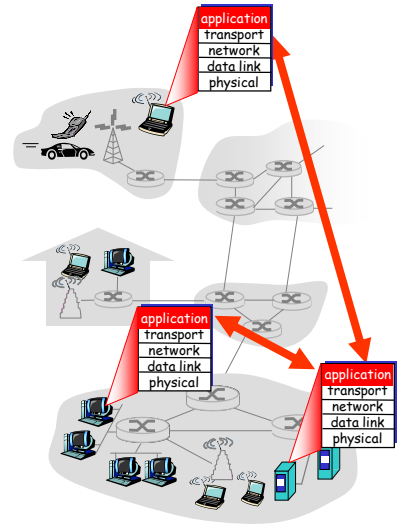
Applicazioni di rete

Programmi
- ❑ in esecuzione su *end systems*
- ❑ che comunicano attraverso la rete
- ❑ esempio: web server comunica con browser

Non è necessario scrivere software per il nucleo della rete
- ❑ i nodi del nucleo della rete non eseguono applicazioni di rete



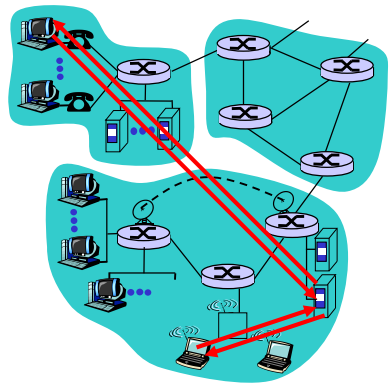
Applicazioni di rete

- ❑ Coppie di *processi*: programmi in esecuzione su end systems

- ❑ Architetture
  - ❍ Client-server
  - ❍ Peer-to-Peer (P2P)
  - ❍ Ibrida (P2P + Client/Server)

- ❑ I commutatori intermedi fanno solo da tramite e ignorano il reale contenuto dei pacchetti

Architettura Client/Server



server:

- Sempre “on”
- Indirizzo IP noto

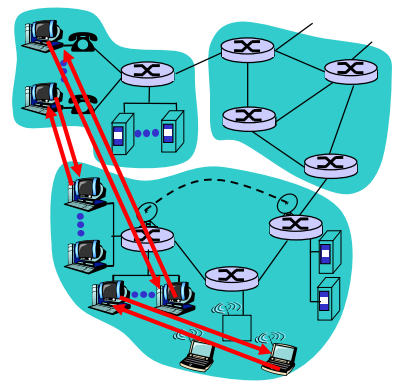
clients:

- comunicano con il server
- non comunicano tra loro direttamente
- Indirizzi IP variabili

Architettura P2P

- Gli end-systems comunicano direttamente tra loro
- I peer richiedono servizio ad altri peer ed offrono servizi
- I peer possono cambiare indirizzo IP

Molto scalabile

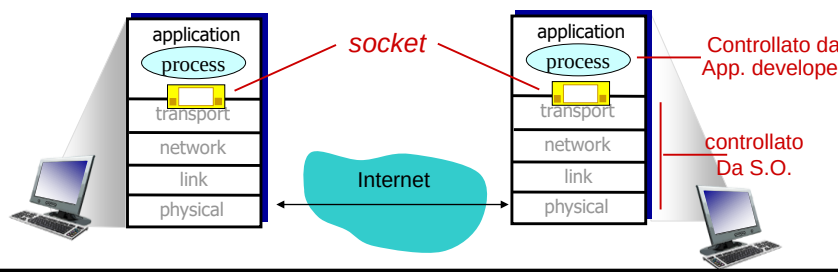


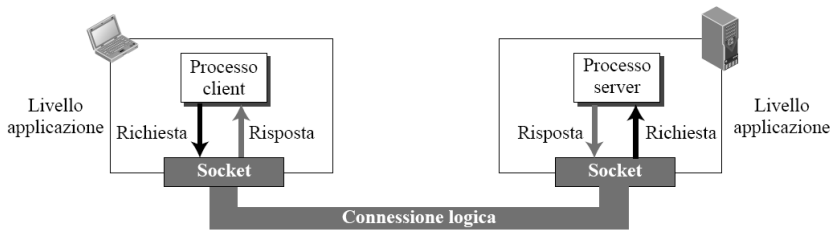
Comunicazione tra processi

- Processo: programma in esecuzione su un host
- Due processi su host diversi comunicano scambiando messaggi
- Processo Client: processo che inizia la comunicazione
- Processo Server: processo che aspetta di essere contattato
- Le applicazioni P2P hanno sia processi client che processi server.

Socket

- Socket: API (Application Programming Interface) di comunicazione tra applicazione e S.O. per l'invio e la ricezione di messaggi a/da un altro processo applicativo
- Introdotte in BSD4.1 UNIX, 1981
- Scelta di:
  - protocollo di trasporto (inaffidabile, affidabile)
  - alcuni parametri a livello trasporto
- Il processo mittente si affida all'infrastruttura offerta dallo strato di trasporto per consegnare il messaggio al socket del processo ricevente





Se si creano due socket definendo gli indirizzi del mittente e del destinatario si possono usare le istruzioni disponibili per inviare e ricevere dati. Il resto è responsabilità del sistema operativo e dei protocolli TCP/IP

Indirizzamento

- Il processo mittente deve poter identificare il processo ricevente
- I nodi di rete hanno un indirizzo, ma non è sufficiente: più processi potrebbero essere in esecuzione sullo stesso nodo
- Socket address: combinazione di indirizzo dell'host (indirizzo IP a 32 bit) e identificatore processo ricevente (numero di porta o socket)
- Ogni interfaccia possiede 65535 "porte"
- http://www.iana.org : coordinamento DNS root, indirizzamento IP, protocolli.

Cosa definiscono i protocolli applicativi

- Tipi di messaggi scambiati (richiesta, risposta)
- Sintassi: i campi di un messaggio e il loro formato
- Semantica dei messaggi: significato delle informazioni nei campi
- Regole su come e quando i processi si scambiano i messaggi
- Protocolli di pubblico dominio (definiti nelle RFC, es. HTTP, SMTP) o proprietari (es. Skype)

Servizi richiesti dai protocolli applicativi

- Affidabilità (integrità dei dati): alcune applicazioni (es., audio) tollerano perdita di dati, altre (es., telnet, ftp) richiedono completa affidabilità
- Garanzia di:
  - Tempo di risposta (latenza): alcune applicazioni (telefonia Internet, giochi), per essere efficaci, tollerano ritardi minimi
  - Throughput: alcune applicazioni (multimedia) richiedono throughput minimo garantito, per essere efficaci, altre si "adattano" alla banda disponibile

Requisiti di alcune applicazioni

Applicazione	Affidabilità	Throughput	Latenza
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few kbps up	yes, 100's msec
instant messaging	no loss	elastic	yes and no

Supporto alle applicazioni

TCP:

- *trasporto affidabile:* dati in ordine e senza errori
- *controllo del flusso*
- *controllo della congestione*
- non fornisce: temporizzazione, garanzia di banda
- *orientato alla connessione*: handshake e apertura della connessione *full-duplex*

UDP:

- trasferimento non affidabile tra mittente e destinatario
- non fornisce:connessione, affidabilità, controllo del flusso e della congestione, garanzia di latenza e banda
- protocollo "leggero": minor overhead

Applicazioni e protocolli di trasporto

Applicazione	Protocollo applicativo	Protocollo di trasporto usato
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854], SSH	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	RTP [RFC 1889]	TCP o UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP o UDP

Socket

Due tipi di socket per due servizi di trasporto:

- *Datagram (UDP):* senza connessione, non affidabile
- *Stream (TCP):* con connessione, affidabile, flusso di byte

Socket API

Solo per streams

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address (port) with a socket
LISTEN	Announce willingness to accept connections
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND(TO)	Send some data over the socket
RECEIVE(FROM)	Receive some data over the socket
CLOSE	Release the socket

Programmazione socket con *UDP*

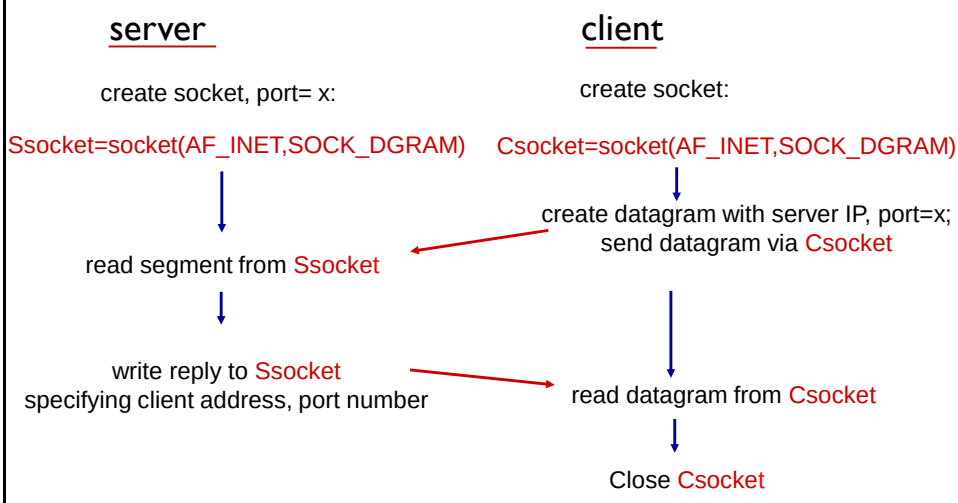
UDP: no "connessione" tra client e server

- Il sender attacca esplicitamente ad ogni pacchetto l'indirizzo IP destinazione e il numero di porta
- Il receiver estrae dal pacchetto l'indirizzo IP del sender e il numero di porta

UDP: I dati possono essere persi o ricevuti fuori ordine

- UDP fornisce un trasferimento inaffidabile di gruppi di bytes (datagrammi) tra client e server

Client/server socket interaction: UDP



Esempio: client UDP

Python UDPClient

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = 'bla bla bla''
clientSocket.sendto(message,(serverName, serverPort))
receivedMessage, serverAddress =
                                clientSocket.recvfrom(2048)
print receivedMessage
clientSocket.close()
```

create UDP socket

Attach server name and port number to message; send packet into socket

read reply characters from socket into string

close socket

Esempio: server UDP

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((' ', serverPort))
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    serverSocket.sendto('ricevuto', clientAddress)
```

create UDP socket

assign socket to local port number 12000

loop forever

Read from UDP socket into message, getting client's address (client IP and port)

send upper case string back to this client

Programmazione socket con TCP

Il client deve contattare il server:

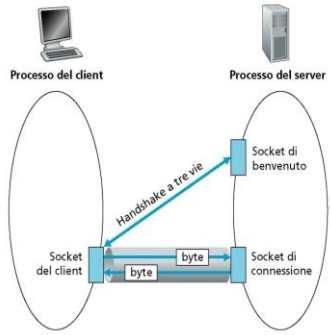
- Sul server deve esserci un processo attivo
- Il server deve aver creato un socket di benvenuto

Il client contatta il server:

1. Creando un socket locale TCP
2. Specificando l'indirizzo IP e numero di porta del server
3. Quando il client crea il socket, il client TCP stabilisce una connessione con il server TCP
4. Contattato dal client, il server TCP crea un nuovo socket per la comunicazione tra processo server e processo client

- Trasferimento affidabile di un flusso di byte ordinato tra client e server

Programmazione socket con TCP



Client/server socket interaction: TCP

server (running on `hostid`)

client

create socket, port=`x`, for incoming request:
serverSocket = socket()

wait for incoming connection request
connectionSocket = serverSocket.accept()

TCP connection setup

create socket, connect to `hostid`, port=`x`
clientSocket = socket()

send request using clientSocket

read request from connectionSocket

write reply to connectionSocket

read reply from clientSocket

close connectionSocket

close clientSocket

Esempio: client TCP

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence "bla bla bla"
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(2048)
print 'From Server:', modifiedSentence
clientSocket.close()
```

create TCP socket for server, remote port 12000

Connection established

No need to attach server name, port

Esempio: server TCP

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((",serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(2048)
    Sentence = 'aaaa'
    connectionSocket.send(Sentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)