Livello applicazione: P2P, programmazione socket

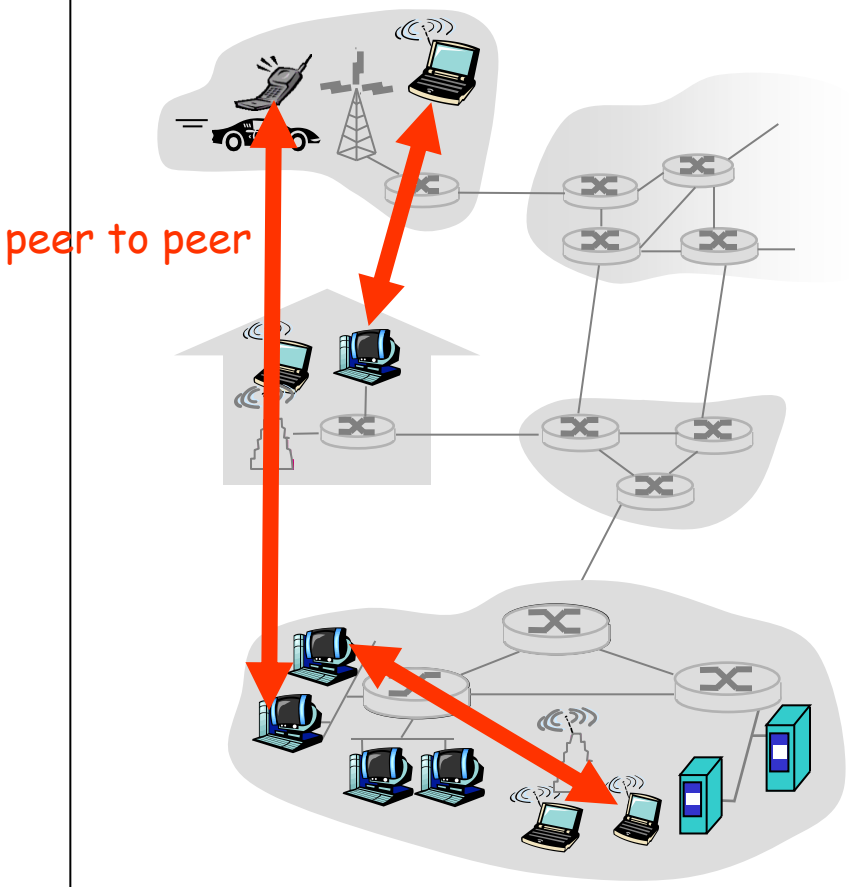
Gaia Maselli

Livello di applicazione

- Condivisione di file P2P
- Programmazione delle socket con TCP
- Programmazione delle socket con UDP

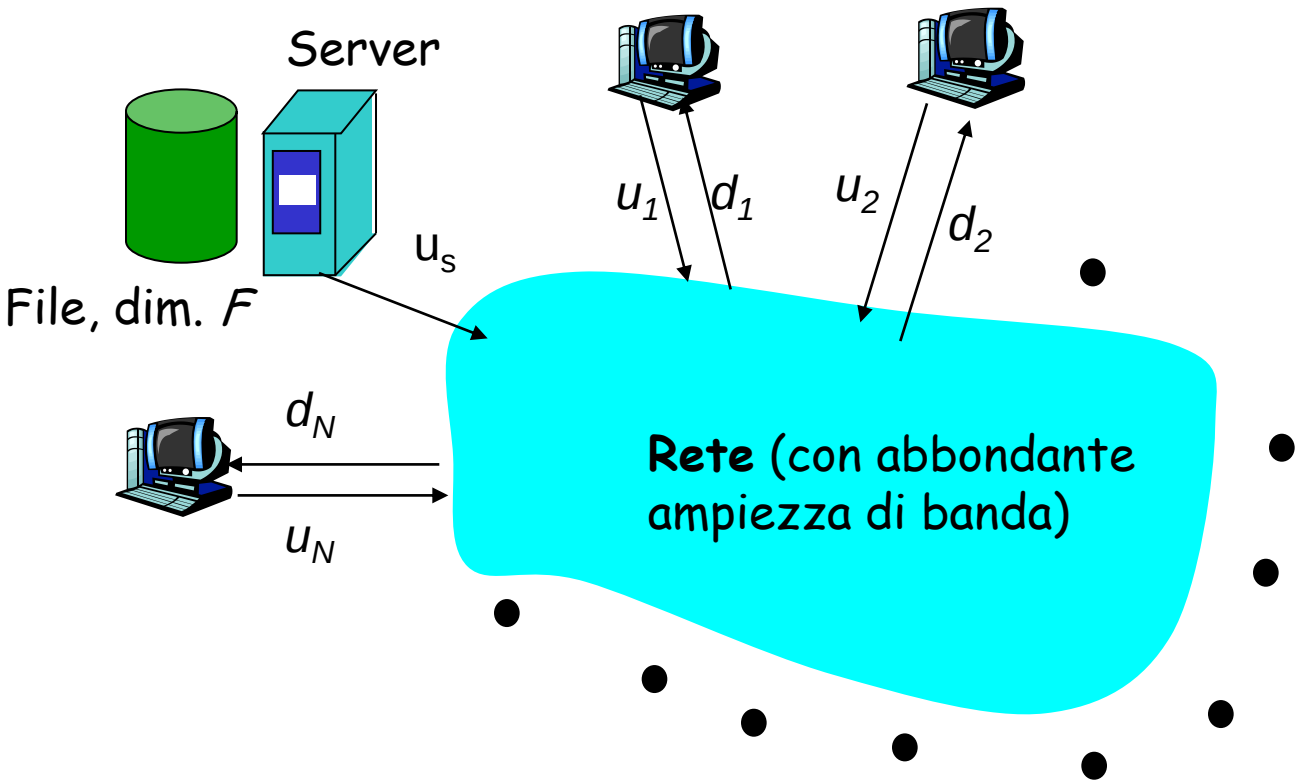
Architettura P2P pura

- non c'è un server sempre attivo
- coppie arbitrarie di host (peer) comunicano direttamente tra loro
- i peer non devono necessariamente essere sempre attivi, e cambiano indirizzo IP
- Tre argomenti chiave:
  - Distribuzione di file
  - Ricerca informazioni
  - Caso di studio: Skype



Distribuzione di file: confronto tra Server-Client e P2P

Domanda : Quanto tempo ci vuole per distribuire file da un server a *N peer (tempo di distribuzione)*?



u_s: frequenza di upload del collegamento di accesso del server

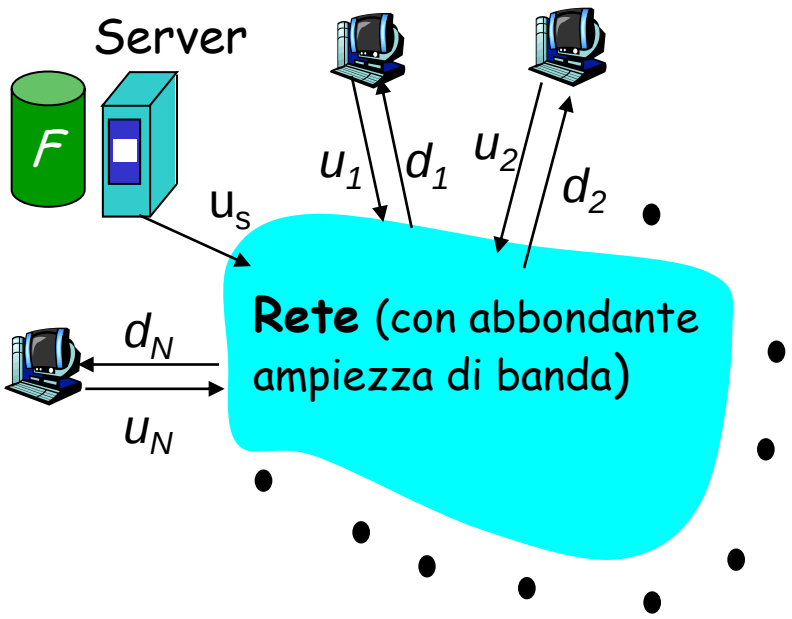
u_i: frequenza di upload del collegamento di accesso dell'*i*-esimo peer

d_i: frequenza di download del collegamento di accesso dell'*i*-esimo peer

Distribuzione di file: server-client

Distribuzione client-server: il server deve inviare una copia del file a ciascun peer

- Il server invia in sequenza *N* copie:
  - *Tempo = NF/u_s*
- Il client impiega il tempo F/d_i per scaricare



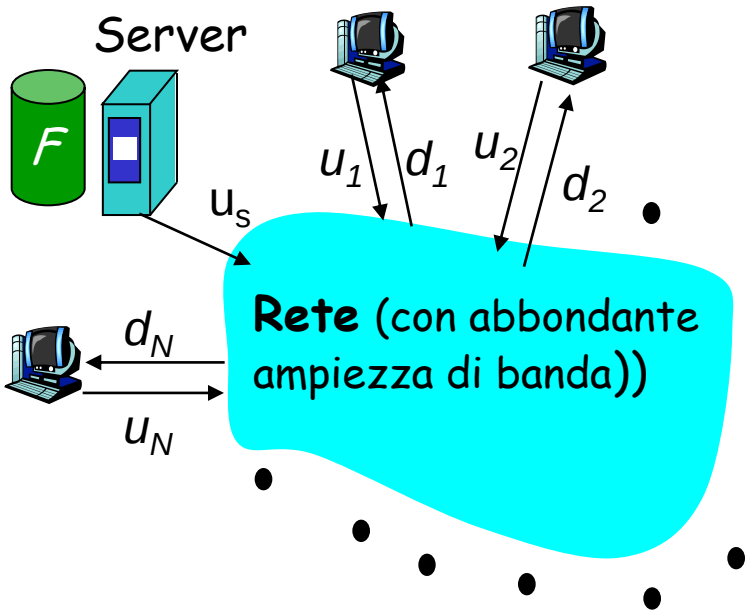
Tempo per distribuire *F* a *N* client usando l'approccio client/server

$$= d_{cs} = \max\left\{\frac{NF}{u_s}, \frac{F}{d_{\min}}\right\}$$

aumenta linearmente con *N* peer

Distribuzione di file: P2P

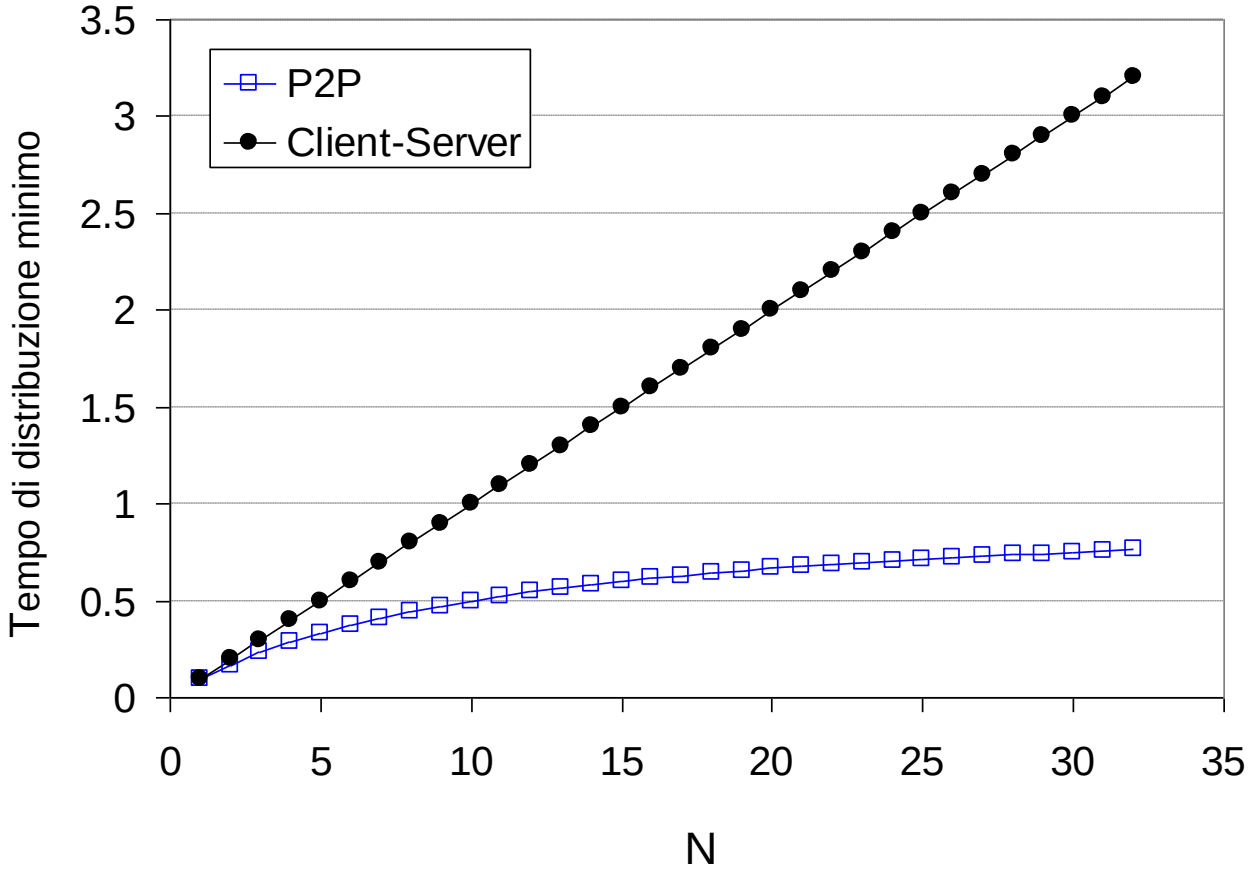
- il server deve inviare una copia (all'inizio solo il server dispone del file) nel tempo F/u_s
- il client impiega il tempo F/d_i per il download
- Devono essere scaricati NF bit
- Il più veloce tasso di upload è: $u_s + \Sigma u_i$



$$d_{P2P} = \max\left\{\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum u_i}\right\}$$

Confronto tra server-client e P2P: un esempio

Tasso di upload dei client $u_i = u$, $F/u = 1$ ora, $u_s = 10u$, $d_{min} \geq u_s$

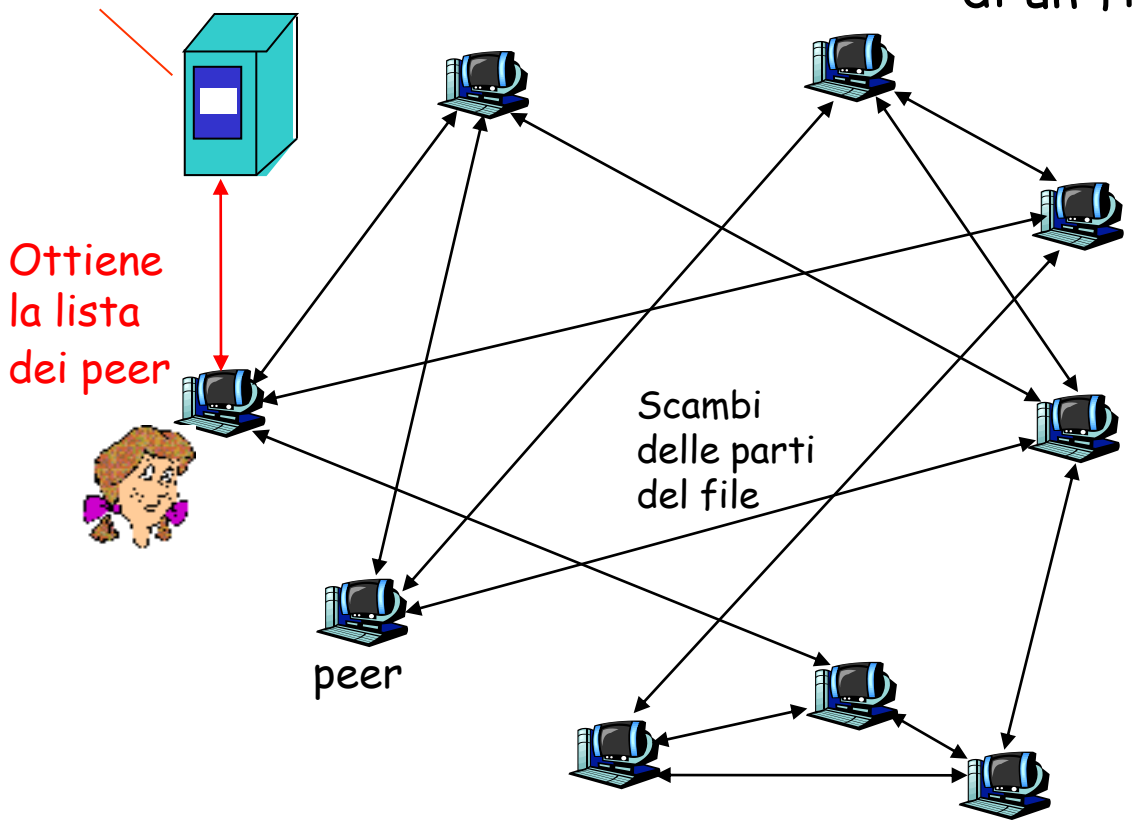


Distribuzione di file: BitTorrent

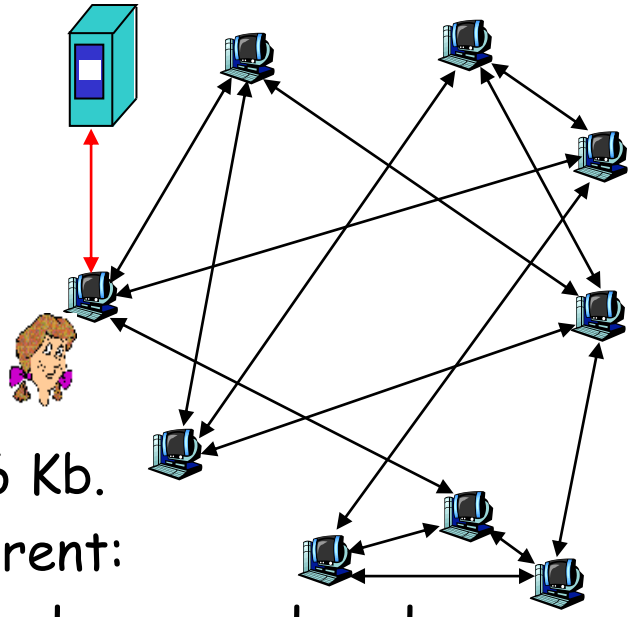
- Distribuzione di file P2P

tracker: tiene traccia dei peer che partecipano

torrent: gruppo di peer che si scambiano parti di un file



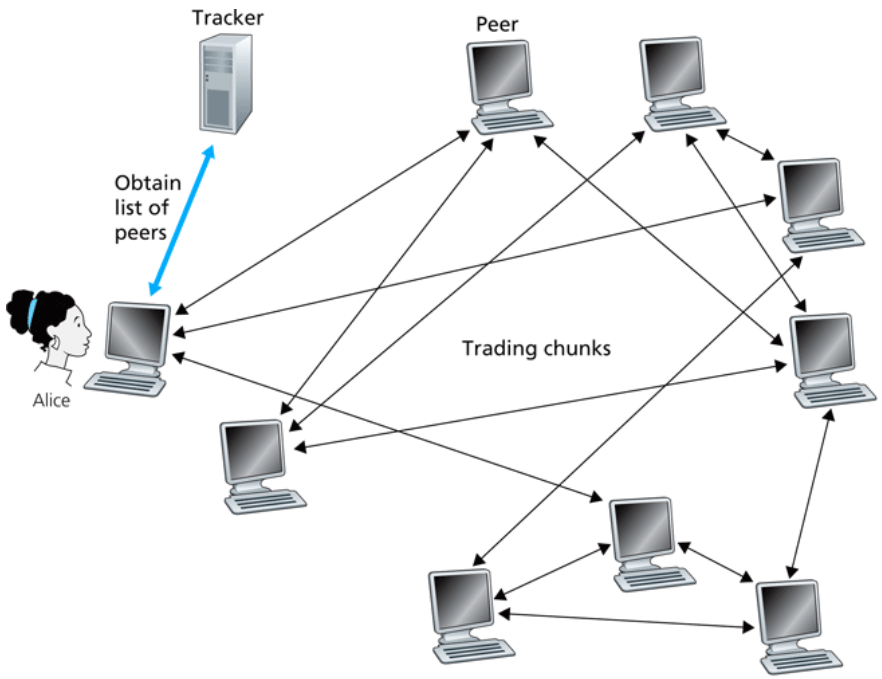
BitTorrent (1)



- Il file viene diviso in parti (*chunk*) da 256 Kb.
- Quando un peer entra a far parte del torrent:
  - non possiede nessuna parte del file, ma le accumula col passare del tempo
  - si registra presso il tracker (e periodicamente lo aggiorna) per avere la lista dei peer, e si collega ad un sottoinsieme di peer vicini ("neighbors")
- Mentre effettua il download, il peer carica le sue parti su altri peer.
- I peer possono entrare e uscire a piacimento dal torrent
- Una volta ottenuto l'intero file, il peer può lasciare il torrent (egoisticamente) o (altruisticamente) rimanere collegato.

BitTorrent (2)

- In un dato istante, peer diversi hanno differenti sottoinsiemi del file
- Peer in grado di inviare file a frequenza compatibili tendono a trovarsi…



Invio di richieste

- periodicamente, un peer (Alice) chiede a ciascun vicino la lista dei chunk che possiede
- Alice invia le richieste per le sue parti mancanti:
  - Adotta la tecnica del *rarest first*

Invio di risposte

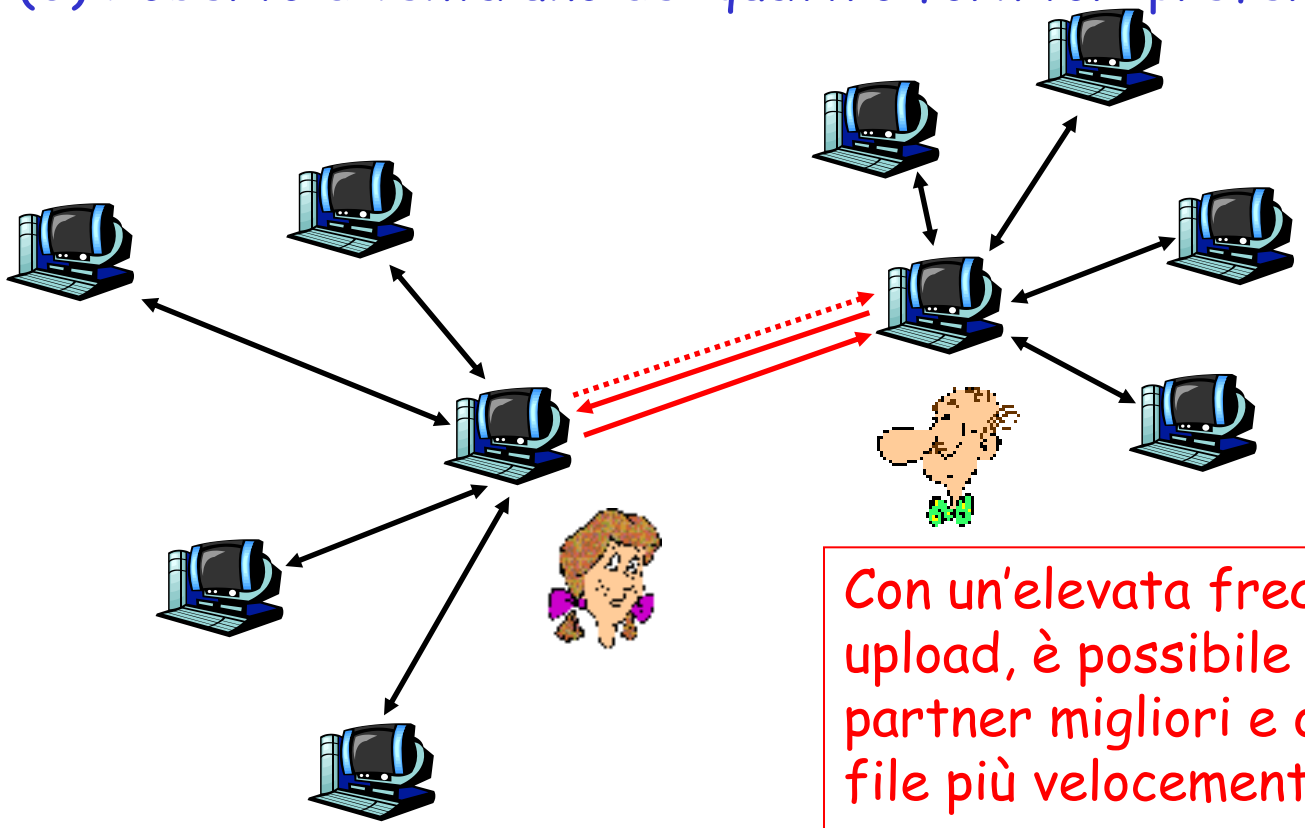
- Algoritmo di trading per decidere a quali richieste di vicini rispondere
- Assegnazione di priorità ai vicini che stanno inviando dati alla frequenza più alta
- Determinazione periodica (ogni 10 secondi) dei 4 peer che inviano alla frequenza maggiore
- Alice invia le sue parti ai 4 vicini
- Ogni 30 secondi seleziona casualmente un altro peer, e inizia a inviargli chunk
  - Il peer appena scelto può entrare a far parte dei top 4
  - A parte i "top 4" e il "nuovo entrato", gli altri peer sono "soffocati", cioè non ricevono nulla.

BitTorrent: occhio per occhio

(1) Alice casualmente sceglie Roberto

(2) Alice diventa uno dei quattro fornitori preferiti di Roberto; Roberto ricambia

(3) Roberto diventa uno dei quattro fornitori preferiti di Alice



L'algoritmo di trading elimina il free-riding (sfruttamento)

Con un'elevata frequenza di upload, è possibile trovare i partner migliori e ottenere il file più velocemente!

P2P: ricerca di informazioni

Indice nei sistemi P2P: corrispondenza tra le informazioni e la loro posizione negli host

File sharing (es. e-mule)

- L'indice tiene traccia dinamicamente della posizione dei file che i peer condividono.
- I peer comunicano all'indice ciò che possiedono.
- I peer consultano l'indice per determinare dove trovare i file.

Messaggistica istantanea

- L'indice crea la corrispondenza tra utenti e posizione.
- Quando l'utente lancia l'applicazione, informa l'indice della sua posizione
- I peer consultano l'indice per determinare l'indirizzo IP dell'utente.

N.B. BitTorrent non fornisce alcuna funzionalità per indicizzare e cercare i file

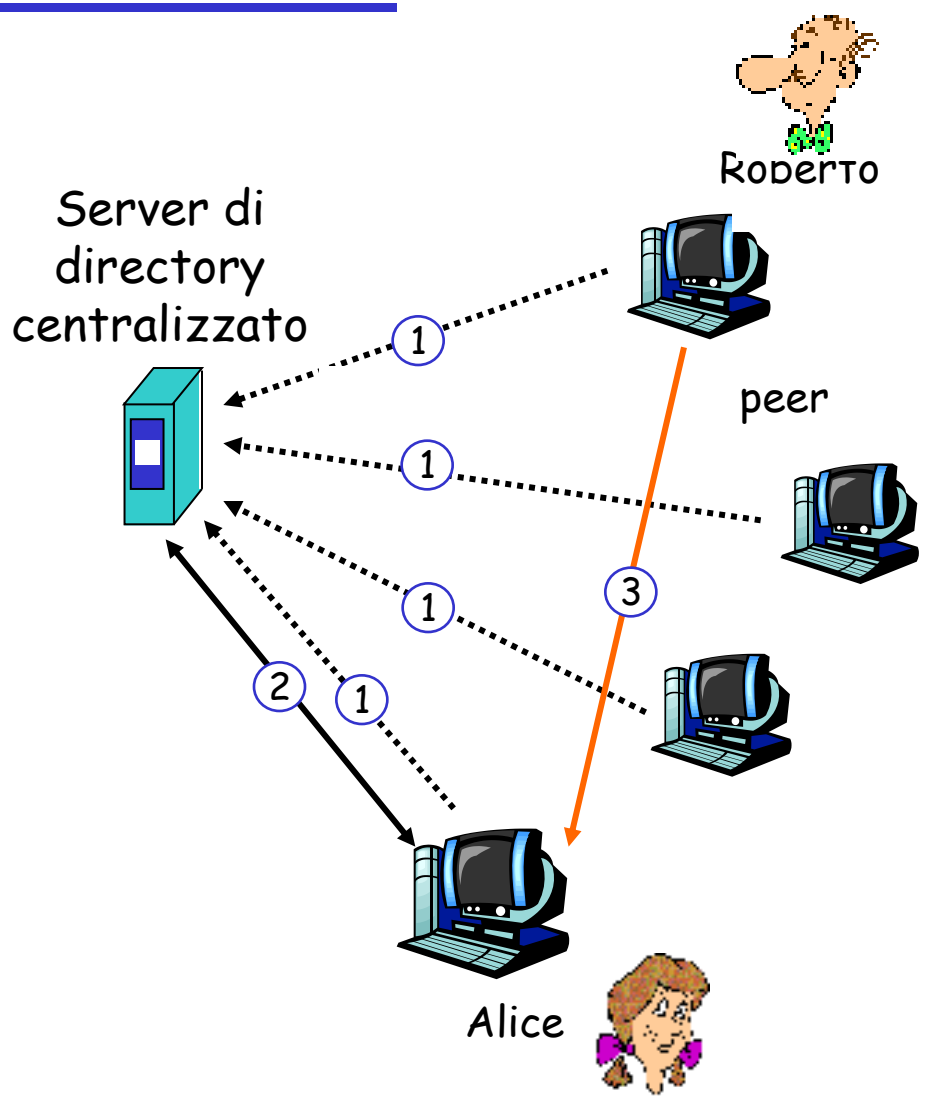
Soluzioni per indicizzazione

1. Directory centralizzata
2. Query flooding
3. Copertura gerarchica

1: directory centralizzata

Progetto originale di "Napster"

1) quando il peer si collega, informa il server centrale:
   - indirizzo IP
   - contenuto
2) Alice cerca la canzone "Hey Jude"
3) Alice richiede il file a Roberto



P2P: problemi con la directory centralizzata

- Unico punto di guasto
- Collo di bottiglia per le prestazioni
- Violazione del diritto d'autore

Il trasferimento dei file è distribuito, ma il processo di localizzazione è fortemente centralizzato

Architettura ibrida

2: Query flooding

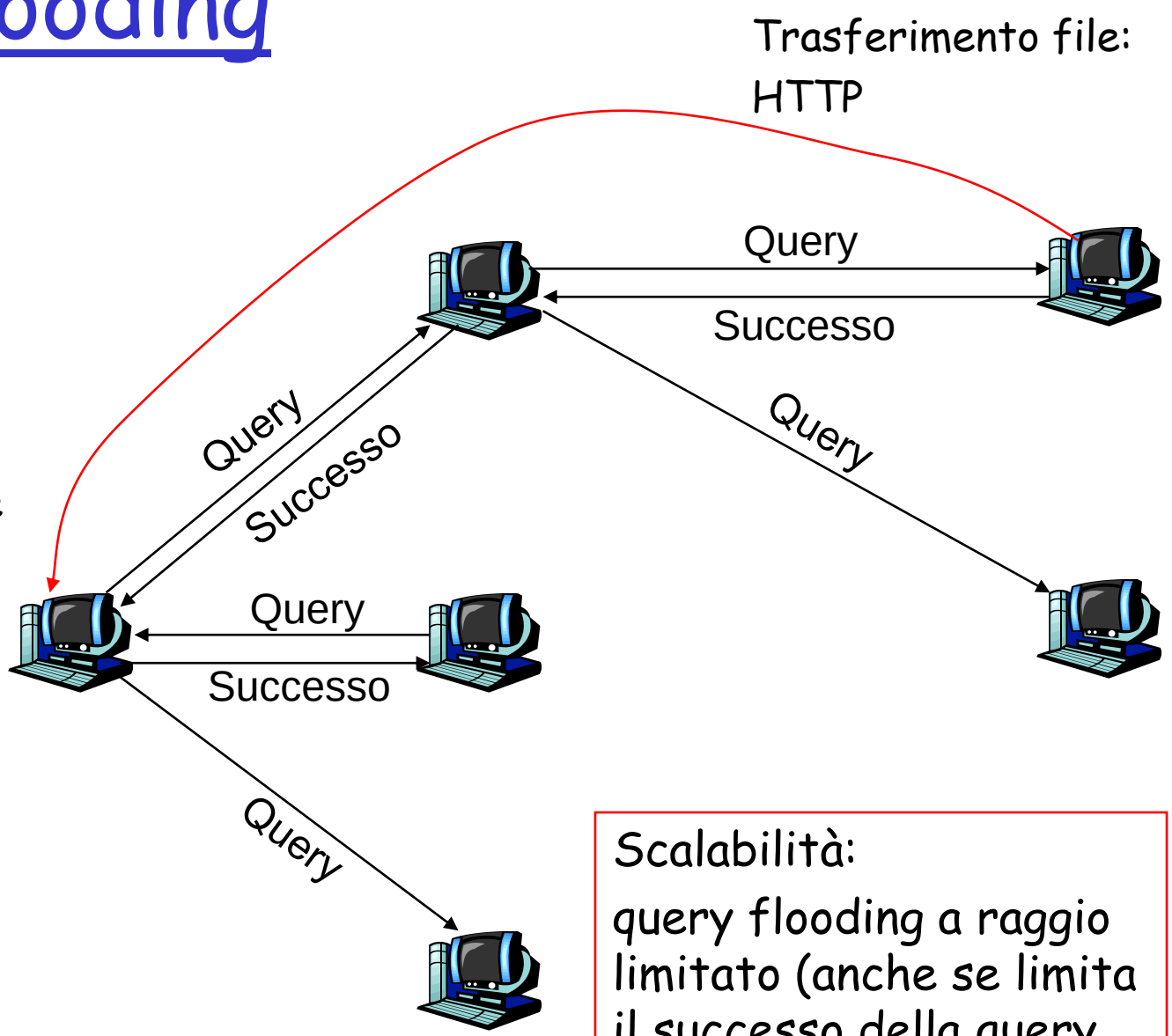
- L'indice è completamente distribuito nella comunità di peer
  - nessun server centrale
- Protocollo di pubblico dominio usato da Gnutella
- Ciascun peer indicizza i file che rende disponibili per la condivisione (e nessun altro)
- I peer formano una rete logica detta di copertura

Rete di copertura (overlay network): grafo

- Arco tra i peer X e Y se c'è una connessione TCP
- Tutti i peer attivi e gli archi formano la rete di copertura
- Un arco è un collegamento virtuale e *non* fisico (es. Brasile-Australia)
- Un dato peer sarà solitamente connesso con meno di 10 peer vicini nella rete di copertura

Query flooding

- Il messaggio di richiesta è trasmesso sulle connessioni TCP esistenti
- Il peer inoltra il messaggio di richiesta
- Quando un peer riceve una richiesta per un file che ha disponibile allora invia un messaggio di query-hit (nome file, dimensione)
- Il messaggio di successo è trasmesso sul percorso inverso
- Alice scopre i peer che hanno il file



Scalabilità:

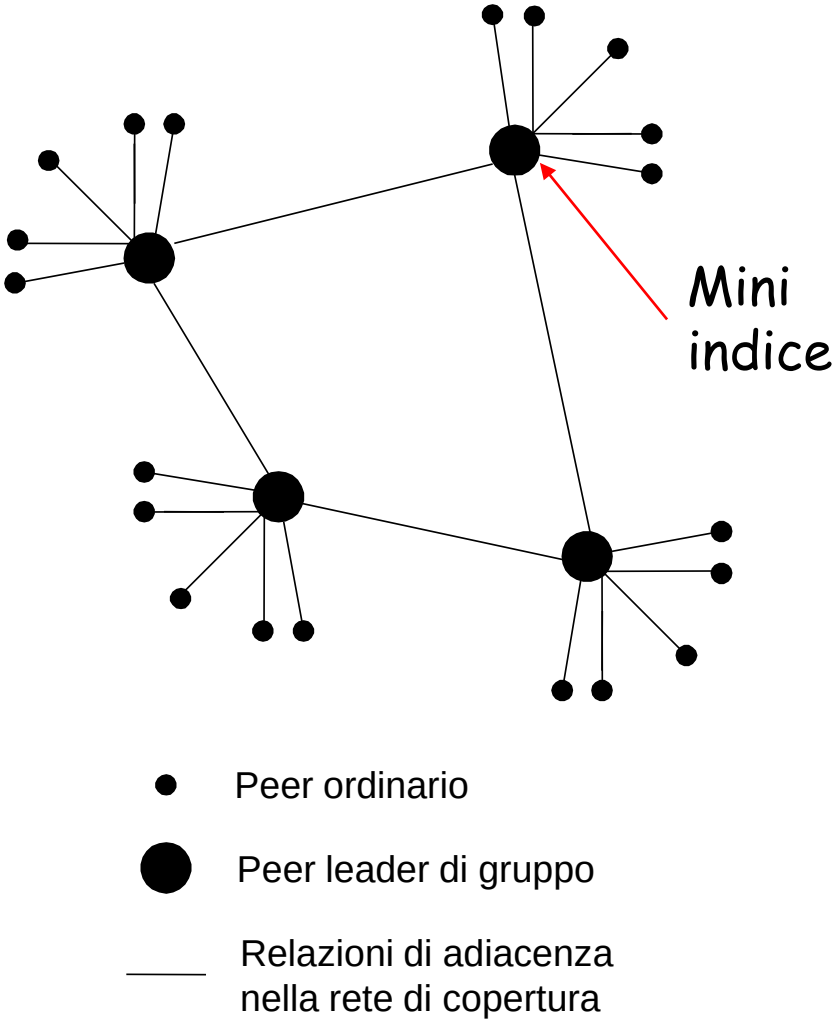
query flooding a raggio limitato (anche se limita il successo della query

Gnutella: unione di peer

1. Per unire il peer X alla rete, bisogna trovare qualche altro peer della rete Gnutella: lista dei peer candidati
2. X tenta in sequenza di impostare una connessione TCP con i peer della lista finché non stabilisce una connessione con un peer Y
3. Flooding: X invia un messaggio Ping a Y; Y inoltra il messaggio Ping ai suoi vicini (che a loro volta lo inviano ai propri vicini…)
4. Tutti i peer che ricevono il messaggio Ping rispondono ad X con un messaggio Pong con il loro IP
5. X riceve molti messaggi Pong. Quindi può impostare delle connessioni TCP addizionali

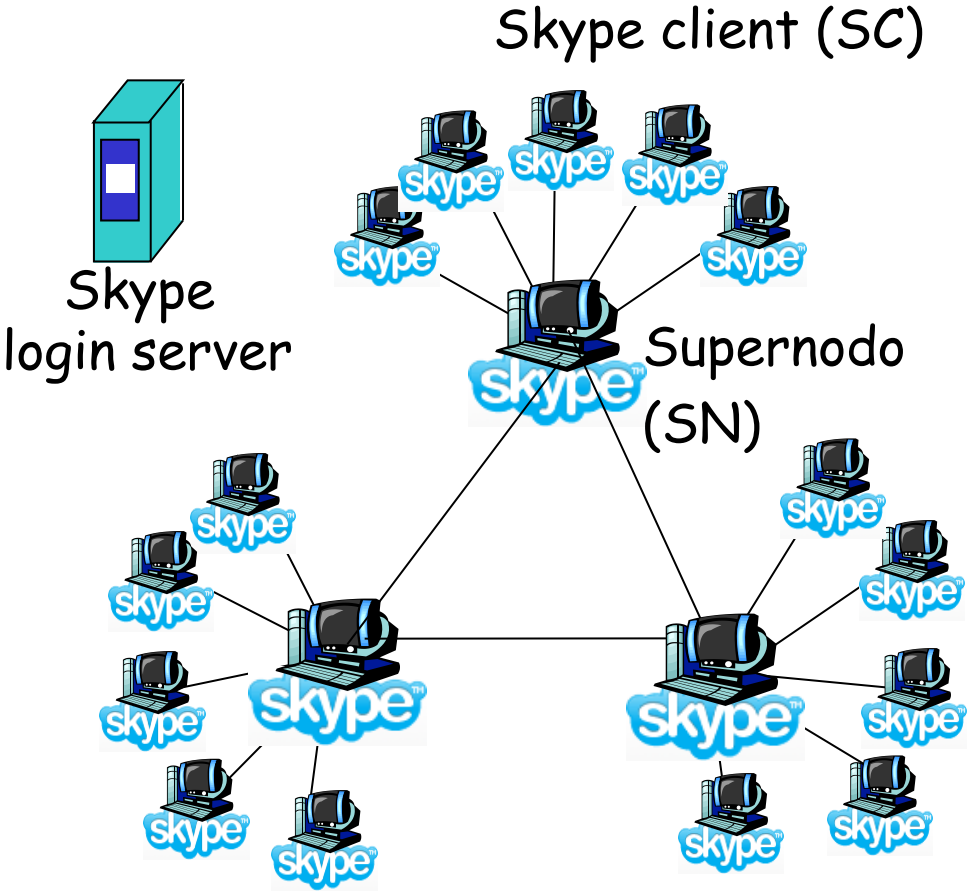
3: Copertura gerarchica

- La copertura gerarchica combina le migliori caratteristiche di indice centralizzato e query flooding
- Ogni peer è un leader di gruppo o è assegnato a un leader di gruppo
  - Connessione TCP tra peer e il suo leader di gruppo
  - Connessioni TCP tra qualche coppia di leader di gruppo
- Il leader di gruppo tiene traccia del contenuto di tutti i suoi figli.
- Kazaa, Morpheus



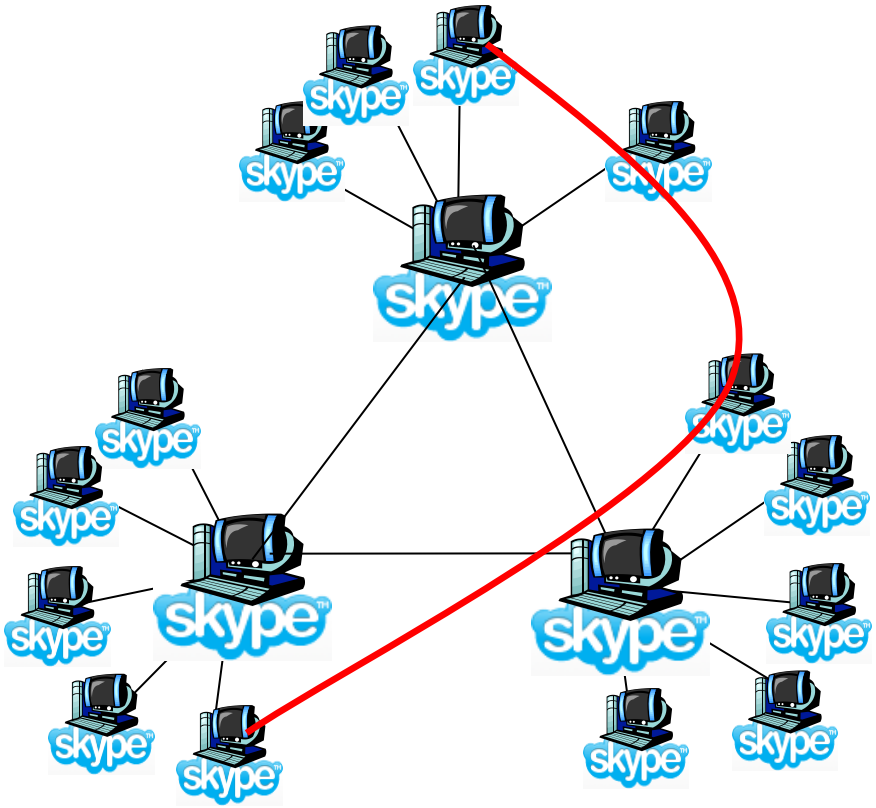
Caso di studio P2P: Skype

- intrinsecamente P2P: coppie di utenti comunicano tra loro
- Protocollo proprietario (dedotto mediante reverse engineering)
- Copertura gerarchica con i supernodi
- L'indice crea corrispondenza tra nomi utente e indirizzi IP



Peer e relay

- Si pone un problema quando sia Alice che Roberto hanno NAT.
  - NAT evita che un host al di fuori della rete domestica crei una connessione con un host all'interno di questa
- Soluzione
  - Usando il supernodo di Alice e Roberto, si sceglie un relay
  - Ciascun peer inizia la sessione con il relay.
  - I peer ora comunicano con NAT attraverso il relay



Capitolo 2: Livello di applicazione

- Applicazioni P2P
- Programmazione delle socket con TCP
- Programmazione delle socket con UDP

Introduzione alle socket

- Applicazioni di rete: coppia di programmi client-server, che risiedono su sistemi diversi
- In esecuzione: processo client e processo server comunicano tramite socket
- Applicazione di rete può implementare
  - Protocollo standard definito (es. in una RFC)
    - Conformità all'RFC (garantita l'interazione)
    - Utilizzo di porte assegnate
    - Sviluppatori differenti e anche solo da un lato
  - Applicazione proprietaria
    - Sviluppo sia del client che del server
    - Utilizzo di porte non assegnate
    - No sviluppatori indipendenti
- Decisione preliminare: TCP o UDP?

Programmazione delle socket

Obiettivo: imparare a costruire un'applicazione client/server che comunica utilizzando le socket

Socket API

- introdotta in BSD4.1 UNIX, nel 1981
- esplicitamente creata, usata, distribuita dalle applicazioni
- paradigma client/server
- due tipi di servizio di trasporto tramite una socket API:
  - datagramma inaffidabile
  - affidabile, orientata ai byte

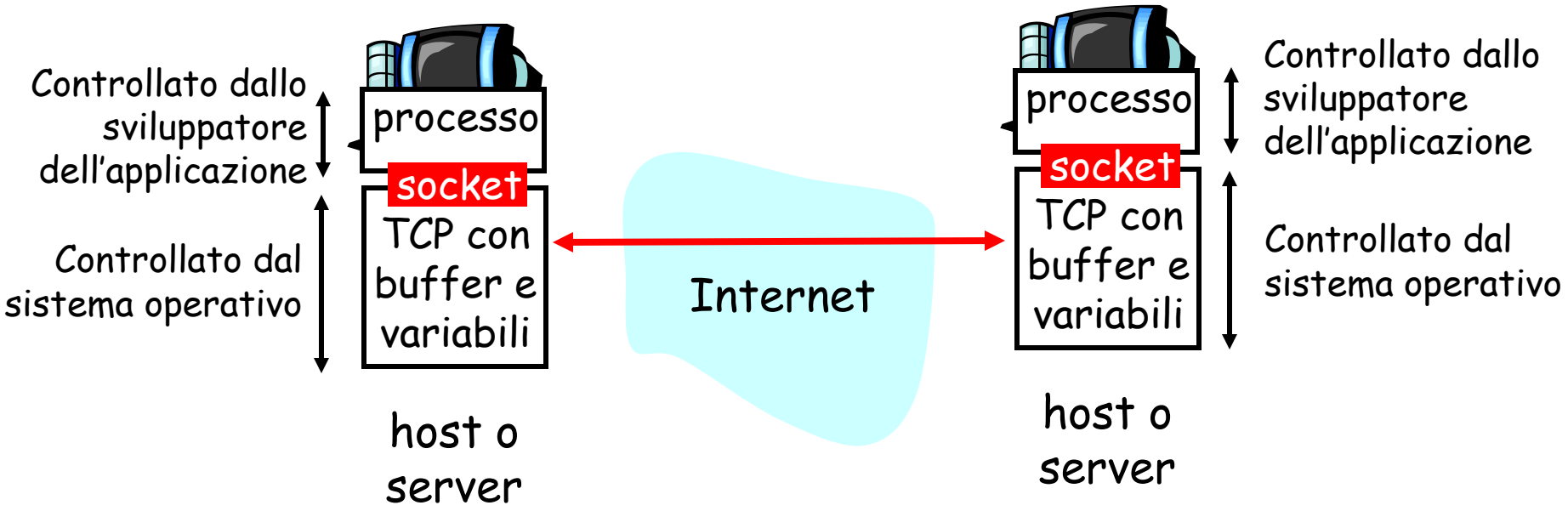
socket

Interfaccia di un *host locale*, *creata dalle applicazioni*, *controllata dal SO* (una "porta") in cui il processo di un'applicazione può inviare e ricevere messaggi al/dal processo di un'altra applicazione

Programmazione delle socket con TCP

Socket: una porta tra il processo di un'applicazione e il protocollo di trasporto end-end (UDP o TCP)

Servizio TCP: trasferimento affidabile di byte da un processo all'altro



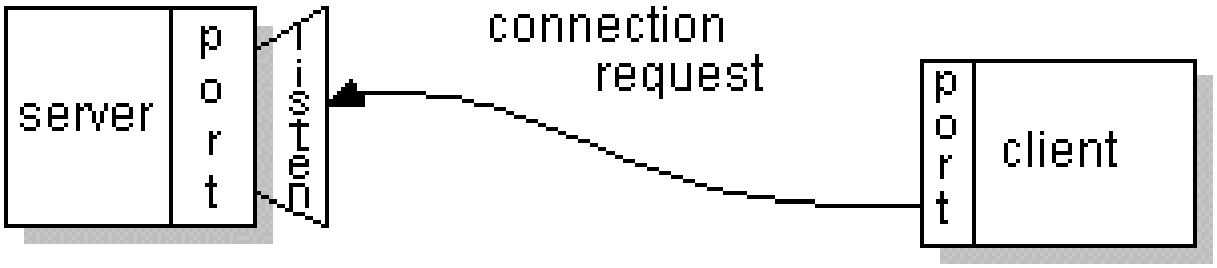
Programmazione delle socket con TCP

Il client deve contattare il server

- Il processo server deve essere in corso di esecuzione
- Il server deve avere creato una socket (porta) che dà il benvenuto al contatto con il client

Il client contatta il server:

- Creando una socket TCP
- Specificando l'indirizzo IP, il numero di porta del processo server
- Quando il client crea la socket: il client TCP stabilisce una connessione con il server TCP

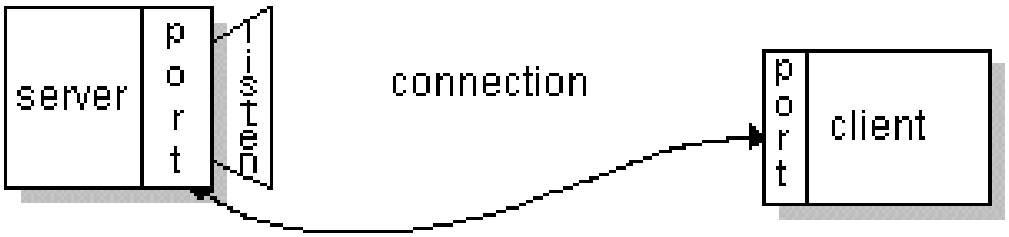


Programmazione delle socket con TCP (2)

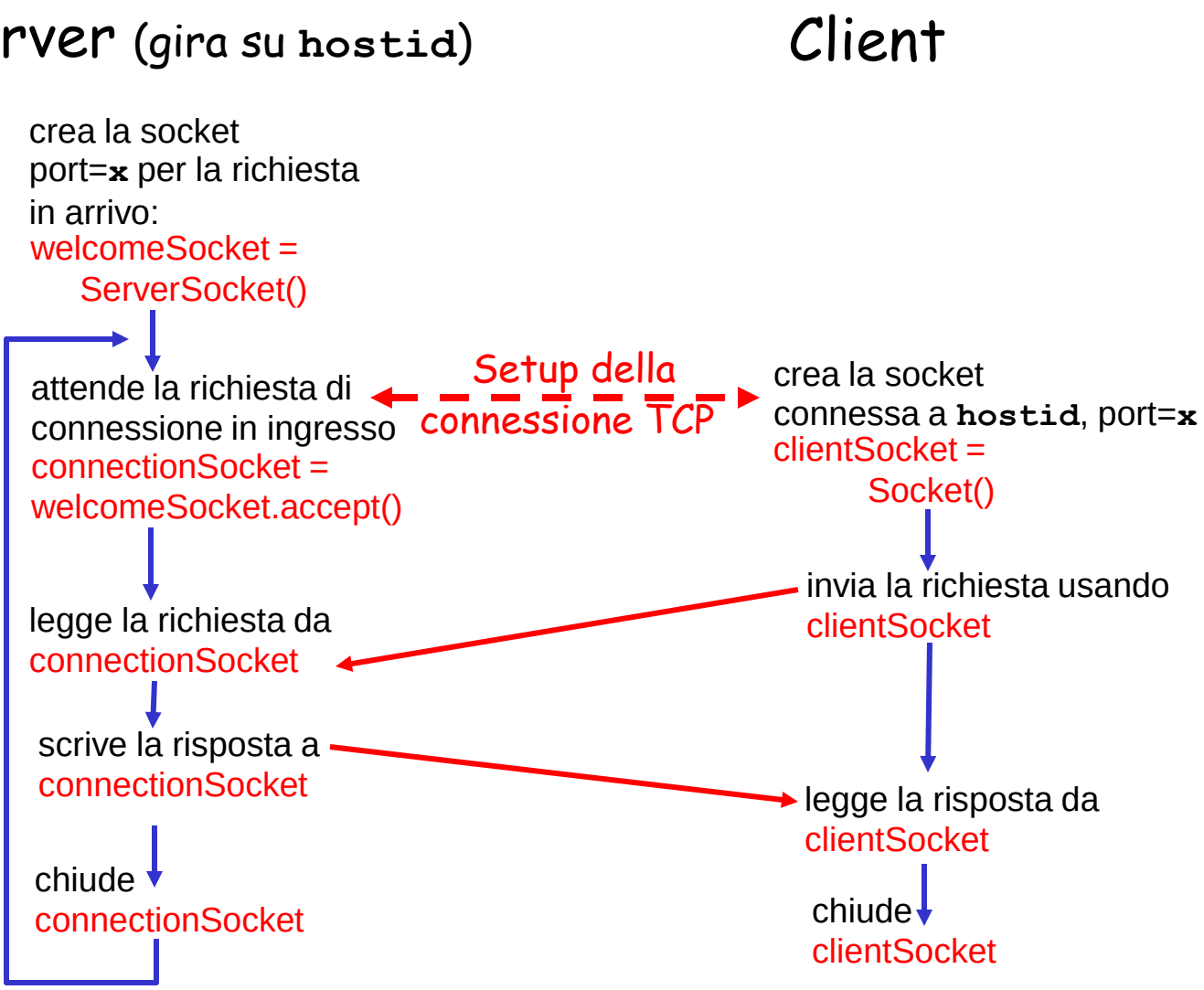
- Quando viene contattato dal client, il server TCP crea una nuova socket per il processo server per comunicare con il client
  - consente al server di comunicare con più client
  - numeri di porta origine usati per distinguere i client (maggiori informazioni nelle lezioni su TCP)

Punto di vista dell'applicazione

TCP fornisce un trasferimento di byte affidabile e ordinato ("pipe") tra client e server

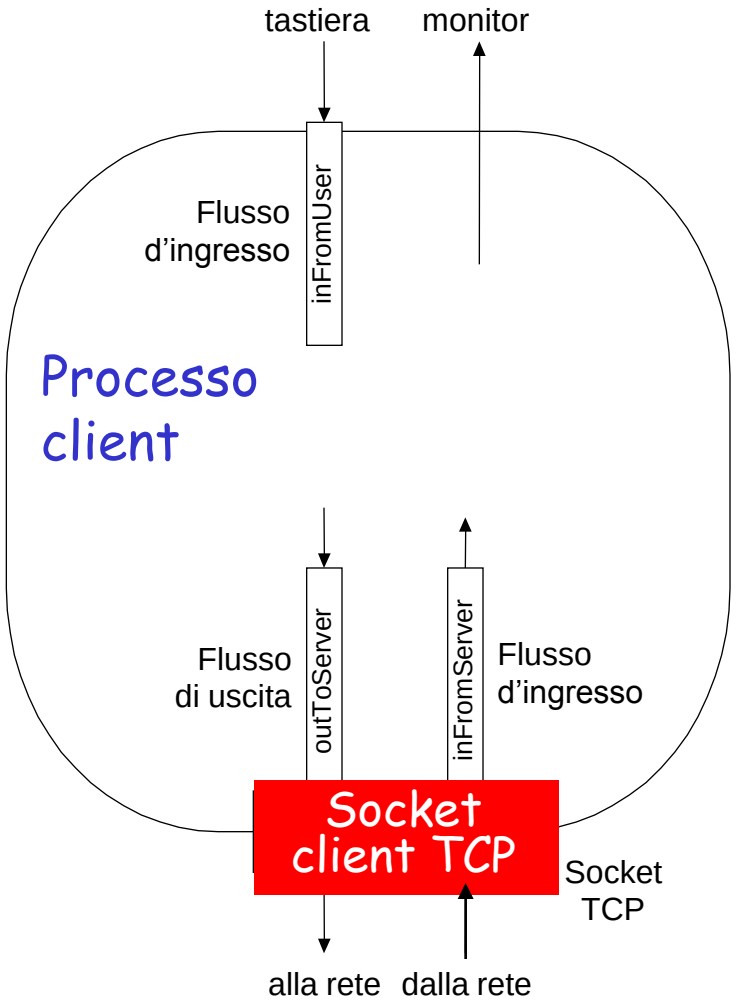


Interazione delle socket client/server: TCP



Termini

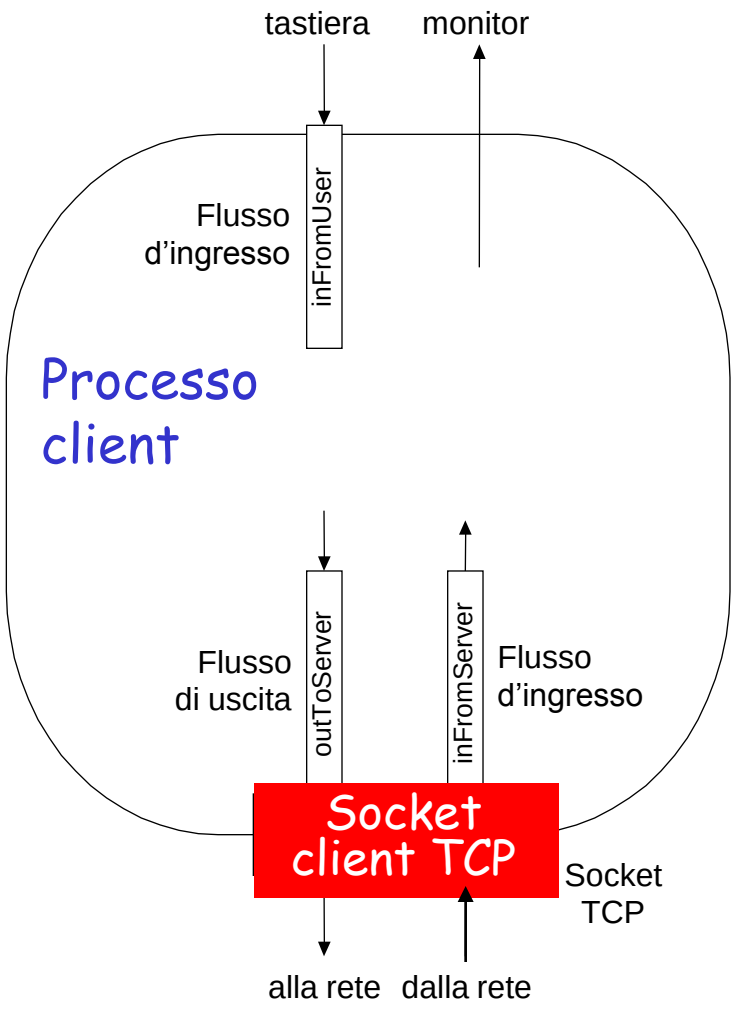
- Un flusso (*stream*) è una sequenza di caratteri che fluisce verso/da un processo.
- Un flusso d'ingresso (*input stream*) è collegato a un'origine di input per il processo, ad esempio la tastiera o la socket.
- Un flusso di uscita (*output stream*) è collegato a un'uscita per il processo, ad esempio il monitor o la socket.



Programmazione delle socket con TCP

Esempio di applicazione client-server:

1) Il client legge una riga dall'input standard (flusso **`inFromUser`**) e la invia al server tramite la socket (flusso **`outToServer`**)
2) Il server legge la riga dalla socket
3) Il server converte la riga in lettere maiuscole e la invia al client
4) Il client legge nella sua socket la riga modificata e la visualizza (flusso **`inFromServer`**)



Esempio: client Java (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Crea un flusso d'ingresso

Crea una socket client, connessa al server

Crea un flusso di uscita collegato alla socket

Esempio: client Java (TCP), continua

Crea un flusso d'ingresso collegato alla socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

    }
}
```

Invia una riga al server

Legge la riga dal server

Esempio: server Java (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

       Socket connectionSocket = welcomeSocket.accept();

      BufferedReader inFromClient =
        new BufferedReader(new
        InputStreamReader(connectionSocket.getInputStream()));
```

Crea una socket di benvenuto sulla porta 6789

Attende, sulla socket di benvenuto, un contatto dal client

Crea un flusso d'ingresso collegato alla socket

Esempio: server Java (TCP), continua

Crea un flusso di uscita collegato alla socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

Legge la riga dalla socket

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Scrive la riga sulla socket

```
outToClient.writeBytes(capitalizedSentence);
            }
        }
    }
```

Fine del ciclo while, ricomincia il ciclo e attende un'altra connessione con il client

from The Java™ Tutorial

- **Definition:** *A socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.
- An endpoint is a combination of an **IP address** and a **port number**. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

Capitolo 2: Livello di applicazione

- Applicazioni P2P
- Programmazione delle socket con TCP
- Programmazione delle socket con UDP

Programmazione delle socket *con UDP*

UDP: non c'è "connessione" tra client e server

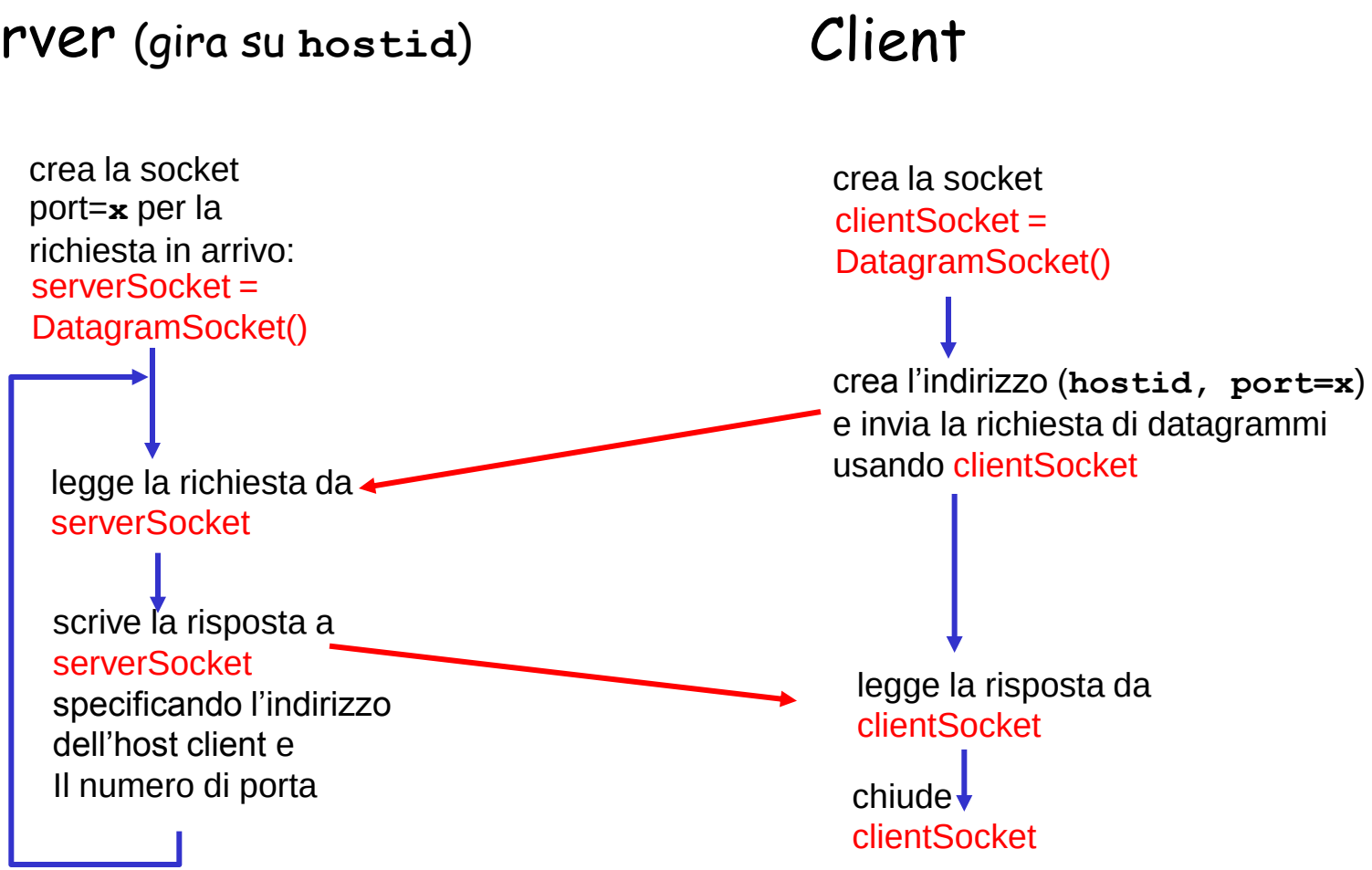
- Non c'è handshaking
- Il mittente allega esplicitamente a ogni pacchetto l'indirizzo IP e la porta di destinazione
- Il server deve estrarre l'indirizzo IP e la porta del mittente dal pacchetto ricevuto

UDP: i dati trasmessi possono perdersi o arrivare a destinazione in un ordine diverso da quello d'invio

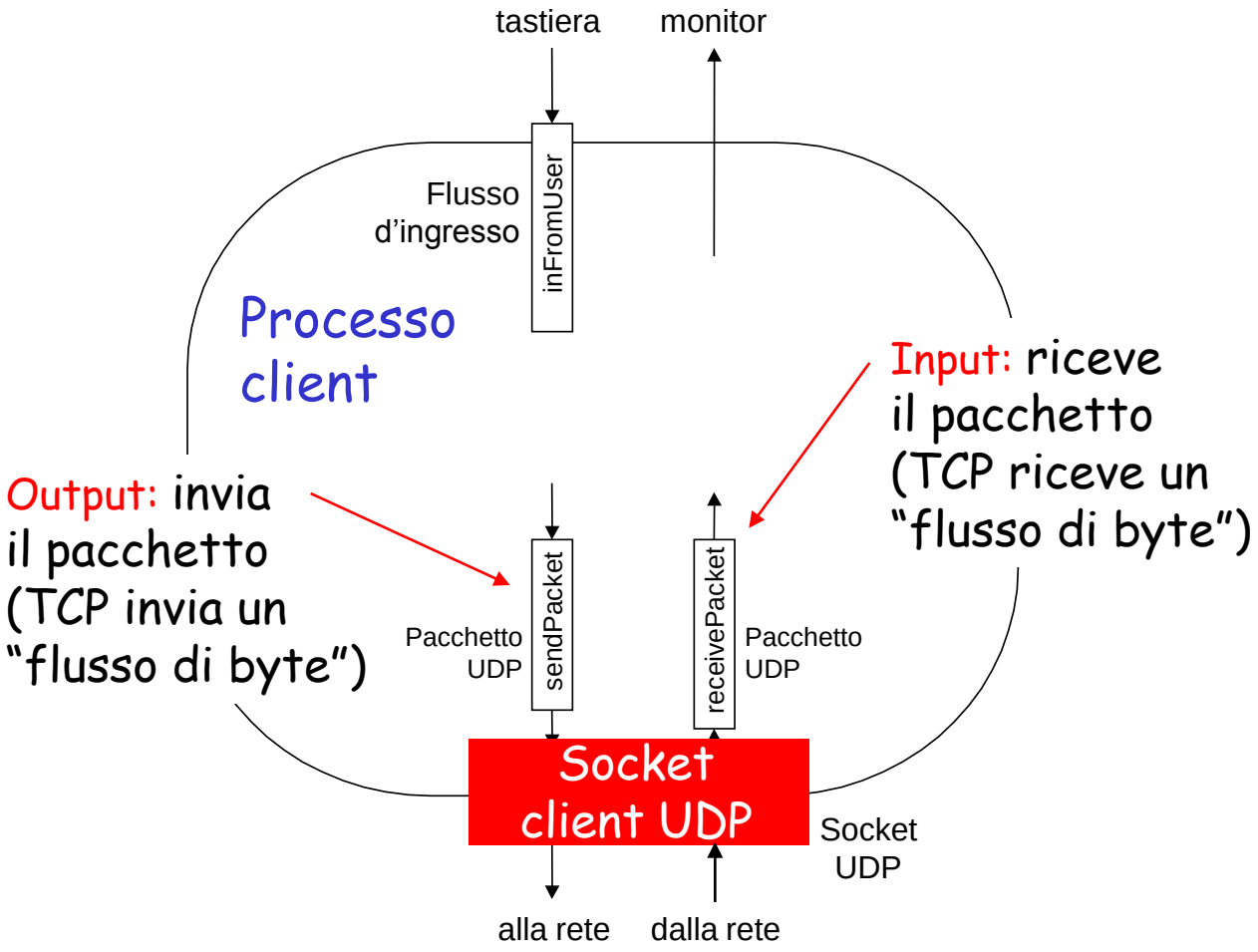
Punto di vista dell'applicazione

UDP fornisce un trasferimento inaffidabile di gruppi di byte ("datagrammi") tra client e server

Interazione delle socket client/server: UDP



Esempio: client Java (UDP)



Esempio: client Java (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
   public static void main(String args[]) throws Exception
   {
      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Crea un flusso d'ingresso

Crea una socket client

Traduce il nome dell'host nell'indirizzo IP usando DNS

Esempio: client Java (UDP), continua

Crea il datagramma con i dati da trasmettere, lunghezza, indirizzo IP, porta

Invia il datagramma al server

Legge il datagramma dal server

```
DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

clientSocket.send(sendPacket);

DatagramPacket receivePacket =
   new DatagramPacket(receiveData, receiveData.length);

clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}

}
```

Il client rimane inattivo fino a quando riceve un pacchetto

Esempio: server Java (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
   {

     DatagramSocket serverSocket = new DatagramSocket(9876);

     byte[] receiveData = new byte[1024];
     byte[] sendData  = new byte[1024];

     while(true)
      {

       DatagramPacket receivePacket =
          new DatagramPacket(receiveData, receiveData.length);

       serverSocket.receive(receivePacket);
```

Crea una socket per datagrammi sulla porta 9876

Crea lo spazio per i datagrammi

Riceve i datagrammi

Esempio: server Java (UDP), continua

```
                String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

                String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress,
                      port);

serverSocket.send(sendPacket);
        }
    }

}
```

Ottiene l'indirizzo IP e il numero di porta del mittente

Crea il datagramma da inviare al client

Scrive il datagramma sulla socket

Fine del ciclo while, ricomincia il ciclo e attende un altro datagramma

Riassunto

Lo studio delle applicazioni di rete adesso è completo!

- Architetture delle applicazioni
  - client-server
  - P2P
  - ibride
- Requisiti dei servizi delle applicazioni:
  - affidabilità, ampiezza di banda, ritardo
- Modello di servizio di trasporto di Internet
  - orientato alle connessioni, affidabile: TCP
  - inaffidabile, datagrammi: UDP
- Protocolli specifici:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, Skype
- Programmazione delle socket

Riassunto

Molto importante: conoscere i *protocolli*

- Tipico scambio di messaggi di richiesta/risposta:
  - il client richiede informazioni o servizi
  - il server risponde con dati e codici di stato
- Formati dei messaggi:
  - intestazioni: campi che forniscono informazioni sui dati
  - dati: informazioni da comunicare
- Controllo o messaggi di dati
  - in banda, fuori banda
- Architettura centralizzata o decentralizzata
- Protocollo senza stato o con stato
- Trasferimento di messaggi affidabile o inaffidabile
- "Complessità nelle parti periferiche della rete"