

Applicazioni di rete

Programmi (processi) in esecuzione su *end systems* che comunicano attraverso la rete.

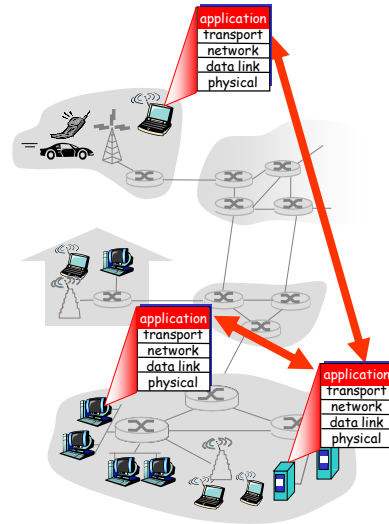
□ Architetture

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

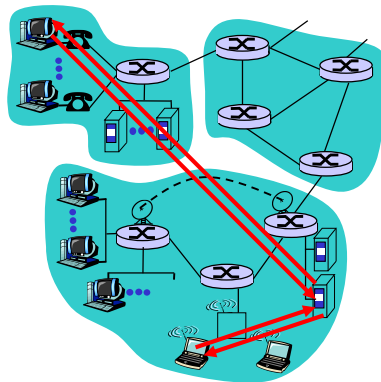
Non è necessario scrivere software per il nucleo della rete

□ i commutatori della sottorete

- non eseguono applicazioni di rete
- fanno da tramite e ignorano il contenuto dei pacchetti



Architettura Client/Server



server:

- sempre "on"
- indirizzo IP noto

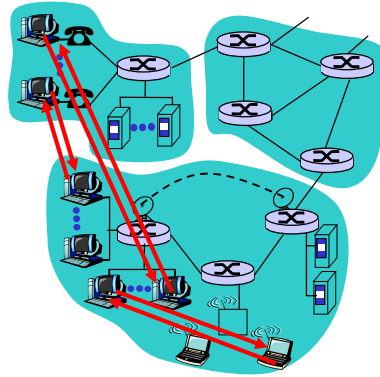
clients:

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

Architettura P2P

- ❑ Comunicazione diretta tra end-systems (peer)
- ❑ I peer richiedono servizio ad altri peer ed offrono servizi
- ❑ I peer possono cambiare indirizzo IP

Molto scalabile: ogni peer aggiunge carico e capacità

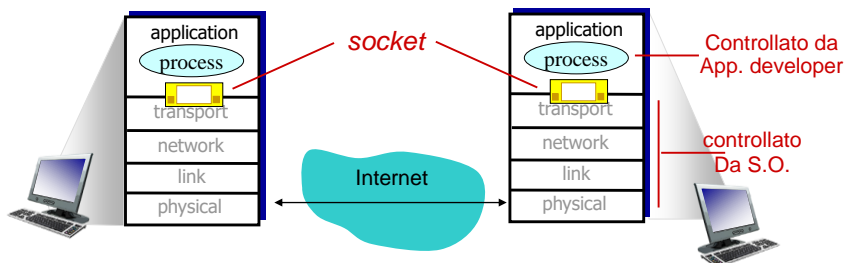


Comunicazione tra processi

- ❑ **Processo** in esecuzione su un end-system
- ❑ Due processi su host diversi comunicano scambiando **messaggi**
- ❑ **Processo Client:** processo che inizia la comunicazione
- ❑ **Processo Server:** processo che aspetta di essere contattato

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



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**)
- ❑ <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 ai protocolli di trasporto

- ❑ **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 (sensibili alla banda), 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

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

Esempio

Python UDPClient

```
serverName = <indirizzo del server>
serverPort = 12000
create UDP socket → clientSocket = socket(AF_INET, SOCK_DGRAM)
Attach server name and port number to message; send packet into socket → message = 'bla bla bla'
clientSocket.sendto(message,(serverName, serverPort))
read reply characters from socket into string → receivedMessage, serverAddress = clientSocket.recvfrom(2048)
print receivedMessage
close socket → clientSocket.close()
```

Python UDPServer

```
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
assign socket to local port number 12000 → serverSocket.bind(('', serverPort))
Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)
send string back to this client → serverSocket.sendto('ricevuto', clientAddress)
```

Programmazione socket con TCP

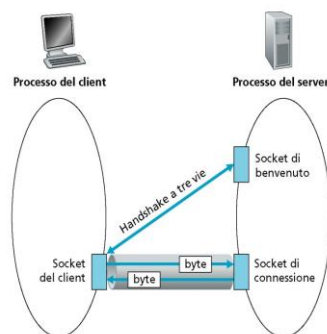
Il client deve contattare il server:

- ❑ 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()`

read request from
`connectionSocket`

write reply to
`connectionSocket`

close
`connectionSocket`

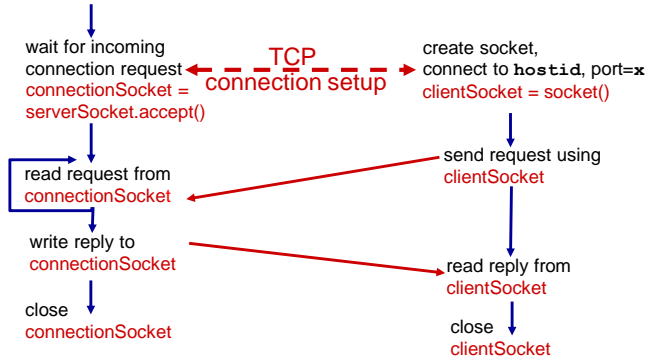
create socket,
connect to `hostid`, port=`x`
`clientSocket = socket()`

send request using
`clientSocket`

read reply from
`clientSocket`

close
`clientSocket`

TCP
connection setup



Esempio: client TCP

Python TCPClient

```
serverName = <indirizzo del server>
serverPort = 12000
create TCP socket for server, remote port 12000 → clientSocket = socket(AF_INET, SOCK_STREAM)
Connection established → clientSocket.connect((serverName,serverPort))
sentence = "bla bla bla"
No need to attach server name, port → clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(2048)
print 'From Server:', modifiedSentence
clientSocket.close()
```

Esempio: server TCP

Python TCPServer

create TCP welcoming socket	→	serverPort = 12000 serverSocket = socket(AF_INET,SOCK_STREAM)
server begins listening for incoming TCP requests	→	serverSocket.bind(('',serverPort)) serverSocket.listen(1) print 'The server is ready to receive'
loop forever	→	while 1:
server waits on accept() for incoming requests, new socket created on return	→	connectionSocket, addr = serverSocket.accept()
read bytes from socket (but not address as in UDP)	→	sentence = connectionSocket.recv(2048) Sentence = 'aaaa'
close connection to this client (but <i>not</i> welcoming socket)	→	connectionSocket.send(Sentence) connectionSocket.close()