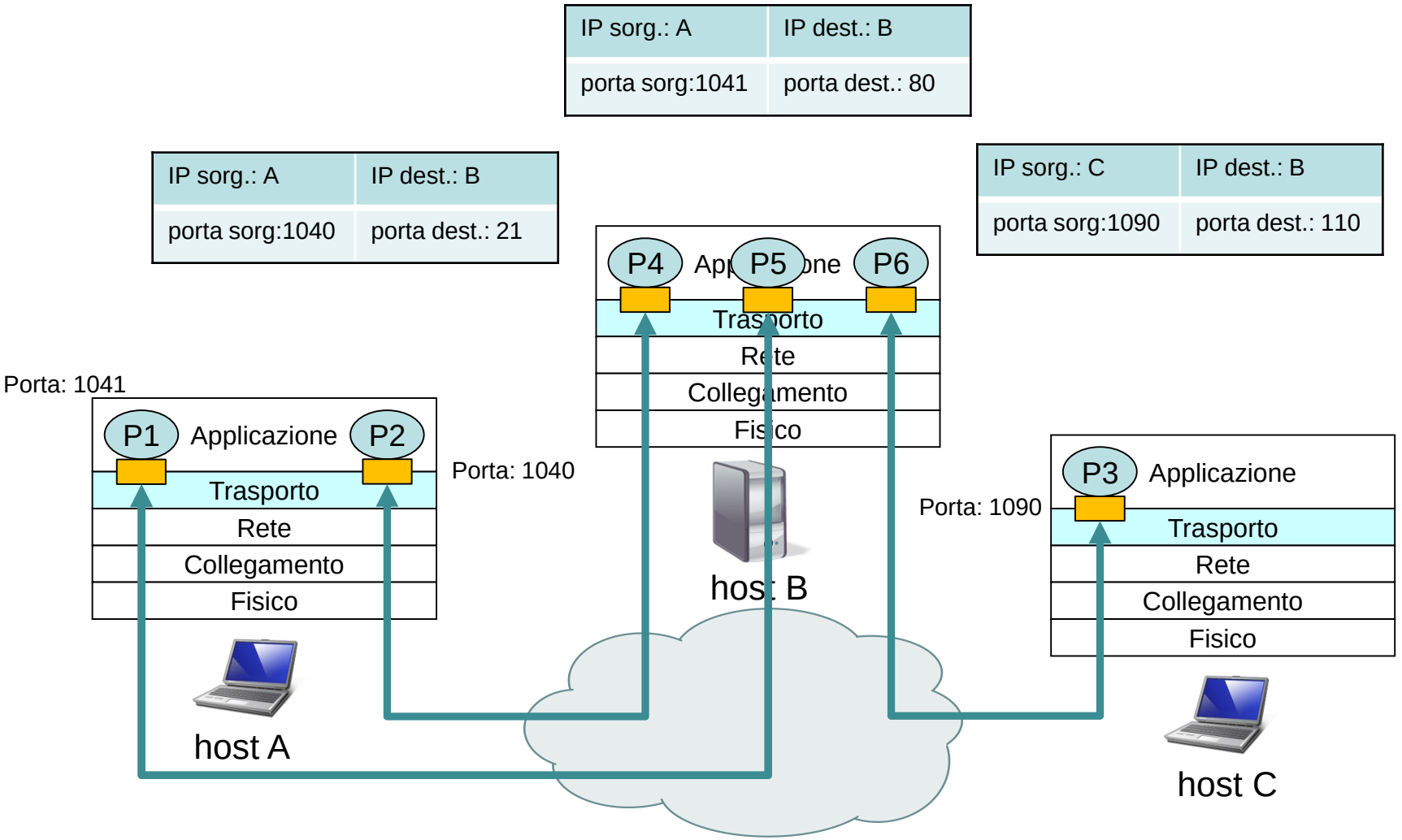
  Socket clientSocket = new Socket (“nomeServer", 1234) ;

- 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 connectionSocket = ascoltoSocket.accept();

- Inoltre, il TCP lato server associa i seguenti quattro valori del segmento di richiesta di connessione al socket di connessione appena creato:
  - **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 di sorgente, la porta di destinazione e l'indirizzo IP di destinazione uguale a questi quattro valori saranno indirizzati a questo socket.
- Una volta instaurata la connessione TCP, il client e il server possono scambiarsi dati.



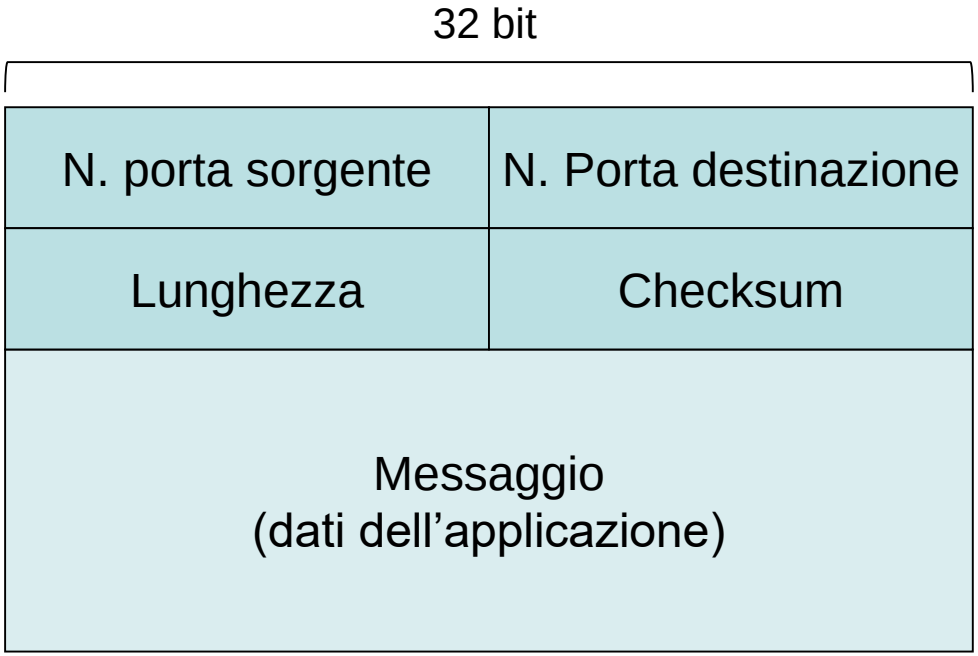
Multiplexing e demultiplexing con TCP

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).



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
                 1
------------------
  0100101011001011
```

Complemento a 1 →

checksum

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.

Trasporto orientato alla connessione: TCP

- Per fornire un trasporto affidabile dei dati, il TCP utilizza molti algoritmi relativi alla ritrasmissione di segmenti, riscontri cumulativi, rilevazione degli errori, timer e campi di intestazione per i numeri di sequenza e numeri di riscontro.

La connessione TCP

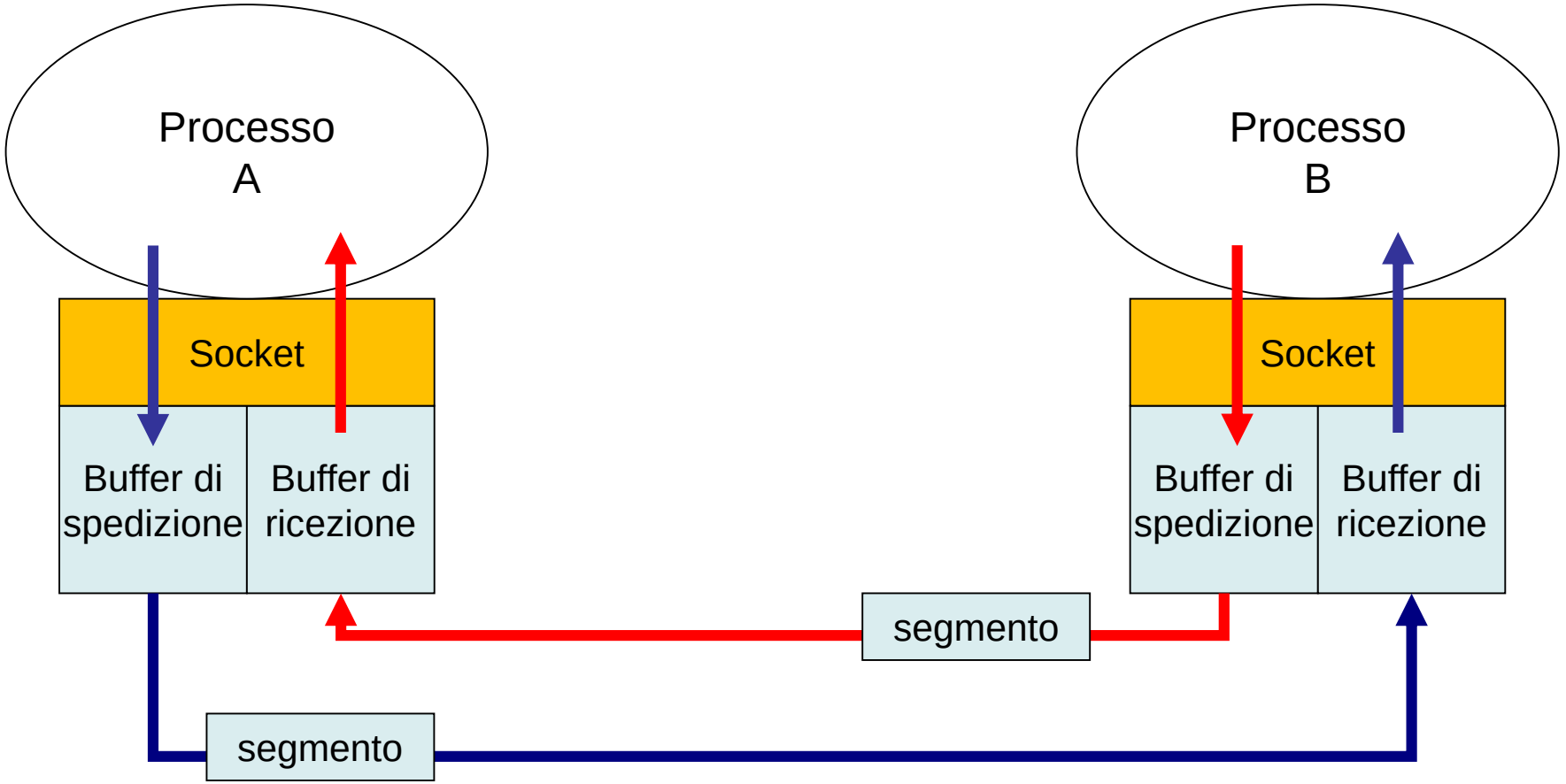
- Col TCP prima che due processi possano trasmettere dati, devono inizialmente eseguire una procedura di handshake. In questa fase, entrambi i lati TCP inizializzeranno diverse "**variabili di stato**" associate alla connessione stessa.
- Una connessione TCP consente un trasferimento dei dati **full duplex** cioè permette di inviare i dati contemporaneamente nelle due direzioni.
- Una connessione TCP è di tipo **unicast punto-punto**, cioè tra un singolo mittente e un singolo destinatario.

- In Java la connessione TCP si realizza creando nel lato client un oggetto della classe Socket:

  Socket clientSocket = new Socket("hostname", numeroPorta);

- Poiché tre segmenti speciali sono scambiati tra i due host, spesso questa procedura è chiamata ***handshake a tre vie***.
- In questa fase,
  - **il client invia uno speciale segmento TCP**;
  - **il server risponde con un secondo segmento speciale TCP**
  - **alla fine il client risponde ancora con un terzo segmento speciale.**
- I primi due segmenti non portano dati dell'applicazione (***carico utile, payload***), mentre il terzo segmento può trasportare dati dell'applicazione.
- Una volta stabilita la connessione TCP, i due processi client e server si possono scambiare dati.

- Consideriamo la trasmissione di dati dal client al server.
- Durante la fase di handshake, il TCP, sia nel lato client che nel lato server, crea un **buffer di spedizione** (send buffer) e un **buffer di ricezione** (receive buffer).
- Il processo client passa uno stream di dati attraverso il socket. Una volta passati al socket, i dati sono gestiti dal TCP del client. Il TCP pone questi dati nel buffer di spedizione. Di tanto in tanto il TCP invierà blocchi di dati prelevandoli dal buffer di spedizione.
- La dimensione massima di dati che può trasportare un segmento è determinato dalla variabile **MSS** (***Maximum Segment Size* - dimensione massima del segmento**)



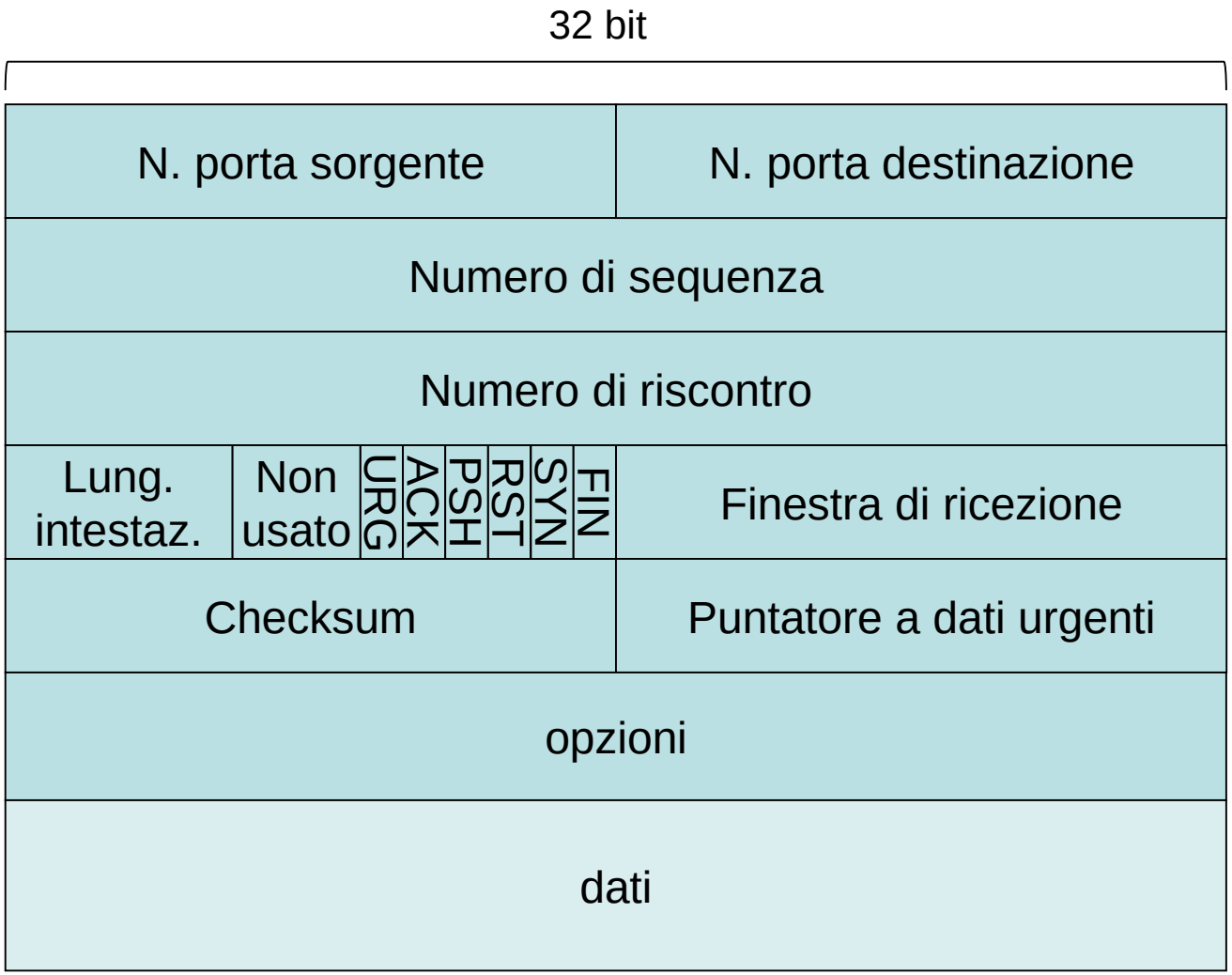
Buffer di invio e ricezione del TCP.

- Il valore di MSS dipende dall'implementazione del TCP e spesso può essere configurato; valori tipici sono, 512, 536 e 1460 byte. Queste dimensioni di segmenti sono scelte soprattutto per evitare la **frammentazione IP**, che esamineremo in seguito.
- Più precisamente, l'MSS è la **massima quantità dei dati dell'applicazione** presente nel segmento, non la massima dimensione del segmento TCP.
- Quando il TCP invia un file di grandi dimensioni, come per esempio un file multimediale, lo **suddivide in parti** di dimensione **MSS** byte (ad eccezione dell'ultima parte, che generalmente ha dimensione inferiore a MSS).
- Tuttavia, le applicazioni interattive generalmente trasmettono parti di dati più piccole di MSS byte. Per esempio, Telnet (o ssh), può avere il campo dati nel segmento di un solo byte. Poiché **l'intestazione TCP tipica è di 20 byte**, la dimensione dei segmenti inviati mediante Telnet può essere di soli **21 byte**.

- Il TCP aggiunge ai dati dell'applicazione un'**intestazione TCP**, formando in tal modo i segmenti TCP.
- I segmenti sono passati allo strato di rete, dove sono incapsulati separatamente nei datagram IP dello strato di rete.
- i segmenti TCP, quando arrivano a destinazione, sono posti nel **buffer di ricezione** del lato ricevente, come mostrato nella figura precedente. L'applicazione legge il flusso di dati da questo buffer.
- Ciascun lato della connessione ha i suoi propri buffer di spedizione e di ricezione.

Struttura del segmento TCP

- La figura seguente mostra l'intestazione del segmento TCP.



- Come per UDP, i **numeri di porta sorgente** e di **destinazione**, sono usati per la multiplazione e demultiplazione.
- **numero di sequenza** e **numero di riscontro** sono usati rispettivamente da mittente e destinatario per implementare un servizio di trasferimento affidabile dei dati.
- **lunghezza intestazione** (4 bit) specifica la lunghezza dell'intestazione del TCP in parole di 32 bit. L'intestazione TCP può avere lunghezza variabile in dipendenza alla lunghezza del campo opzioni. Generalmente il campo opzioni non è usato, pertanto la lunghezza standard dell'intestazione TCP è di 20 byte.
- **finestra di ricezione** (16 bit) è usato per il **controllo del flusso di dati**. Indica il numero di byte che il destinatario è in grado di ricevere.
- Il campo **checksum** è simile a quello dell'UDP.
- ;

- **opzioni** è un campo facoltativo e di lunghezza variabile. Le opzioni più importanti sono negoziabili durante l'instaurazione della connessione, e sono:
  - dimensione massima dei segmenti da spedire (**MSS: Maximum Segment Size**), serve soprattutto per adattare la dimensione del segmento in modo che ogni segmento venga incluso in un datagram IP che non debba essere frammentato
  - Scelta dell'algoritmo di controllo del flusso come ad esempio *selective repeat* invece che *go-back-n.*
- **flag** (6 bit). I bit **RST**, **SYN** e **FIN** sono usati per instaurare e chiudere la connessione.
  - Il bit **ACK** se settato a 1 indica che il valore presente nel campo **numero di riscontro** è valido.
  - I bit PSH e URG sono raramente usati. Il bit **PSH** quando è settato, indica che i dati in arrivo non devono essere bufferizzati ma dovrebbero essere passati immediatamente allo strato superiore. Il bit **URG** se settato a 1 indica che in questo segmento ci sono dati che il mittente ha contrassegnato come "urgenti".