DNS: Domain Name System

Persone: molte mezzi di identificazione:

- CF, nome, # Passaporto

Host, router Internet:

- Indirizzi IP (32 bit) – usati per indirizzare i datagrammi IP
- "Nome", es., gaia.cs.umass.edu – usati dagli utenti

D.: corrispondenza tra indirizzo IP e nome?

Domain Name System:

- *Database distribuito* implementato come una gerarchia di molti *name server*
- *Protocollo applicativo*

  usato da host, router, name server per comunicare allo scopo di *risolvere* (tradurre) i nomi in indirizzi IP
  - Nota: funzione di base di Internet implementata come protocollo applicativo
  - La complessità trasferita al "bordo" della rete

Name server DNS

Perché non un server DNS centralizzato?

- Minor tolleranza ai guasti
- Traffico eccessivo
- Database centrale troppo distante in molti casi
- Scarsa scalabilità!

- Nessun name server contiene tutte le associazioni nome simbolico/indirizzo IP

Name server locali :

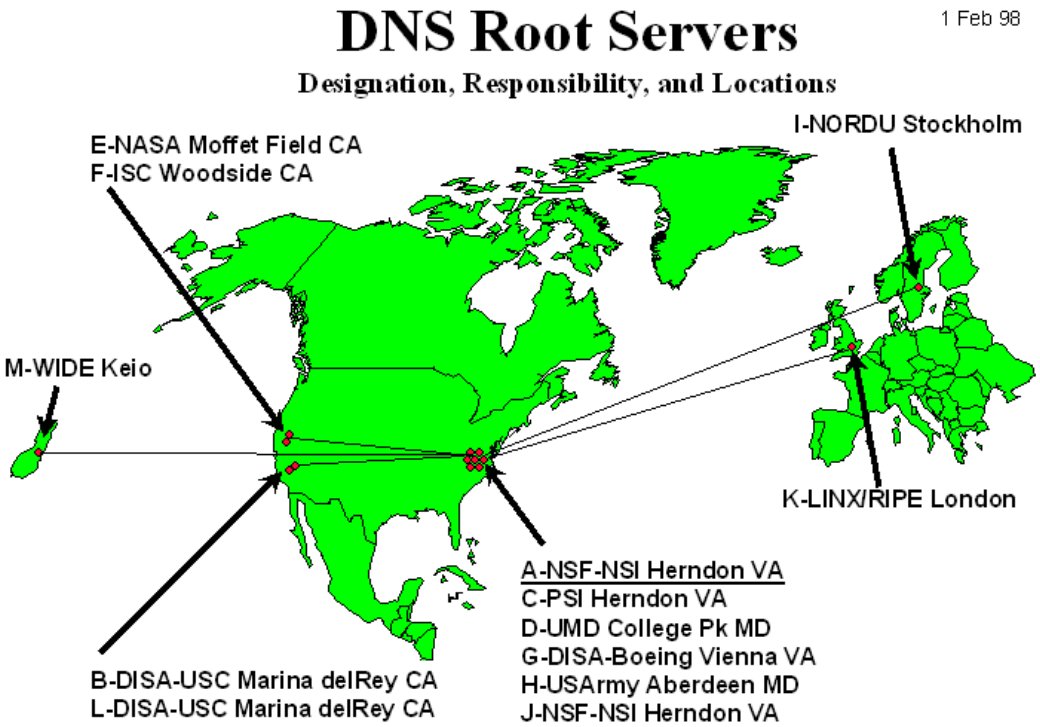
- Ogni ISP o compagnia ha un *name server locale (default)*
- La richiesta di traduzione (mapping) di un host è prima rivolta al name server locale

Name server di riferimento :

- Per un host: per definizione è quello che è sempre in grado di eseguire la traduzione (mapping) nome simbolico/indirizzo IP dell'host

DNS: Root name server

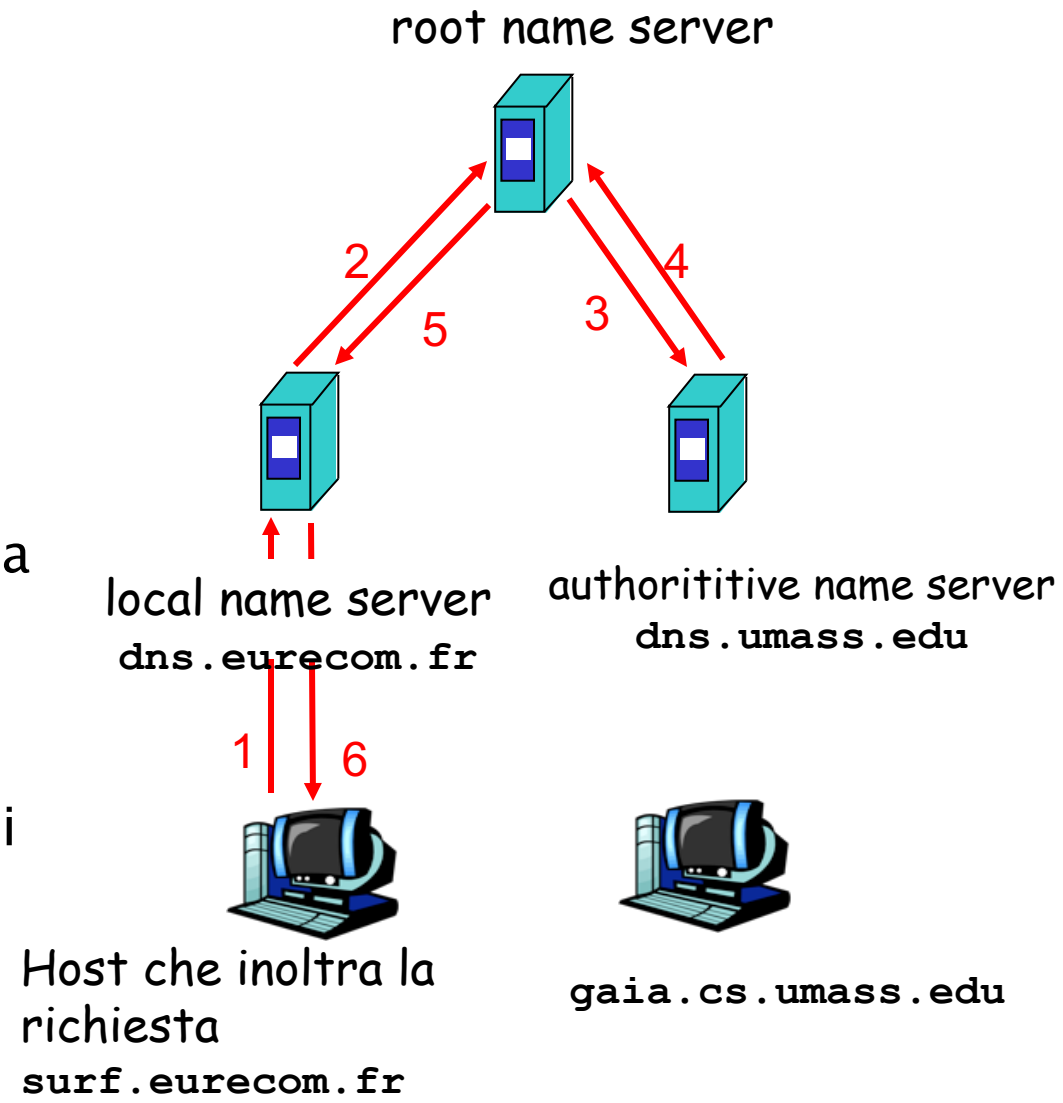
- Contattato dal name server locale che non riesce a risolvere un nome
- root name server:
  - Contatta il name server di riferimento (authoritative) se la traduzione non è nota
  - Ottiene la traduzione
  - Restituisce la traduzione al name server locale
- ~ una dozzina di root name server nel mondo



Esempio

L' host **surf.eurecom.fr** vuole l'indirizzo IP di **gaia.cs.umass.edu**

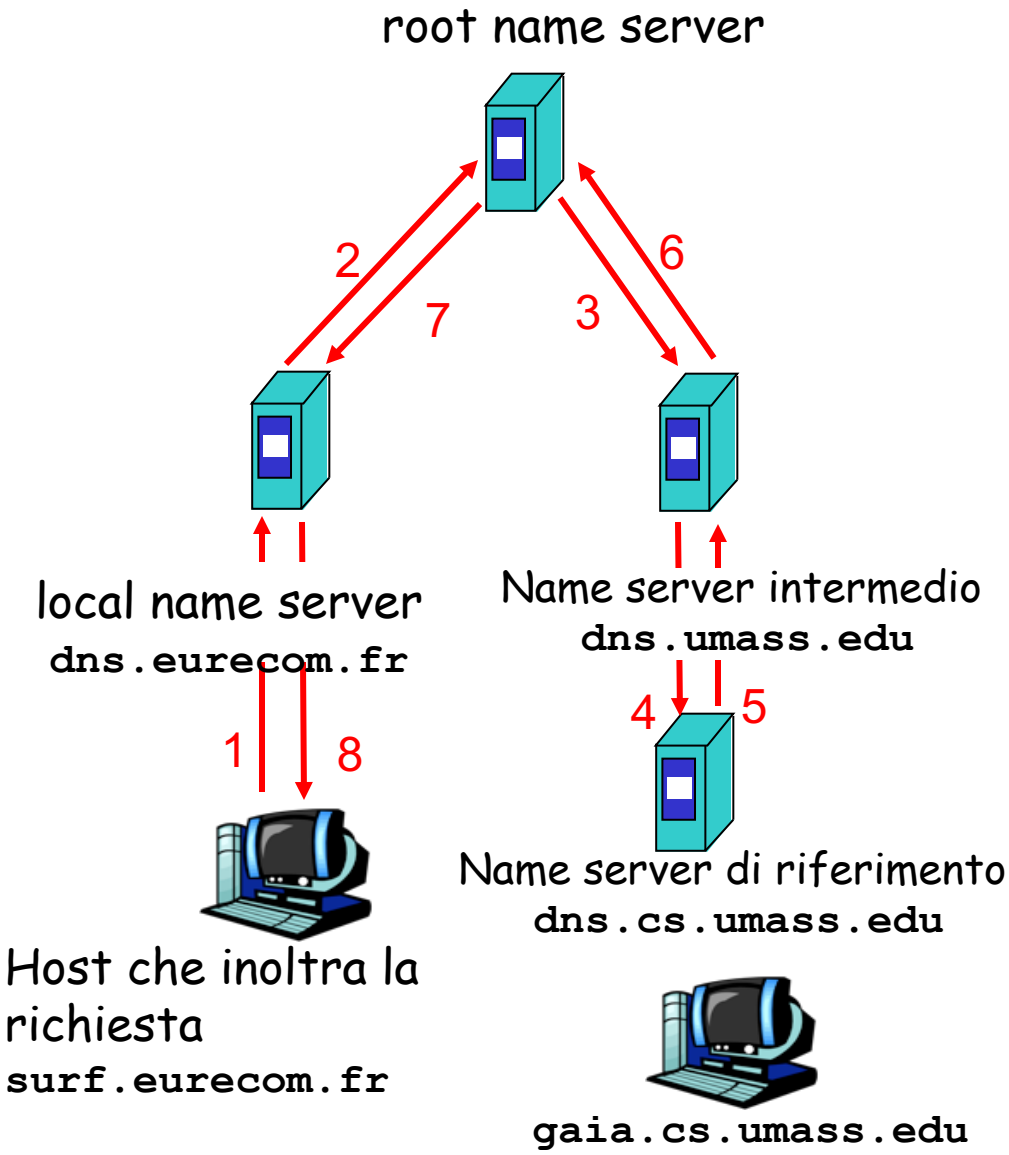
1. Contatta il server DNS locale, **dns.eurecom.fr**
2. **dns.eurecom.fr** contatta il root name server, se necessario
3. Il root name server contatta il name server di riferimento, **dns.umass.edu**, se necessario



Esempio (2)

Root name server:

- Può non essere a conoscenza di un name server di riferimento
- Può tuttavia conoscere un *name server intermedio* che contatta per avere raggiungere quello di riferimento



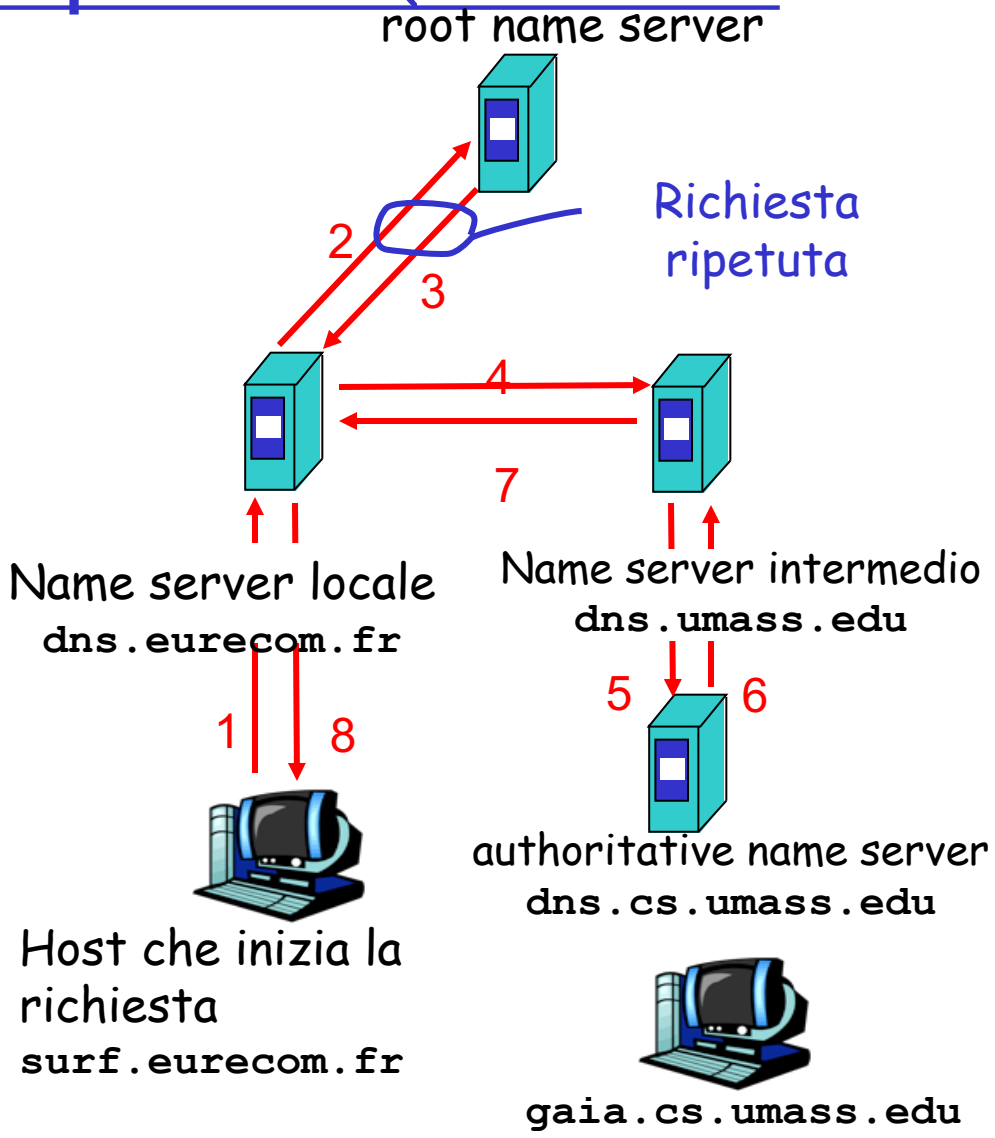
DNS: richieste ripetute (iterated queries)

Richieste ricorsive (recursive query):

- Trasferisce il carico della traduzione al name server contattato
- Carico eccessivo?

Richieste ripetute (iterated query):

- Il name server contattato risponde con l'indirizzo del prossimo name server da contattare
- "Non conosco questo nome, ma prova a rivolgerti a quest'altro server"



DNS: caching e aggiornamento

- Quando un qualsiasi name server apprende una traduzione la memorizza localmente (caching)
  - Le traduzioni memorizzate nella cache (cache entries) scadono (timeout) dopo un certo tempo (di solito un paio di giorni)
- Se possibile, richieste successive vengono servite usando la traduzione presente in cache
- I meccanismi di aggiornmento/modifica in studio da parte dell' IETF
  - RFC 2136
  - http://www.ietf.org/html.charters/dnsind-charter.html

Record DNS

DNS: database distribuito che memorizza Resource Record (RR)

Formato RR: **(nome, valore, tipo, ttl)**

- Tipo=A
  - **nome** è il nome dell' host
  - **valore** è l' indirizzo IP
- Tipo=NS
  - **nome** è il dominio (es. foo.com)
  - **valore** è l'indirizzo IP del name server di riferimento per questo dominio
- Tipo=CNAME
  - **nome** è un alias di qualche nome reale ("canonico")
  - **valore** è il nome canonico
- Tipo=MX
  - **valore** è il nome di un mailserver associato a **nome**

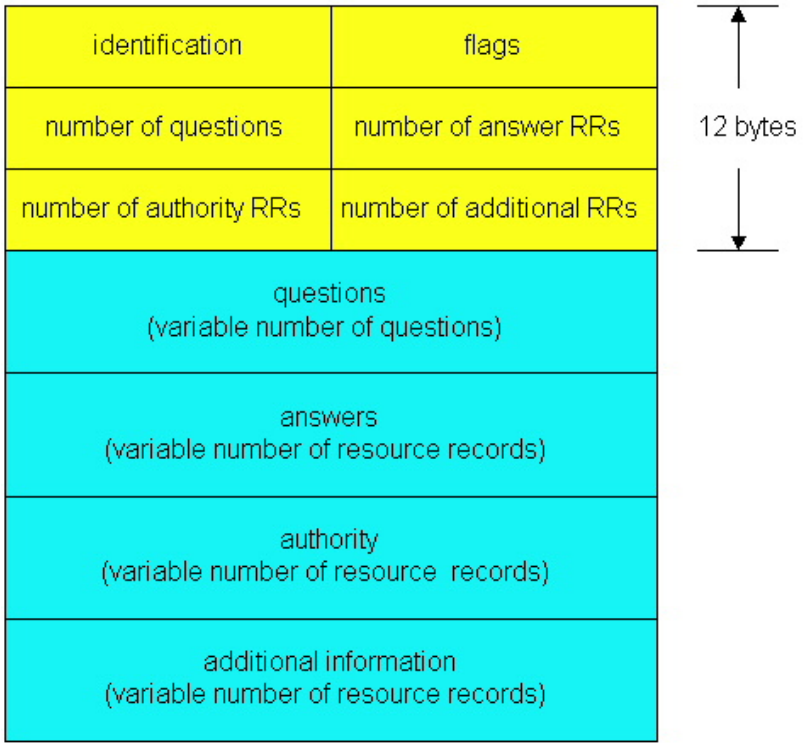
Protocollo DNS, messaggi

Protocollo DNS : messaggi di *richiesta (query)* e *risposta (reply)*, → *client server*

Header di messaggio

- identification: numero a 16 bit per la richiesta, la risposta usa lo stesso numero
- flag:
  - Richiesta o risposta
  - Chiesta la ricorsione (Q)
  - Ricorsione disponibile (R)
  - Il server che risponde è di riferimento per la richiesta (R)

Nota: richiesta e risposta hanno lo stesso formato



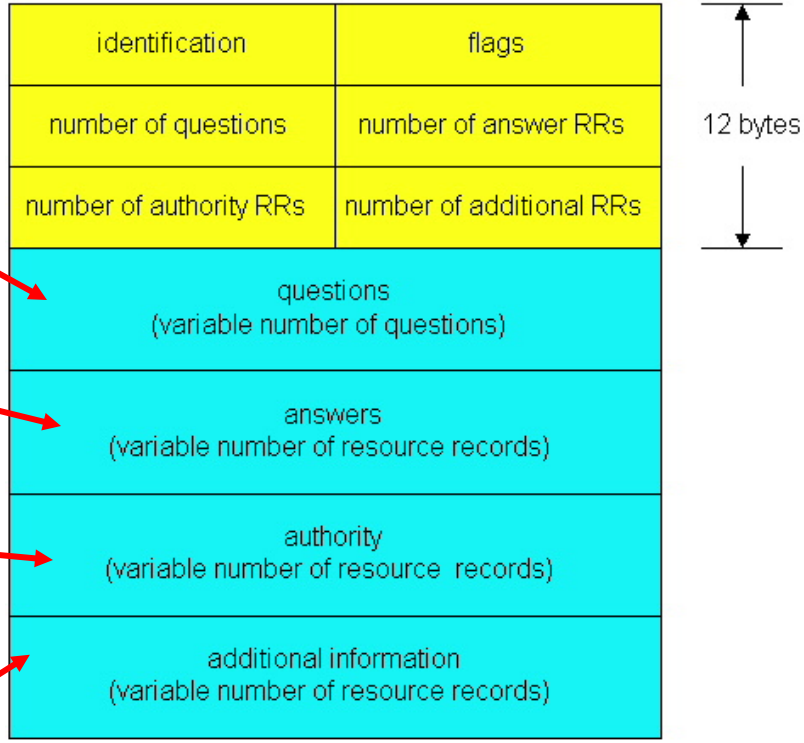
Protocollo DNS, messaggi (2)

Nome, campi tipo per una richiesta

RR in risposta a una richiesta

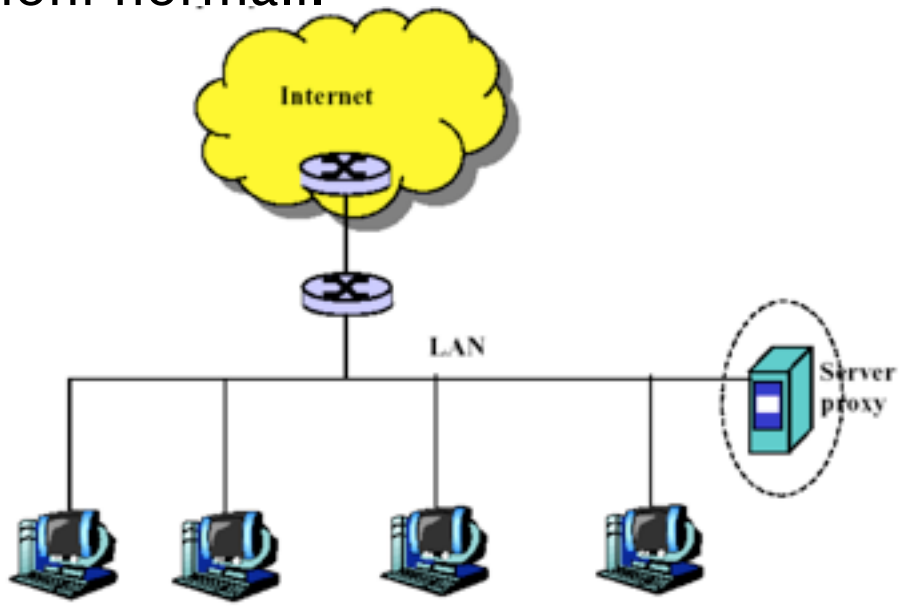
Record per server di riferimento

Informazioni addizionali
Es.: RR di tipo A contenente indirizzo IP di un mail server il cui nome canonico è contenuto nella answer section



Esercizi

- Si supponga che l'Hit Ratio del server proxy sia 0.4. Si assuma che in condizioni normali (link verso Internet non congestionato) il tempo di risposta per una pagina presente nel proxy sia 0.2 s, indipendentemente dalla pagina richiesta. Si assuma che il tempo di risposta per una pagina non in cache sia invece 2 s, indipendentemente dalla pagina richiesta. Determinare il tempo medio di risposta del sistema in condizioni normali.



Esercizi/cont.

- Descrivere cosa succede a livello applicativo da quando si digita www.google.it nell'apposita finestra del browser e poi si dà invio a quando la pagina HTML di base è ricevuta dal browser.
- Si stimi il tempo necessario a scaricare una pagina Web che contiene, oltre al file .html base, 2 riferimenti ad altrettanti oggetti, assumendo connessioni non persistenti e un RTT (Round Trip Time) di 1 secondo. *Si motivi la risposta, mostrando il diagramma temporale della comunicazione tra client e server, illustrando gli aspetti rilevanti del protocollo e le assunzioni considerati ai fini del calcolo.*

Esercizi/cont.

- La comunicazione di controllo SMTP è in banda o fuori banda? Perché?
- Alice invia un messaggio di posta elettronica a Bob. A e B sono i mail server di Alice e Bob. Bob legge la posta elettronica usando il protocollo POP3. Descrivere cosa accade dal momento in cui, dopo aver composto il messaggio, Alice preme "Send" a quello in cui Bob legge il messaggio.

Esercizi/cont.

- Una pagina Web contiene 5 oggetti. Quanti RTT sono necessari per una richiesta http nel caso di
  a. Connessioni non persistenti
  b. Connessioni persistenti non incanalate
  c. Connessioni persistenti incanalate

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

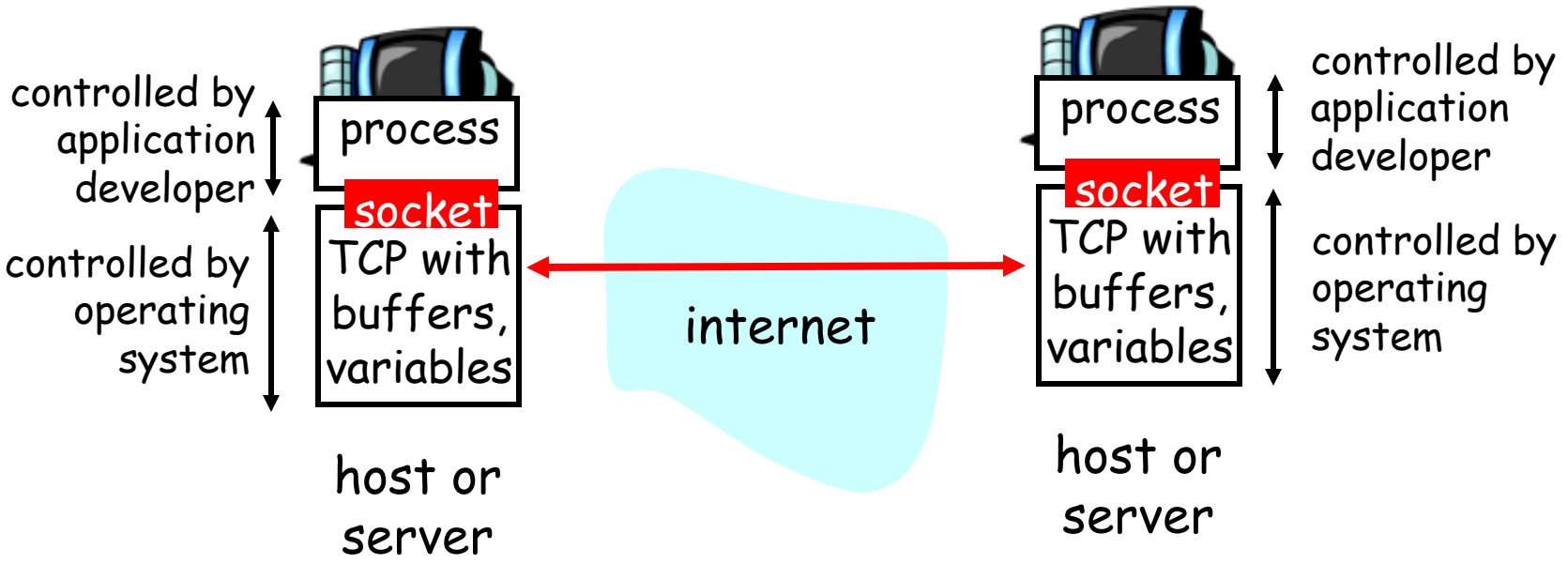
socket

a *host-local*, *application-created/owned*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another (remote or local) application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of bytes from one process to another



Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

- creating client-local TCP socket
- specifying IP address, port number of server process

- When client creates socket: client TCP establishes connection to server TCP
- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

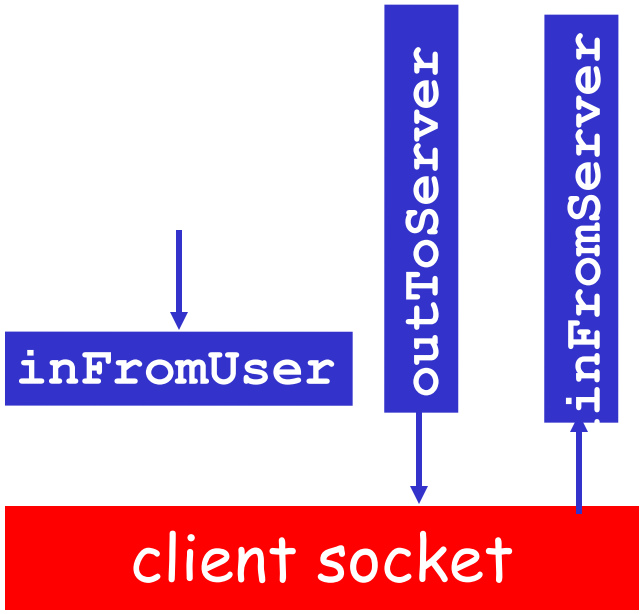
Socket programming with TCP

Example client-server app:

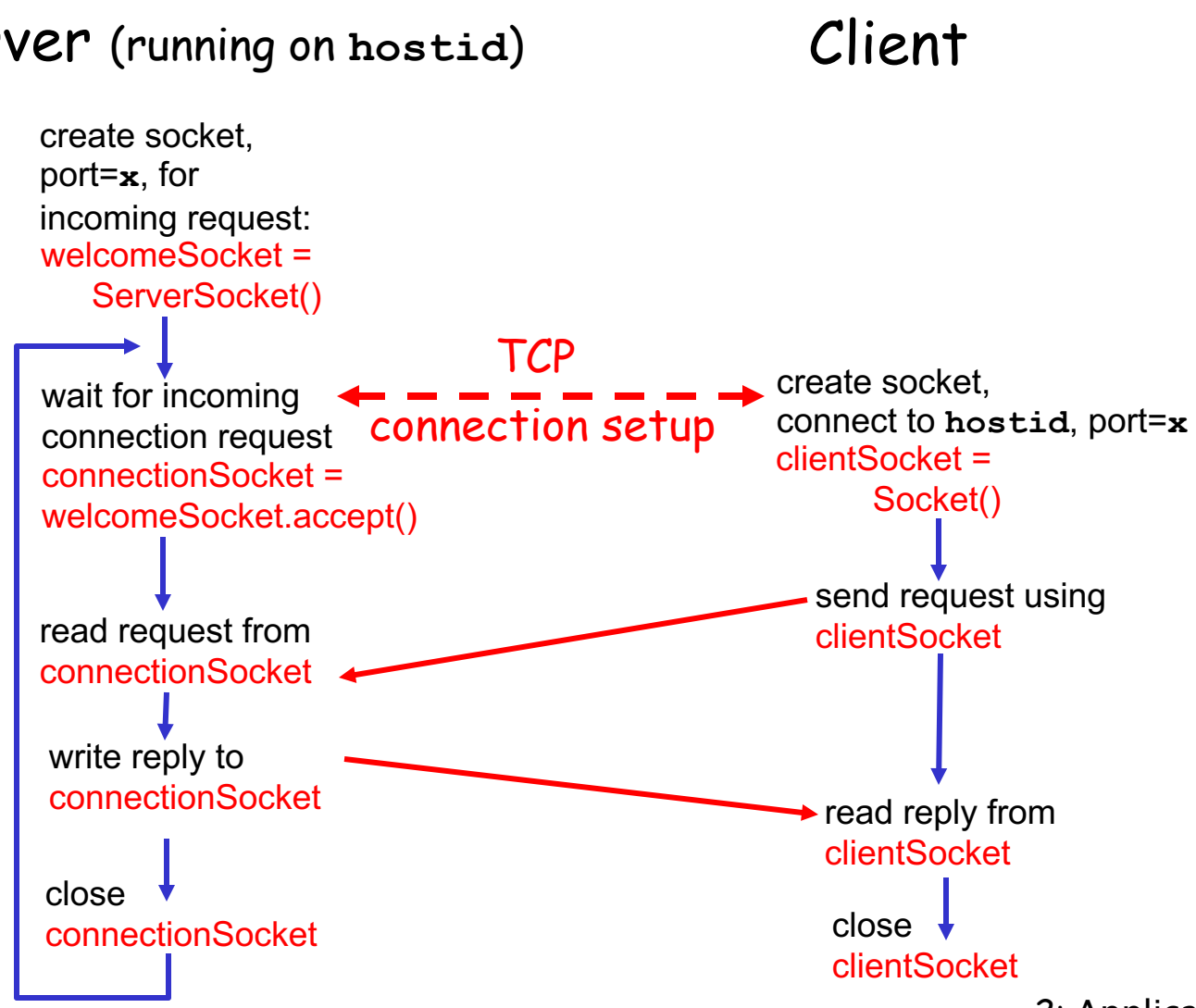
- client reads line from standard input (**`inFromUser`** stream) , sends to server via socket (**`outToServer`** stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (**`inFromServer`** stream)

Input stream: sequence of bytes into process

Output stream: sequence of bytes out of process



Client/server socket interaction: TCP



Example: Java client (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());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

Create input stream attached to 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();

    }
}
```

Send line to server

Read line from server

Example: Java server (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()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

Create output stream, attached to socket

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

Read in line from socket

```
clientSentence = inFromClient.readLine();

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

Write out line to socket

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

End of while loop, loop back and wait for another client connection

Socket programming with UDP

UDP: no "connection" between client and server

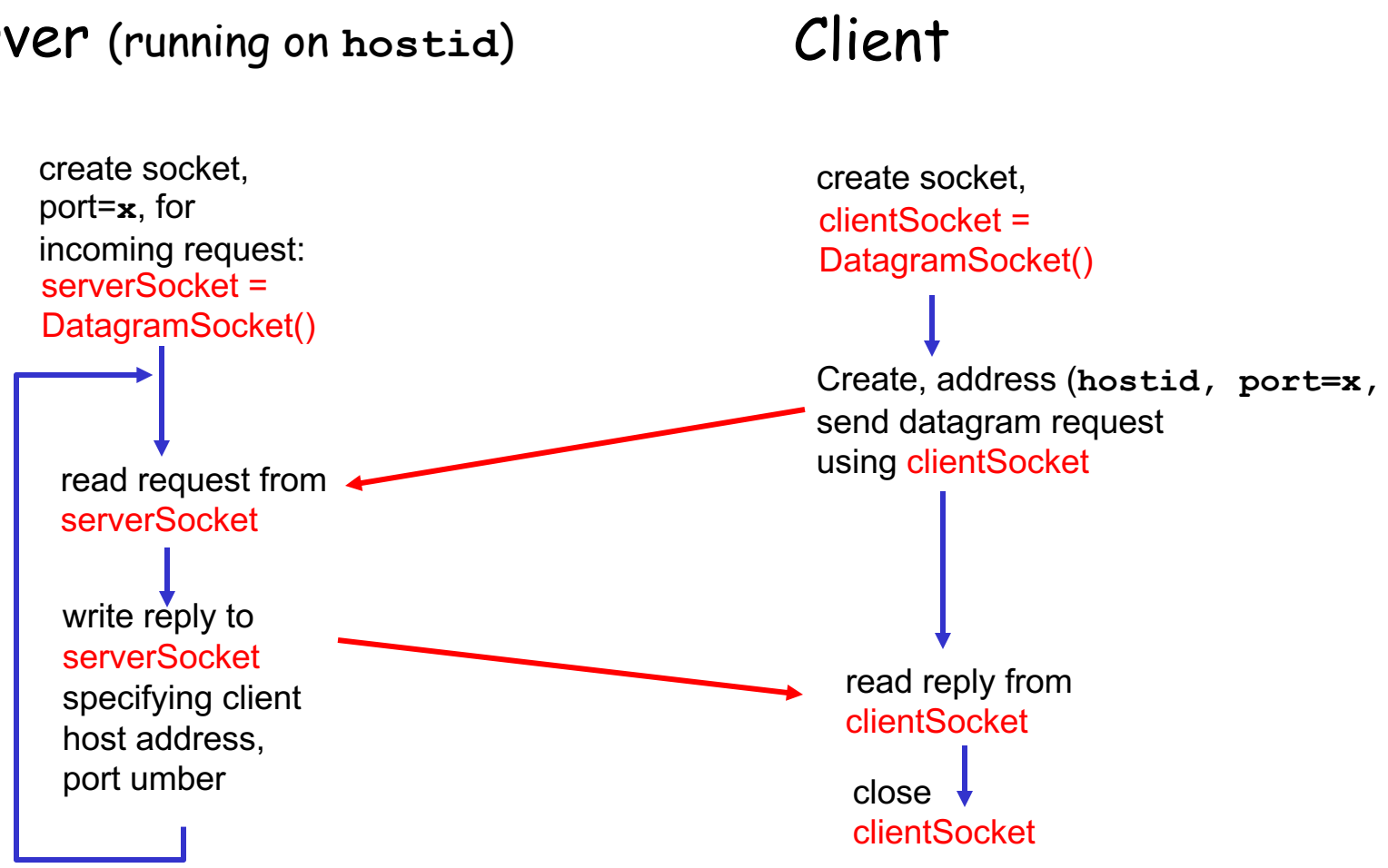
- no handshaking
- sender explicitly attaches IP address and port of destination
- server must extract IP address, port of sender from received datagram

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Example: Java client (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();
```

Create input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from 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();
                }

            }
```

Example: Java server (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);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

```
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);
    }
  }

}
```

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

End of while loop, loop back and wait for another datagram

Chapter 2: Summary

Our study of network apps now complete!

- application service requirements:
  - reliability, bandwidth, delay
- client-server paradigm
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - http
  - ftp
  - smtp, pop3
  - dns
- socket programming
  - client/server implementation
  - using tcp, udp sockets

Chapter 2: Summary

Most importantly: learned about *protocols*

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated

- control vs. data msgs
  - in-based, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"
- security: authentication