

[www.dis.uniroma1.it/~midlab](http://www.dis.uniroma1.it/~midlab)

---

# Sistemi di Calcolo

## Corso di Laurea in Ingegneria Informatica e Automatica

**Prof. Roberto Baldoni**

[www.dis.uniroma1.it/~baldoni](http://www.dis.uniroma1.it/~baldoni)

## Il Sistema Operativo e la rete

*Principali riferimenti: W.R. Stevens “Unix Network Programming” Prentice Hall, 1999*

*Peterson - Davie “Computer Networks: A system approach” Morgan Kaufmann 2000*

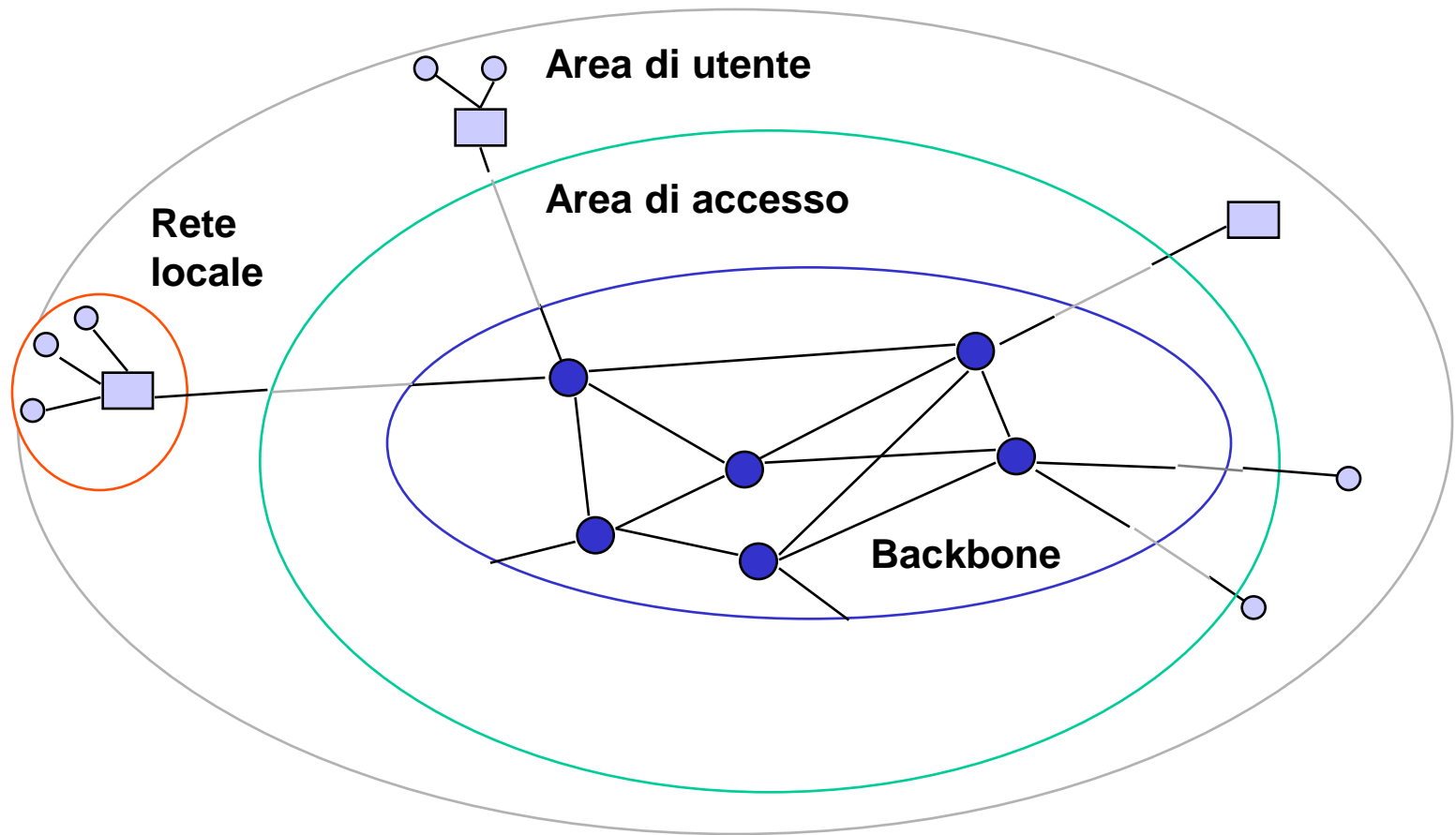
# Contenuti

---

- Architettura di Internet
- Richiami di TCP/IP
- Sockets
- Network Adaptors

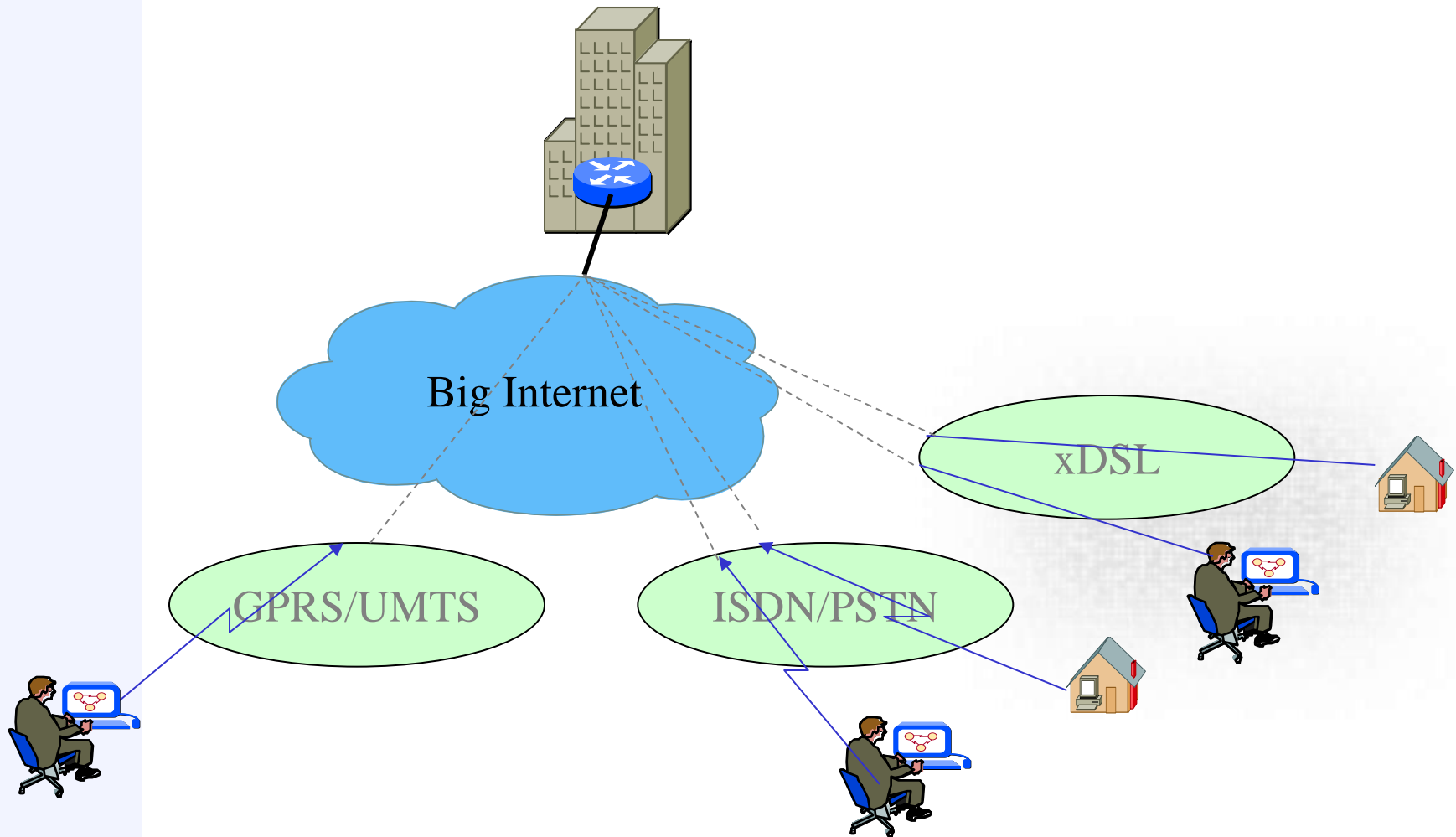
# Architettura di Internet

# Rete geografica per trasmissione dati

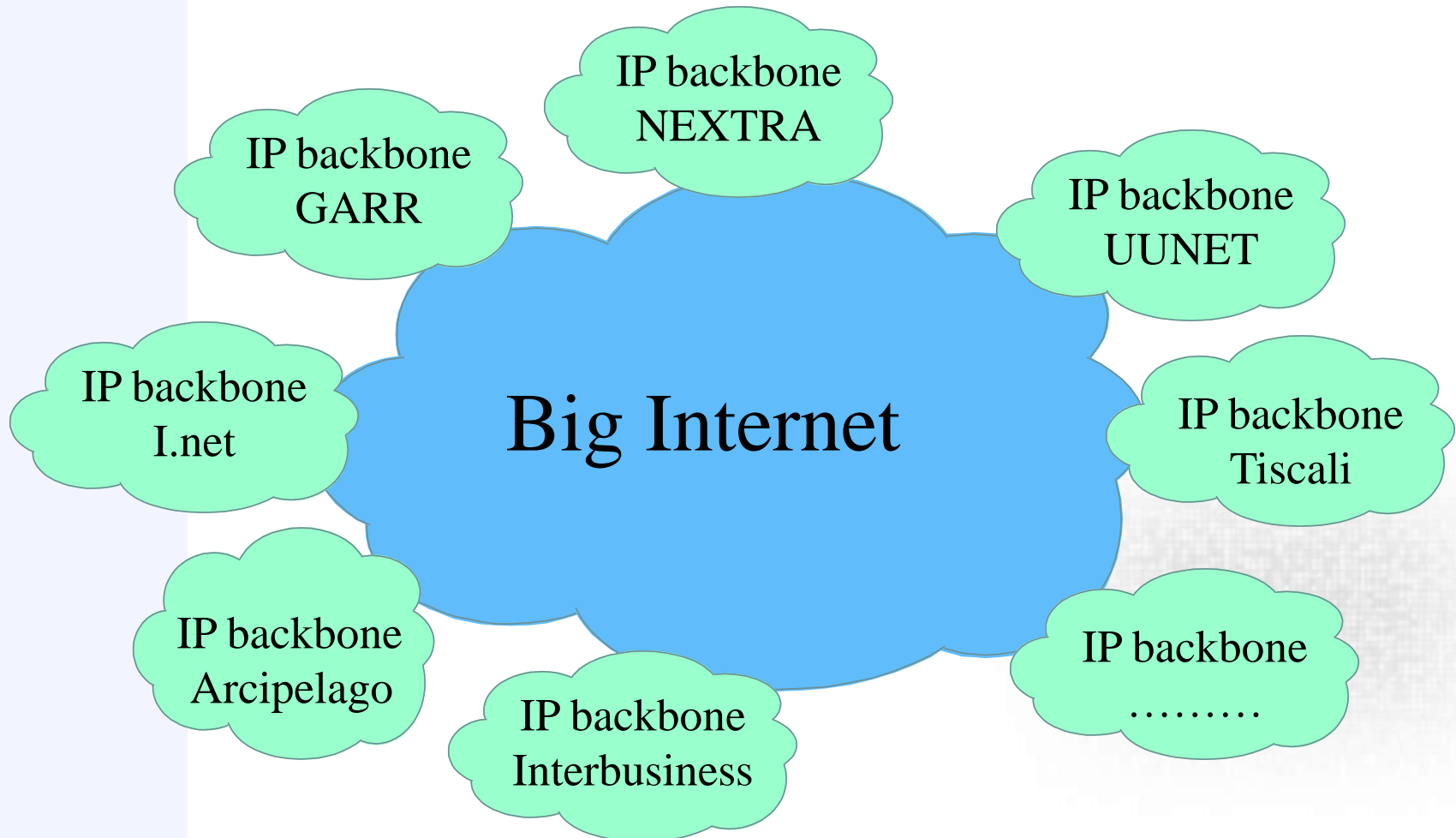


- = terminale di utente
- = unità di accesso
- = nodo del sottosistema di comunicazione

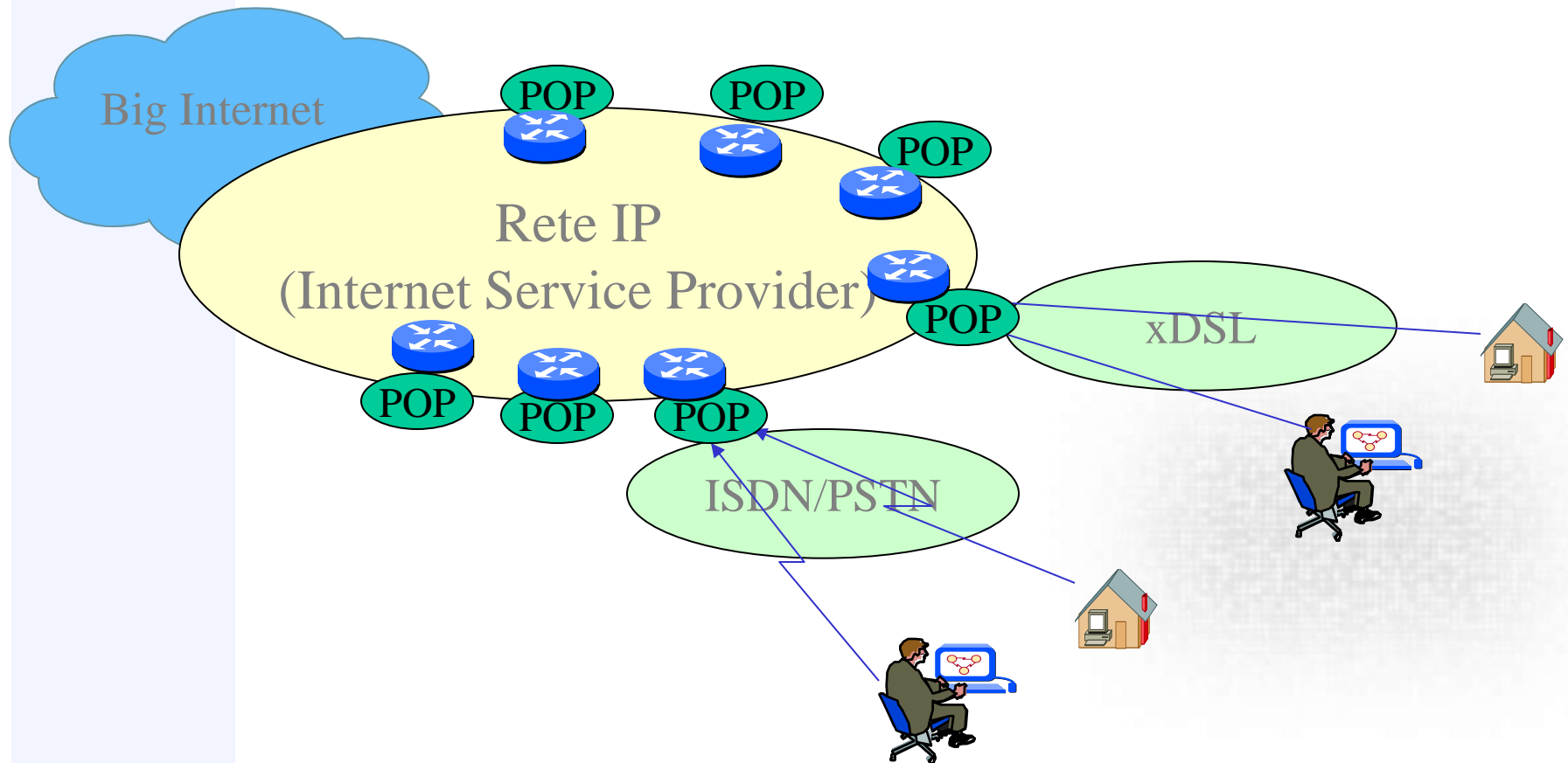
# Internet



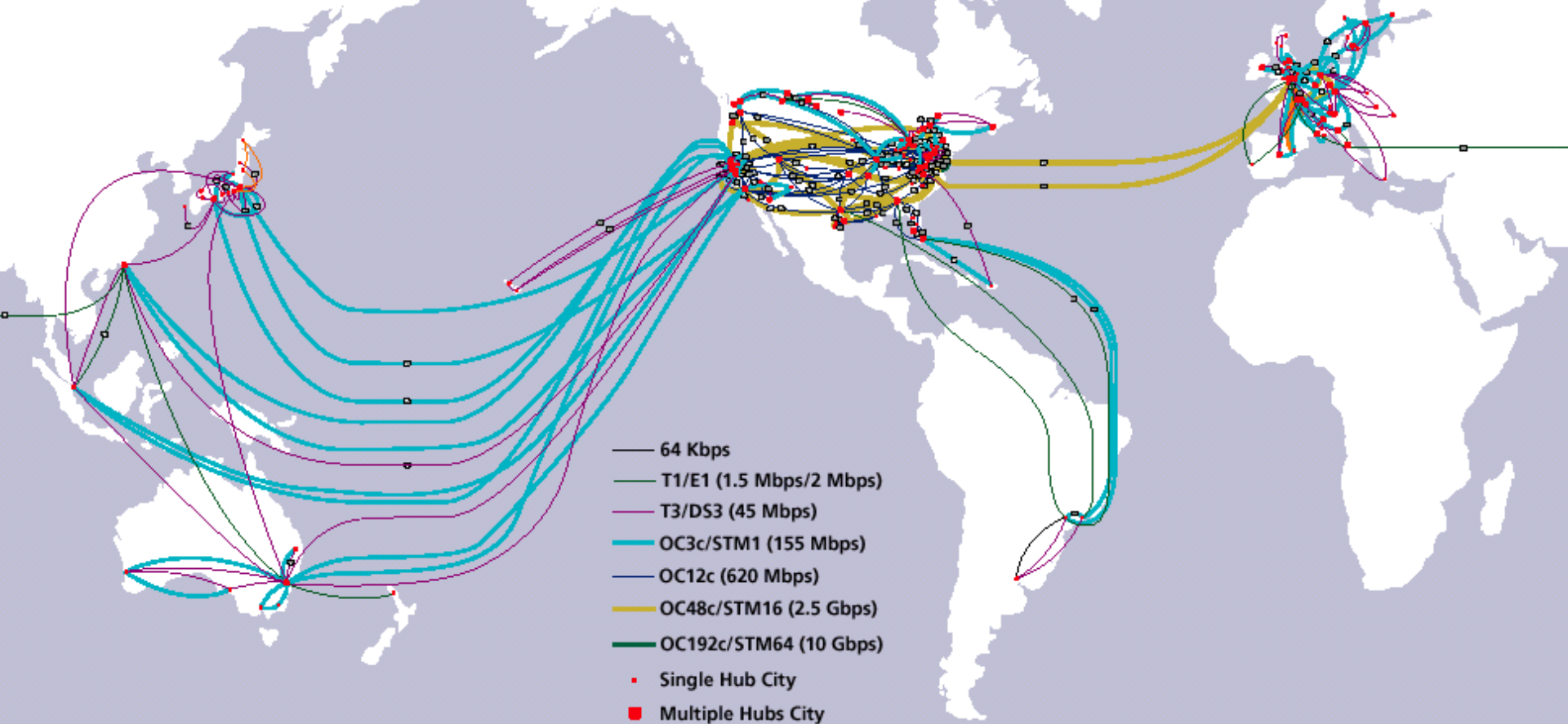
# Internet Architecture



# Internet Service Providers



# WorldCom's Global UUNET Internet network

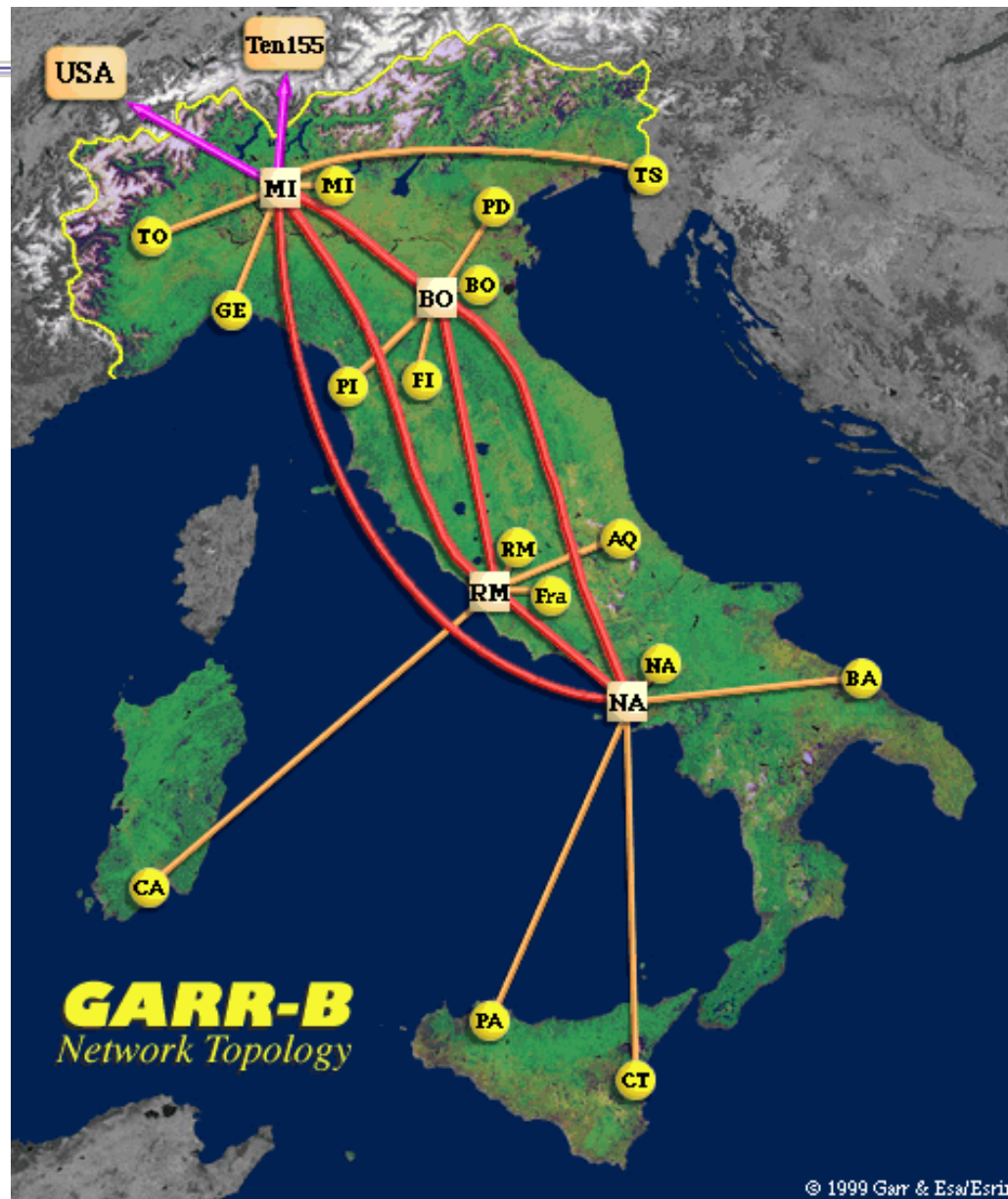


For more information see [www.uu.net/network/maps](http://www.uu.net/network/maps)

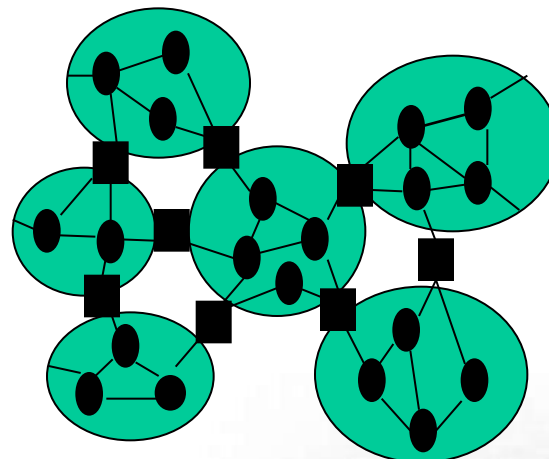
NB: UUNET also has infrastructure within individual countries, which is not shown on this map.

January 2001





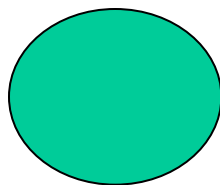
# Network Access Point (NAP) anche Neutral Access Point o Internet Exchange Point



Punto “neutrale” di scambio dati tra ISPs

Localizzato in aree metropolitane

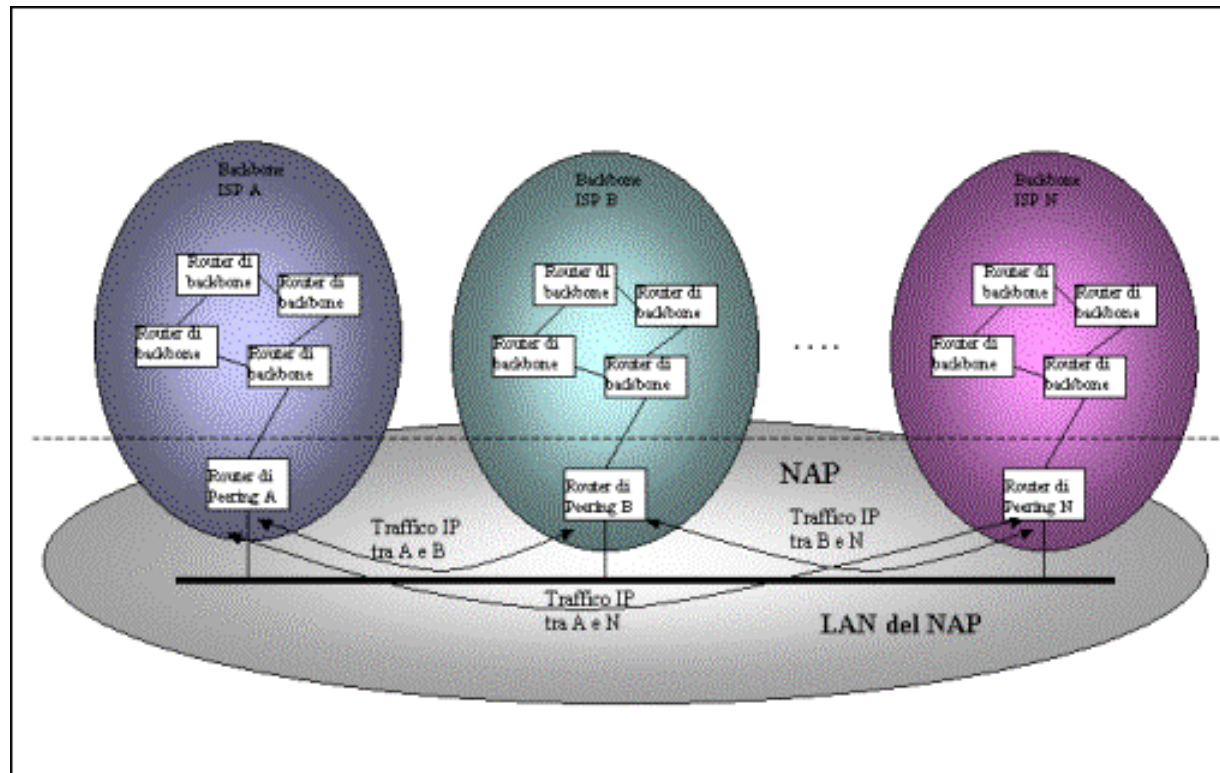
■ Network access point



ISP o NSP

# Network Access Point (NAP)

Lo scambio di dati tra diversi ISP avviene in base ai cosiddetti "accordi di peering"



# NAP in Italia

---

- Milano: MIX - Milan Internet eXchange
- Roma: NaMeX - Nautilus Mediterranean eXchange
- Torino: TOP-IX - TOrino Piemonte Internet eXchange
- Udine: FVG-IX - Friuli Venezia Giulia Internet eXchange
- Firenze: TIX - Tuscany Internet eXchange
- Padova: VSIX - Nap del Nord Est

# NAP NAMEX (Roma)

Il Nautilus Mediterranean eXchange point (NaMeX) è un punto d'interscambio e interconnessione, neutrale e senza fini di lucro, tra Internet Service Provider e operatori di rete nazionali ed internazionali.

NaMeX consente agli operatori di rete di usufruire di servizi per lo scambio di traffico IP attraverso peering pubblici e privati e realizzazione di circuiti fisici tra operatori.

## Organizzazione

### Consiglio direttivo:

- Riccardo de Sanctis - ANFoV (**Presidente**)
- Renato Brunetti - Unidata (**Vice Presidente**)
- Antonio Baldassarra - Seeweb
- Silvano Fraticelli - MC-link
- Alberto Maria Langellotti - Telecom Italia
- Danilo Lanzoni - Wind
- Stefano Merigliano - CINECA
- Giuliano Peritore - Panservice
- Rosario Pingaro - Convergenze

### Direttore generale:

- Maurizio Goretti

### Direttore tecnico:

- Francesco Ferreri

### Comitato tecnico:

- Antonio Baldassarra - Seeweb
- Prof. Giuseppe Di Battista - DIA, Università Roma Tre
- Silvano Fraticelli - MC-link
- Maurizio Goretti - CINECA
- Gabriella Paolini - GARR
- Luca Rea - FUB
- Giampaolo Rossini - Unidata
- Gianpaolo Scassellati - Wind
- Antonio Soldati - Telecom Italia

# Membri NAMAX

- 2006**
1. [Agora'](#)
  2. [CASPUR](#)
  3. [Cybernet](#)
  4. [GARR](#)
  5. [MClick](#)
  6. [Unidata](#)
  7. [InterBusiness](#)
  8. [Unisource](#)
  9. [Pronet](#)
  10. [Infostrada](#)
  11. [Wind](#)
  12. [Tiscali](#)
  13. [UUnet](#)
  14. [Cubecom](#)
  15. [Atlanet](#)
  16. [Galactica](#)
  17. [Postecom](#)
  18. [Edisontel](#)

**2015**

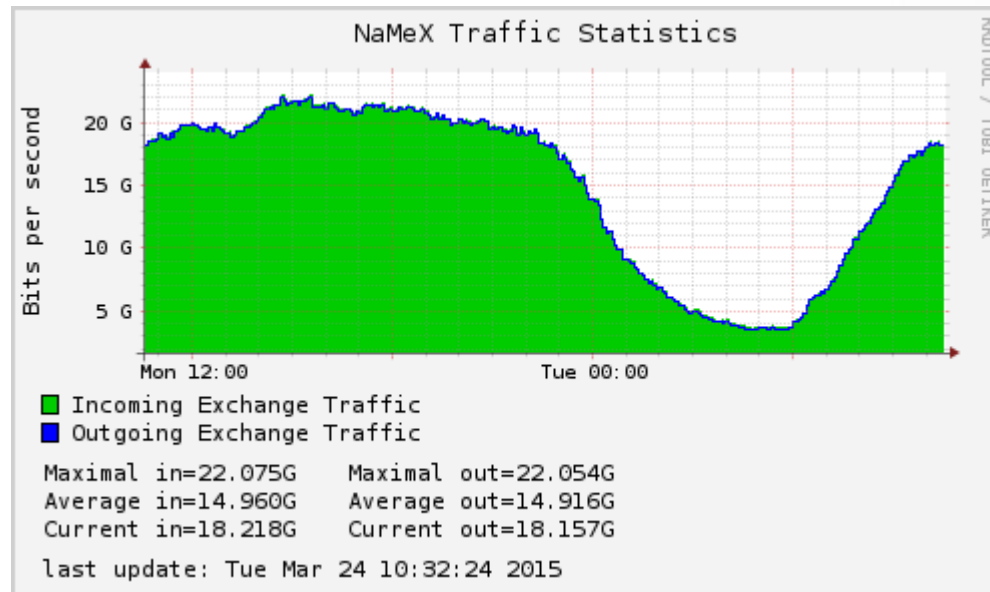
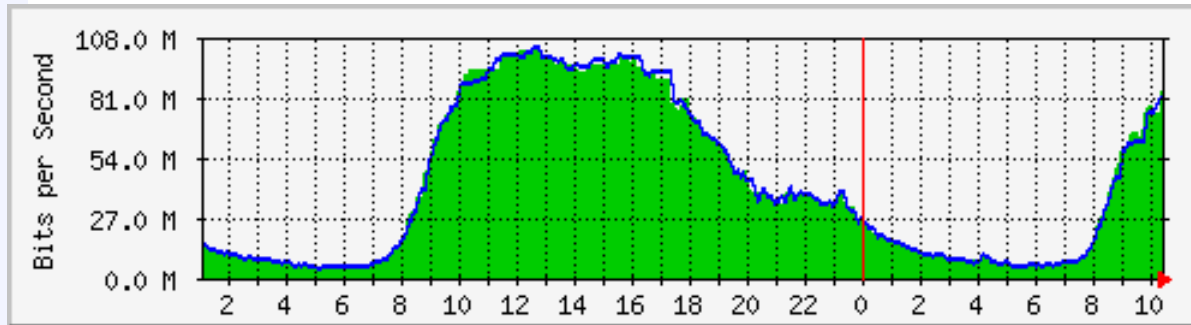
## Members

Total number of members: 59

Member info		Network	
Name	AS Number	Peering bandwidth	RS
<a href="#">ACI Informatica</a>	<a href="#">AS42515</a>	n/a	✗
<a href="#">Active Network</a>	<a href="#">AS197075</a>	200 Mbps	✓
<a href="#">Aqesci</a>	<a href="#">AS42463</a>	200 Mbps	✗
<a href="#">Akamai</a>	<a href="#">AS20940</a>	10 Gbps	✓
<a href="#">Almaviva</a>	<a href="#">AS29419</a>	200 Mbps	✓
<a href="#">Aruba</a>	<a href="#">AS31034</a>	2 Gbps	✗
<a href="#">BT Italia</a>	<a href="#">AS8968</a>	4 Gbps	✗
<a href="#">Caspur</a>	<a href="#">AS5397</a>	2 Gbps	✓
<a href="#">Clicom</a>	<a href="#">AS9104</a>	100 Mbps	✗
<a href="#">Clouditalia Communications</a>	<a href="#">AS15589</a>	2 Gbps	✓
<a href="#">Cogent</a>	<a href="#">AS174</a>	100 Mbps	✗
<a href="#">Colt Technology</a>	<a href="#">AS8220</a>	200 Mbps	✗
<a href="#">Convergenze</a>	<a href="#">AS39120</a>	2 Gbps	✓
<a href="#">E4A</a>	<a href="#">AS34695</a>	200 Mbps	✓
<a href="#">Engineering.IT</a>	<a href="#">AS21176</a>	2 Gbps	✓
<a href="#">Eurnetcity</a>	<a href="#">AS20794</a>	100 Mbps	✗
<a href="#">F-Root</a>	<a href="#">AS27320</a>	200 Mbps	✓
<a href="#">Fastnet</a>	<a href="#">AS8265</a>	1 Gbps	✓
<a href="#">Fastweb</a>	<a href="#">AS12874</a>	10 Gbps	✗
<a href="#">Foxtel</a>	<a href="#">AS56754</a>	n/a	✗
<a href="#">Frosinone Wireless</a>	<a href="#">AS50627</a>	200 Mbps	✓
<a href="#">FUB</a>	<a href="#">AS50112</a>	2 Gbps	✗
<a href="#">GARR</a>	<a href="#">AS137</a>	20 Gbps	✓
<a href="#">Google</a>	<a href="#">AS36040</a>	20 Gbps	✓
<a href="#">H3G</a>	<a href="#">AS24608</a>	10 Gbps	✓
<a href="#">Holy See</a>	<a href="#">AS8978</a>	2 Gbps	✓
<a href="#">Hurricane Electric</a>	<a href="#">AS6939</a>	n/a	✗
<a href="#">I.NET</a>	<a href="#">AS3313</a>	2 Gbps	✗
<a href="#">ICT Valle Umbra</a>	<a href="#">AS15605</a>	2 Gbps	✓
<a href="#">Infracom</a>	<a href="#">AS3302</a>	1 Gbps	✗



# Traffico Giornaliero NAMEX



# Google



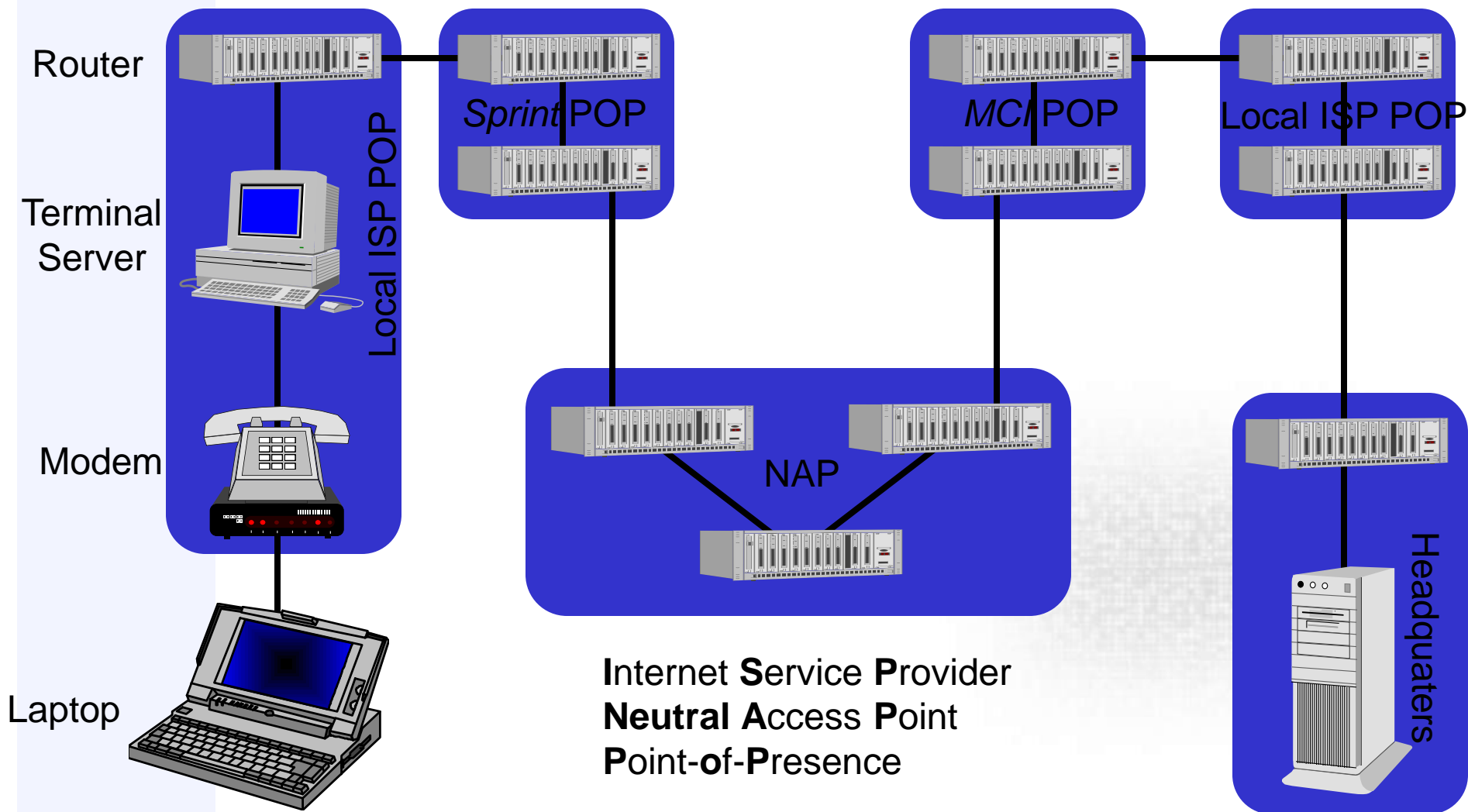
The company has essentially two huge networks: the one that connects users to Google services (Search, Gmail, YouTube, etc.) and another (internal) that connects Google data centers to each other.

Google is in control of scheduling internal traffic (bursty), but it faces difficulties in traffic engineering.

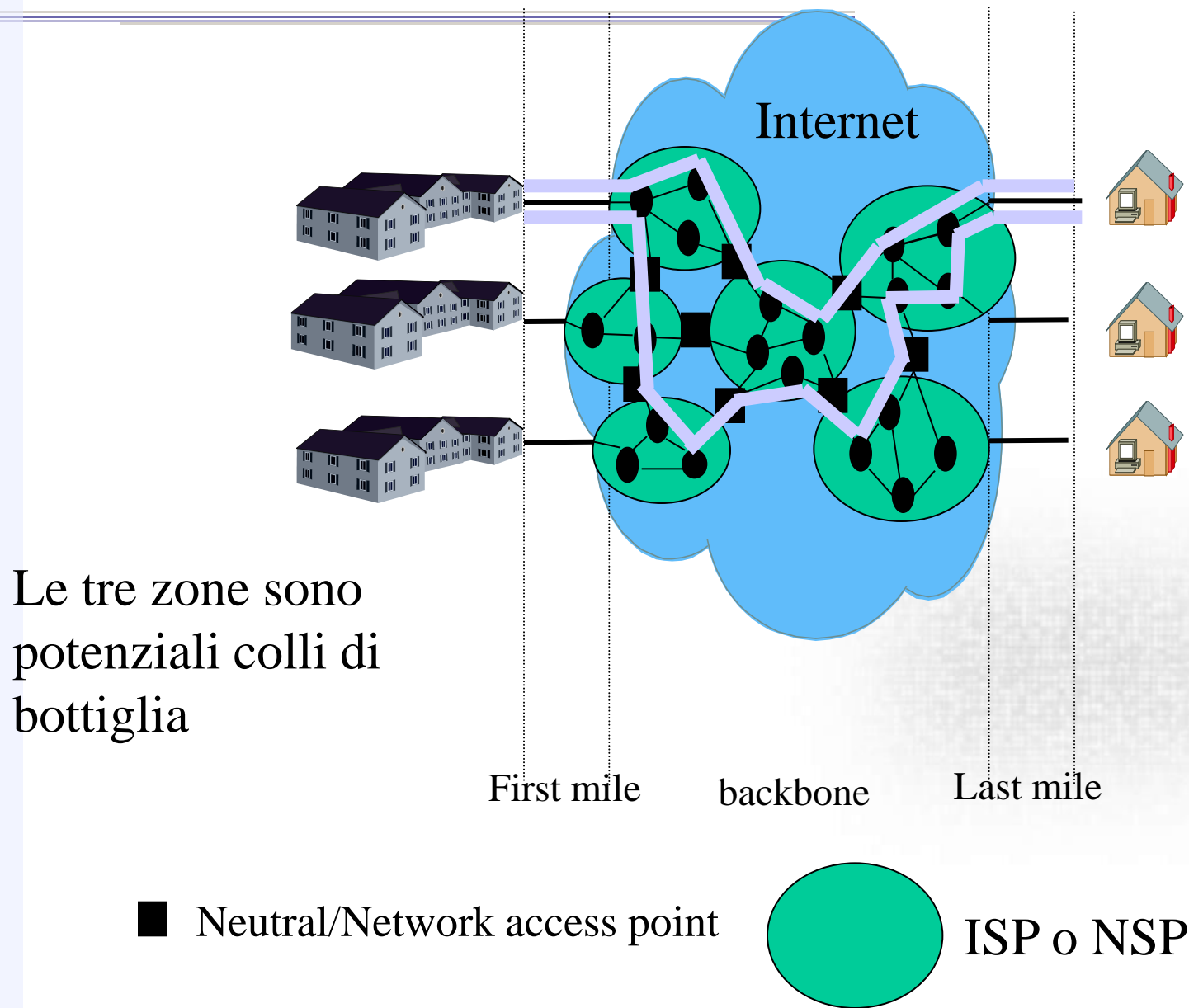
Often Google has to move many petabytes of data (indexes of the entire web, millions of backup copies of user Gmail) from one place to another.



# Comunicazione via ISPs, POPs e NAPs



# Architettura a tre livelli di Internet



# Architettura a tre livelli di Internet

Eliminare colli di bottiglia (soluzioni hardware)

- first mile, last mile -> aumentare la banda che connette al provider
- Backbone -> dipende dal miglioramento delle infrastrutture di rete dei singoli ISP (non controllabile dagli utenti finali)

Eliminare il collo di bottiglia di backbone (soluzione software)

- Content Delivery Networks. caching di pagine vicino a dove risiede l'utente completamente trasparente all'utente (e.g. AKAMAI). In questo modo si spera che l'utente possa accedervi con larga banda

Nota: idea di soluzione simile a quella della gerarchia di caching delle memorie nei SO

## Akamai's Global Platform

### ■ Akamai's Internet Platform

- 100,000+ servers
- 72 countries
- 1,500+ locations
- 1,000 networks

### ■ Ginormous Daily Traffic

- Carries 15-30% of the world's web traffic on any given day
- More than 1 trillion requests
- More than 30 petabytes
- 10 million+ concurrent video streams



# Il modello di comunicazione OSI

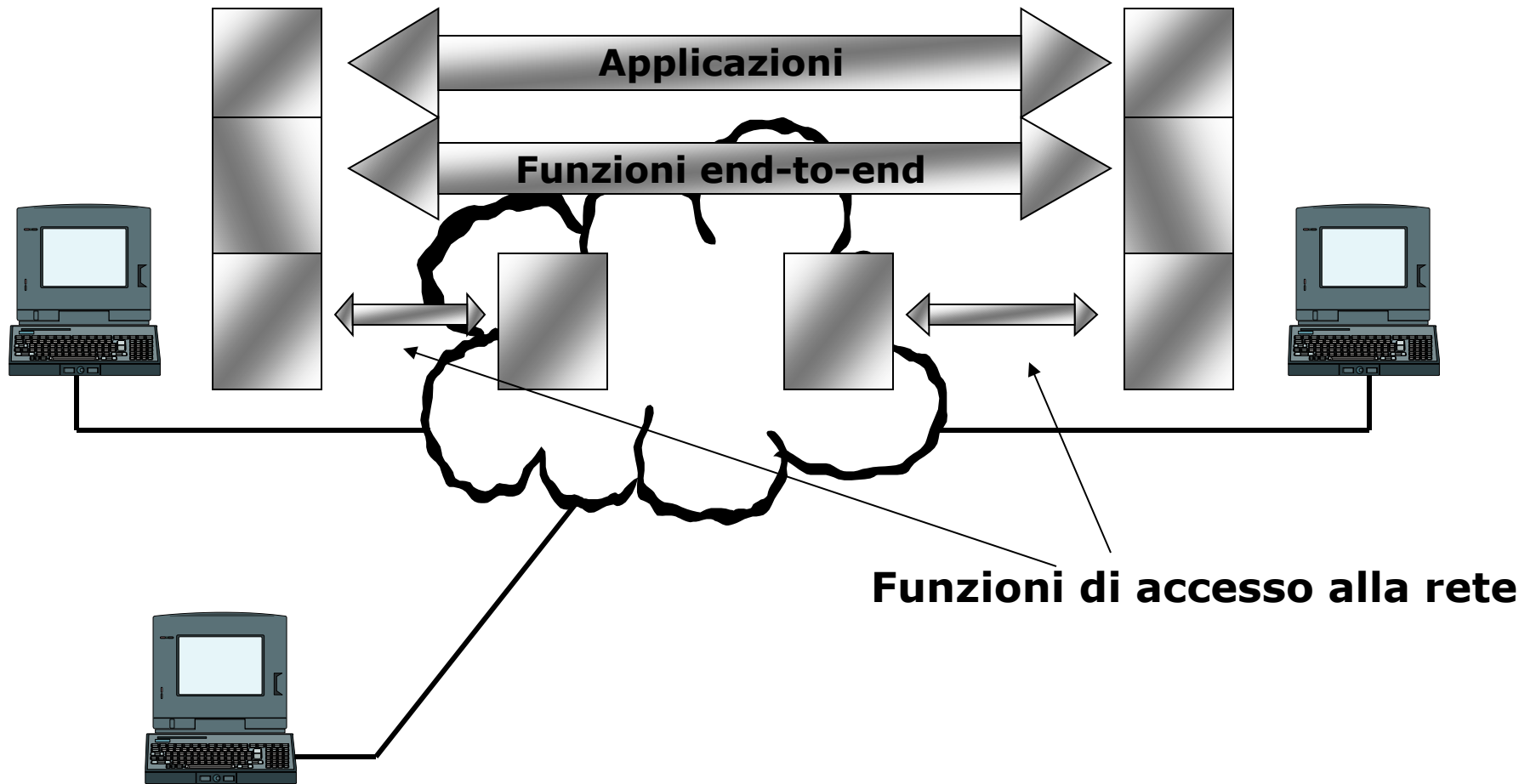
## ESEMPIO DI PROFILO DEI PROTOCOLLI PER IL PIANO UTENTE (commutazione di pacchetto)



## ESEMPIO DI PROFILO DEI PROTOCOLLI PER IL PIANO UTENTE (commutazione di circuito)



# Rete geografica di calcolatori



# Struttura a tre livelli di una rete di calcolatori

**Area Applicativa**

**Interoperabilità trasporto dell'informazione**

**Infrastruttura di trasporto dell'informazione**

# Struttura a tre livelli di una rete di calcolatori

**Area Applicativa**

**Interoperabilità trasporto dell'informazione**

**TRASPORTO**

**RETE**

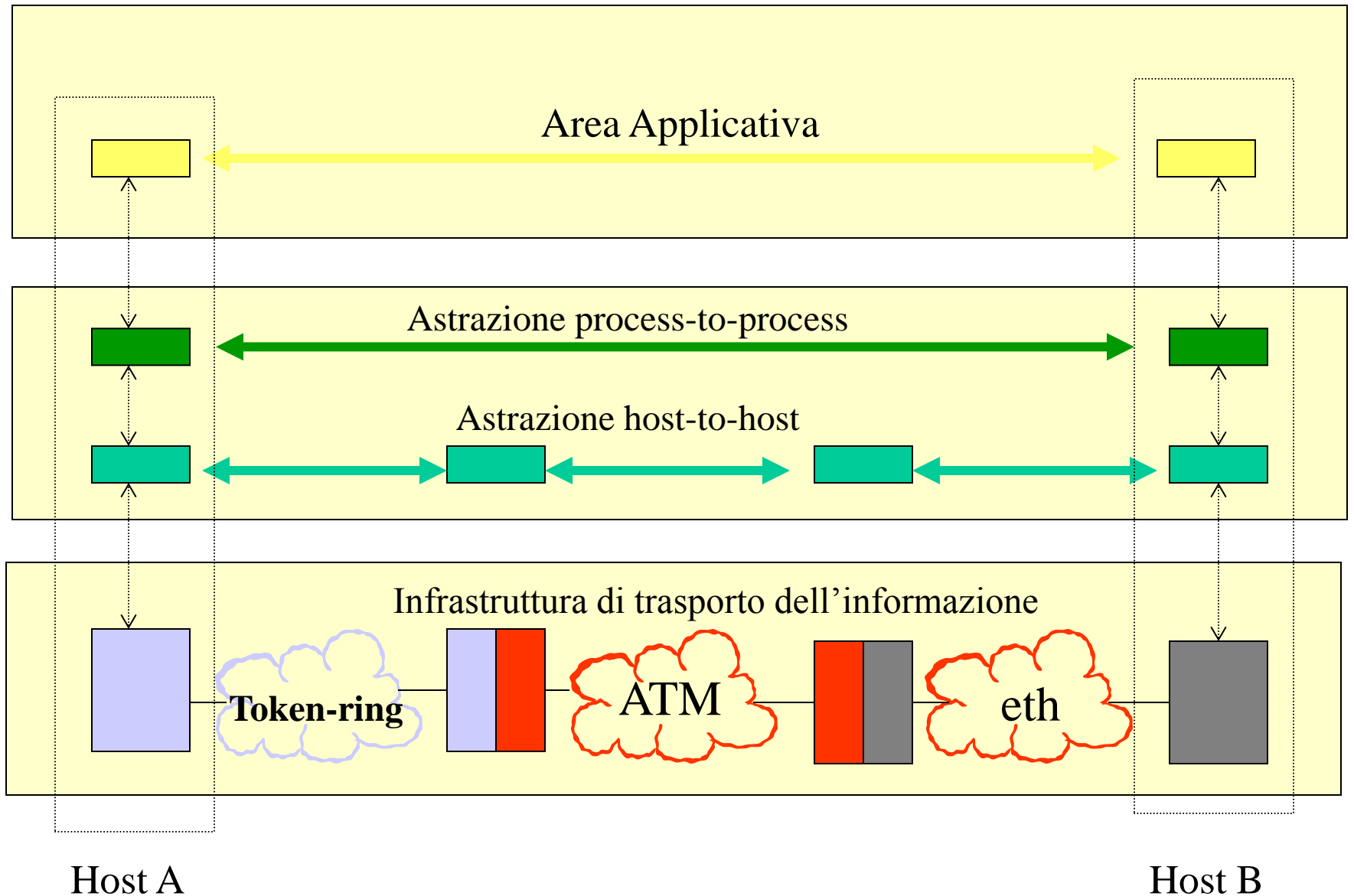
**Infrastruttura di trasporto dell'informazione**

**LINK**

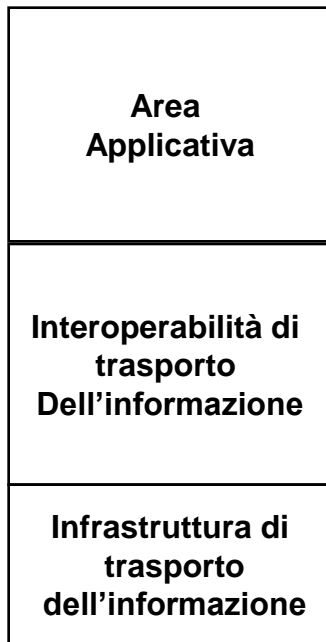
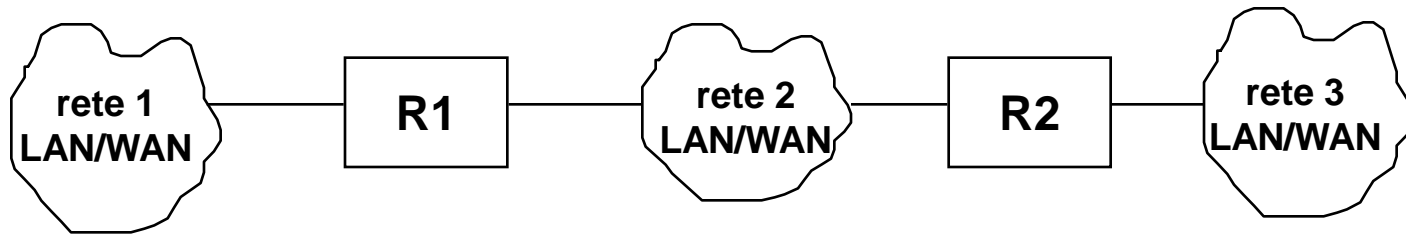
**FISICO**



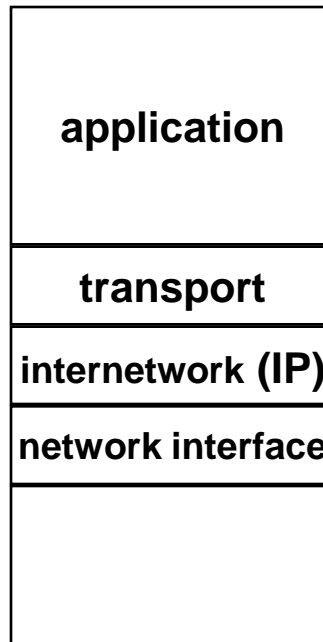
# Rete geografica di calcolatori



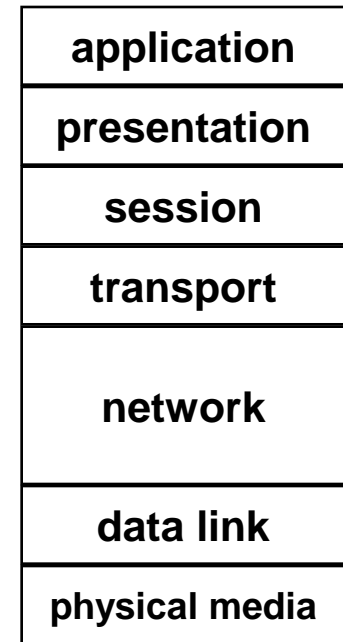
# Interoperabilità Trasporto dell'informazione: Internet



**Struttura a  
tre livelli**

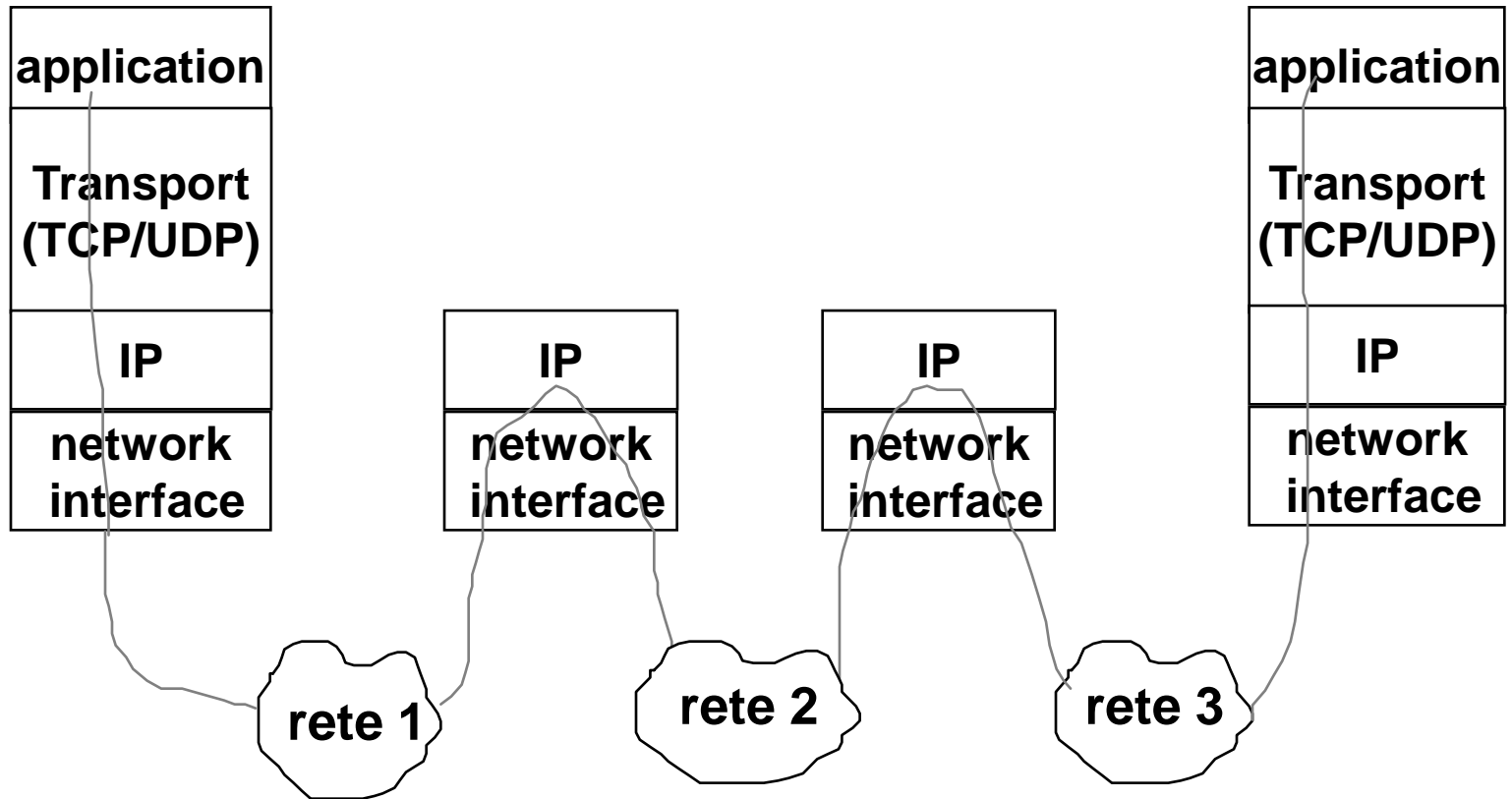


**Internet**



**OSI**

# L'ARCHITETTURA TCP/IP E LA RETE INTERNET

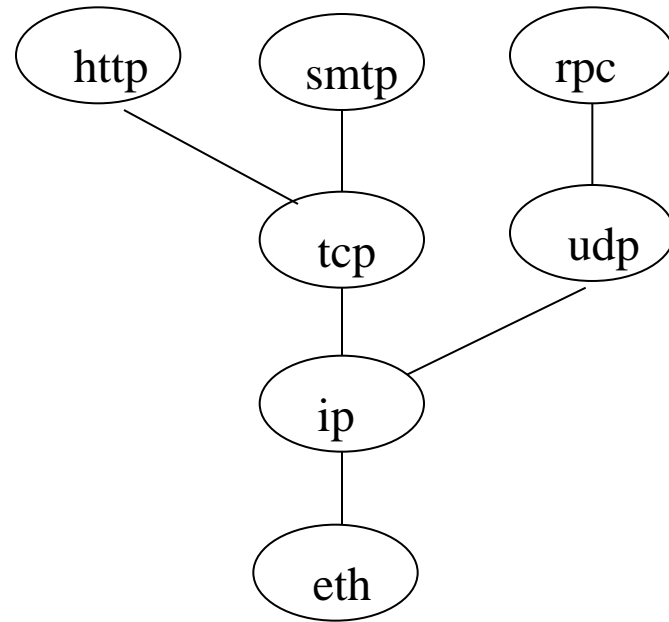


# Protocol Stack: esempi

http= hyper text tranfer protocol

smtp= simple mail transfer protocol

Rpc= remote procedure call



# Esempi di problematiche comuni

---

Area Applicativa

**Indirizzamento**

**Routing**

**Frammentazione/Riassemblaggio**

Interoperabilità trasporto dell'informazione

**Indirizzamento**

**Routing**

**Frammentazione/Riassemblaggio**

Infrastruttura di trasporto dell'informazione

**Indirizzamento**

**Routing**

**Frammentazione/Riassemblaggio**

# Esempi di problematiche comuni: Indirizzamento

Area Applicativa

**Indirizzamento DNS “www.uniroma1.it”**



Interoperabilità trasporto  
dell'informazione

**Indirizzamento IP “151.100.16.1”**

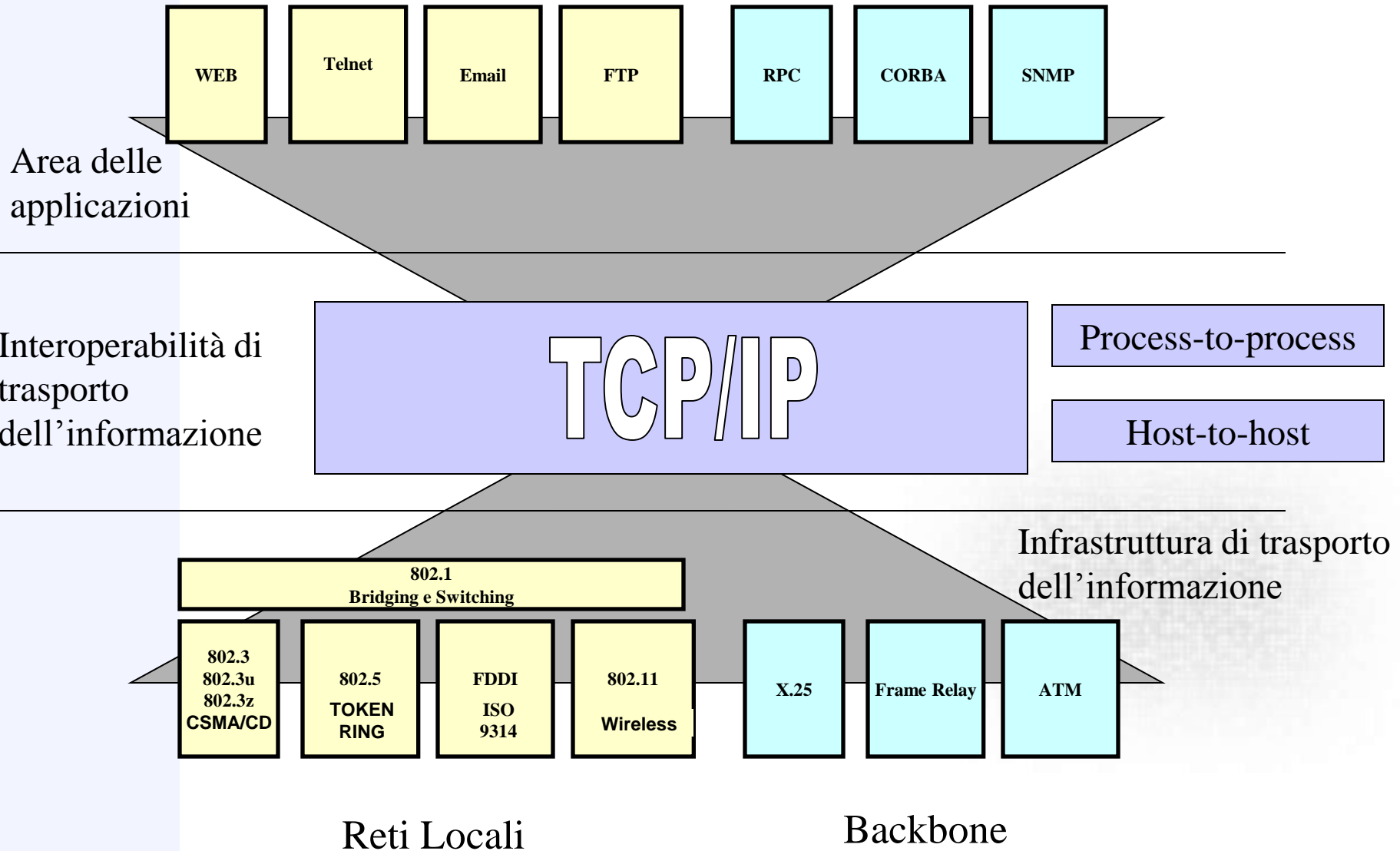


Infrastruttura di trasporto  
dell'informazione

**Indirizzamento MAC “ABC123578ABB”**

## Applicazioni di base

## Supporto per interoperabilità applicativa



# Richiami di TCP/IP



# Il protocollo IP

- IP e' una grande coperta che nasconde ai protocolli sovrastanti tutte le disomogeneità della infrastruttura di trasporto dell'informazione
- Per far questo necessità di due funzionalità di base:
  - Indirizzamento di rete (indirizzi omogenei a dispetto della rete fisica sottostante)
  - Instradamento dei pacchetti (Routing) (capacità di inviare pacchetti da un host ad un altro utilizzando gli indirizzi definiti al punto precedente)

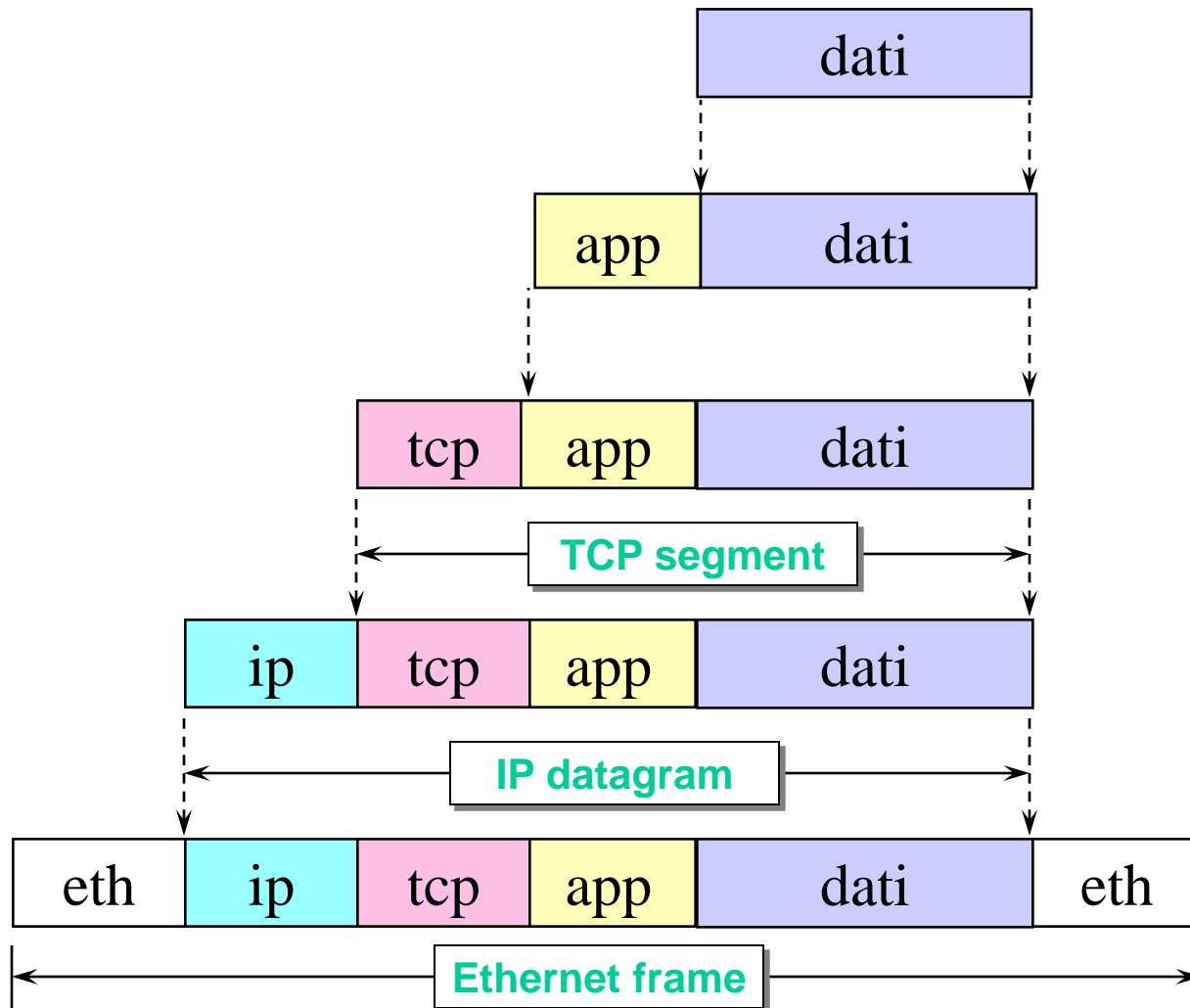
## Proprietà di IP

- Senza connessione (datagram based)
- Consegna Best effort
  - I pacchetti possono perdersi
  - I pacchetti possono essere consegnati non in sequenza
  - I pacchetti possono essere duplicati
  - I pacchetti possono subire ritardi arbitrari

## Servizi di compatibilità con l'hardware sottostante

- Frammentazione e riassemblaggio
- Corrispondenza con gli indirizzi dei livelli sottostanti (ARP)

# Il protocollo IP



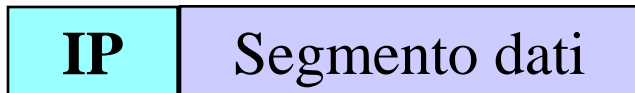
# Il protocollo IP

## *In Trasmissione, IP*

- riceve il segmento dati dal livello di trasporto

Segmento dati

- inserisce header e crea datagram



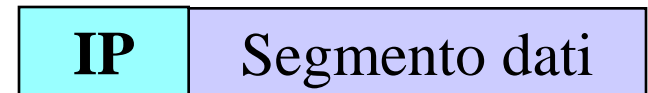
- applica l'algoritmo di routing
- invia i dati verso l'opportuna interfaccia di rete

## *In Ricezione, IP*

- consegna il segmento al protocollo di trasporto individuato

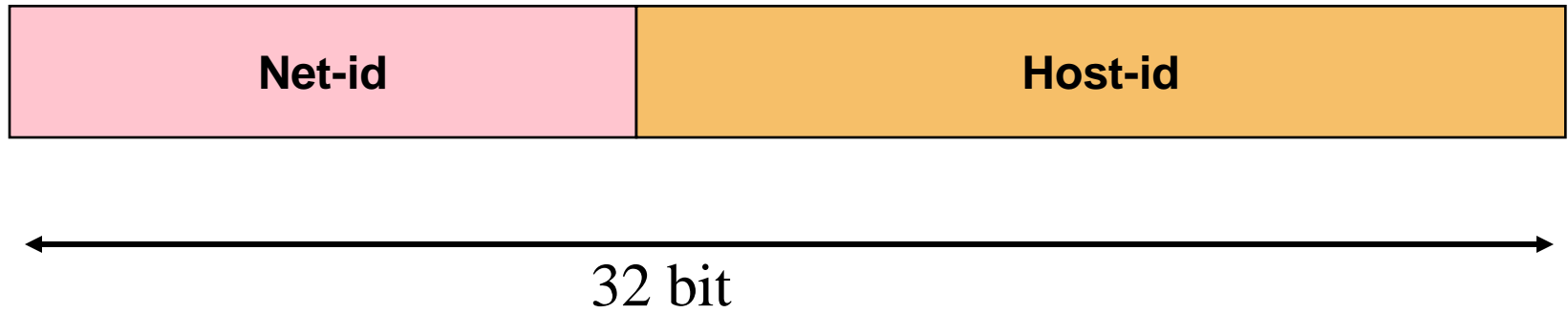
Segmento dati

- se sono dati locali, individua il protocollo di trasporto, elimina l'intestazione



- verifica la validità del datagram e l'indirizzo IP
- riceve i dati dalla interfaccia di rete

# Indirizzamento



# Classi di indirizzi

**Classe A** (0.0.0.0 - 127.255.255.255)

127.0.0.0 riservato



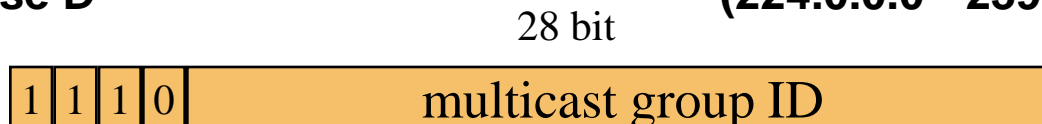
**Classe B** (128.0.0.0 - 191.255.255.255)



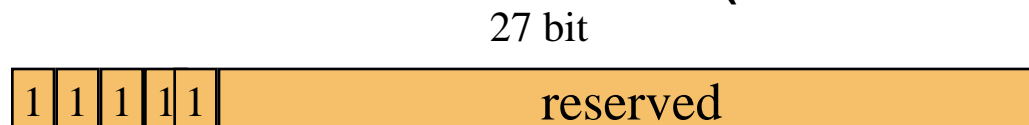
**Classe C** (192.0.0.0 - 223.255.255.255)



**Classe D** (224.0.0.0 - 239.255.255.255)



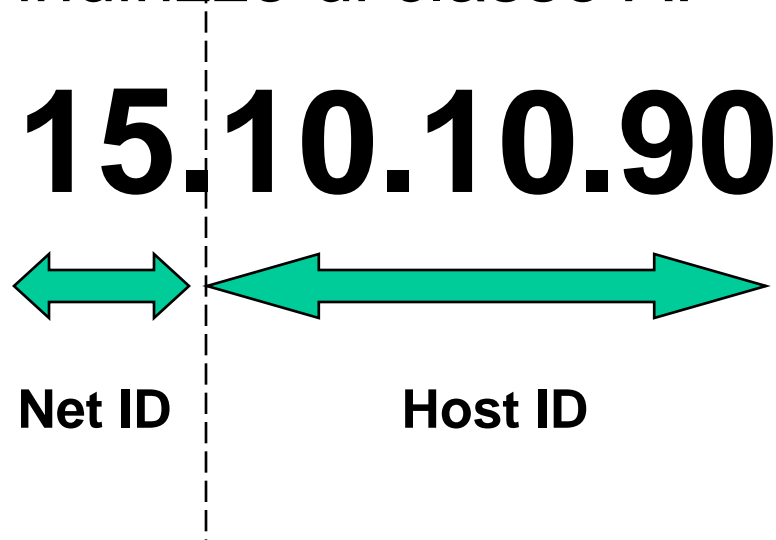
**Classe E** (240.0.0.0 - 255.255.255.254)



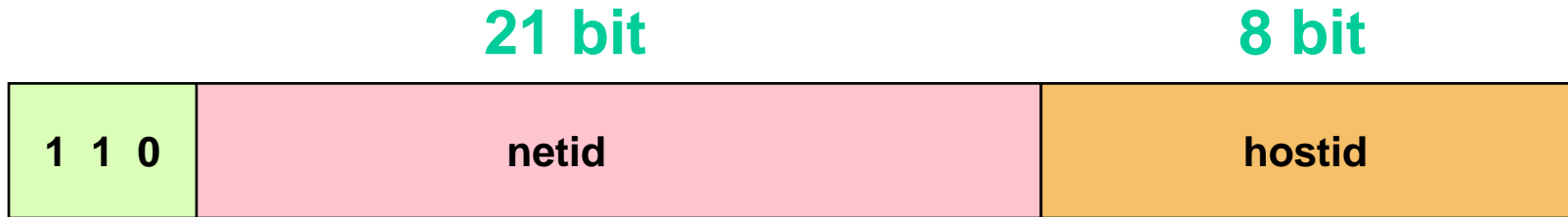
# Indirizzi di classe A



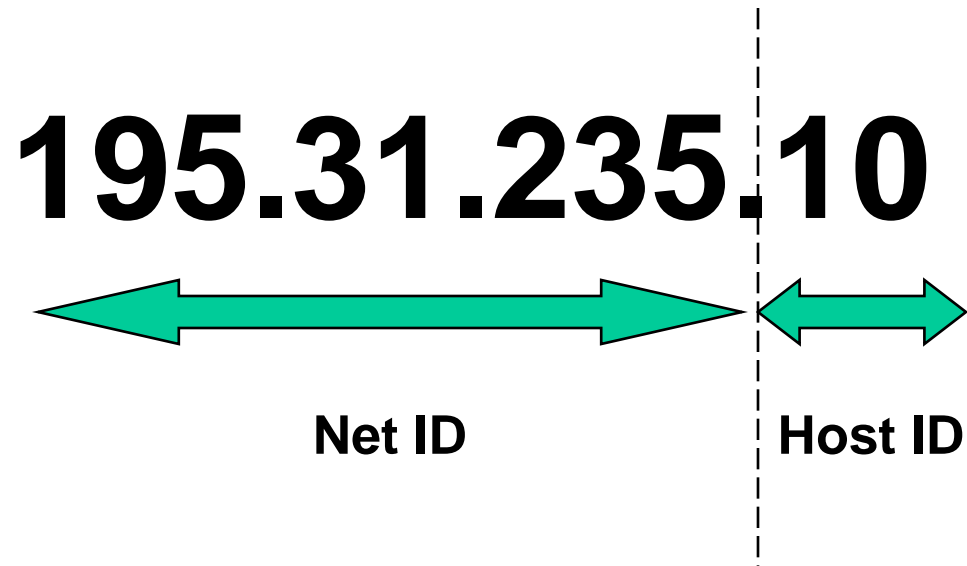
- Esempio di indirizzo di classe A:



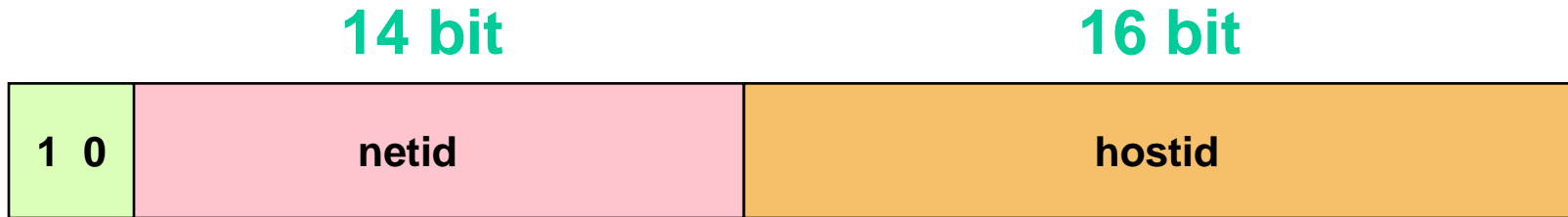
# Indirizzi di classe C



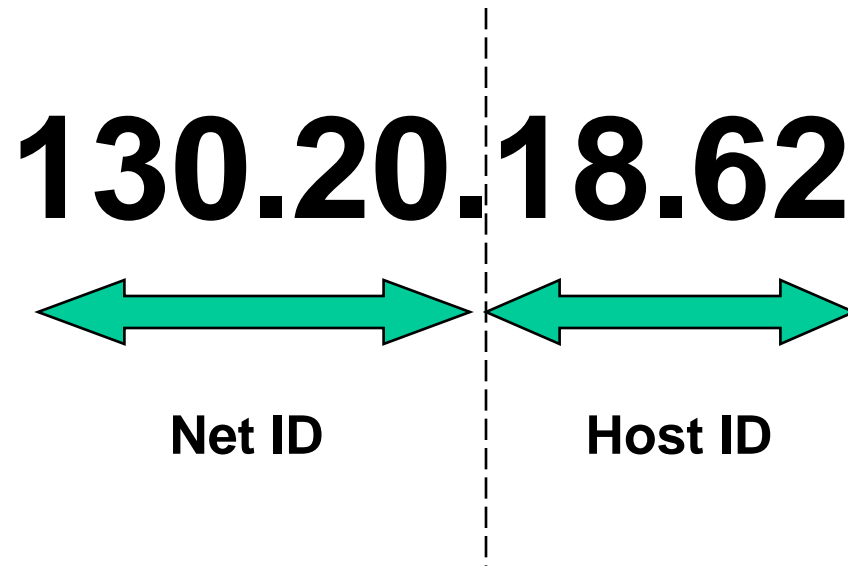
Esempio di indirizzo di classe C:



# Indirizzi di classe B

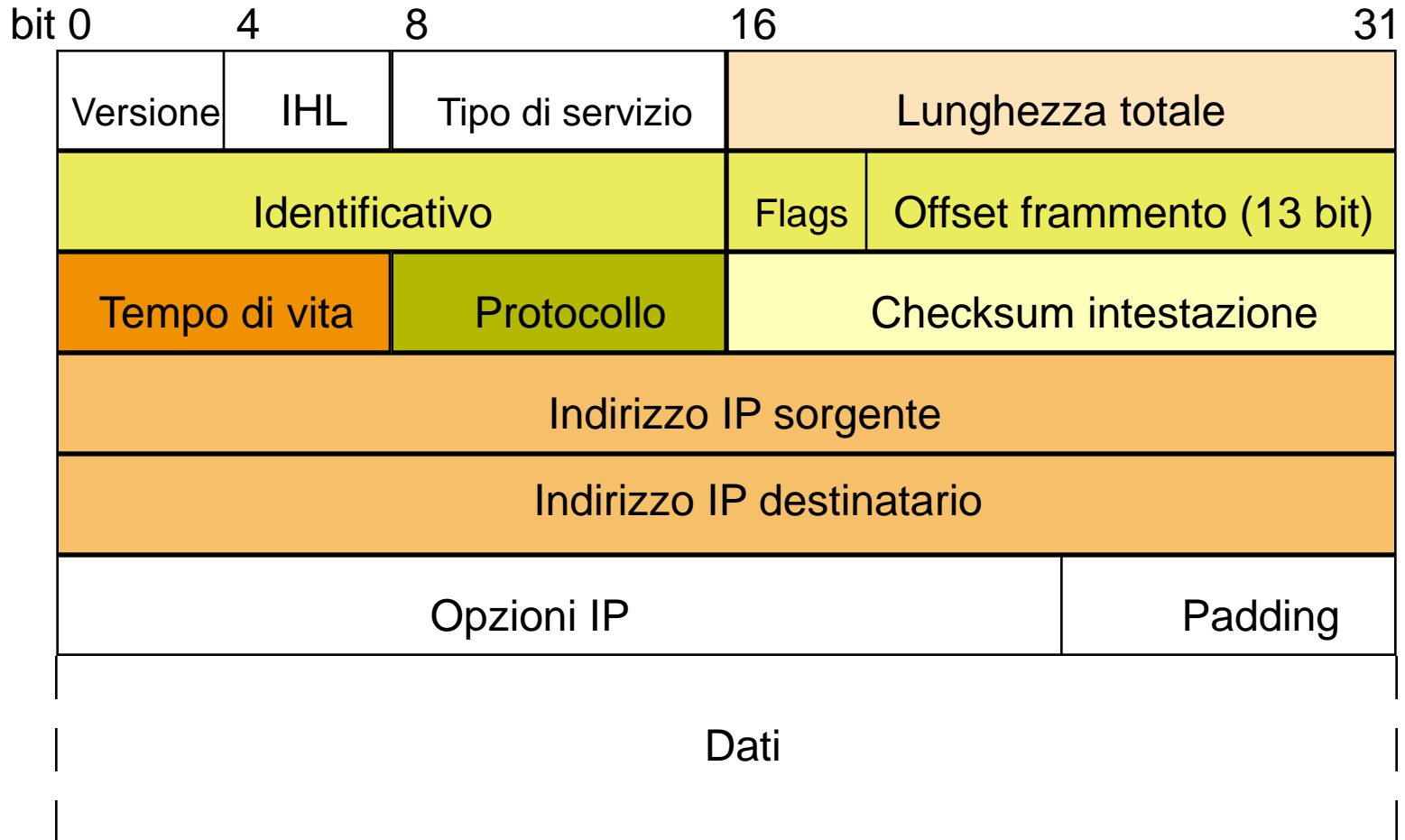


Esempio di indirizzo di classe B:

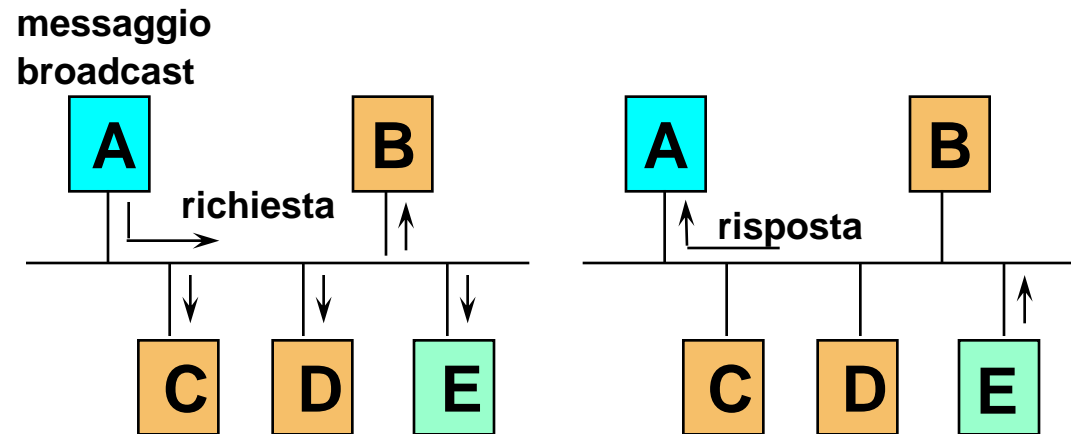




# Il protocollo IP

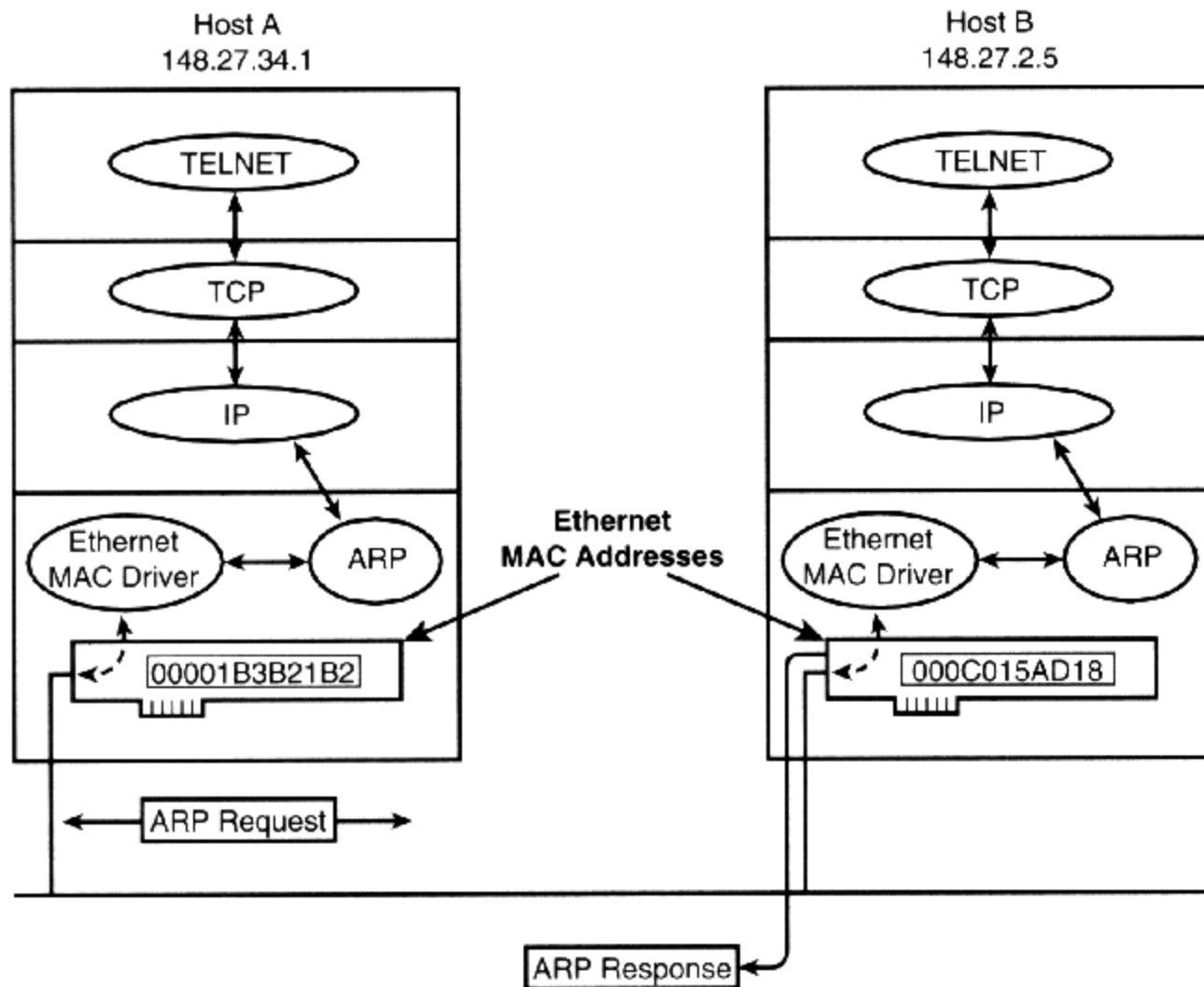


# Address Resolution Protocol: ARP

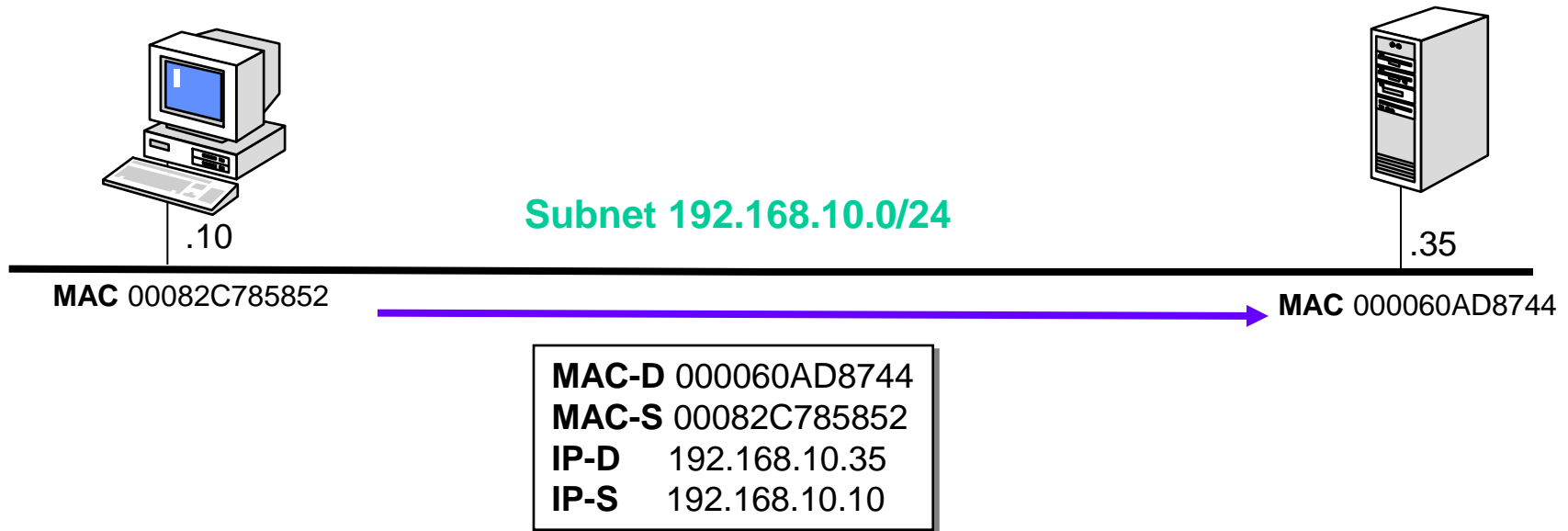


**FIGURE 4.9.**

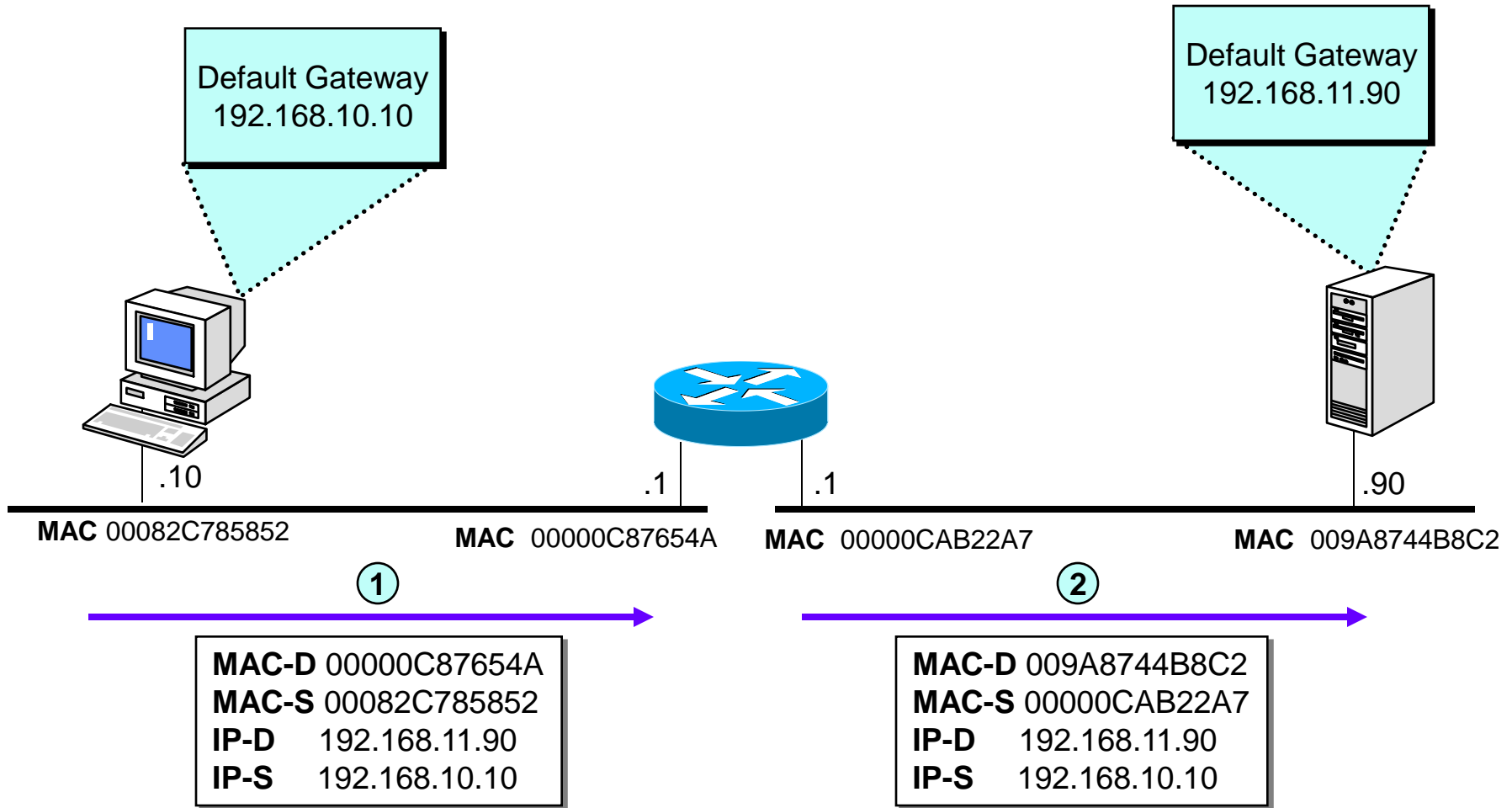
*Resolution of an IP address into its MAC address using ARP.*



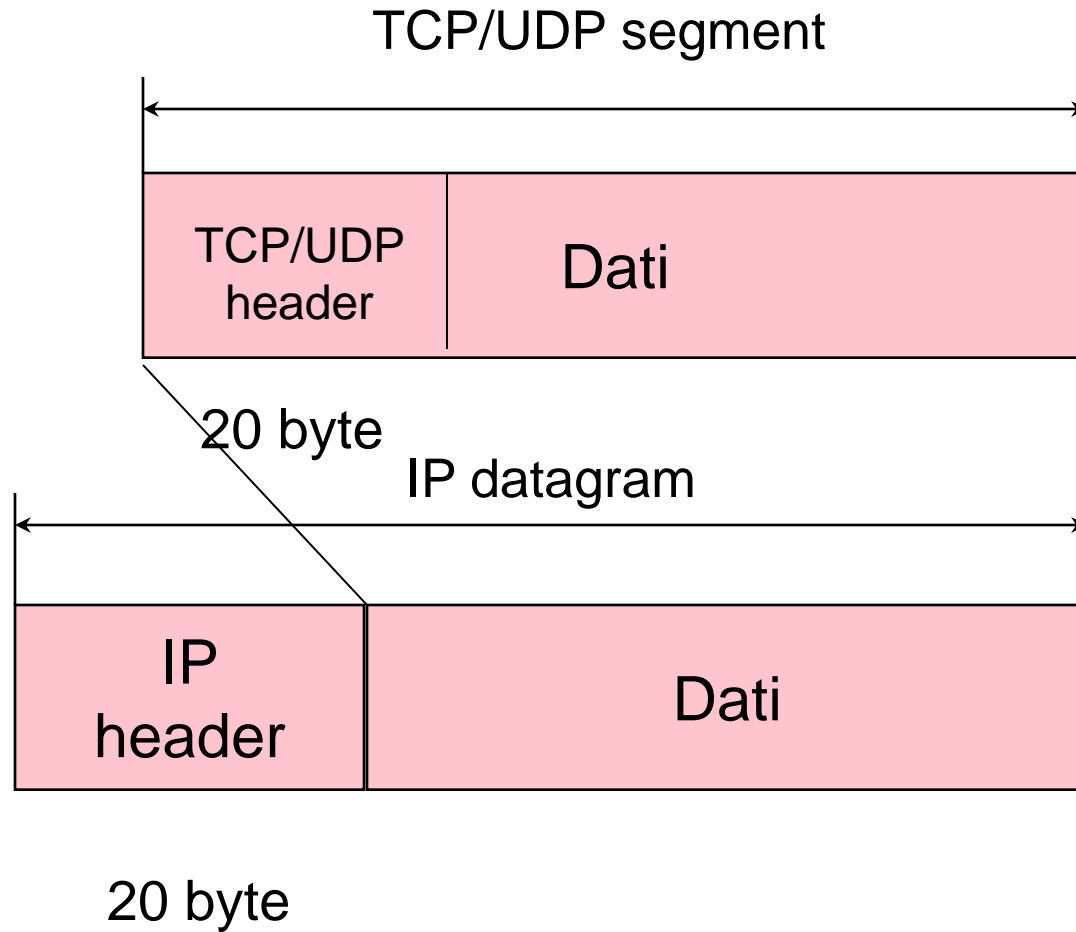
# Forwarding diretto: esempio



# Forwarding indiretto: esempio



# Strato di Trasporto



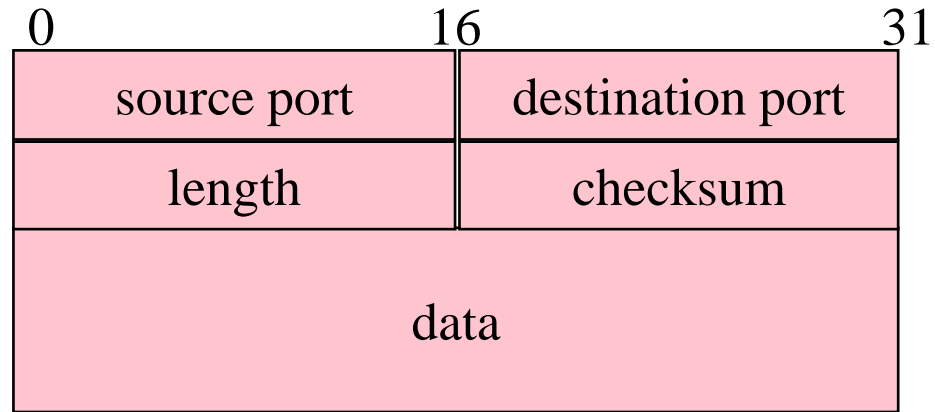
# Strato di Trasporto



# Strato di Trasporto: UDP

Il pacchetto UDP viene imbustato in IP ed indirizzato con il campo protocol pari a 17.

L'intestazione di UDP è lunga 8 byte



**port number**, sorgente e destinazione, servono a moltiplicare, su una connessione tra due macchine, diverse sessioni e individuano i protocolli di livello superiore;

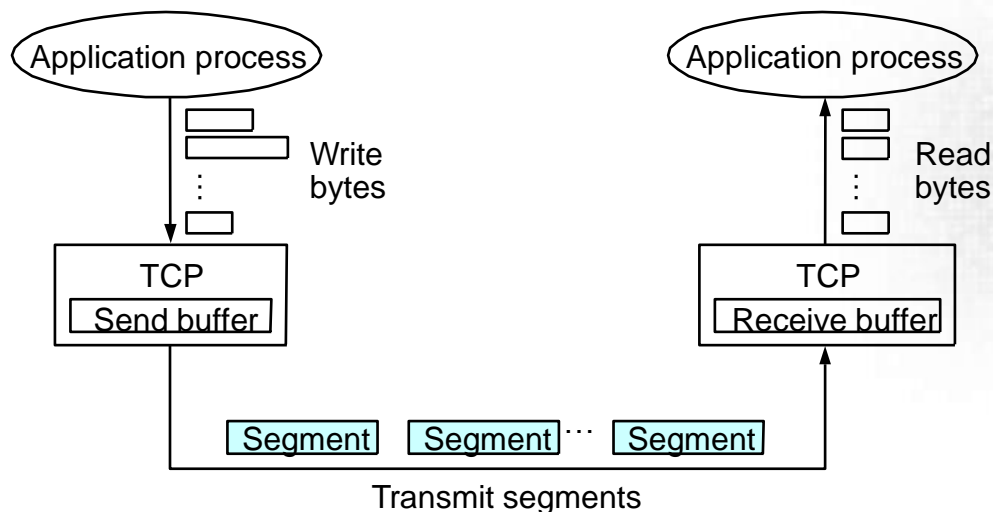
**length**, è la lunghezza in byte del pacchetto UDP, header e dati; il minimo valore per questo campo è di 8 byte, quando la parte dati è vuota; questa informazione è ridondante perché nell'intestazione IP è presente il campo length, relativo alla lunghezza di tutto il pacchetto IP; visto che l'intestazione UDP ha una lunghezza fissa di 8 byte, la lunghezza della parte dati potrebbe essere ricavata sottraendo al contenuto del campo length dell'header IP 8 byte;

**checksum**, campo per il controllo di errore, che copre tutto il pacchetto UDP, header e dati; in realtà oltre al pacchetto UDP, il checksum è applicato anche ad una parte dell'intestazione IP, composta tra l'altro dagli indirizzi IP sorgente e destinazione e dal campo protocol, detta UDP-pseudo-header.

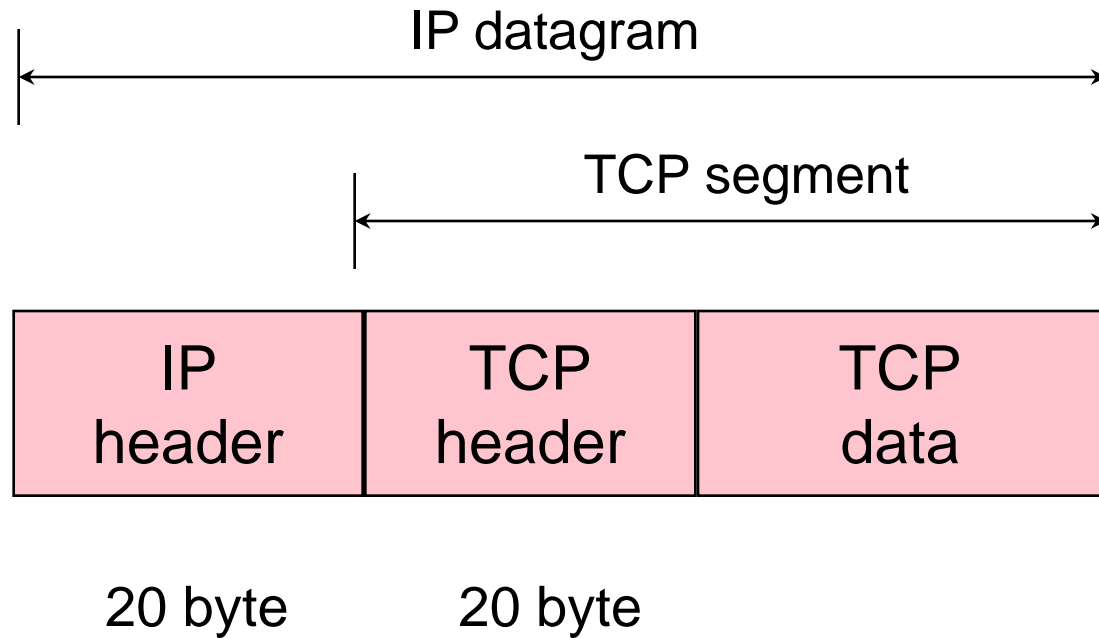


# TCP Overview

- Connection-oriented
- Byte-stream
  - app writes bytes
  - TCP sends *segments*
  - app reads bytes
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network

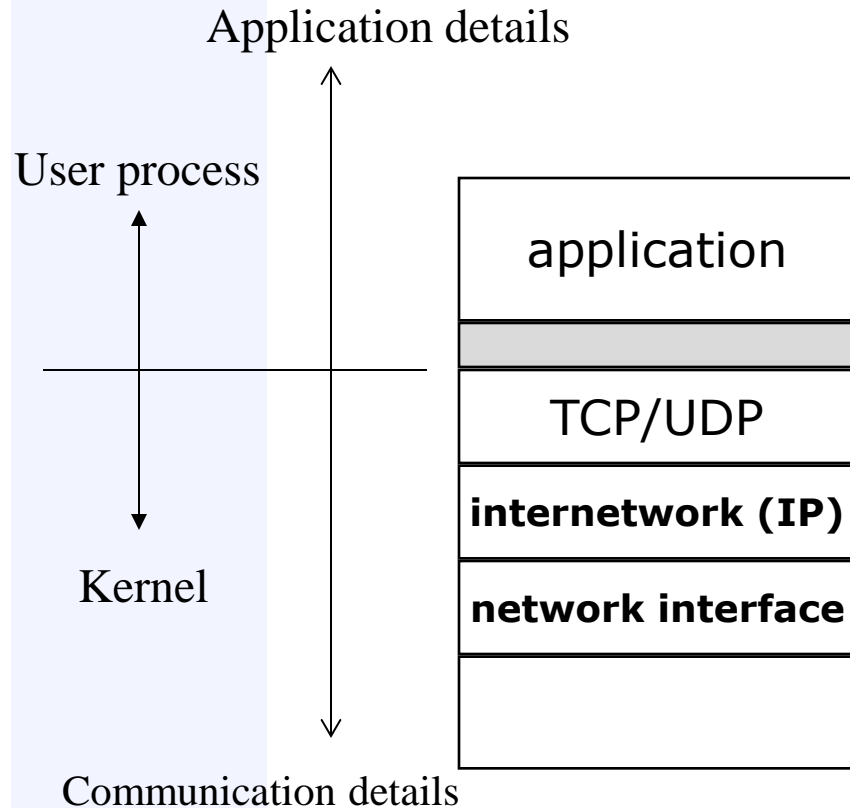


# Strato di Trasporto: TCP



# Sockets

# Sockets

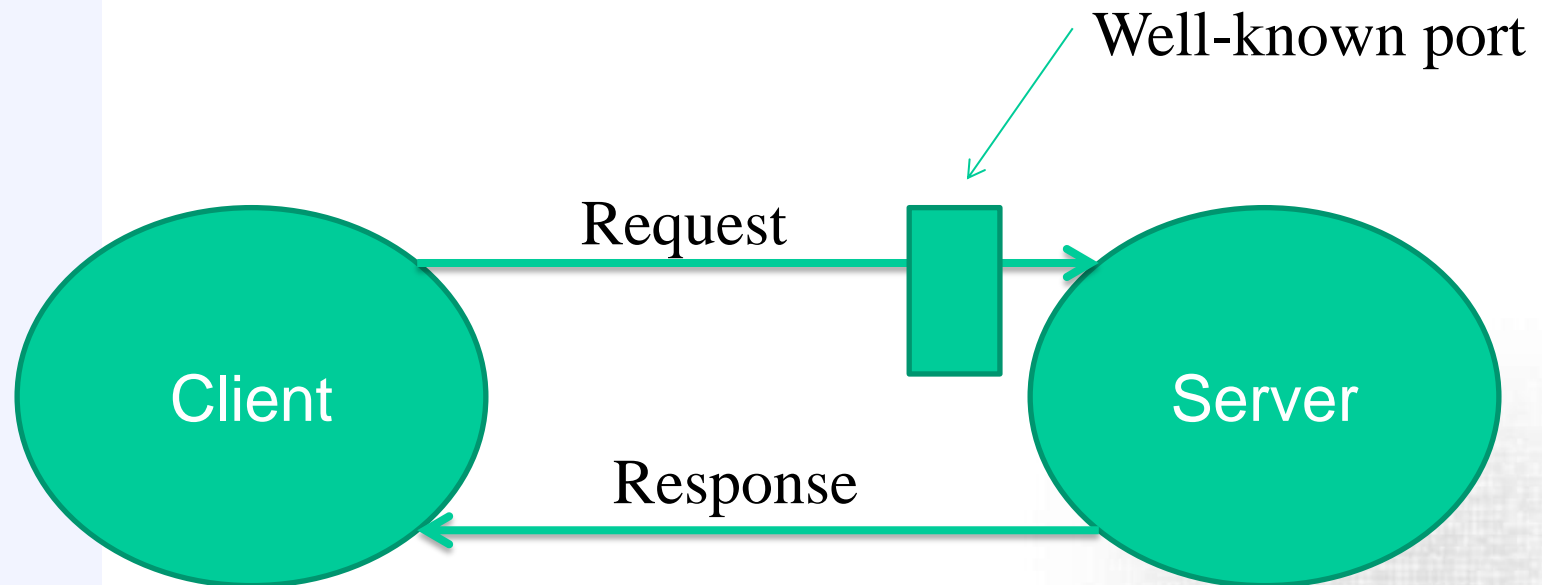


**Interfaccia socket  
TLI (Transport Layer Interface)**

# Socket Basics

- An end-point for a IP network connection
  - what the application layer “plugs into”
  - programmer cares about Application Programming Interface (API)
- End point determined by two things:
  - Host address: IP address is Network Layer
  - Port number: is Transport Layer
- Two end-points determine a connection: socket pair
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1500
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1499

# Client-Server Approach

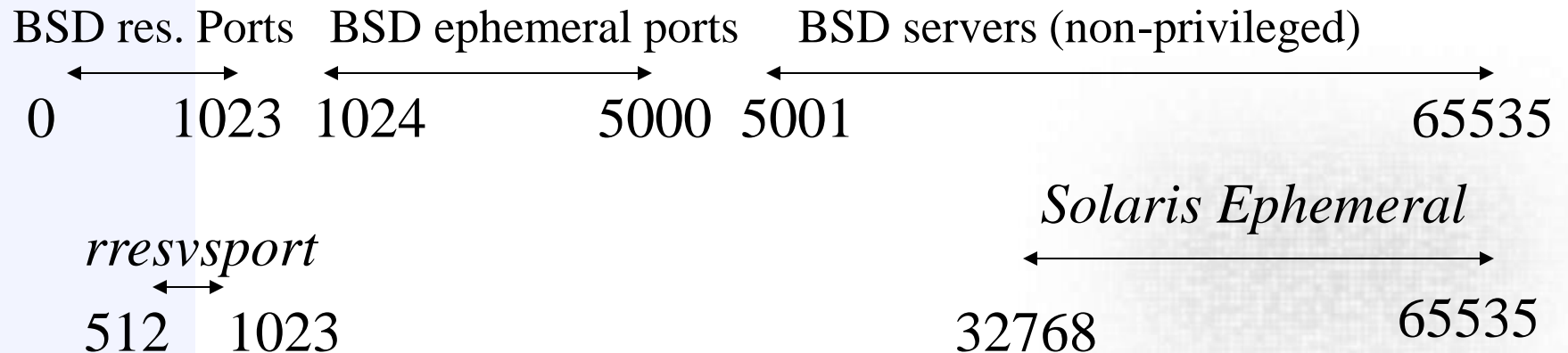


# Ports

---

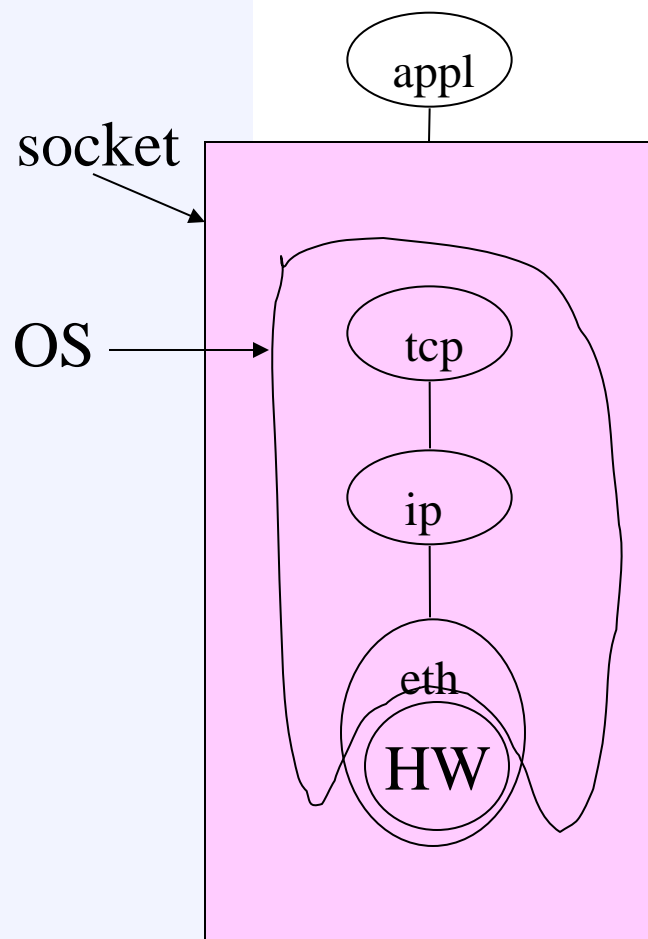
- Numbers (vary in BSD, Solaris, Linux):
  - 0-1023 “reserved”, must be root
  - 1024 - 5000 “ephemeral” (short-lived ports assigned automatically by the OS to clients)
  - however, many systems allow > 3977 ephemeral ports due to the number of increasing handled by a single PC. (Sun Solaris provides 30,000 in the last portion of BSD non-privileged)
- Well-known, reserved services /etc/services:
  - ftp 21/tcp
  - telnet 23/tcp
  - finger 79/tcp
  - snmp 161/udp
- Several client program needs to be a server at the same time (rlogin, rsh) as a part of the client-server authentication. These clients call the library function rresvport to create a socket and to assign an unused port in the range 512-1024.

# Ports





# Sockets and the OS



- User sees “descriptor”, integer index
  - like: `FILE *`, or file index from `open()`
  - returned by `socket()` call (more later)

# Socket Address Structure

```
struct in_addr {  
    in_addr_t s_addr;           /* 32-bit IPv4 addresses */  
};  
  
struct sockaddr_in {  
    uint8_t      sin_len;       /* length of structure */  
    sa_family_t  sin_family;    /* AF_INET */  
    in_port_t    sin_port;      /* TCP/UDP Port num */  
    struct in_addr sin_addr;     /* IPv4 address (above) */  
    char sin_zero[8];           /* unused */  
}
```

- Use **bzero()** before using **sockaddr**

# Addresses and Sockets

- Structure to hold address information
- Functions pass address from user to OS
  - `bind()`
  - `connect()`
  - `sendto()`
- Functions pass address from OS to user
  - `accept()`
  - `recvfrom()`

# TCP Client-Server

**Server**

socket()

bind()

listen()

“well-known”  
port

accept()

*(Block until connection)*

recv()

send()

recv()

close()

**Client**

socket()

connect()

send()

recv()

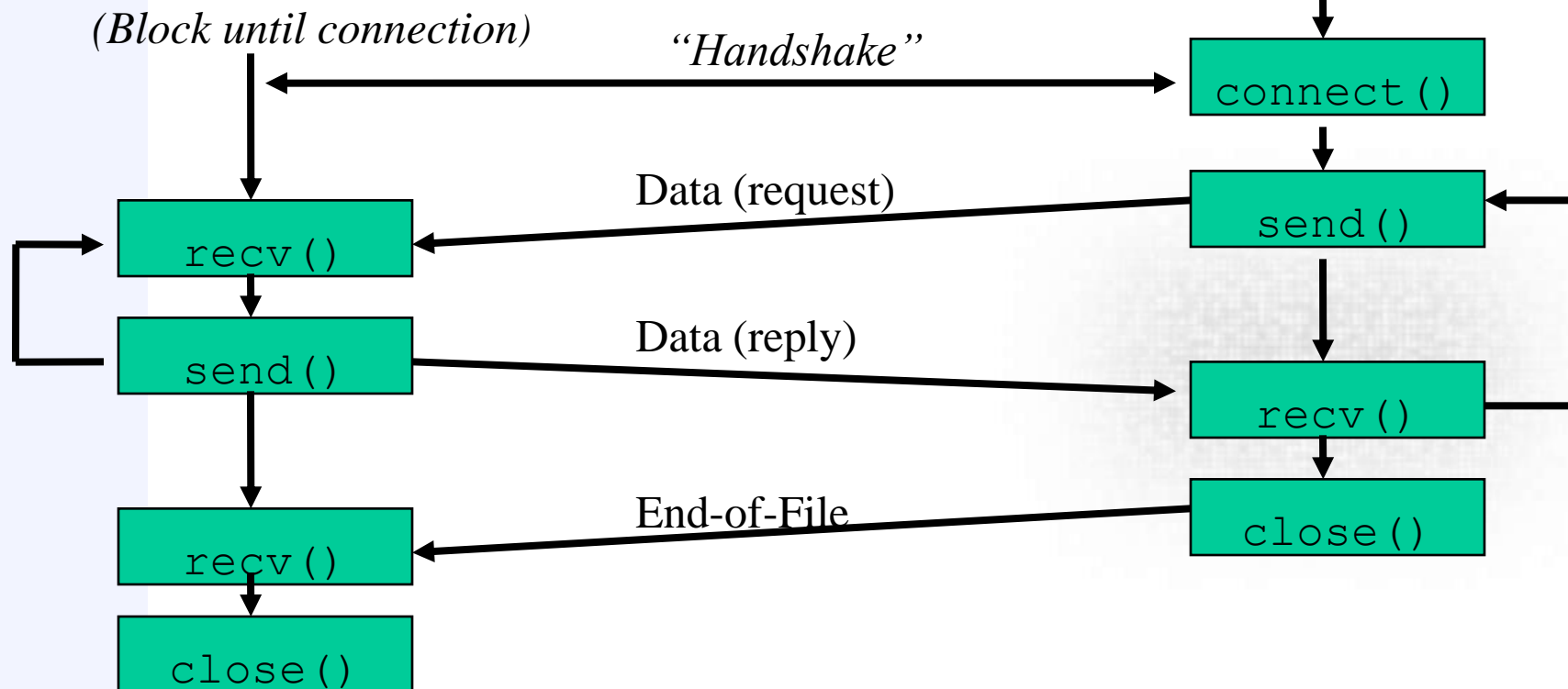
close()

*“Handshake”*

Data (request)

Data (reply)

End-of-File



# socket ()

---

```
int socket(int family, int type, int protocol);
```

Create a socket, giving access to transport layer service.

- *family* is one of
  - AF\_INET (IPv4), AF\_INET6 (IPv6), AF\_LOCAL (local Unix),
  - AF\_ROUTE (access to routing tables), AF\_KEY (new, for encryption)
- *type* is one of
  - SOCK\_STREAM (TCP), SOCK\_DGRAM (UDP)
  - SOCK\_RAW (for special IP packets, PING, etc. Must be root)
- *protocol* is 0 (used for some raw socket options)
- upon success returns socket descriptor
  - Integer, like file descriptor
  - Return -1 if failure

# bind()

---

```
int bind(int sockfd, const struct sockaddr *myaddr,  
         socklen_t addrlen);
```

Assign a local protocol address (“name”) to a socket.

- *sockfd* is socket descriptor from `socket()`
- *myaddr* is a pointer to address struct with:
  - *port number* and *IP address*
  - if port is 0, then host OS will pick ephemeral port
- *addrlen* is length of structure
- returns 0 if ok, -1 on error
  - EADDRINUSE (“Address already in use”)

# Argomenti Valore-risultato

- Nelle chiamate che passano la struttura indirizzo da processo utente a OS (esempio bind) viene passato un puntatore alla struttura ed un intero che rappresenta il sizeof della struttura (oltre ovviamente ad altri parametri). In questo modo l'OS kernel sa esattamente quanti byte deve copiare nella sua memoria.
- Nelle chiamate che passano la struttura indirizzo dall'OS kernel al processo utente (esempio accept) vengono passati nella system call eseguita dall'utente un puntatore alla struttura ed un puntatore ad un intero in cui è stata preinserita la dimensione della struttura indirizzo. In questo caso, sulla chiamata della system call, il kernel dell'OS sa la dimensione della struttura quando la va a riempire e non ne oltrepassa i limiti. Quando la system call ritorna inserisce nell'intero la dimensione di quanti byte ha scritto nella struttura.
- Questo modo di procedere è utile in system call come la select e la getsockopt che vedrete durante le esercitazioni.

# `listen()`

```
int listen(int sockfd, int backlog);
```

Change socket state for TCP server.

- *sockfd* is socket descriptor from `socket()`
- *backlog* is maximum number of *incomplete* connections
  - historically 5
  - rarely above 15



## accept ()

---


```
int accept(int sockfd, struct sockaddr  
          cliaddr, socklen_t *addrlen);
```

Return next completed connection.

- *sockfd* is socket descriptor from `socket ()`
- *cliaddr* and *addrlen* return protocol address from client
- returns new descriptor, created by OS
- if used with `fork ()`, can create concurrent server. If used with `pthread_create ()` can create a multithread server

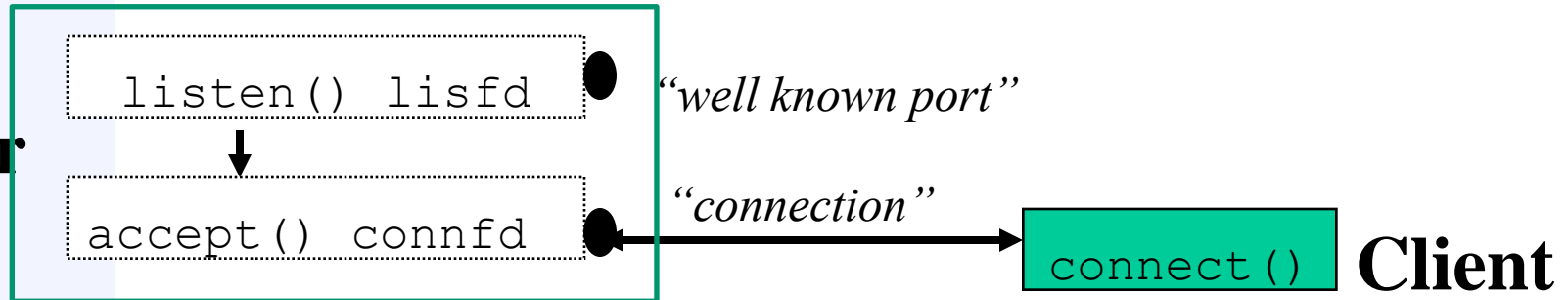
# Accept()+fork()

```
.....  
Lisfd=socket (.....);  
Bind(lisfd,....);  
Listen(lisfd,5);  
For ( ; ; ) {  
    connfd=accept(lisfd,.....);  
    If (pid=Fork()==0) {  
        close(lisfd);  
        doit(connfd);  
        close(connfd);  
        exit(0);  
    }  
    Close(connfd)  
}  
.....
```

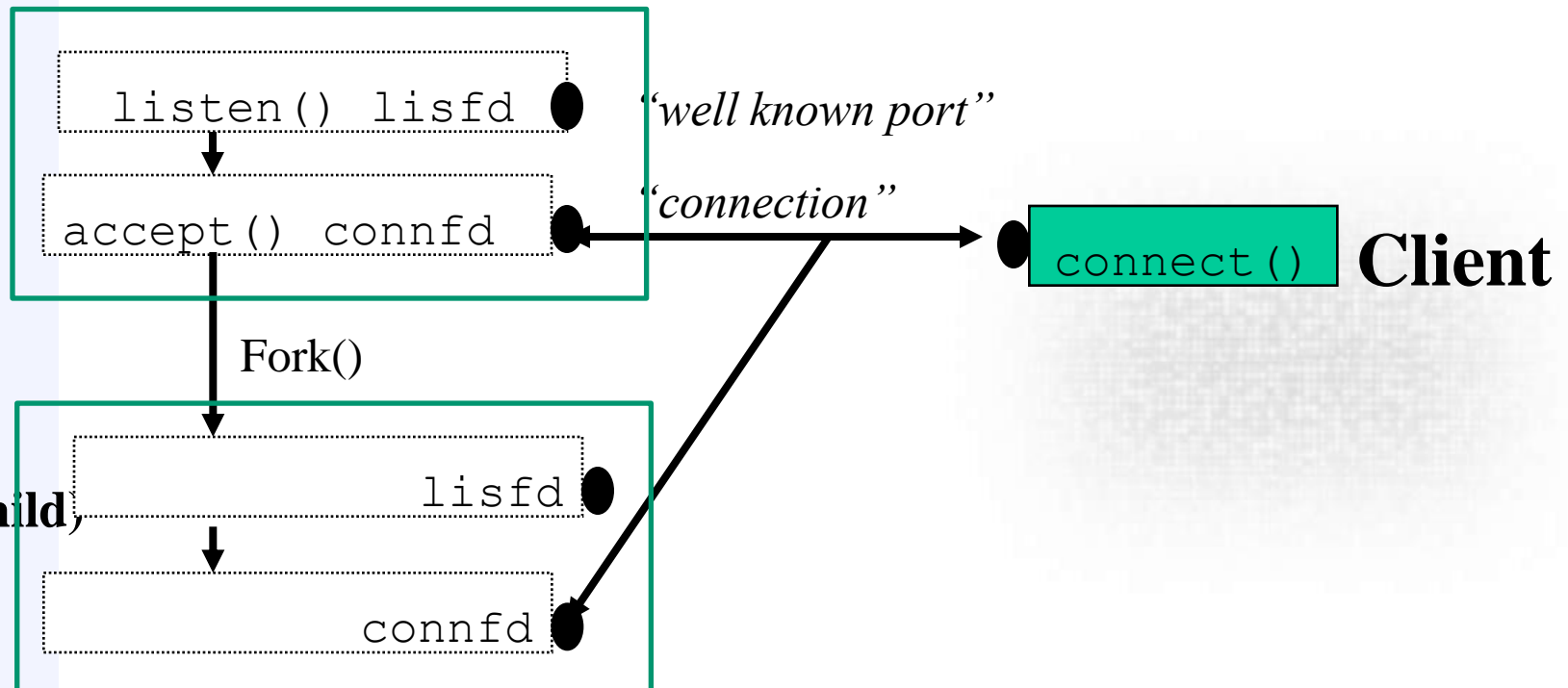


# Status Client-Server after Fork returns

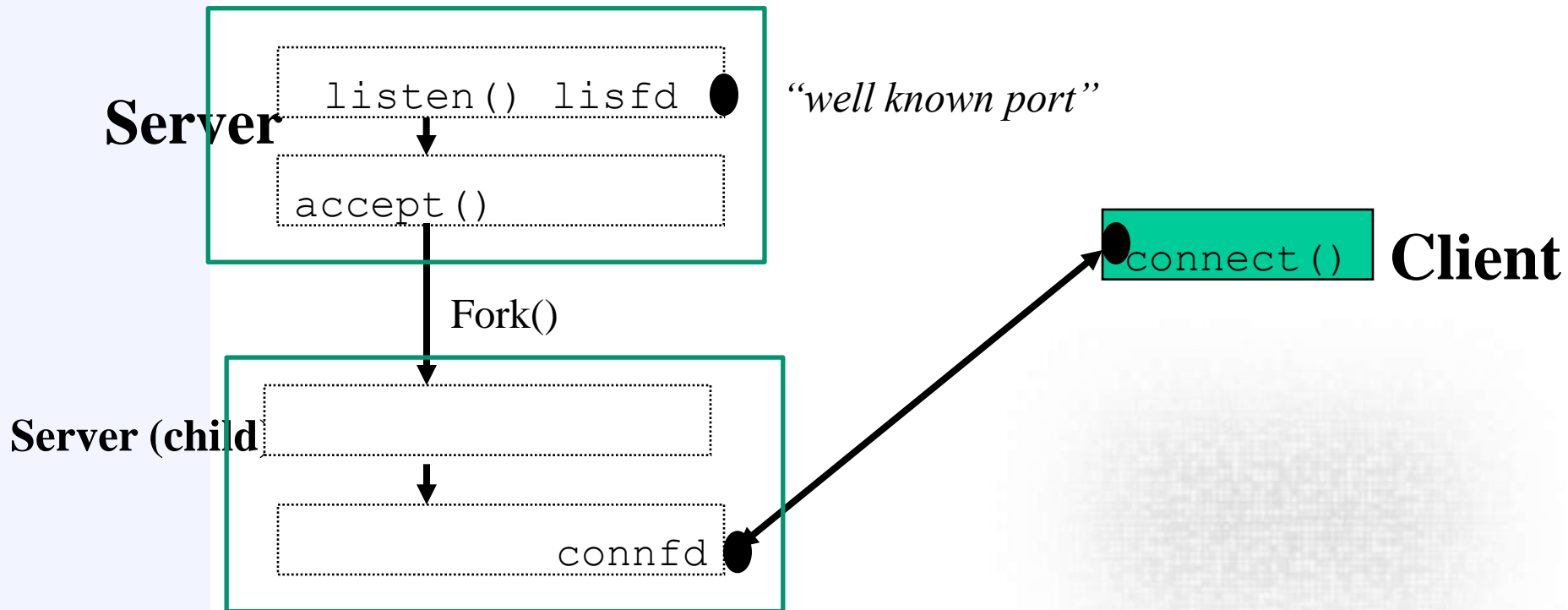
**Server**



**Server**



# Status Client-Server after closing sockets



# close()

```
int close(int sockfd);
```

Close socket for use.

- *sockfd* is socket descriptor from `socket()`
- closes socket for reading/writing
  - returns (doesn't block)
  - attempts to send any unsent data
  - socket option `SO_LINGER`
    - block until data sent
    - or discard any remaining data
  - Returns -1 if error

# TCP Client-Server

**Server**`socket()``bind()`

“well-known”  
port

`listen()``accept()`

*(Block until connection)*

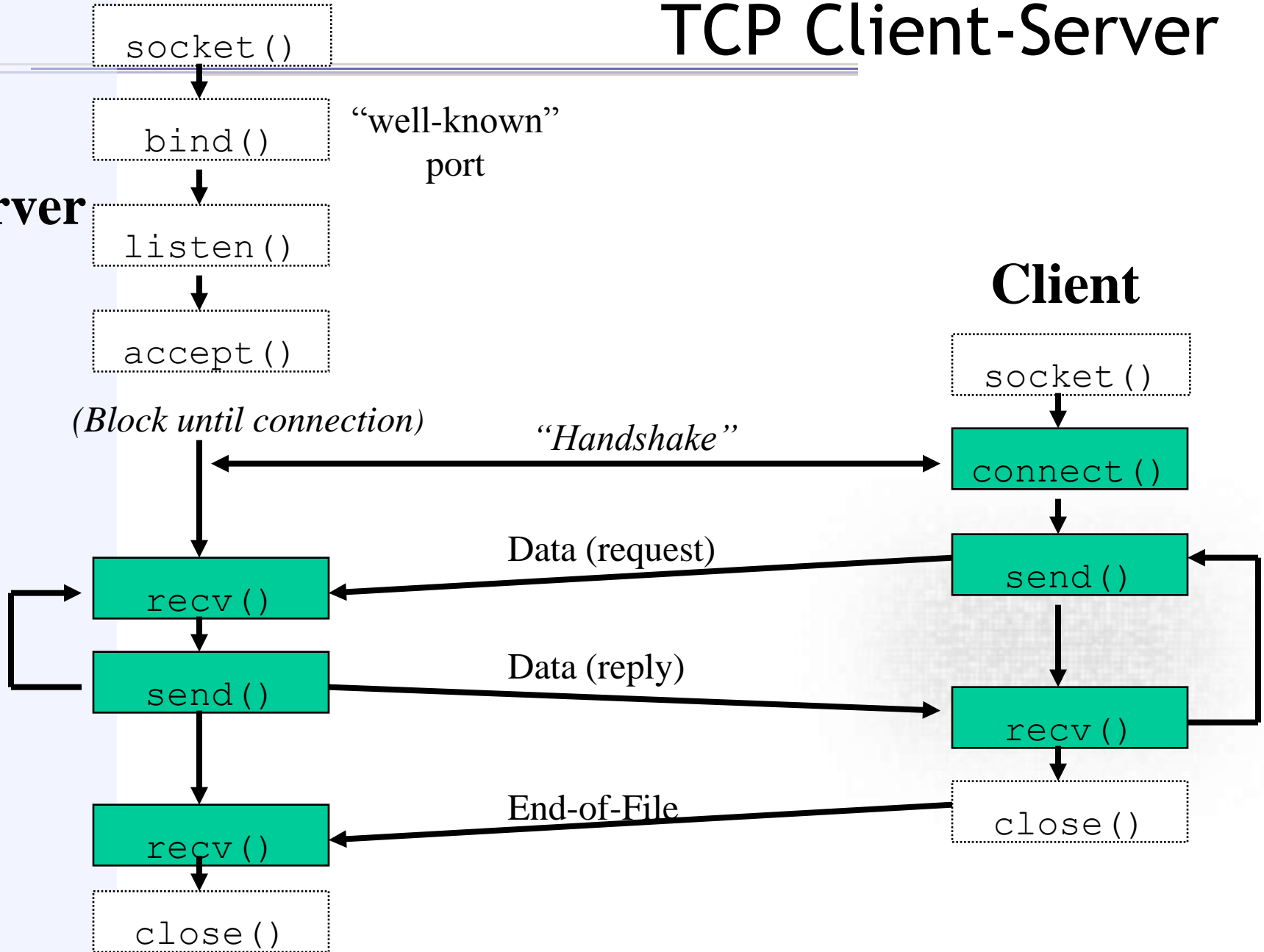
`recv()``send()``recv()``close()`**Client**`socket()``connect()``send()``recv()``close()`

*“Handshake”*

Data (request)

Data (reply)

End-of-File



# connect ()

---

```
int connect(int sockfd, const struct  
sockaddr *servaddr, socklen_t addrlen);
```

Connect to server.

- *sockfd* is socket descriptor from `socket()`
- *servaddr* is a pointer to a structure with:
  - *port number* and *IP address*
  - must be specified (unlike `bind()`)
- *addrlen* is length of structure
- client doesn't need `bind()`
  - OS will pick ephemeral port
- returns socket descriptor if ok, -1 on error

# Sending and Receiving

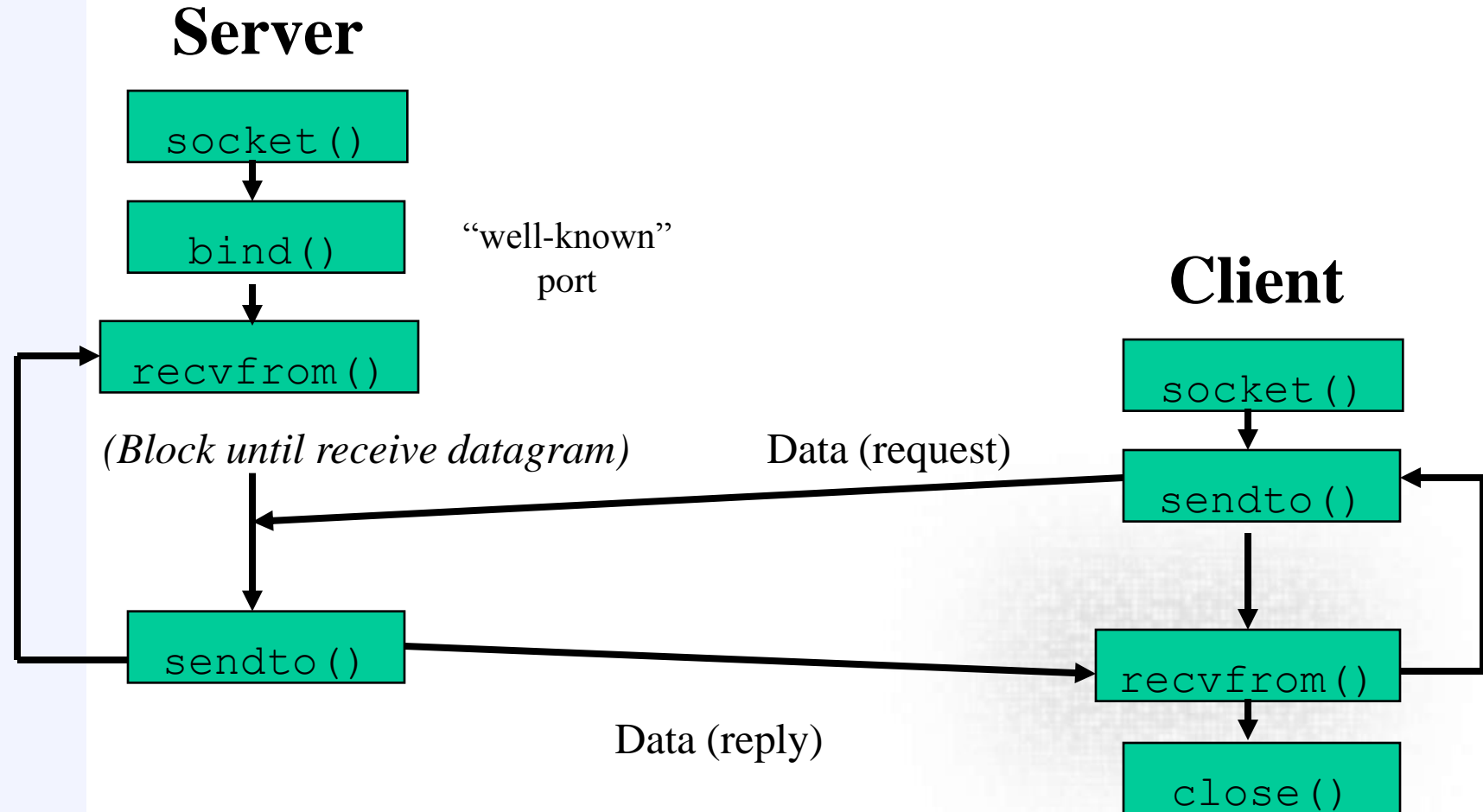
```
int recv(int sockfd, void *buff, size_t  
#bytes, int flags);
```

```
int send(int sockfd, void *buff, size_t  
#bytes, int flags);
```

- Same as `read()` and `write()` but for *flags*
  - `MSG_DONTWAIT` (this send non-blocking)
  - `MSG_OOB` (out of band data, 1 byte sent ahead)
  - `MSG_PEEK` (look, but don't remove)
  - `MSG_WAITALL` (don't give me less than max)
  - `MSG_DONTROUTE` (bypass routing table)



# UDP Client-Server



- No "handshake"
- No simultaneous close
- No `fork()` for concurrent servers!

# Sending and Receiving

```
int recvfrom(int sockfd, void *buff, size_t #bytes, int
    flags, struct sockaddr *from, socklen_t *addrlen);
int sendto(int sockfd, void *buff, size_t #bytes, int
    flags, const struct sockaddr *to, socklen_t
    addrlen);
```

- Same as `recv()` and `send()` but for *addr*
  - `recvfrom` fills in address of where packet came from
  - `sendto` requires address of where sending packet to

# gethostname()

---

- Get the name of the host the program is running on.
  - `int gethostname(char *hostname, int bufferLength)`
    - Upon return `hostname` holds the name of the host
    - `bufferLength` provides a limit on the number of bytes that `gethostname` can write to `hostname`.

## Internet Address Library Routines (inet\_addr() and inet\_ntoa())

---

- `unsigned long inet_addr(char *address);`
  - converts address in dotted form to a 32-bit numeric value in network byte order
    - (e.g., “128.173.41.41”)
- `char* inet_ntoa(struct in_addr address)`
  - `struct in_addr`
    - `address.S_addr` is the long int representation

## Domain Name Library Routine (gethostbyname)

---

- `gethostbyname()`: Given a host name (such as `acavax.lynchburg.edu`) get host information.
  - `struct hostent* getbyhostname(char *hostname)`
    - `char* h_name;` // official name of host
    - `char** h_aliases;` // alias list
    - `short h_addrtype;` // address family (e.g., `AF_INET`)
    - `short h_length;` // length of address (4 for `AF_INET`)
    - `char** h_addr_list;` // list of addresses (null pointer terminated)

## Internet Address Library Routines (gethostbyaddr)

---

- Get the name of the host the program is running on.
  - `struct hostent* gethostbyaddr(char *address, int addressLength, int type)`
    - address is in network byte order
    - addressLength is 4 if type is AF\_INET
    - type is the address family (e.g., AF\_INET)

# WinSock

---

- Derived from Berkeley Sockets (Unix)
  - includes many enhancements for programming in the windows environment
- Open interface for network programming under Microsoft Windows
  - API freely available
  - Multiple vendors supply winsock
  - Source and binary compatibility
- Collection of function calls that provide network services

## Differences Between Berkeley Sockets and WinSock

Berkeley	WinSock
bzero()	memset()
close()	closesocket()
read()	not required
write()	not required
ioctl()	ioctlsocket()



## Additional Features of WinSock 1.1

---

- WinSock supports three different modes
  - Blocking mode
    - socket functions don't return until their jobs are done
    - same as Berkeley sockets
  - Nonblocking mode
    - Calls such as `accept()` don't block, but simply return a status
  - Asynchronous mode
    - Uses Windows messages
      - `FD_ACCEPT` - connection pending
      - `FD_CONNECT` - connection established
      - etc.

# WinSock 2

---

- Supports protocol suites other than TCP/IP
  - DecNet
  - IPX/SPX
  - OSI
- Supports network-protocol independent applications
- Backward compatible with WinSock 1.1
- Today it also means a more limited user-base

# WinSock 2 (Continued)

- Uses different files
  - winsock2.h
  - different DLL (WS2\_-32.DLL)
- API changes
  - accept() becomes WSAAccept()
  - connect() becomes WSAConnect()
  - inet\_addr() becomes WSAAddressToString()
  - etc.

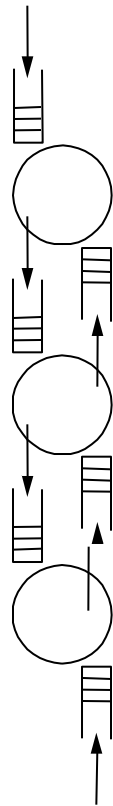
# Socket implementation issues

- Process Models
- Message Buffers

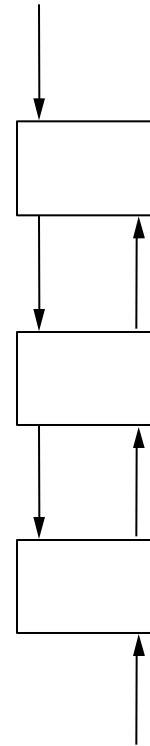
## Critical factors

1. Memory-Memory copies
2. Context Switches

# Socket implementation: Process Model

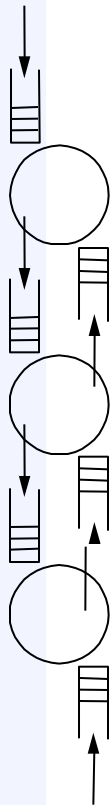


Process-per-protocol



Process-per-message

# Socket implementation: Process Model



Each protocol layer is implemented by one process

Each process has an incoming and outgoing queue

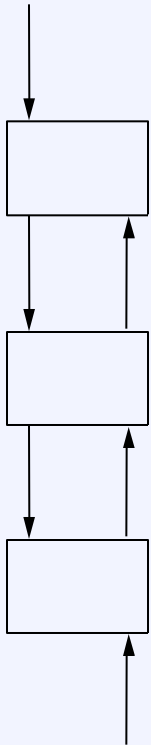
Easy to understand and to implement

Very low performance due to

1. Many context switches
2. Many memory to memory message copies

Process-per-protocol

# Socket implementation: Process Model



- Each time a message is sent (resp. Received) by the applicatio layer (resp. the network adaptor) a process (or thread) is created
- The message goes up and down through the protocol stack using simple procedure call
- The cost of a procedure call is much less than a process context switch

Process-per-message

# Socket implementation: Message Buffers

Socket implementation is not efficient

- `send()` and `receive()` need a buffer at the application level

This forces the copy of the message by the OS from the application buffer to the kernel buffer

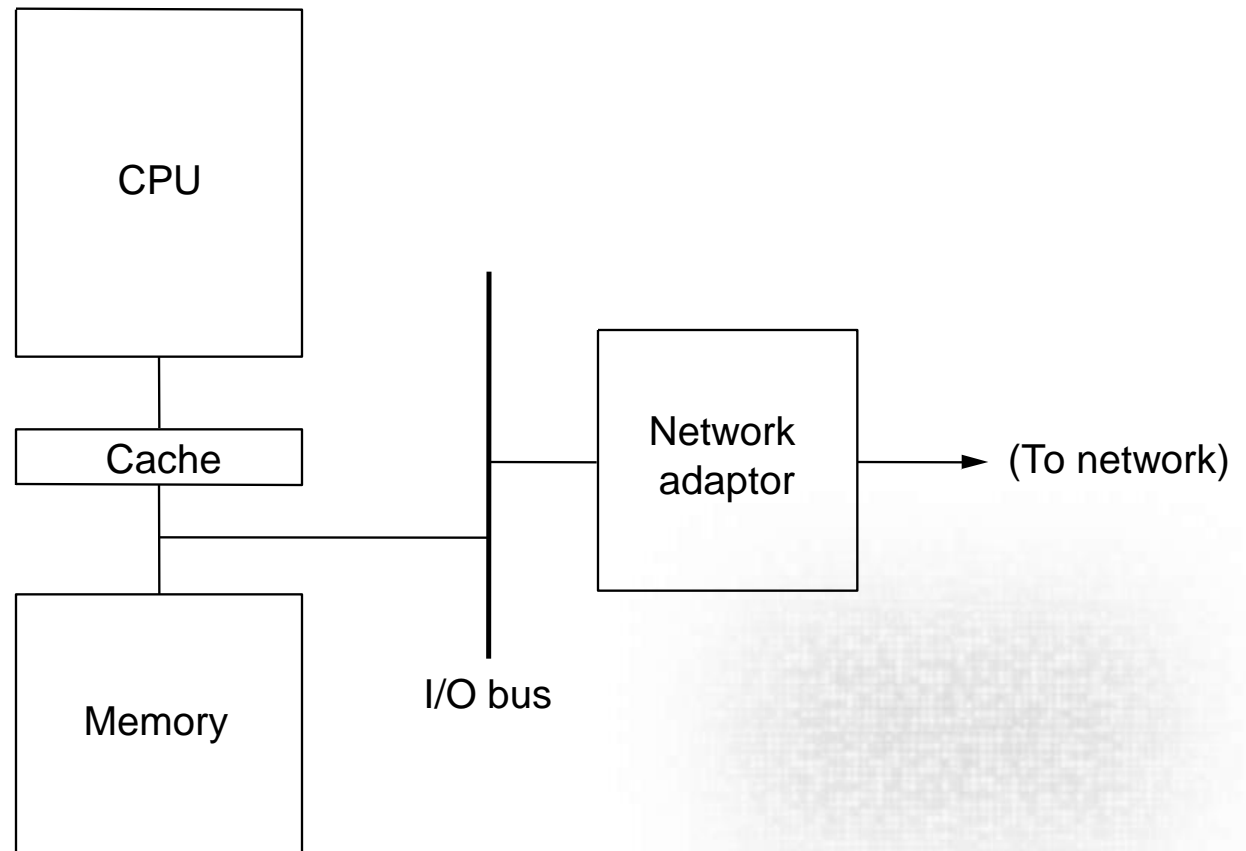
Instead of copying a message up/down the protocol stack layer provides a set of primitives to all the layers to manipulate a message (e.g. append header, strip header, fragment, reassemble etc.).

In this way protocol layers need just passes the reference of the message (not copying)



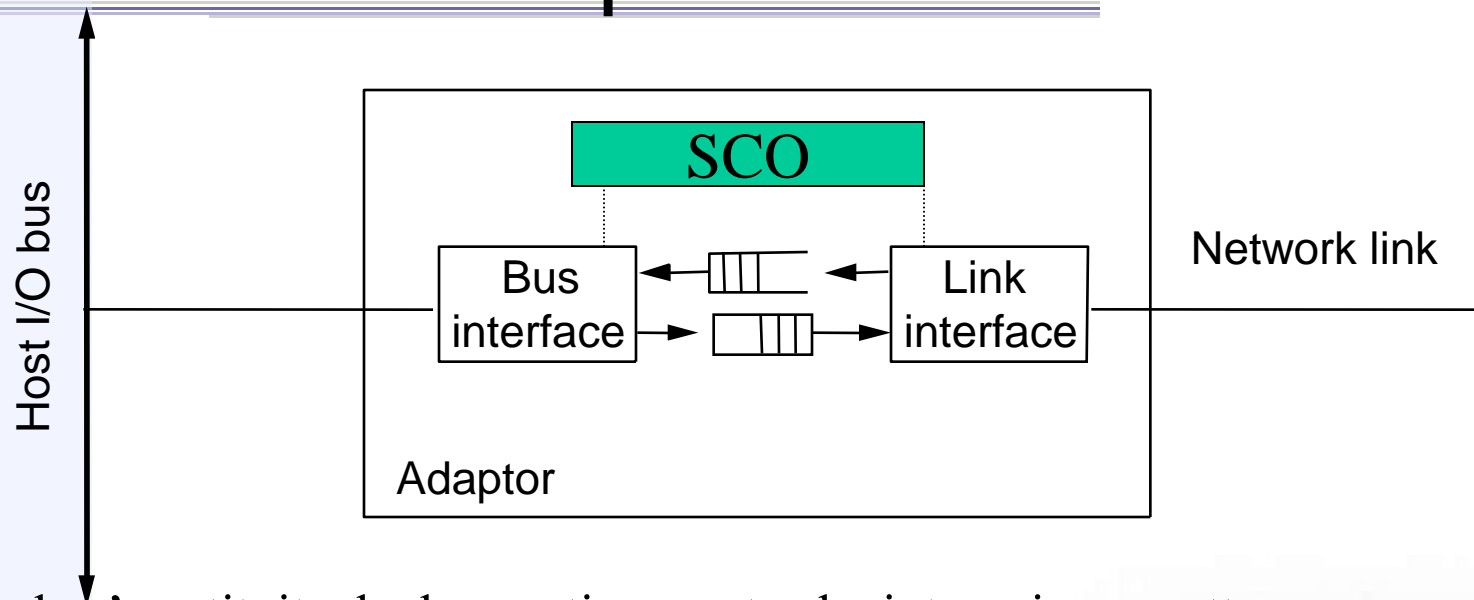
# Network Adaptors

# Network Adaptors



Interfaccia tra Host e Rete

# Network Adaptors



La scheda è costituita da due parti separate che interagiscono attraverso una FIFO che maschera l'asincronia tra la rete e il bus interno

La prima parte interagisce con la CPU della scheda

La seconda parte interagisce con la rete (implementando il livello fisico e di collegamento)

Tutto il sistema è comandato da una SCO (sottosistema di controllo della scheda)

# Vista dall'host

- L'adaptor esporta verso la CPU uno o piu' registri macchina (Control Status Register)
- CRS è il mezzo di comunicazione tra la SCO della scheda e la CPU

```
/* CSR
* leggenda: RO - read only; RC - Read/Clear (writing 1 clear, writing 0 has no effect);
* W1 write-1-only (writing 1 sets, writing 0 has no effect)
* RW - read/write; RW1 - Read-Write-1-only
*/
#define LE_ERR 0X8000
.....
#define LE_RINT 0X0400 /* RC richiesta di interruzione per ricevere un pacchetto */
#define LE_TINT 0X0200 /* RC pacchetto trasmesso */
.....
#define LE_INEA 0X0040 /* RW abilitazione all'emissione di un interrupt da parte
..... dell'adaptor verso la CPU */
#define LE_TDMD 0X0008 /* W1 richiesta di trasmissione di un pacchetto dal device
..... driver verso l'adaptor */
.....
```

# Vista dall'host

---

L'host può controllare cosa accade in CSR in due modi

- Busy waiting (la cpu esegue un test continuo di CSR fino a che' CSR non si modifica indicando la nuova operazione da eseguire. Ragionevole solo per calcolatori che non devono fare altro che attendere e trasmettere pacchetti ad esempio routers)
- Interrupt (l'adaptor invia un interrupt all'host e l'host fa partire un interrupt handler che va a leggere CSR per capire l'operazione da fare)

# Trasferimento dati da adaptor a memoria (e viceversa)

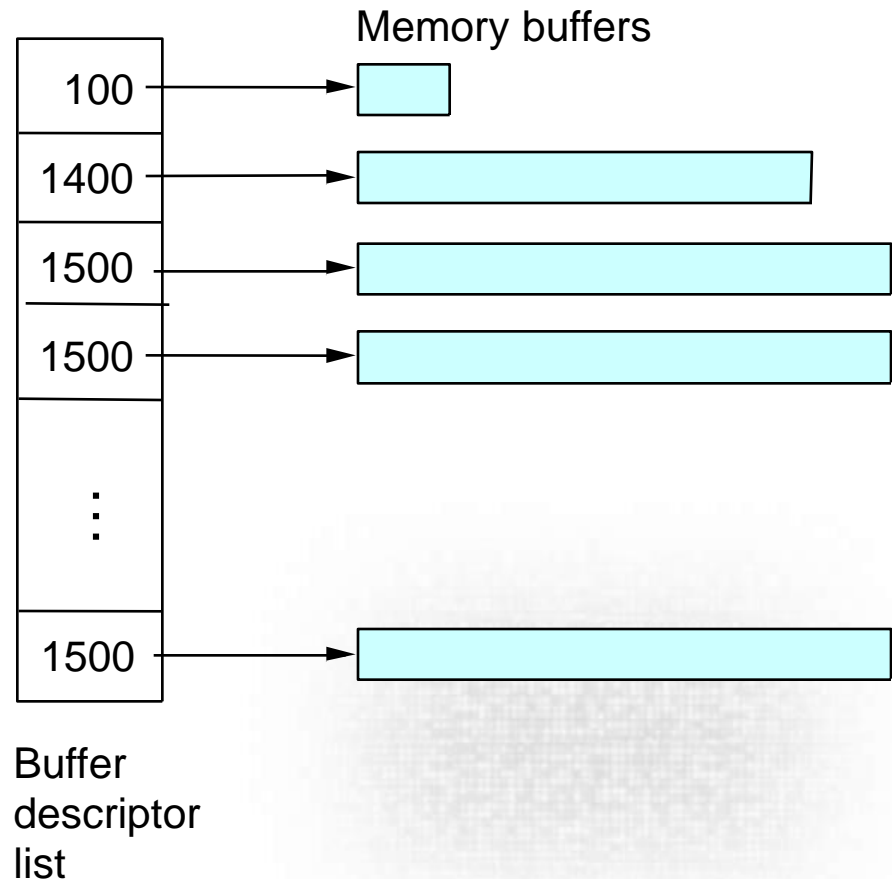
- Direct Memory Access
  - Nessun coinvolgimento della CPU nello scambio dati
  - Pochi bytes di memoria necessari sulla scheda
  - Frame inviati immediatamente alla memoria di lavoro dell'host gestita dal SO
- Programmed I/O
  - Lo scambio dati tra memoria e adaptor passa per la CPU
  - Impone di bufferizzare almeno un frame sull'adaptor
  - La memoria deve essere di tipo dual port
    - Il processore e l'adaptor possono sia leggere che scrivere in questa porta

# Buffer Descriptor list (BD)

La memoria dove allocare i frames e' organizzata attraverso buffer description list

Un vettore di puntatori ad aree di memoria (buffers) dove e' descritta la quantità di memoria disponibile in quell'area

In ethernet vengono tipicamente preallocati 64 buffers da 1500 bytes



# Buffer Descriptor list

---

Tecnica usata per frame che arrivano dall'adaptor:

*scatter read / gather write*

*“frame distinti sono allocati in buffer distinti, un frame puo’ essere allocato su piu’ buffer”*



# Viaggio di un messaggio all'interno dell'SO

Quando un messaggio viene inviato da un utente in un certo socket

1. Il sistema operativo copia il messaggio dal buffer della memoria utente in una zona di BD
2. tale messaggio viene processato da tutti i livelli protocollari (esempio tcp, IP, device driver) che provvedono ad inserire gli opportuni header ed ad aggiornare gli opportuni puntatori presenti nel buffer description list in modo da poter sempre ricostruire il messaggio
3. Quando il messaggio ha completato l'attraversamento del protocol stack, viene avvertita la SCO dell'adaptor dal device driver attraverso il set dei bit del CSR (LE\_TDMD e LE\_INEA). Il primo invita la SCO ad inviare il messaggio sulla linea. Il secondo abilita la SCO ad inviare una interruzione
4. La SCO dell'adaptor invia il messaggio sulla linea
5. Una volta terminata la trasmissione notifica il termine alla CPU attraverso il set del bit (LE\_TINT) del CSR e scatena una interruzione
6. Tale interruzione avvia un interrupt handler prende atto della trasmissione resetta gli opportuni bit (LE\_TINT e LE\_INEA) e libera le opportune risorse (operazione semsignal su xmit\_queue)

# Device Drivers

Il device driver e' una collezione di routines (inizializzare l'adaptor, invio di un frame sul link etc.) di OS che serve per "ancorare" il SO all'hardware sottostante specifico dell'adaptor

Esempio routine di richiesta di invio di un messaggio sul link

```
#define csr ((u_int) 0xffff3579 /*CSR address*/
Transmit(Msg *msg)
{
    descriptor *d;
    semwait(xmit_queue);      /* abilita non piu' di 64 accessi al BD*/
    d=next_desc();
    prepare_desc(d,msg);
    semwait(mutex);          /* abilita a non piu' di un processo (dei potenziali 64)
                              alla volta la trasmissione verso l'adaptor */
    disable_interrupts();     /* il processo in trasmissione si protegge da eventuali
                              interruzioni dall'adaptor */
    csr= LE_TDMD | LE_INEA;   /* una volta preparato il messaggio invita la SCO dell'adaptor a
                              trasmetterlo e la abilita la SCO ad emettere una interruzione
                              una volta terminata la trasmissione */
    enable_interrupts();      /* riabilita le interruzioni */
    semsignal(mutex);         /* sblocca il semaforo per abilitare un altro processo a
                              trasmettere */
}
```

"next\_desc()" ritorna il prossimo buffer descriptor disponibile nel buffer descriptor list  
"prepare\_desc(d,msg)" il messaggio msg nel buffer d in un formato comprensibile dall'adaptor

# Interrupt Handler

- Disabilita le interruzioni
- Legge il CSR per capire che cosa deve fare: tre possibilità
  1. C'e' stato un errore
  2. Una trasmissione è stata completata
  3. Un frame e' stato ricevuto
- Noi siamo nel caso 2
  - LE\_TINT viene messo a zero (RC bit)
  - Ammette un nuovo processo nell'area BD poiche un frame è stato trasmesso
  - Abilita le interruzioni

```

lance_interrupt_handler()
{
    disable_interrupts();

    /* some error occurred */
    if (csr & LE_ERR)
    {
        print_error(csr);
        /* clear error bits */
        csr = LE_BABL | LE_CERR | LE_MISS | LE_MERR | LE_INEA;
        enable_interrupts();
        return();
    }

    /* transmit interrupt */
    if (csr & LE_TINT)
    {
        /* clear interrupt */
        csr = LE_TINT | LE_INEA;

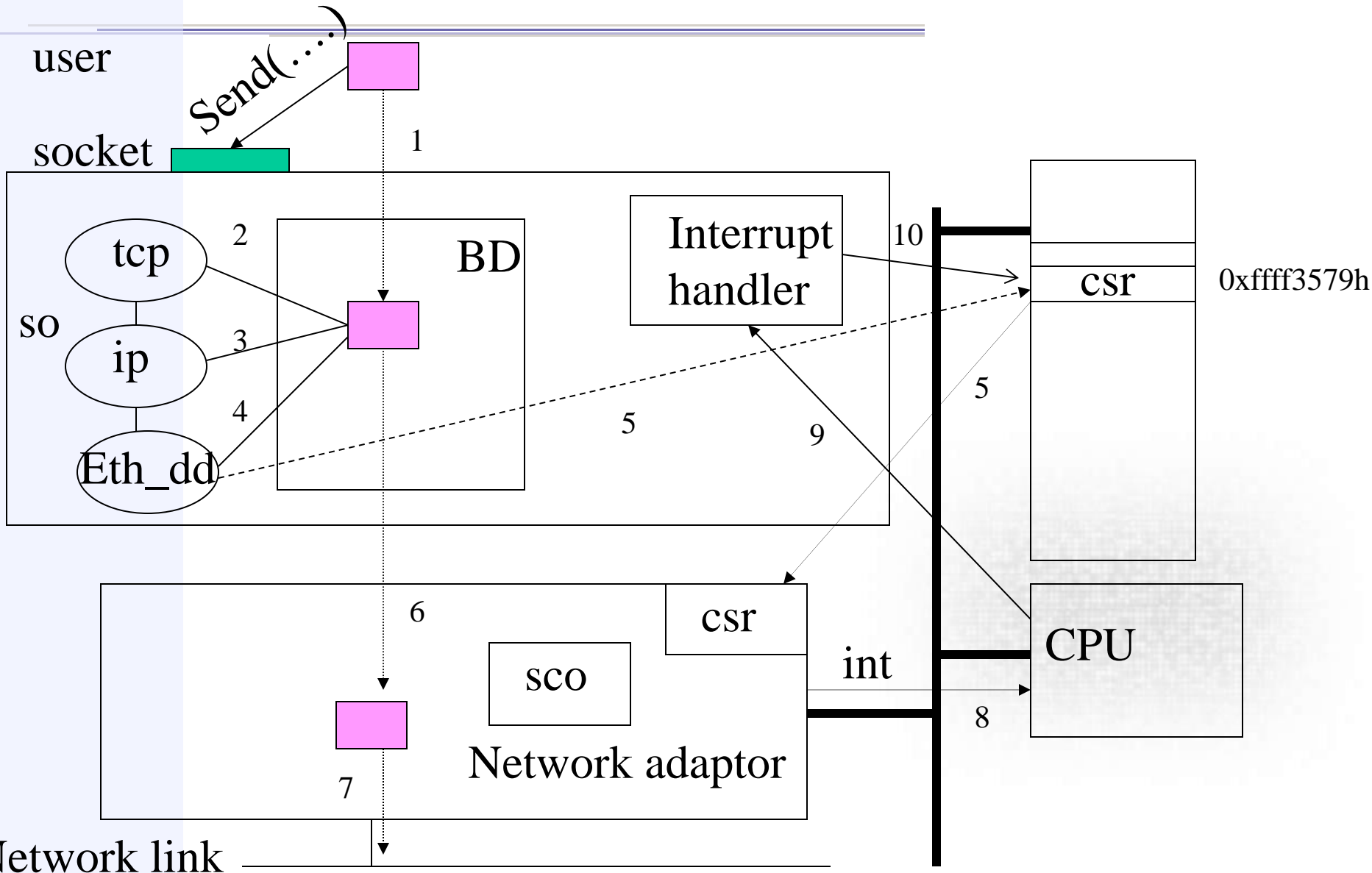
        /* signal blocked senders */
        semSignal(xmit_queue);

        enable_interrupts();
        return(0);
    }

    /* receive interrupt */
    if (csr & LE_RINT)
    {
        /* clear interrupt */
        csr = LE_RINT | LE_INEA;

        /* process received frame */
        lance_receive();
        enable_interrupts();
        return();
    }
}

```



# Main References

---

- W.R. Stevens “Unix Network Programming” Prentice Hall, 1999
- Peterson – Davie “Computer Networks: A system approach” Morgan Kaufmann 2000