

RETI INFORMATICHE

APPUNTI DI LABORATORIO

A.A. 2022/2023

Lorenzo Mancinelli

SOMMARIO

1	INTRODUZIONE	6
2	CONFIGURAZIONE DI INTERFACCE DI RETE, GATEWAY E DNS	6
2.1	CONFIGURARE L'INTERFACCIA DI RETE.....	8
2.1.1	COMANDO <i>ip</i>	8
2.1.2	COMANDO <i>ip addr show</i>	9
2.1.3	COMANDO <i>ip</i> (2)	9
2.1.4	PACCHETTO <i>ifupdown</i>	10
2.1.5	INVIO DEI PACCHETTI.....	10
2.2	CONFIGURARE IL DEFAULT GATEWAY.....	11
2.2.1	FILE DI CONFIGURAZIONE	11
2.2.2	CONFIGURAZIONE DEL GATEWAY	12
2.3	SISTEMA DI RISOLUZIONE DEI NOMI	12
2.3.1	DNS	12
2.3.2	NAME SERVICE SWITCH	13
2.4	ESERCIZI PRIMA PARTE.....	13
2.5	ESERCIZI SECONDA PARTE.....	14
3	CONFIGURAZIONE DI DHCP E TEST DI CONNETTIVITA'	16
3.1	DHCP (DYNAMIC HOST CONFIGURATION PROTOCOL)	16
3.1.1	RETI CON DHCP	16
3.1.2	CONFIGURAZIONE SERVER DHCP	17
3.1.3	CONFIGURAZIONE CLIENT DHCP	17
3.2	TEST DI CONNETTIVITA'	18
3.2.1	INTERNET CONTROL MESSAGE PROTOCOL	18
3.2.2	COMANDO <i>ping</i>	18
3.2.3	COMANDO <i>traceroute</i>	19
4	PROGRAMMAZIONE DISTRIBUITA IN C	22
4.1	DEFINIZIONE DI VARIABILI	22
4.2	STRUTTURE	22
4.3	MEMORIA DINAMICA (HEAP)	22
4.3.1	ALLOCAZIONE CON COERCIZIONE	23
4.4	INGRESSO-USCITA (LIBRERIA <i>stdio.h</i>)	24
4.4.1	GESTIONE DELL'USCITA	24
4.4.2	GESTIONE DELL'INGRESSO	24
4.5	STRINGHE	25
4.5.1	DIMENSIONI	25
4.5.2	CONFRONTO	25
4.5.3	COPIA E CONCATENAZIONE	25
4.6	GESTIONE DEI FILE	26
4.6.1	APERTURA FILE	26
4.6.2	LETTURA E SCRITTURA	26
4.6.3	STATISTICHE E CHIUSURA FILE	26
4.7	COMPILAZIONE	27
4.8	PROGRAMMAZIONE DISTRIBUITA IN C	27
4.8.1	COOPERAZIONE TRA PROCESSI	27
4.8.2	MODELLO CLIENT-SERVER	27
4.8.3	SOCKET	28
4.8.4	PROCESSO SERVER	28

4.8.5	<i>PRIMITIVA socket()</i>	28
4.8.6	<i>STRUTTURE PER GLI INDIRIZZI</i>	29
5	PROGRAMMAZIONE DISTRIBUITA IN C – PARTE 1	30
5.1.1	<i>ENDIANNESS (ORDINE DEI BYTE)</i>	30
5.1.2	<i>FUNZIONI DI CONVERSIONE (FUNZIONI NINJA)</i>	30
5.1.4	<i>FORMATO DEGLI INDIRIZZI</i>	31
5.1.5	<i>CREAZIONE E INIZIALIZZAZIONE DI UN SOCKET</i>	31
5.2	PROGRAMMAZIONE DISTRIBUITA – LATO SERVER	32
5.2.1	<i>PRIMITIVA bind()</i>	32
5.2.2	<i>PRIMITIVA listen()</i>	32
5.2.3	<i>PRIMITIVA accept()</i>	33
5.2.4	<i>PROCESSO SERVER</i>	34
5.3	PROGRAMMAZIONE DISTRIBUITA – LATO CLIENT	34
5.3.1	<i>PRIMITIVA connect()</i>	34
5.3.3	<i>PROCESSO CLIENT</i>	35
5.3.4	<i>INIZIALIZZAZIONE</i>	35
5.3.5	<i>ACCETTAZIONE DI CONNESSIONE</i>	36
6	PROGRAMMAZIONE DISTRIBUITA – SCAMBIO DI DATI E PROTOCOLLI TEXT E BINARY	37
6.1	PROGRAMMAZIONE DISTRIBUITA – SCAMBIO DI DATI	37
6.1.1	<i>PRIMITIVA send()</i>	37
6.1.2	<i>PRIMITIVA recv()</i>	38
6.1.3	<i>PRIMITIVA close()</i>	39
6.1.4	<i>ESEMPIO DI UTILIZZO SEND E RECEIVE</i>	39
6.2	GESTIONE DEGLI ERRORI	41
6.3	PROTOCOLLI TEXT E BINARY	42
6.3.1	<i>ESEMPIO DI INVIO DEI DATI SBAGLIATO</i>	42
6.3.2	<i>ESEMPIO: CODIFICA DEL NUMERO 1234 IN FORMATO BINARY E TEXT</i>	43
6.3.3	<i>TEXT PROTOCOLS</i>	44
6.3.4	<i>BINARY PROTOCOL</i>	45
6.3.5	<i>VANTAGGI E SVANTAGGI DEI DUE PROTOCOLLI</i>	46
6.4	ESERCIZIO 1	46
6.4.1	<i>CLIENT</i>	46
6.4.2	<i>SERVER</i>	47
6.5	ESERCIZIO 2	48
6.5.1	<i>CLIENT</i>	48
6.5.2	<i>SERVER</i>	50
7	PROGRAMMAZIONE CONCORRENTE	52
7.1	SERVER CONCORRENTI	52
7.1.1	<i>TIPI DI PROCESSI SERVER</i>	52
7.1.2	<i>PRIMITIVA fork()</i>	52
7.1.3	<i>USO DI fork()</i>	53
7.1.4	<i>ESEMPIO DI SERVER CONCORRENTE</i>	54
7.2	ESERCIZIO: ECHO SERVER CONCORRENTE	54
7.2.1	<i>CLIENT</i>	55
7.2.2	<i>SERVER</i>	56
8	SOCKET NON BLOCCANTI, I/O MULTIPLEXING	59
8.1	MODELLI DI I/O	59
8.1.1	<i>SOCKET BLOCCANTI</i>	59
8.1.2	<i>SOCKET NON BLOCCANTE</i>	59

8.2	I/O MULTIPLEXING: GESTIRE PIU' DESCRITTORI/SOCKET CONTEMPORANEAMENTE	60
8.2.1	<i>MULTIPLEXING I/O SINCRONO</i>	60
8.2.2	<i>PRIMITIVA select()</i>	61
8.3	ESERCIZIO: TIME SERVER TCP	64
8.3.1	<i>STAMPARE L'ORA</i>	64
8.3.2	<i>CLIENT</i>	64
8.3.3	<i>SERVER</i>	65
8.3.4	<i>CLIENT CON SELECT</i>	68
8.3.5	<i>SERVER CON SELECT</i>	69
9	SOCKET UDP	73
9.1	SOCKET UDP.....	73
9.1.1	<i>TCP VS UDP</i>	73
9.1.2	<i>PRIMITIVA sendto()</i>	73
9.1.3	<i>PRIMITIVA recvfrom()</i>	74
9.1.4	<i>CODICE DEL SERVER</i>	74
9.1.5	<i>CODICE DEL CLIENT</i>	75
9.2	SOCKER UDP "CONNESSO"	75
10	FIREWALL, PACKET FILTERING, IPTABLES	76
10.1	FIREWALL.....	76
10.1.1	<i>TIPI DI FIREWALL</i>	76
10.2	PACKET FILTERING (FIREWALL DI LIVELLO NETWORK)	77
10.2.1	<i>TIPI DI PACKET FILTERING</i>	77
10.2.2	<i>FUNZIONAMENTO</i>	77
10.2.3	<i>FUNZIONAMENTO PER OGNI PACCHETTO</i>	77
10.3	NETFILTER E IPTABLES (PACKET FILTERING SU LINUX)	79
10.3.1	<i>IPTABLES</i>	79
10.4	NAT E PAT/NAPT.....	82
10.4.1	<i>NETWORK ADDRESS TRANSLATION (NAT)</i>	82
10.4.2	<i>PORT ADDRESS TRANSLATION (PAT)</i>	84
10.4.3	<i>NETWORK AND PORT TRANSLATION</i>	84
10.4.4	<i>IPTABLES E NA[P]T</i>	85
11	RICHIAMI HTTP, APACHE HTTP SERVER	89
11.1	HTTP	89
11.1.1	<i>HTTP 1.0</i>	89
11.1.2	<i>HTTP 1.1</i>	89
11.1.3	<i>MESSAGGI HTTP</i>	90
11.2	APACHE HTTP SERVER.....	90
11.2.1	<i>FILE DI CONFIGURAZIONE</i>	91
11.2.2	<i>MODULI</i>	92
11.3	DIRETTIVE GLOBALE	92
11.3.1	<i>DIRETTIVA SERVERROOT</i>	92
11.3.2	<i>DIRETTIVE KEEPALIVE E KEEPALIVETIMEOUT</i>	93
11.3.3	<i>DIRETTIVA LISTEN</i>	93
11.3.4	<i>DIRETTIVA ERRORLOG</i>	93
11.4	DIRETTIVE PER I SITI WEB (VIRTUAL HOST)	93
11.4.1	<i>HOSTING</i>	93
11.4.2	<i>VIRTUAL HOST</i>	93
11.4.3	<i>DIRETTIVA VIRTUALHOST</i>	94
11.4.4	<i>DIRETTIVA SERVERNAME</i>	94
11.4.5	<i>DIRETTIVA DOCUMENTROOT</i>	94

<i>11.4.6 FILE DI CONFIGURAZIONE</i>	95
11.5 MULTI-PROCESSING MODULES.....	95
<i>11.5.1 MPM PREFORK.....</i>	<i>95</i>
<i>11.5.2 MPM WORKER.....</i>	<i>96</i>
<i>11.5.3 MPM EVENT.....</i>	<i>97</i>
12 ALGORITMI DI ROUTING.....	98
<i>12.1 INTERAZIONE TRA ROUTING E FORWARDING</i>	<i>98</i>
<i>12.2 GRAFO</i>	<i>98</i>
<i>12.3 CLASSIFICAZIONE DEGLI ALGORITMI DI ROUTING.....</i>	<i>99</i>
<i>12.4 ALGORITMO DI DIJKSTRA: ALGORITMO LINK STATE</i>	<i>99</i>
<i>12.5 ALGORITMI DISTANCE VECTOR</i>	<i>100</i>
<i>12.5.1 ESEMPIO BELLMAN-FORD.....</i>	<i>100</i>
<i>12.5.2 RITORNANDO ALL'ALGORITMO GENERALE.....</i>	<i>101</i>
<i>12.5.3 CAMBIAMENTI NEI COSTI DEI LINK.....</i>	<i>101</i>
<i>12.5.4 CONFRONTO TRA ALGORITMO LS E DV.....</i>	<i>103</i>
<i>12.6 ROUTING GERARCHICO</i>	<i>103</i>
<i>12.6.1 AS INTERCONNESSI</i>	<i>104</i>
<i>12.6.2 BPG BASICS</i>	<i>106</i>
<i>12.6.3 PERCHE' ABBIAMO DIVERSI ROUTING INTRA-AS E INTER-AS?</i>	<i>111</i>

APPUNTI DELLE LEZIONI DI LABORATORIO INTEGRATI CON:

- Slide del docente reperibili sul sito personale del docente:
<http://docenti.ing.unipi.it/f.pistolesi/teaching.html>
- Soluzioni degli esercizi proposti in classe, sempre reperibili sul sito personale del docente.

1 INTRODUZIONE

La prima lezione, essendo in comune con Sistemi Operativi, è riportata sugli appunti di Sistemi Operativi.

2 CONFIGURAZIONE DI INTERFACCE DI RETE, GATEWAY E DNS

PILA PROTOCOLLARE: pila di livelli di astrazione con i quali i dati possono essere rappresentati.

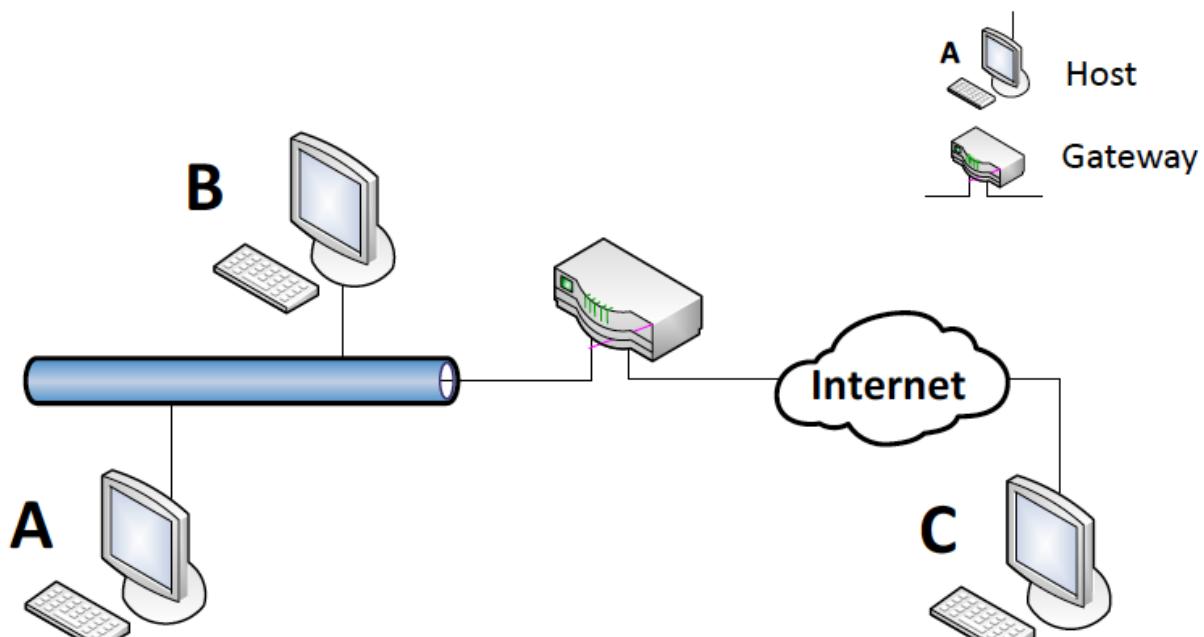
Il livello più alto della pila è il **livello application**: l'applicazione potrebbe aver bisogno di scambiare dati con un'altra applicazione che si trova su un'altra macchina, quindi devo confezionare un messaggio (payload), che appunto si trova a livello applicazione.

È necessario interagire con il **kernel** (chi spedisce il messaggio) utilizzando delle **system call**.

1. Predispongo i vari parametri per chiamare una send. Attraverso dei protocolli, ho la sicurezza che il messaggio venga inviato interamente.
2. Preparando questo messaggio scendo sempre di più nella pila di astrazione: il messaggio viene arricchito di informazioni di controllo necessarie al destinatario per ricevere correttamente il messaggio.
 - a. FRAMMENTAZIONE e AFFIDABILITÀ.

L'effetto che si ha è come se ogni livello dialogasse con il corrispettivo dell'altra applicazione.

- ! Più un dispositivo è specializzato e più livelli sono presenti (il router si ferma al livello trasporto).

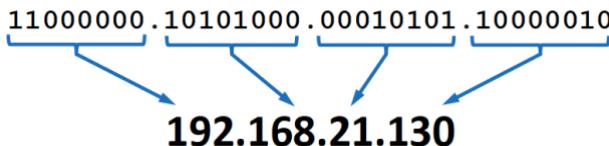


Di cosa ha bisogno un host per comunicare?

1. **INDIRIZZO IP:** sequenza di 32 bit che identifica una scheda di rete (uno stesso host può avere più indirizzi IP).

! Il router ha questo indirizzo visibile all'esterno perché decide a quale dispositivo inviare i pacchetti.

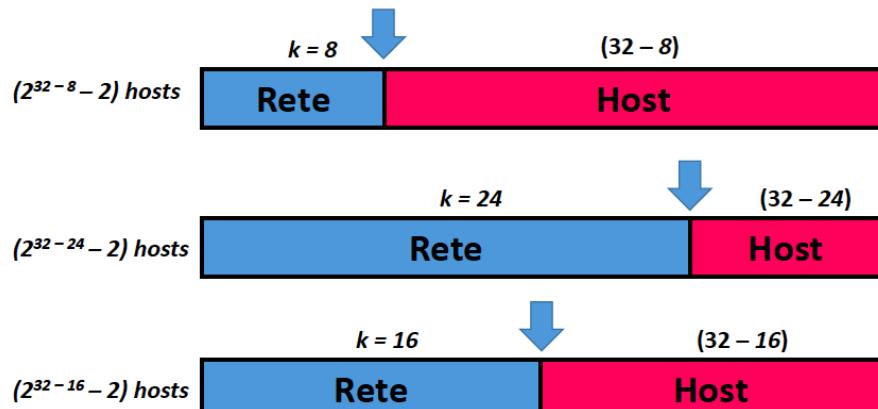
Per comodità, si usa la notazione decimale puntata:



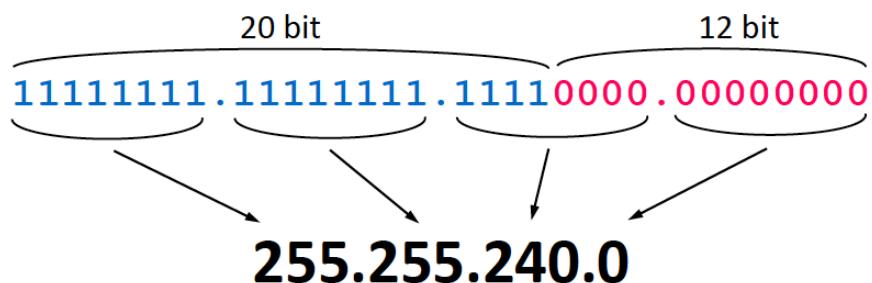
I primi **k bit** a sinistra identificano **la rete** (indirizzo di rete), gli altri **(32 – k) bit** identificano **l'host** in quella rete (indirizzo dell'host).



Classless Inter-Domain Routing: posso utilizzare quanti bit voglio per l'indirizzo di rete e ridimensionare quindi il numero di host.



2. **MASCHERA DI RETE (netmask):** sequenza di 32 bit con **tanti bit a 1** nella parte a sinistra **quanti sono i bit che identificano la rete** (i restanti $32 - k$ bit a destra sono a 0, in corrispondenza dell'indirizzo dell'host).



Notazione compatta: /20

Indirizzo di rete: si ricava tramite un and bit a bit tra IP e netmask.

IP	11000000.10101000.00010101.10000010
Netmask	11111111.11111111.11110000.00000000



11000000.10101000.00010000.00000000

192.168.16.0

Indirizzo di broadcast: si ricava con un **or bit a bit** tra IP e netmask negata.

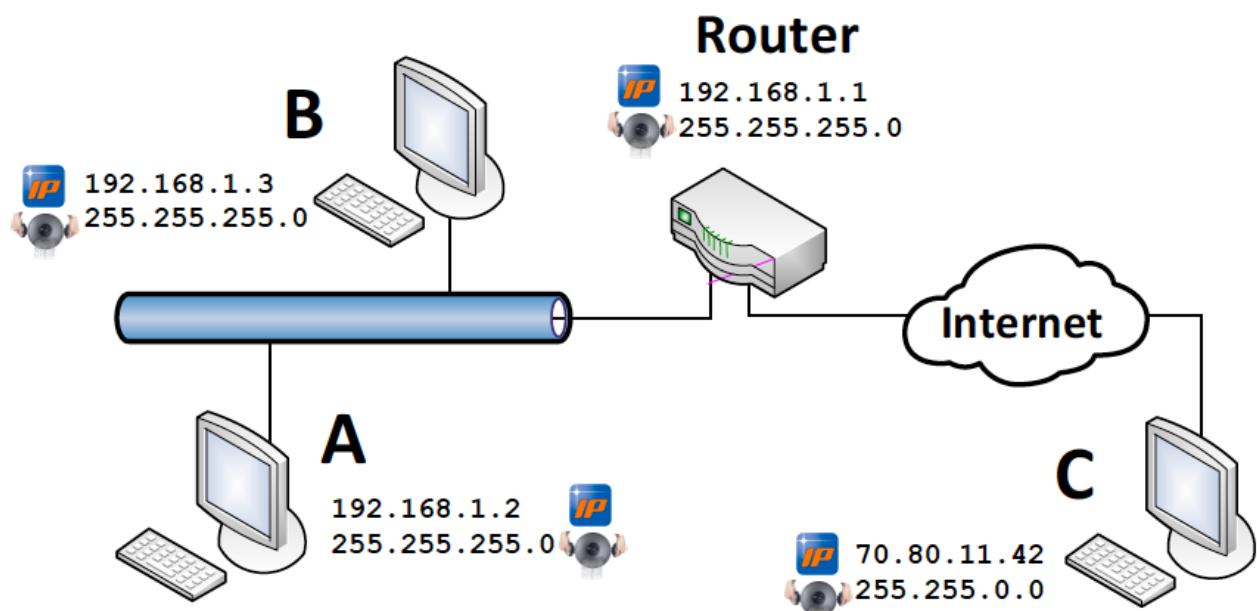
IP 11000000.10101000.00010101.10000010
Netmask 11111111.11111111.11110000.00000000

IP 11000000.10101000.00010101.10000010
Netmask 00000000.00000000.00001111.11111111


11000000.10101000.00011111.11111111

192.168.31.255

3. **INDIRIZZO DEL GATEWAY.**
4. **INDIRIZZO DEL SERVER DNS.**



2.1 CONFIGURARE L'INTERFACCIA DI RETE

man comando

Rimanda al manuale, accessibile da terminale con il comando **man**, seguito dal nome del comando, per dettagli e curiosità sul comando **comando**.

man ip

2.1.1 COMANDO ip

Visualizza e manipola le impostazioni di rete. Se digitato da solo, mostra la sintassi:

```
Usage: ip [ OPTIONS ] OBJECT { COMMAND | help }
WHERE OBJECT := { link | address | route | ... }
...
```

! In debian8, **ip** sostituisce **ifconfig** e **route**.

2.1.2 COMANDO ip addr show

man ip-address

\$ip addr show	Per mostrare tutte le interfacce.
\$ip addr show up	Per mostrare solo le interfacce accese.

OUTPUT DI ip addr show

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:28:fd:4c brd ff:ff:ff:ff:ff:ff
    inet 192.168.50.2/24 brd 192.168.50.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe28:fd4c/64 scope link
        valid_lft forever preferred_lft forever
```

UP: scheda abilitata (interfacce accese).

LOWER_UP: cavo collegato.

mtu: *Maximum Transmission Unit*, dimensione massima (in byte) del pacchetto IP.

qdisc: *Queuing discipline*, stabilisce il prossimo pacchetto da inoltrare.

state: scheda abilitata (UP) o disabilitata (DOWN).

qlen: *Transmission Queue Length*, lunghezza della coda di trasmissione.

link/ether 00:0c:29:28:fd:4c brd ff:ff:ff:ff:ff:ff

link/ether: significa che il protocollo data link Ethernet e l'indirizzo fisico (MAC) della scheda di rete è 00:0c:29:28:fd:4c.

brd: significa *broadcast*, ed è seguito dall'indirizzo ff:ff:ff:ff:ff:ff impostato dall'interfaccia come destination MAC quando invia un broadcast.

inet 192.168.50.2/24 brd 192.168.50.255 scope global eth0

inet: si sta usando il protocollo Internet (IPv4) a livello network, con indirizzo IP 192.168.50.2 e maschera /24, in notazione compatta.

brd: indirizzo di broadcast (192.168.50.255).

2.1.3 COMANDO ip (2)

man ip-link

#ip link set eth0 {up | down}

Abilitare e disabilitare un'interfaccia.

#ip addr add 192.168.1.42/24 broadcast 192.168.1.255 dev eth0

Impostare l'IP di un'interfaccia.

```
#ip addr del 192.168.1.42/24 dev eth0
```

Oppure

```
#ip addr flush dev eth0
```

Rimuovere l'IP a un'interfaccia.

! La configurazione fatta col comando **ip** è annullata dal riavvio della macchina.

2.1.4 PACCHETTO **ifupdown**

Visto il problema appena accennato, dobbiamo trovare un modo di rendere la configurazione manuale **permanente**. Lo facciamo usando il file di configurazione

```
/etc/network/interfaces
```

E usando i comandi **ifup** e **ifdown**.

Di seguito vediamo un esempio di **FILE DI CONFIGURAZIONE**:

man interfaces

```
auto lo      All'avvio viene inizializzata l'interfaccia di lookback.
```

```
iface lo inet loopback
```

```
iface eth0 inet static      I suoi parametri di rete si possono configurare leggendo i file di configurazione e non cambiano.
    address 192.168.1.2
    netmask 255.255.255.0
    broadcast 192.168.1.255
```

COMANDI ifup E ifdown

#ifup eth0	Abilita l'interfaccia eth0 con la configurazione descritta in /etc/network/interfaces .
#ifdown eth0	Disabilita l'interfaccia eth0 .
#ifup -a	Abilita tutte le interfacce della sezione auto nel file di configurazione, nello stesso ordine. È eseguito all'avvio.

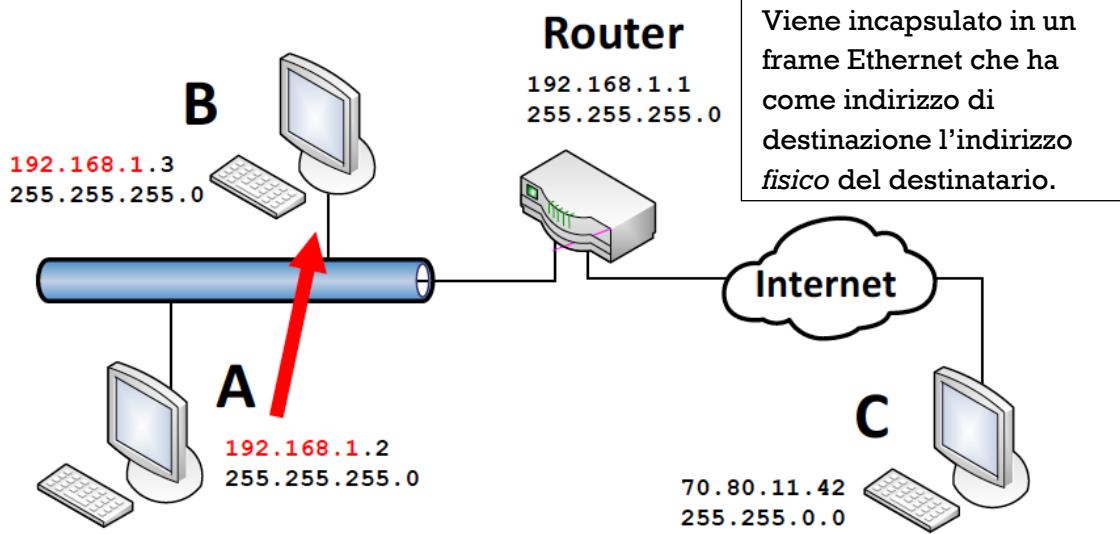
2.1.5 INVIO DEI PACCHETTI

Se un host deve inviare un pacchetto ad un altro host:

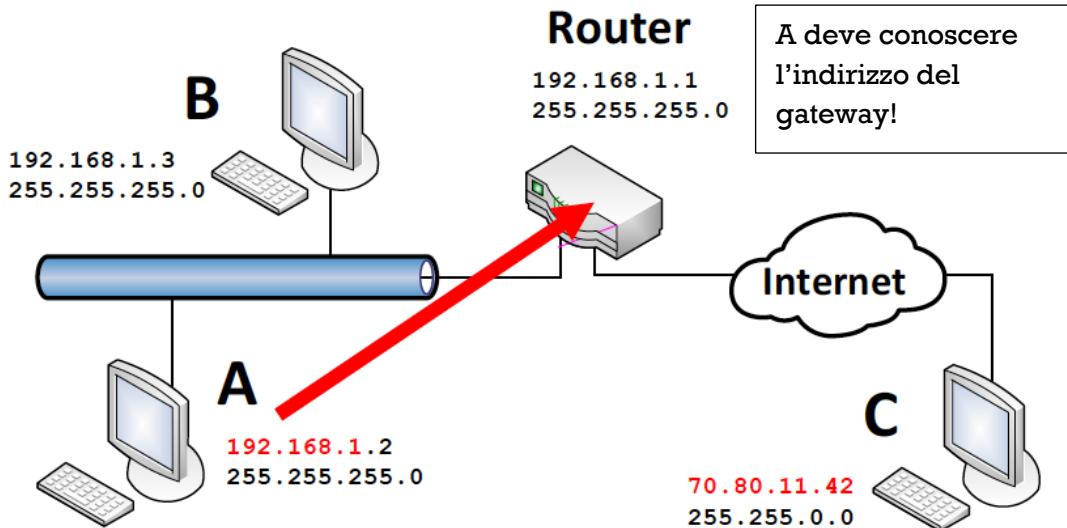
```
dest_subnet := my_netmask & dest_addr;

if (dest_subnet == my_subnet) then
    deliver to dest_addr;
else
    forward to default_gateway;
end if
```

STESSA SOTTORETE: se il destinatario è nella stessa sottorete, il pacchetto viene inviato direttamente all'altro host.



ALTRA SOTTORETE: se il destinatario è in un'altra sottorete, la consegna del pacchetto è delegata al *router* (o *gateway*).



2.2 CONFIGURARE IL DEFAULT GATEWAY

2.2.1 FILE DI CONFIGURAZIONE

Innanzitutto, si deve aggiungere il gateway al file `/etc/network/interfaces`:

```
auto lo

iface lo inet loopback

iface eth0 inet static
    address 192.168.1.2
    netmask 255.255.255.0
    broadcast 192.168.1.255
    gateway 192.168.1.1
```

2.2.2 CONFIGURAZIONE DEL GATEWAY

VISUALIZZARE LA TABELLA DI ROUTING:

COMANDO	OUTPUT
\$ip route show	default via 192.168.1.1 dev eth0 192.168.1.0/24 dev eth0 ... src 192.168.1.2

AGGIUNGERE ROTTE:

#ip route add 192.168.1.0/24 dev eth0	Invio diretto nella rete locale.
#ip route add default via 192.168.1.1	Default gateway.

#ip route get 70.143.3.67	Scoprire la rotta usata.
---------------------------	--------------------------

Adesso sappiamo che, per comunicare, un host ha bisogno di:

1. Indirizzo IP
2. Maschera di rete
3. Indirizzo del default gateway

2.3 SISTEMA DI RISOLUZIONE DEI NOMI

Per semplificare l'uso quotidiano della rete, solitamente si associa un **nome** all'indirizzo IP.

131.114.73.85 = www.unipi.it

L'host deve essere in grado di ricavare l'indirizzo IP a partire dal nome (risoluzione del nome).

Nel file /etc/host troviamo un **elenco di associazioni indirizzo-nome**:

man hosts

```
127.0.0.1 localhost
127.0.1.1 studenti
151.101.37.140 www.reddit.com
131.114.73.85      www.unipi.it
```

2.3.1 DNS

- Database **distribuito e gerarchico** su più server DNS.
- Il client **effettua una richiesta** a un server DNS, che risponde con l'indirizzo IP (se il server non conosce la risposta, inoltra la richiesta a un server più grande).

Per effettuare una richiesta, l'host deve conoscere l'IP di almeno un server DNS.

Il file /etc/resolv.conf contiene gli IP dei server DNS che l'host può contattare:

man resolv.conf

```
nameserver 8.8.8.8
```

```
nameserver 8.8.4.4
```

```
$nslookup nome_dominio
```

Per effettuare una richiesta manualmente.

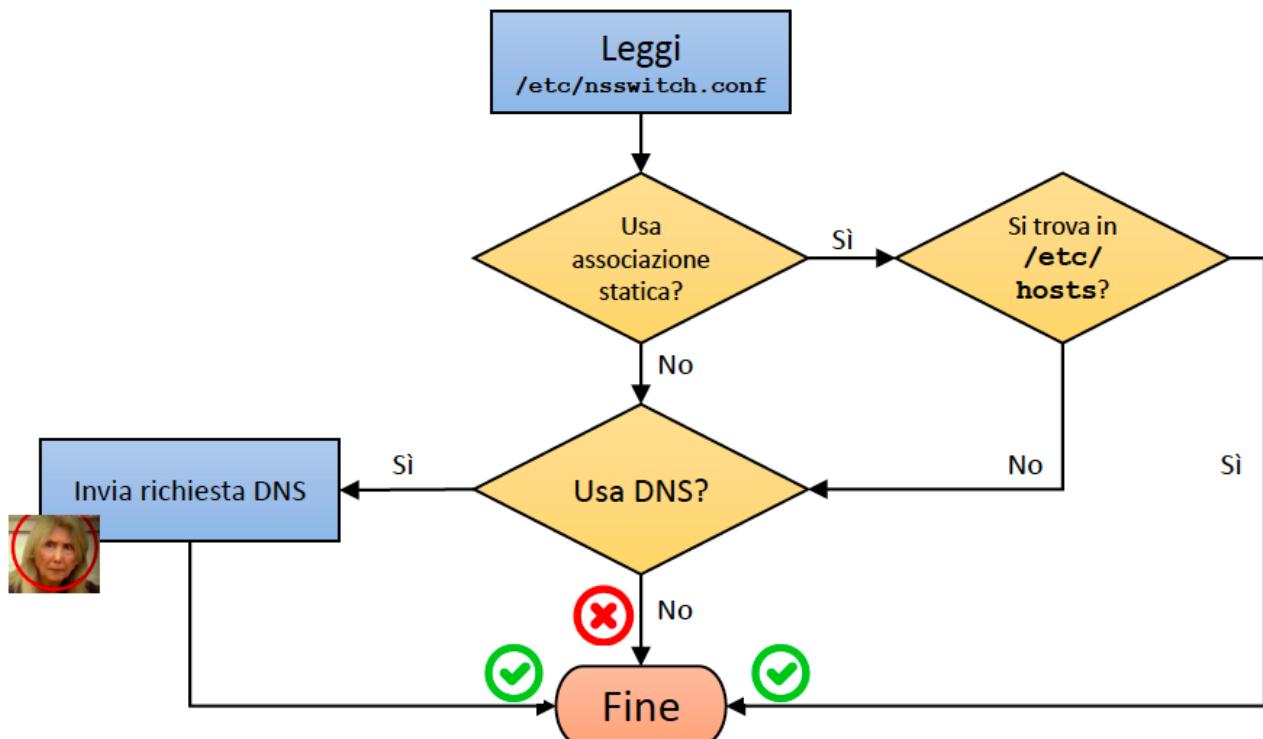
2.3.2 NAME SERVICE SWITCH

Il Name Service Switch (NSS) è il meccanismo che i sistemi Unix usano per **ricavare nomi di "cose"** (nel nostro caso host) da diverse fonti.

Il file `/etc/nsswitch.conf` specifica le fonti da usare e l'ordine in cui usarle:

```
hosts:      files dns
```

```
man nsswitch.conf
```



2.4 ESERCIZI PRIMA PARTE

- Visualizzare la configurazione delle interfacce di rete

```
$ ip addr show
```

- Visualizzare solo la configurazione di eth0

```
$ ip addr show eth0
```

Supponendo il seguente output del comando:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:47:fe:bc brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe47:febc/64 scope link
        valid_lft forever preferred_lft forever
```

- Che indirizzo IP e che maschera sono impostate?

```
IP: 10.0.2.15; maschera: 255.255.255.0
```

- Che indirizzo MAC ha la scheda di rete?

```
08:00:27:47:fe:bc
```

- Cos'è l'MTU e quanto vale?

È la dimensione massima del pacchetto in IP. Vale 1500 byte per il campo dati.

- Impostare come indirizzo IP dell'interfaccia eth0 l'indirizzo 10.0.2.15, maschera 255.255.255.0

```
# ip addr add 10.0.2.15/24 broadcast 10.0.2.255 dev eth0
```

- Abilitare l'interfaccia di rete

```
# ip link set eth0 up
```

- Visualizzare la tabella di routing

```
$ ip route show
```

Supponendo il seguente output del precedente comando:

```
default via 10.0.2.2 dev eth0
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.15
169.254.0.0/16 dev eth0 scope link metric 1000
```

- Qual è il default gateway?

```
10.0.2.2
```

- Impostare come default gateway l'host 10.0.2.2

```
# ip route add default via 10.0.2.2
```

- Visualizzare la rotta usata per raggiungere 192.168.0.27

```
$ ip route get 192.168.0.27
```

2.5 ESERCIZI SECONDA PARTE

- Visualizzare l'indirizzo del server DNS in uso

```
$ cat /etc/resolv.conf
```

- Cambiare l'indirizzo del server DNS in 8.8.8.8

```
# echo "nameserver 8.8.8.8" > /etc/resolv.conf
```

- Scoprire l'indirizzo IP di lmgtfy.com

```
$ nslookup lmgtfy.com
```

11. Rimuovere l'indirizzo IP di eth0 tramite ip

```
# ip addr flush eth0
```

12. Disattivare eth0

```
# ip link set eth0 down
```

13. Fate una copia di backup del file interfaces (es. interfaces.bak)

```
# cp /etc/network/interfaces /etc/network/interfaces.bak
```

14. Impostare l'indirizzo 10.0.2.15, la maschera 255.255.255.0, e accendere la scheda
usando ifup e il file interfaces (Cancellare le impostazioni di eth0 già presenti)

```
# nano /etc/network/interfaces
iface eth0 inet static
    address 10.0.2.15
    netmask 255.255.255.0
    gateway 10.0.2.2
# ifup eth0
```

15. Aggiungete al file la sezione per impostare la scheda all'avvio

```
# nano /etc/network/interfaces
auto eth0
```

3 CONFIGURAZIONE DI DHCP E TEST DI CONNETTIVITÀ

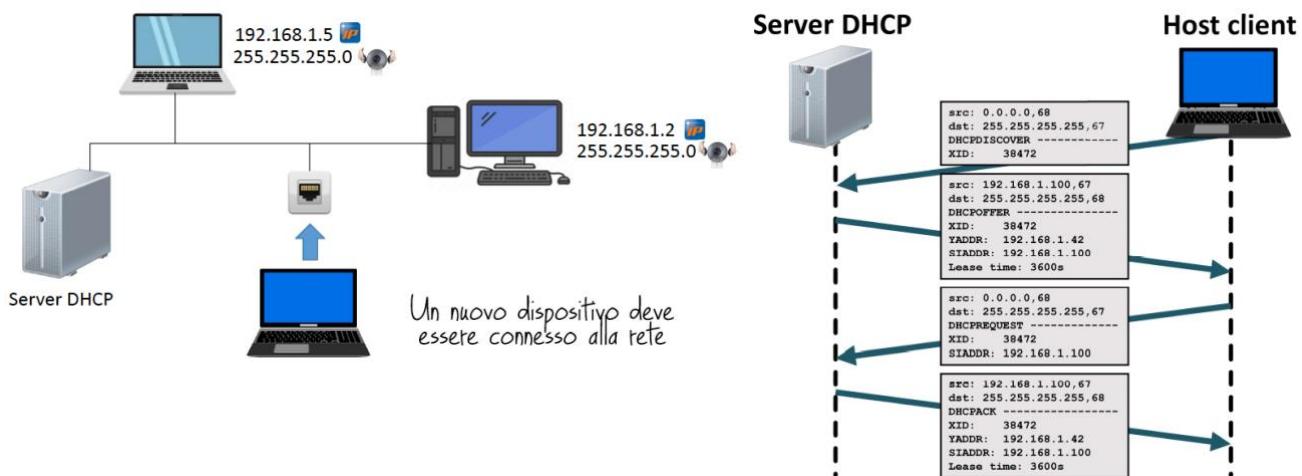
3.1 DHCP (DYNAMIC HOST CONFIGURATION PROTOCOL)

Consente la configurazione **automatica** e **dinamica** dei parametri TCP/IP degli host.

3.1.1 RETI CON DHCP

All'interno della rete c'è un **server DHCP** che configura i parametri di rete degli host: il server/router rileva i nuovi dispositivi che vogliono connettersi alla rete e gli assegna i 4 elementi necessari per la connessione alla rete.

- ! In una rete possono esserci più server DHCP: ciascuno di essi gestisce una sottorete.
- ! Il client che deve collegarsi alla rete tramite DHCP deve essere configurato DHCP.



1. Un dispositivo manda un segnale (**DHCPDISCOVER**) che verrà rilevato da uno o più server DHCP (il messaggio è stato inviato in broadcast, dst: 255.255.255.255). Il sorgente è 0.0.0.0, 68 perché:
 - a. 0.0.0.0 → Indirizzo fittizio per indicare che non ho ancora un indirizzo IP all'interno della rete.
 - b. 68 → Porta riservata al server DHCP.
2. Il client riceve la risposta (**DHCPOFFER**) da parte del server, che sarà composta da:
 - a. src: 192.168.1.100, 67 → Indirizzo IP del server DHCP.
 - b. dst: 255.255.255.255, 68 → Anche il server manda il messaggio in broadcast, perché l'host non può ancora essere indirizzato.
 - c. YADDR: 192.168.1.42 → Indirizzo IP offerto.
 - d. Lease time: 3600s → Tempo di vita dell'IP (durata dei parametri di rete a partire dal momento in cui il server capisce che l'host ha accettato l'IP offerto).
3. Il client accetta l'indirizzo IP proposto tramite la **DCHPREQUEST**.
4. Il server dà l'ok con **DHCPACK** inviando le stesse informazioni inviate al punto 2.

Perché ogni volta il server manda tutto in broadcast? Se ci sono più server DHCP, tutti invieranno le proprie "credenziali" di rete, e sarà l'host a decidere quale utilizzare: tutti i server devono sapere quale indirizzo IP è stato accettato, tra tutti quelli inviati.

Gli indirizzi che possono essere assegnati a un indirizzo IP sono statici: il server ha un certo numero di indirizzi IP da assegnare dinamicamente, mentre tutti gli altri sono assegnati staticamente.

3.1.2 CONFIGURAZIONE SERVER DHCP

```
#apt-get install isc-dhcp-server
```

Installazione.

Nel file di configurazione /etc/default/isc-dhcp-server dobbiamo scrivere

```
INTERFACES="eth0"
```

in quanto INTERFACES ci dice l'interfaccia sulla quale agisce il server DHCP (in questo caso eth0).

Nel file di configurazione /etc/dhcp/dhcpd.conf dobbiamo andare a

[man dhcpd.conf](#)

scrivere

```
option domain-name-servers 192.168.0.2, 8.8.8.8; #server DNS  
option routers 192.168.0.1; #default gateway  
default-lease-time 3600;  
#subnet -> sottorete; netmask -> maschera di sottorete  
#range di indirizzi IP (dinamici) gestiti dal server DHCP: tutti gli  
#altri sono indirizzi statici o non gestiti dal server DHCP in questione  
subnet 192.168.0.0 netmask 255.255.255.0 {  
    range 192.168.0.10 192.168.0.100  
}
```

Gli indirizzi IP assegnabili dinamicamente sono divisi negli altri server: il complemento dell'unione di tutti questi indirizzi è assegnato staticamente.

Dopo le modifiche dobbiamo scrivere

```
#systemctl restart isc-dhcp-server.service
```

3.1.3 CONFIGURAZIONE CLIENT DHCP

Nel client è sufficiente andare a modificare il file di configurazione /etc/network/interfaces, in particolare si sostituisce **dhcp** alla parola **static** (che indica la configurazione statica).

[man interfaces](#)

```
auto lo eth0  
  
iface lo inet loopback  
  
iface eth0 inet dhcp
```

3.2 TEST DI CONNETTIVITÀ'

3.2.1 INTERNET CONTROL MESSAGE PROTOCOL

Protocollo di servizio che **rileva malfunzionamenti**, e scambia informazioni di controllo e messaggi di errore.



3.2.2 COMANDO ping

man ping

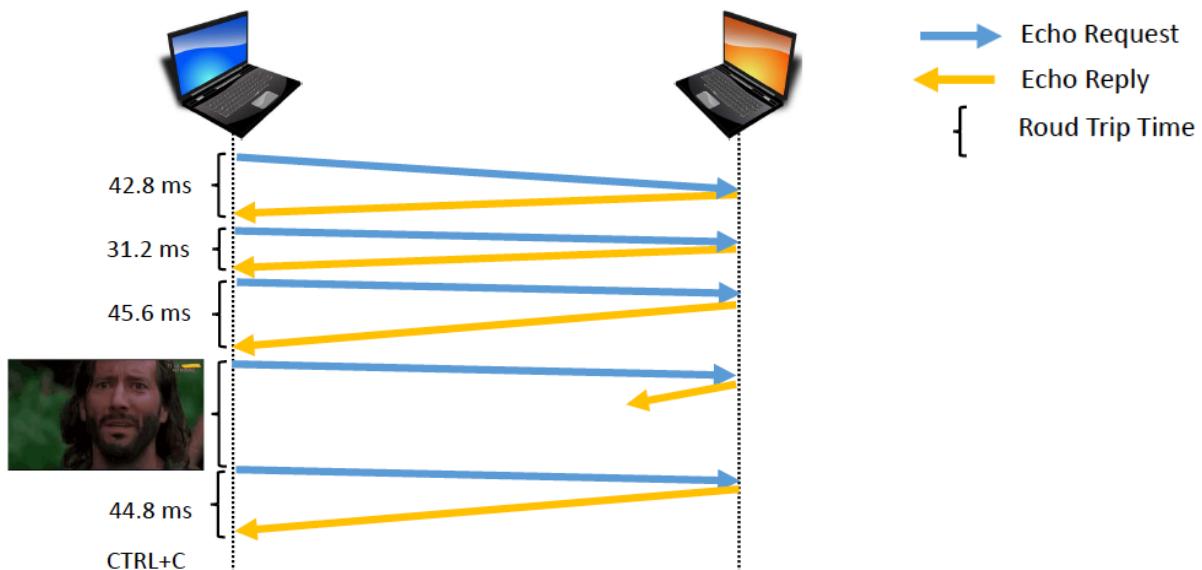
Testa la connettività tra l'host che lo esegue e un host remoto, contattandolo con brevi messaggi per assicurarsi del fatto che sia "still alive".

```
$ping www.apple.com
```

```
$ping 192.168.2.34
```

Iterativamente, invia a un destinatario un messaggio ICMP **Echo Request**, attende un messaggio ICMP **Echo Reply** e misura il tempo in millisecondi impiegato a raggiungere l'host destinatario e tornare indietro.

- ! Questo comando non è eseguibile verso reti private, in quanto sono protette da firewall e per comunicare con l'esterno fanno uso di una porzione di rete, chiamata DMZ, nella quale rimangono eventuali problemi o errori (non si propagano a tutta la rete).



- ! Perché cambia il **round trip time**?
 - Cambia il traffico all'interno del percorso.
 - Il percorso seguito dai pacchetti non è sempre lo stesso.

Nel dettaglio:

1. ping, eseguito da A, invia a B una serie di pacchetti *Echo Request* (per default, uno al secondo).
2. Quando B riceve un *Echo Request*, invia un pacchetto *Echo Reply* ad A.
3. ping calcola la percentuale di pacchetti ricevuti e il *Round Trip Time (RTT)*.
4. Al termine del comando, **mostra le statistiche**.
 - a. Se non specificato con le opzioni, termina solo se l'utente lo interrompe con Ctrl+C.

La freccia gialla che si interrompe (nella figura sopra) sta ad indicare che al primo host non è arrivato nulla entro un intervallo di tempo (time-out) e quindi si ha **packet-loss** (il pacchetto si considera perso). Si può stabilire se il pacchetto è stato perso con *Echo Request* o *Echo Reply*? **No**, posso sapere solo che la risposta non è arrivata.

Se utilizzo ping posso controllare se effettivamente la rete funziona, però non vale il contrario: se ping **non funziona**, non è detto che sia sbagliata la configurazione di rete. Possono esserci infatti diversi problemi:

1. **Network unreachable**: l'host locale non ha route valide per raggiungere l'host remoto.
2. **100% packet lost**: l'host locale non ha ricevuto alcun pacchetto di risposta.
3. **Unknown host**: non è stato possibile risolvere il nome di host specificato.

OPZIONI ping	
-c (count)	Specifica il numero di richieste da inviare.
-i (interval)	Specifica l'intervallo tra le richieste.
-q (quiet)	Visualizza solamente le statistiche finali.
-s (size)	Dimensione in byte del pacchetto, <u>al netto</u> degli header ICMP di 8 byte.

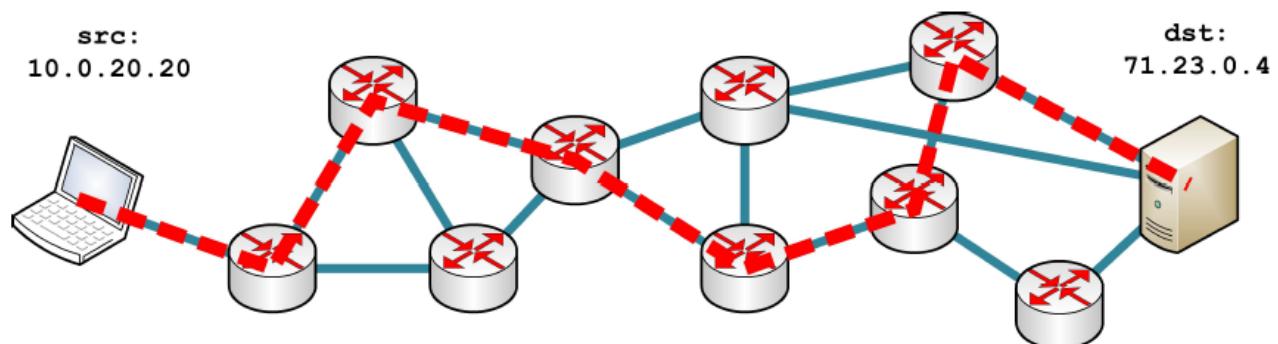
3.2.3 COMANDO traceroute

[man traceroute](#)

È un'estensione del comando ping. Oltre a dire se il destinatario ha risposto, **mostra il percorso** che un pacchetto IP effettua per raggiungere un host destinatario.

Con percorso percorso si intendono **gli indirizzi IP dei router attraversati**.

\$ traceroute www.unipi.it



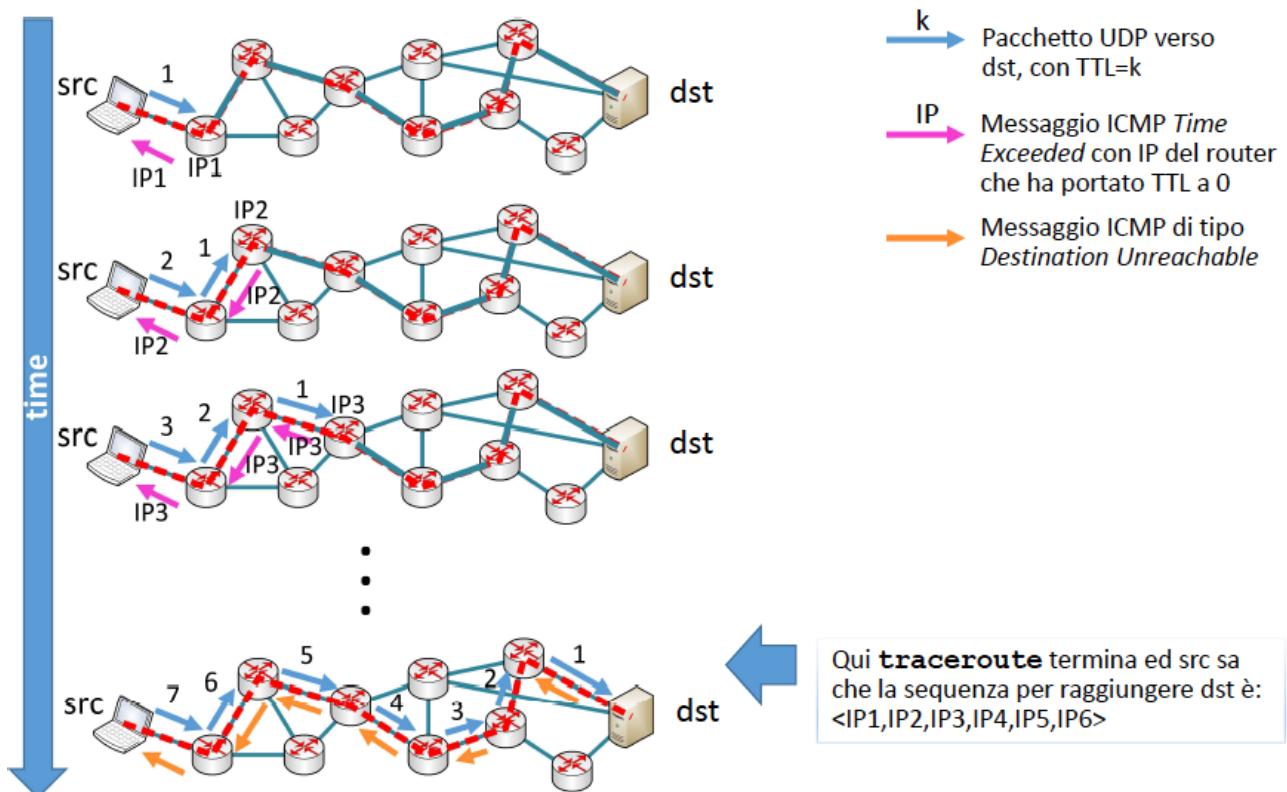
GESTIONE DI time-to-leave

Il campo **TTL** (time-to-live) del datagram IP specifica il numero di apparati di rete che il pacchetto può attraversare prima di “scadere”.

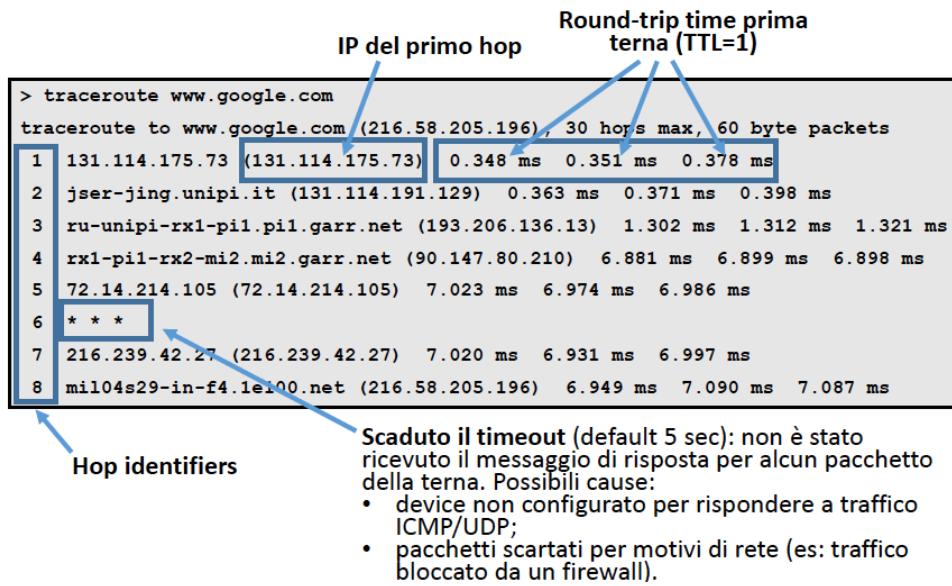
Ogni router che riceve un pacchetto, prima di inoltrarlo, **diminuisce TTL di 1**. Se TTL = 0, invia al mittente un messaggio di errore ICMP di tipo **Time Exceeded con l'indirizzo del router che ha portato TTL a 0**.

Funzionamento:

1. traceroute sfrutta la gestione di **TTL**.
2. Invia all'host destinatario una **serie di terne di pacchetti UDP con TTL crescente**:
 - a. La prima terna con TTL = 1, la seconda con TTL = 2, e così via.
3. Tieni traccia degli indirizzi IP all'interno dei messaggi **ICMP Time Exceeded** ricevuti finché l'ultimo pacchetto della serie raggiunge l'host destinatario.
4. Il destinatario, quando viene raggiunto, non manda ICMP Time Exceeded, ma un **ICMP Destination Unreachable**.
 - a. UDP ha bisogno di una porta. I pacchetti verso il destinatario necessitano di una porta in ascolto. Scegliendone una alla cieca, è improbabile beccarne una in ascolto, ecco perché l'host invia Destination Unreachable (con code = 3, cioè *port unreachable*).
5. traceroute ordina gli IP in base al TTL crescente e ricostruisce il percorso. Fornisce anche i round-trip dei pacchetti.



OUTPUT DI traceroute



- ! Perché traceroute stampa *? Il timeout è scaduto: non riceviamo mai la risposta.
- ! Quale potrebbe essere la motivazione per non ricevere pacchetti di un certo tipo? Ad esempio, non subire attacchi dall'esterno (attacchi DOS).

SVANTAGGI

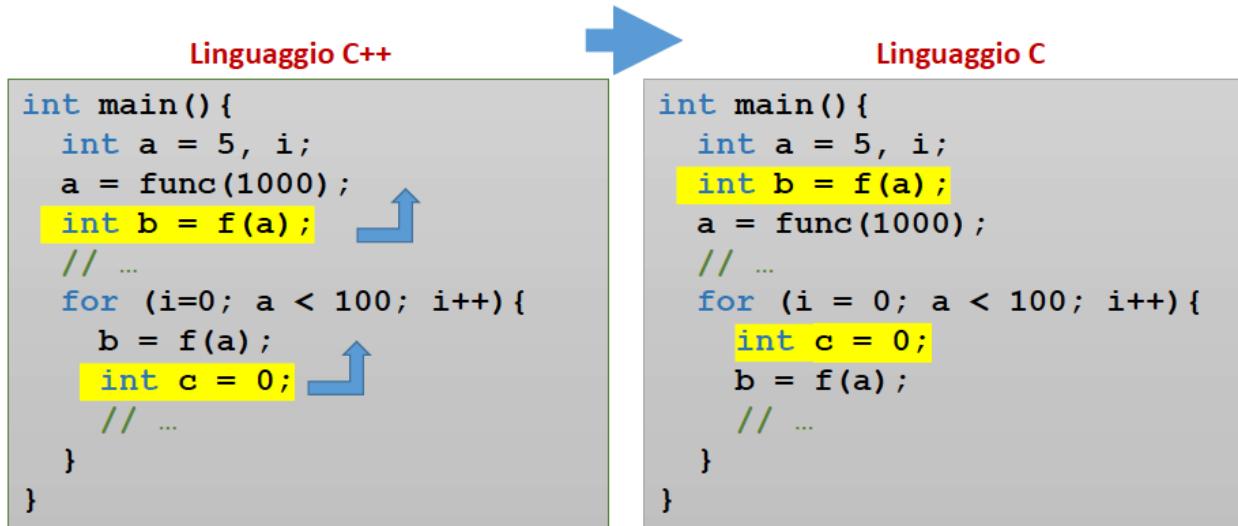
1. I pacchetti possono seguire più percorsi, quindi **gli IP ricavati possono riferirsi a più percorsi**.
2. Se i pacchetti e i messaggi ICMP seguono percorsi diversi, **il calcolo del round-trip è inaffidabile**.

4 PROGRAMMAZIONE DISTRIBUITA IN C

4.1 DEFINIZIONE DI VARIABILI

Possono essere definite solo all'inizio di un blocco.

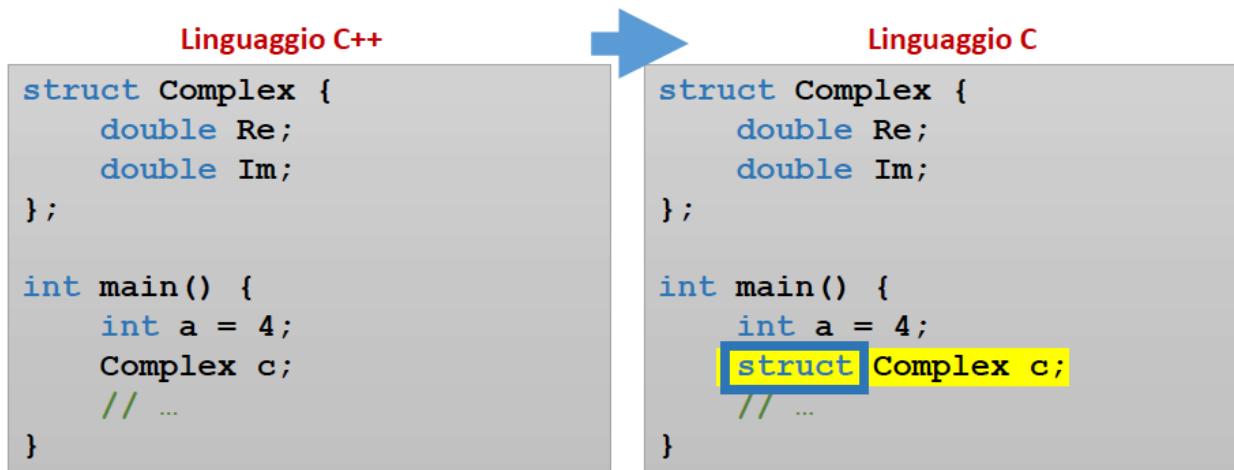
- Facciamo riferimento allo standard C89.



4.2 STRUTTURE

- Non esistono le classi in C.

Si deve inserire sempre **struct** quando si crea una variabile di tipo struttura.



4.3 MEMORIA DINAMICA (HEAP)

Si gestisce con le funzioni **malloc()** per l'allocazione, e **free()** per la deallocazione.

man malloc
man 3 free

PARALLELISMO CON C++

malloc()	new()
free()	delete()

```
#include <stdlib.h>

int main() {
    int dimensione = 5;
    void* punt;
    punt = malloc(dimensione);
    if (punt == NULL) {
        // Gestione errore
    }
    // ...
    free(punt);
}
```

allocà un'area di memoria di 5 byte che sarà puntata da *punt*

rilascia l'area di memoria di 5 byte

- malloc() restituisce un puntatore all'area di memoria allocata.

4.3.1 ALLOCAZIONE CON COERCIZIONE

Per coercizione si intende il **cast** → Se ho bisogno di allocare un'area di memoria di un certo numero di double devo allocare un'area di memoria uguale a **dimensione double per n** numero di elementi: per fare in modo che quest'area di memoria sia trattata come un double devo fare il cast.

(double*)

- La dimensione del double (e in generale di tutti i tipi) dipende dall'architettura: in alcune situazioni devo avere la **certezza** che, ad esempio, se mando un double chi riceve deve allocarlo con 8 byte (come quelli mandati) → In rete non si può dare nulla per scontato: serve un modello di standardizzazione, per far sì che i programmi si capiscano.

```
#include <stdlib.h>

int main() {
    int dimensione = 10;
    double* punt;
    punt = (double*)malloc(sizeof(double)*dimensione);
    if (punt == NULL) {
        // Gestione errore
    }
    // ...
    free(punt);
}
```

l'area di memoria
 è trattata come
 array di double

numero di byte da
 allocare (8*10=80)

4.4 INGRESSO-USCITA (LIBRERIA stdio.h)

man 3 printf
man scanf
man stdio

4.4.1 GESTIONE DELL'USCITA

```
int printf(char* formato, lista_argomenti)
```

Stampa `lista_argomenti` sullo standard output (video) nel `formato` ricevuto come primo parametro.

Restituisce il **numero di caratteri stampati**.

- Se si usa, per esempio, "%3.2f" si stampa un valore in virgola mobile con 3 cifre e 2 cifre di precisione (dopo la virgola).

Formato	Cosa viene stampato
%d, %i	Intero decimale
%f	Valore in virgola mobile
%c	Un carattere
%s	Una stringa di caratteri
%o	Numero ottale
%x, %X	Numero esadecimale
%u	Intero senza segno
%f	Numero reale (float o double)
%e, %E	Formato scientifico
%%	Carattere '%'

ESEMPI DI USCITA	
ESEMPIO	OUTPUT
#include <stdio.h> char* str = "Com'è bello il C!\n"; //Es. 1 printf(str); //Es. 2 printf("Messaggio:\n %s", str); //Es. 3 printf("Da oggi pane e C!"); int i = 5; //Es. 4 printf("i = %d\n", i);	//Es.1 >Com'è bello il C! — //Es.2 >Messaggio: Com'è bello il C! — //Es.3 >Da oggi pane e C!_ //Es.4 >i = 5 —

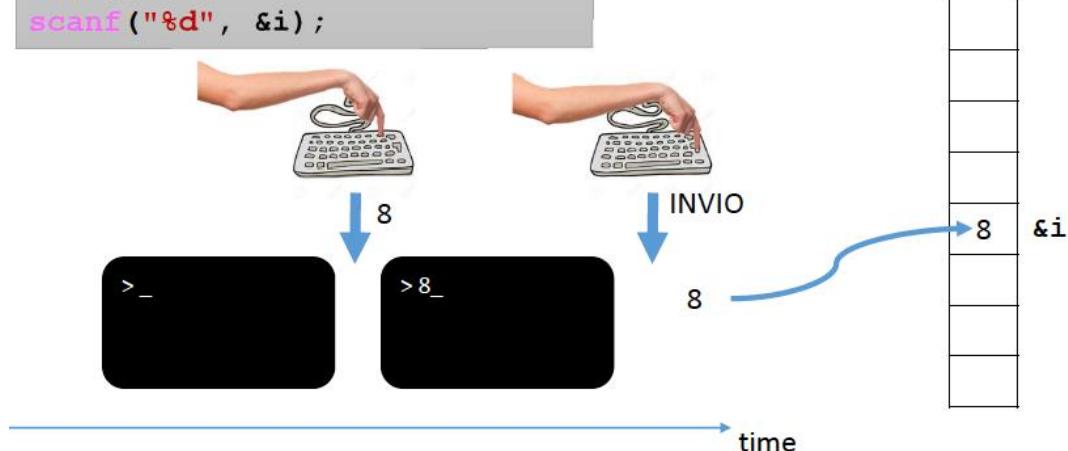
4.4.2 GESTIONE DELL'INGRESSO

```
int scanf(char* formato, lista_indirizzi)
```

Legge dallo standard in (tastiera) una sequenza di caratteri (cifre o lettere) e li memorizza agli indirizzi ricevuti in `lista_indirizzi` nel `formato` ricevuto come primo parametro.

```
#include <stdio.h>

int i;
scanf("%d", &i);
```



4.5 STRINGHE

Nel C non ci sono le stringhe (e non ci sono librerie per simularle): sono **array di caratteri**.

4.5.1 DIMENSIONI

Restituisce il **numero di celle escludendo il carattere di fine stringa**.

man strlen
man strcmp

- Se vogliamo inviare anche il carattere di fine stringa, dobbiamo inviare **strlen + 1**.

```
#include <string.h>

//Lunghezza
char* str1 = "Gatto \n";
int len;
len = strlen(str1);
```

G a t t o \n \0

8 byte allocati, ma **strlen()**
restituisce 7!

4.5.2 CONFRONTO

```
#include <string.h>

//Confronto
char* str2 = "Gattone \n";
int ret;
ret = strcmp(str1, str2);
```

strcmp() confronta le due stringhe un carattere alla volta:

ret = 0	Se le stringhe sono identiche.
ret < 0	Se il primo carattere di str1 diverso dal corrispondente di str2 ha codifica ASCII minore.
ret > 0	Viceversa.

4.5.3 COPIA E CONCATENAZIONE

```
#include <string.h>

//Copia
char str1[20];
int n = sizeof(str1);
strncpy(str1, "Gatto \n", n);
//Concatenazione
char* str2 = "nero\n";
strncat(str1, str2, 6);
```

G a t t o \n n e r o \n \0

- Fare attenzione a **copiare anche il terminatore ('0')**.
- Le funzioni **non controllano che ci sia spazio** nella stringa di destinazione.

man strncpy
man strncat

4.6 GESTIONE DEI FILE

Introduciamo il tipo **FILE**, un tipo opaco (non è fornita l'implementazione).

4.6.1 APERTURA FILE

```
#include <stdio.h>

//Apertura file
FILE* fd;
fd = fopen("/tmp/gatto.txt", "r");
if (fd == NULL) {
    //Gestione errore
}
```

man fopen

Si specifica un **percorso** e una **modalità di apertura**:

r	Sola lettura.
w	Sola scrittura.
r+	Lettura e scrittura.
a	Append.
a+	Lettura e append.

Se si specifica scrittura o append e il file non esiste, **il file viene creato**.

4.6.2 LETTURA E SCRITTURA

```
#include <stdio.h>

//Lettura da file
int ret, n;
FILE *fd1;
fd1 = fopen("/tmp/gatti.txt", "r");
ret = fscanf(fd1, "%d", &n);

//Scrittura su file
char* str = "Gatto!\n";
FILE* fd2;
fd2 = fopen("/tmp/gatti2.txt", "w");
ret = fprintf(fd2, "%s", str);
```

Funzionano come **scanf()** e **printf()**, ma usano il file specificato anziché **stdin** e **stdout**.

man fscanf
man fprintf

4.6.3 STATISTICHE E CHIUSURA FILE

```
#include <stdio.h>
#include <sys/stat.h>

//Dimensione
int ret, size;
struct stat info;
ret = stat("/tmp/foo.txt", &info);
size = info.st_size;

//Chiusura file
FILE* fd;
fd = fopen("/tmp/bar.txt", "w");
fclose(fd);
```

Non serve aprire il file per usare **stat()**.

man 2 stat
man fprintf

! È necessario chiudere **sempre** il file terminate le operazioni.

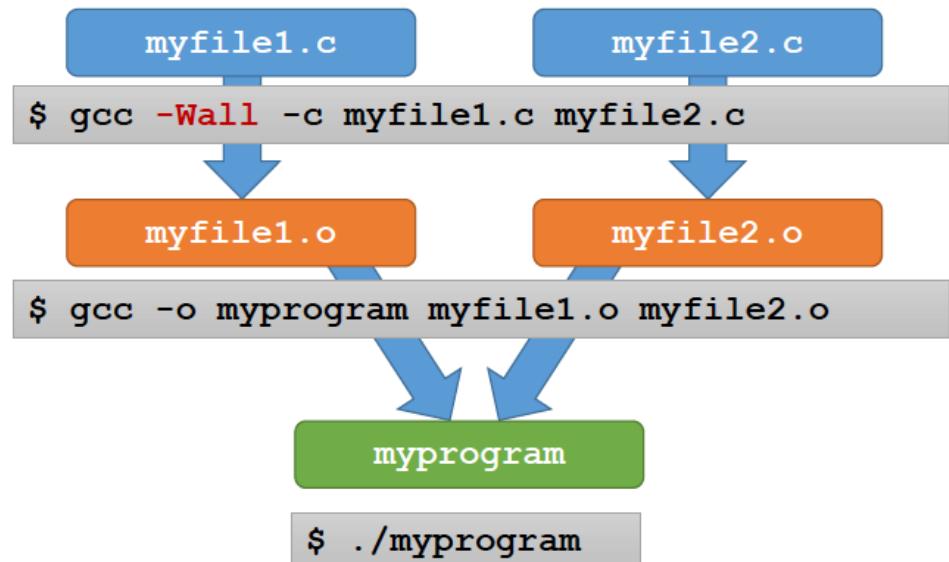
4.7 COMPILAZIONE

Useremo la *GNU Compiler Collection (GCC)*.

Con la compilazione, si creano i file oggetto a partire dai file sorgente

Con il linking, si crea un file eseguibile a partire dai file oggetto

Esecuzione



4.8 PROGRAMMAZIONE DISTRIBUITA IN C

4.8.1 COOPERAZIONE TRA PROCESSI

Due processi possono essere **indipendenti** o **cooperanti**, su una o più macchine (in questo caso si parla di sistemi distribuiti).

Abbiamo diverse modalità di cooperazione.

SINCRONIZZAZIONE

Tramite **semafori**. Calcolatori Elettronici.

COMUNICAZIONE

Memoria condivisa. Sistemi Operativi.

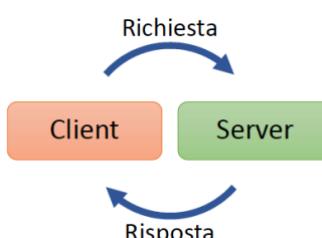
Chiamata a procedura remota. Basi di Dati.

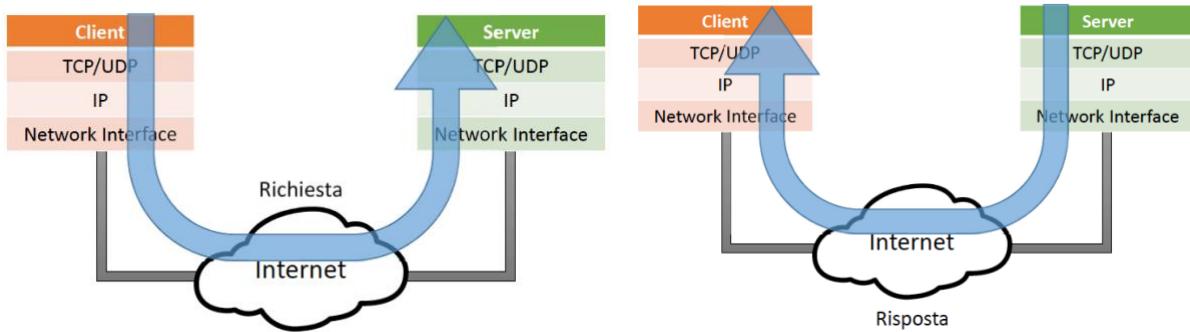
Scambio di messaggi. Reti Informatiche.

4.8.2 MODELLO CLIENT-SERVER

Paradigma basato su **scambio di messaggi**, usato principalmente per sistemi distribuiti.

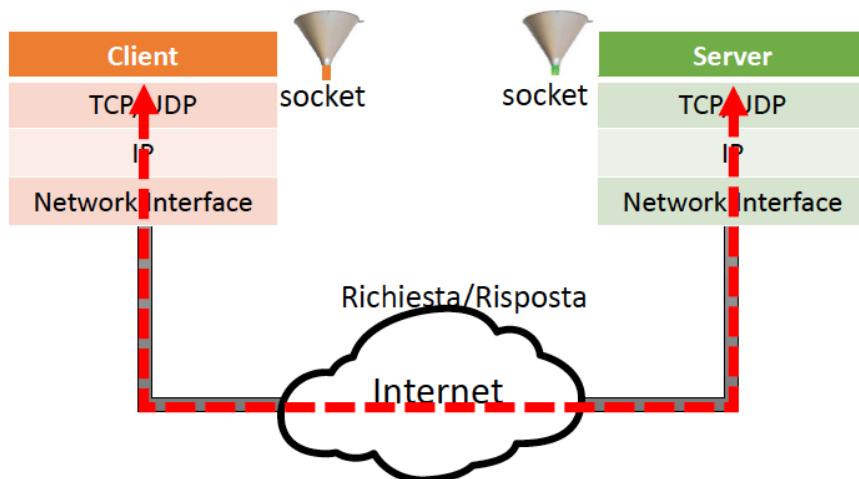
- Con client e server si intendono sia le applicazioni che i dispositivi hardware.





4.8.3 SOCKET

Un socket è un'estremità di un canale di comunicazione fra due processi (programmi) in esecuzione su macchine connesse in rete.



Il sistema operativo offre le primitive per:

1. **Creare** un socket.
2. Assegna**rgli** un **indirizzo**: identificarlo in rete.
3. **Connettersi** a un altro socket.
4. **Accettare** una connessione.
5. **Inviare e ricevere** dati attraverso i socket.

4.8.4 PROCESSO SERVER

Programma in esecuzione sulla macchina server (strato applicazione): offre un servizio (o più) e fa uso di **system call** per utilizzare i socket.

- Le system call utilizzano **primitive del sistema operativo**.

4.8.5 PRIMITIVA `socket()`

Creazione di un socket: viene chiamata quando dobbiamo creare un endpoint, un punto di connessione tra un processo e l'altro.

man 7 ip
man 2 socket
man 7 socket

```
#include <sys/types.h>
#include <sys/socket.h>

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

domain	Famiglia di protocolli da usare: 1. AF_LOCAL – Comunicazione locale. 2. AF_INET – Protocolli IPv4, TCP e UDP.
type	Tipologia di socket: 1. SOCK_STREAM – Connessione affidabile, bidirezionale (TCP). 2. SOCK_DGRAM – Connectionless, invio di pacchetti UDP.
protocol	Sempre a 0.

La `socket()` restituisce un **descrittore di file** (-1 in caso di errore), cioè un numero che rappresenta un file, una pipe, o un socket (come in questo caso) aperto al processo e sul quale il processo può effettuare operazioni.

- Dopo la chiamata `socket()`, il socket non è ancora associato né a un indirizzo IP né a una porta.

4.8.6 STRUTTURE PER GLI INDIRIZZI

```
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t sin_family;      /* Address Family: AF_INET */
    in_port_t sin_port;         /* Port (espresso in network order) */
    struct in_addr sin_addr;    /* Address */
};

/* Internet address */
struct in_addr {
    uint32_t s_addr; /* Address (espresso in network order)*/
};
```

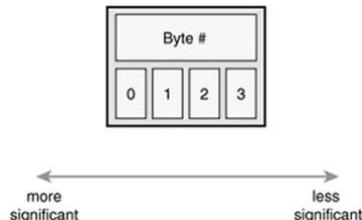
_t – I tipi con questo underscore è come se fossero dei `typedef`: sono garantiti in termini di dimensioni, numero di byte, in modo da poter essere utilizzati in tutte le architetture (tipi TIMBRATI).

5 PROGRAMMAZIONE DISTRIBUITA IN C

– PARTE 1

5.1.1 ENDIANCESS (ORDINE DEI BYTE)

Calcolatori diversi possono usare modi diversi di **posizionare i byte** all'interno di una word.



ESEMPIO: numero **422990**, in 32 bit (4 byte).

- **Memorizzazione big-endian:** per primo il byte più significativo (MSB).

Indirizzo A	Indirizzo A+1	Indirizzo A+2	Indirizzo A+3
0000 0000	0000 0110	0111 0100	0100 1110

- **Memorizzazione little-endian:** per primo il byte meno significativo (LSB).

Indirizzo A	Indirizzo A+1	Indirizzo A+2	Indirizzo A+3
0100 1110	0111 0100	0000 0110	0000 0000

ENDIANNESS IN RETI E HOST: il formato usato in rete (network order) è **big-endian**, quello dell'host (host order) dipende dall'host.

5.1.2 FUNZIONI DI CONVERSIONE (FUNZIONI NINJA)

man 3 byteorder

- Come argomento devono passare lo stesso tipo di ritorno.
- Non si possono usare per convertire dati da meno di 2 byte.
- Servono per salvare nelle architetture dati tipi di dati da almeno 2 byte: se fosse solo 1 byte, non si potrebbe cambiare l'ordine.

```
#include <arpa/inet.h>

//conversione verso network
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);

//conversione verso l'host
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- I tipi `uint16_t` e `uint32_t` sono **intei senza segno** che occupano **sempre** 16 e 32 bit, rispettivamente, a prescindere dall'host (calcolatore) usato per compilare.
Sono utili quando c'è bisogno di avere il controllo sulla dimensione delle variabili.
Sono definiti nell'header `<stdint.h>`.

5.1.4 FORMATO DEGLI INDIRIZZI

- **Formato numerico:** 32-bit, usato dal computer.
- **Formato presentazione:** stringa in notazione decimale puntata.

man inet_pton
man inet_ntop

```
int inet_pton(int af, const char* src, void* dst);
```

Quando la uso nell'inizializzazione, poi non posso ancora usare l'indirizzo.

af	Famiglia (AF_INET).
src	Stringa del tipo "ddd.ddd.ddd.ddd".
dst	Puntatore ad un'istanza di struttura in_addr.

```
const char* inet_ntop(int af, const void* src, char* dst,
                      socklen_t size);
```

af	Famiglia (AF_INET).
src	Puntatore ad un'istanza di struttura in_addr.
dst	Puntatore a un buffer di caratteri di lunghezza size.
size	Dimensione dell'area di memoria che conterrà l'indirizzo in formato presentazione, deve avere almeno INET_ADDRSTRLEN (16 byte).

5.1.5 CREAZIONE E INIZIALIZZAZIONE DI UN SOCKET

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main () {
    /* Creazione socket */
    int sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del socket */
    struct sockaddr_in indirizzo;
    memset(&indirizzo, 0, sizeof(indirizzo)); // Pulizia
    indirizzo.sin_family = AF_INET;
    indirizzo.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &indirizzo.sin_addr);

    // to be continued...
```

5.2 PROGRAMMAZIONE DISTRIBUITA – LATO SERVER

5.2.1 PRIMITIVA bind()

Assegna un indirizzo a un socket: specifica IP e porta dove il server riceve richieste di connessione.

[man 2 bind](#)

! Il client non ha di solito bisogno di eseguire la bind().

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

sockfd	Descrittore del socket.
addr	Puntatore a struttura di tipo sockaddr. ! Usiamo sockaddr_in tra gli argomenti, quindi occorre un cast del puntatore.
addrlen	Dimensione di addr.

La primitiva bind() restituisce 0 se ha successo, -1 se si verifica un errore.

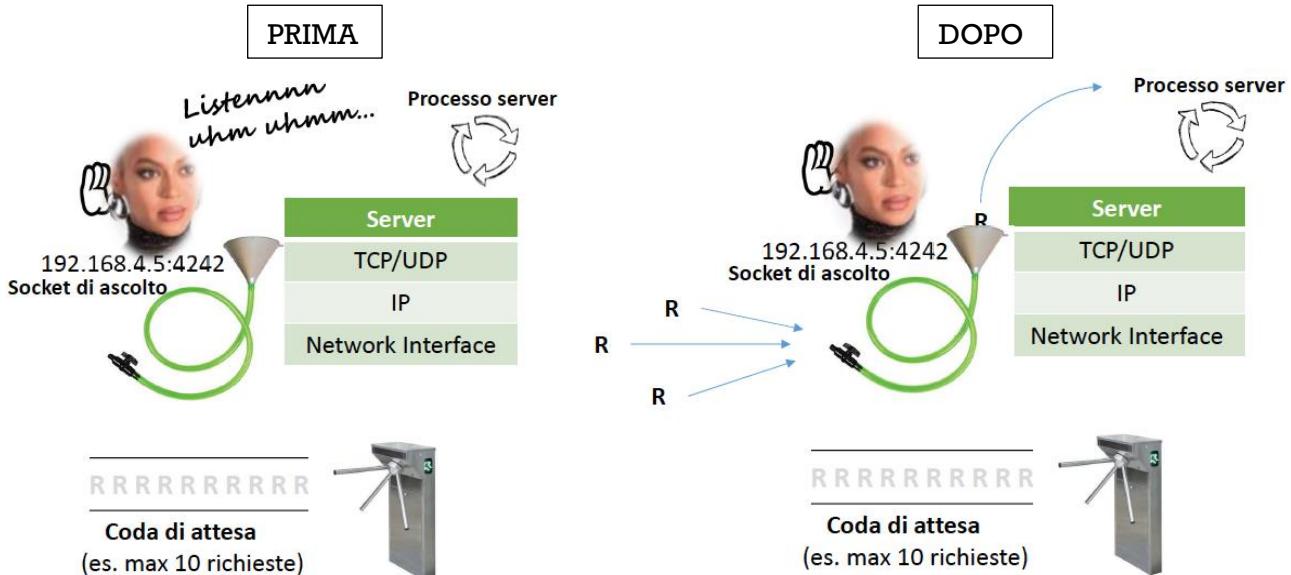
```
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
```

Il socket, però, non è ancora utilizzabile, perché non è in ascolto.

5.2.2 PRIMITIVA listen()

Mette in **ascolto** il socket e crea una **coda di attesa** con un numero **massimo** di richieste.

! È una primitiva non bloccante.



Ascolta le richieste ma non può soddisfarle finché non viene chiamata la `accept()` (che vedremo più avanti).

! Se arriva una richiesta in più rispetto al massimo, questa viene scartata. Come scegliamo la lunghezza della coda?

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

[man 2 listen](#)

Specifica che il socket è **usato per ricevere richieste** di connessione (*socket passivo*).

- ! Si possono mettere in attesa solo i socket `SOCK_STREAM`.

<code>sockfd</code>	Descrittore del socket.
<code>backlog</code>	Dimensione della coda: quante richieste dai client possono essere in attesa di essere gestite.

La funzione restituisce 0 se ha successo, -1 se errore.

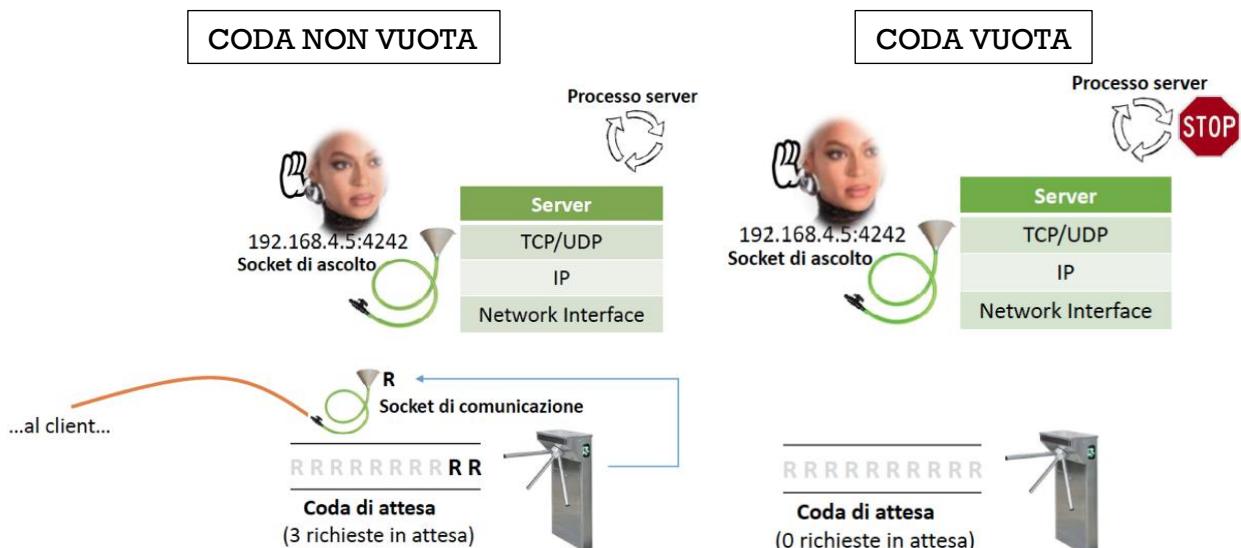
```
ret = listen(sd, 10);
```

5.2.3 PRIMITIVA `accept()`

Viene creato un ulteriore socket di comunicazione per collegare le richieste in coda con il server: è a questo socket che si collega il client.

[man 2 accept](#)

- ! **Primitiva bloccante:** quando il processo server effettua la system call `accept()`, se la coda è vuota, si blocca in attesa che arrivi una richiesta.



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

Accetta una richiesta di connessione giunta al socket di ascolto.

- ! Anche in questo caso, ha senso solo sui socket `SOCK_STREAM`.

<code>sockfd</code>	Descrittore del socket.
<code>addr</code>	Puntatore a struttura (vuota) di tipo <code>struct sockaddr</code> dove viene salvato l'indirizzo del client.
<code>addrlen</code>	Dimensione di <code>addr</code> .

All'arrivo della richiesta, restituisce il **descrittore di un nuovo socket** (il socket di comunicazione) che sarà usato per lo scambio di dati. Altrimenti, viene restituito -1 in caso di errore.

```
struct sockaddr_in cl_addr;
int len = sizeof(cl_addr);
new_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
```

5.2.4 PROCESSO SERVER

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main () {
    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, client_addr; // Due strutture

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo */
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
    my_addr.sin_family = AF_INET ;
    my_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &my_addr.sin_addr);
    // In alternativa: my_addr.sin_addr.s_addr = INADDR_ANY;

    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
    ret = listen(sd, 10);
    len = sizeof(client_addr);
    new_sd = accept(sd, (struct sockaddr*)&client_addr, &len);
    //...
}
```

Essendo un server iterativo, dopo la send e la receive si ritorna alla accept per servire un'altra richiesta.

5.3 PROGRAMMAZIONE DISTRIBUITA – LATO CLIENT

5.3.1 PRIMITIVA connect()

man 2 connect

Permette a un socket locale di inviare una **richiesta di connessione** a un socket remoto.

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

sockfd	Descrittore del socket locale.
addr	Puntatore alla struttura contenente l'indirizzo del socket remoto.
addrlen	Dimensione di addr.

La primitiva restituisce 0 se ha successo, -1 se si verifica un errore.

- ! Gemella di accept: è una primitiva **bloccante** → Il programma si ferma in attesa che la richiesta di connessione sia accettata.

```
ret = connect(sd, (struct sockaddr*)&sv_addr, sizeof(sv_addr));
```

5.3.3 PROCESSO CLIENT

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

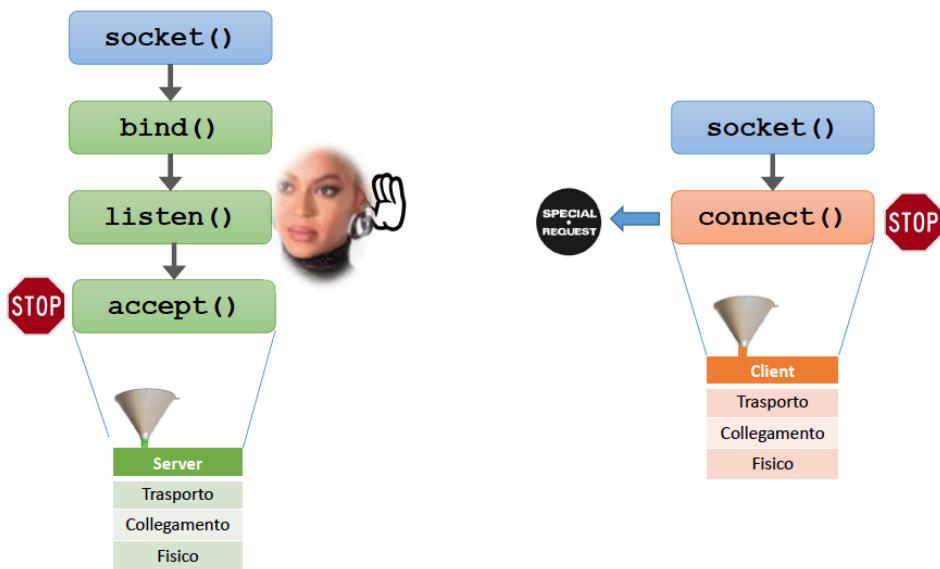
int main () {
    int ret, sd;
    struct sockaddr_in server_addr; // per il server

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

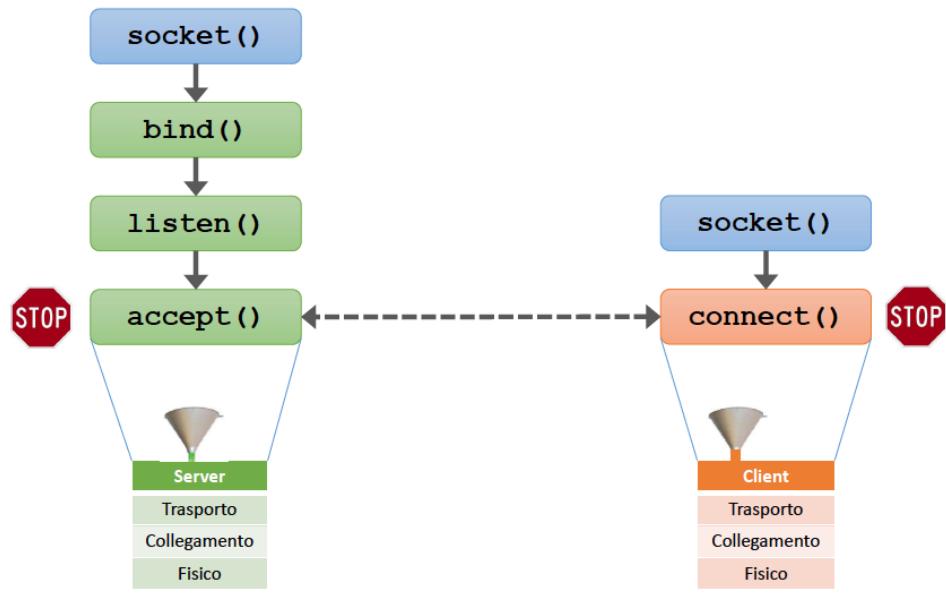
    /* Creazione indirizzo del server */
    memset(&server_addr, 0, sizeof(server_addr)); // Pulizia
    server_addr.sin_family = AF_INET ;
    server_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &server_addr.sin_addr);

    ret = connect(sd, (struct sockaddr*)&server_addr,
                  sizeof(server_addr));
    // ...
}
```

5.3.4 INIZIALIZZAZIONE



5.3.5 ACCETTAZIONE DI CONNESSIONE



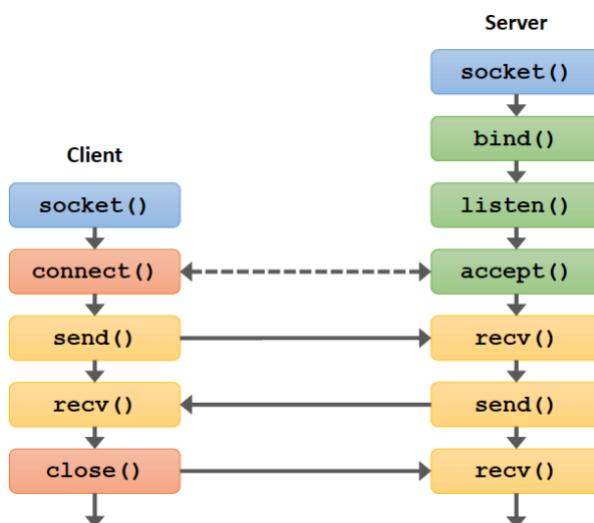
6 PROGRAMMAZIONE DISTRIBUITA – SCAMBIO DI DATI E PROTOCOLLI TEXT E BINARY

6.1 PROGRAMMAZIONE DISTRIBUITA – SCAMBIO DI DATI

In una comunicazione, i due endpoint non possono dare tutto per scontato. In particolare, il destinatario non sa il formato del messaggio: server e client devono mettersi d'accordo sul formato dei dati da scambiarsi.

Ad ogni socket il kernel associa un **buffer in ricezione** e un **buffer in trasmissione** a livello sistema: possiamo quindi bufferizzare i dati da un buffer a livello applicazione ad uno sistema.

- ! Il programmatore ha la certezza che il messaggio arrivi dall'altra parte, ma **non che il messaggio sia stato inviato tutto insieme**: questo dipende dal buffer di trasmissione associato al socket.



6.1.1 PRIMITIVA send()

Invia un messaggio attraverso un socket connesso.

[man 2 send](#)

- ! **Socket connesso:** socket di **comunicazione**, dove fuiscono effettivamente i dati, creato dalla `accept()` quando viene accettata una richiesta.
- ! Cosa si intende per **messaggio**? A livello applicazione abbiamo la corrispondenza
Pacchetto = Messaggio

Ad ogni livello della pila protocollare abbiamo un diverso significato per la parola pacchetto.

```
ssize_t send(int sockfd, const void* buf, size_t len, int flags);
```

sockfd	Descrittore del socket.
buf	Puntatore al buffer contenente il messaggio da inviare.

	! Questo buffer è un'area di memoria in cui il programmatore salva il messaggio da inviare <u>nel suo formato definitivo</u> .
len	Dimensione del messaggio in byte.
flags	Per settare le opzioni (per adesso mettiamolo a 0).

La funzione restituisce il **numero di byte inviati**, oppure -1 se errore.

- Se il valore di ritorno è diverso dal numero di byte dell'area di memoria, non siamo riusciti a copiare tutto correttamente: vuol dire che il buffer di sistema è pieno.
 - Quando potrebbe essere pieno il buffer? Quando i dati non vengono inviati in maniera abbastanza veloce, e quindi non viene vuotato del tutto: il buffer "si ingolfa".
- Cosa fa il programmatore? **Effettua il resto del trasferimento scorrendo il puntatore buf fino all'ultimo byte inviato e richiamando la send()**.

In generale, nella programmazione distribuita non si può supporre che a una sola send() corrisponda una sola receive(), e viceversa: **send() e receive() andrebbero sempre messe in un loop con condizione di uscita ritorno della send() == dimensione memoria**.

La funzione è **bloccante**: il programma si ferma finché non ha scritto il messaggio (tutto o in parte) nel buffer di invio.

```
int ret, sd, len;
char buffer[1024];

//creazione socket

strcpy(buffer, "Hello Server!");
len = strlen(buffer);

// invio
ret = send(sd, (void*)buffer, len, 0);

if (ret < len) {
    // Gestione errore (o ricezione parziale)
}
```

6.1.2 PRIMITIVA recv()

Riceve un messaggio da un socket connesso.

man 2 recv

- ! La `recv()` lavora sul socket **di ascolto**: la richiesta del client arriva sul socket di ascolto.

```
ssize_t recv(int sockfd, const void* buf, size_t len, int flags);
```

sockfd	Descrittore del socket.
buf	Puntatore al buffer in cui salvare il messaggio. ! Questo buffer è l'area di memorizzazione <u>del socket</u> .
len	Dimensione del messaggio desiderato in byte.
flags	Per settare le opzioni.

La funzione restituisce il **numero di byte ricevuti** (copiati nel buffer di ricezione dell'applicazione presi dal buffer di invio), -1 se errore oppure 0 se il socket remoto si è chiuso.

La funzione è **bloccante**: il programma si ferma finché non ha letto almeno un byte.

! Per questo motivo, il valore 0 non può essere confuso con “ricevuti 0 byte”.

```
int ret, sd, bytes_needed;
char buffer[1024];

//...

bytes_needed = 20;

// ricezione
ret = recv(sd, (void*)buffer, bytes_needed, 0);

// Adesso 0 <= ret <= bytes_needed
if (ret < bytes_needed) {
    // Gestione errore (o ricezione parziale)
}

ret = recv(sd, (void*)buffer, bytes_needed, MSG_WAITALL);
// Adesso ret == bytes_needed
```

6.1.3 PRIMITIVA close()

Chiude un socket: non può più essere usato per inviare o ricevere dati.

[man 2 close](#)

! Ogni volta che si apre un socket, lo si deve chiudere: dobbiamo rilasciare tutte le risorse utilizzate.

```
#include <unistd.h>

int close(int fd);
```

fd	Descrittore del socket.
----	-------------------------

La funzione restituisce 0 se ha successo, -1 se errore.

! L'host remoto riceve 0 dalla recv().

6.1.4 ESEMPIO DI UTILIZZO SEND E RECEIVE RECEIVE

Ricezione di un messaggio di lunghezza *n*.

```
//...

int s;           //socket
char* buf;       //buffer
int n = 50;      //dimensione del messaggio in byte
```

```

int letti, letti_dopo;

/* Il ciclo interno verifica che la recv() non ritorno un messaggio più
corto di quello atteso (n byte). recv() può restituire un numero
inferiore di byte (risorse limitate, buffer condivisi, messaggi di
grandi dimensioni...) */
/* Faccio la prima recv */
letti = recv(s, (void*)buf, n, 0);

/* Se ho un errore, esco */
if(letti == -1) {
    perror("Errore in lettura!");
    exit(1);
}

/* Se ho ricevuto un numero di byte inferiore a n, invoco di nuovo la
recv() */
while(letti < n) {
    letti_dopo = recv(s, (void*)&buf[letti], n - letti, 0);
    if(letti_dopo == -1) {
        perror("Errore in lettura!");
        exit(1);
    }
    letti += letti_dopo;
}

//...

```

SEND

Per messaggi di piccola dimensione la frammentazione è poco probabile, ma con dimensioni superiori il pacchetto può essere suddiviso dai livelli sottostanti, e una ricezione parziale diventa più probabile. In caso di congestione, i buffer potrebbero ricevere i dati un po' alla volta.

In ogni caso, in ricezione occorre sempre aspettare l'intero messaggio.

Lo stesso accade quando si spediscono messaggi. Anche la `send()` deve assicurarsi di aver spedito tutti i byte. Se non è così, occorre scorrere il buffer e invocare di nuovo la `send()` sui restanti byte, come visto sopra per la `recv()`.

Invio di un messaggio di lunghezza *n*.

```

int s;                  //socket
char* buf;              //buffer
int n = 50;              //dimensione del messaggio in byte

int spediti, spediti_dopo;

//...

* Faccio la prima send */

```

```

spediti = send(s, (void*)buf, n, 0);

/* Se ho un errore, esco */
if(spediti == -1) {
    perror("Errore in scrittura!");
    exit(1);
}

/* Se ho spedito un numero di byte inferiore a n, invoco di nuovo la
send() */
while(spediti < n) {
    spediti_dopo = send(s, (void*)&buf[spediti], n - spediti, 0);
    if(spediti_dopo == -1) {
        perror("Errore in lettura!");
        exit(1);
    }
    spediti += spediti_dopo;
}

//...

```

6.2 GESTIONE DEGLI ERRORI

Le primitive viste restituiscono -1 quando c'è un errore. In più, settano una variabile `errno` che può essere letta per scoprire il motivo dell'errore.

[man 3 errno](#)

! Nel manuale di ogni funzione c'è l'elenco degli errori possibili.

```

#include <errno.h>

//...

ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) {
    if (errno == EADDRINUSE) /* Gestisci errore */
    if (errno == EINVAL) /* Gestisci errore */
    //...
}

```

EADDRINUSE	Si cerca di fare l'aggancio di un nuovo socket in un indirizzo di cui il sistema operativo non si è ancora liberato.
EINVAL	Indirizzo non valido.

A volte vogliamo solo sapere l'errore e uscire: `perror()` legge `errno` e stampa l'errore su schermo in forma leggibile.

[man 3 perror](#)

```

#include <stdio.h>

//...

ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) {
    perror("Error: ");
}

```

```
    exit(1);
}
//...
```

6.3 PROTOCOLLI TEXT E BINARY

Abbiamo due modi di inviare dati e messaggi:

1. **Protocolli text:** inviano messaggi in formato testo → Prendo il testo, lo converto in una enorme stringa e lo invio in un formato concordato tra mittente e destinatario.
2. **Protocolli binary:** invio direttamente le strutture dati → Invio i dati “grezzi”, binari, così come sono rappresentati sulla macchina, senza convertirli in testo.

Per il testo si usa solitamente la codifica ASCII.

- ! Quanto occupa un intero nel protocollo testo? Dipende dal valore scritto nell'intero.
- ! Devo mandare il carattere di fine stringa? Dipende da come si mettono d'accordo mittente e destinatario: se i messaggi sono di lunghezza variabile, ad esempio, mi fa comodo avere il carattere di fine stringa.

6.3.1 ESEMPIO DI INVIO DEI DATI SBAGLIATO

Un utente sta cercando di passare un'intera struttura dal client al server e viceversa.

Assumiamo che la struttura dati sia la seguente:

```
struct temp {
    int a;
    char b;
}
```

L'utente usa **sendto** (stesso funzionamento di **send**) e invia l'indirizzo della struttura e lo riceve dall'altro lato usando la funzione **recvfrom** (stesso funzionamento di **recv**).

Il problema è che non riesce a riottenere i dati originali nel destinatario. Nella **recvto** salva i dati ricevuti in una variabile di tipo struttura **temp**:

```
n = sendto(sock, &pkt, sizeof(struct temp), 0, &server, lenght);
n = recvfrom(sock, &pkt, sizeof(struct temp), 0,
             (struct sockaddr *)&from, &fromlen);
```

dove **pkt** è la variabile di tipo struttura **temp**.

Nonostante riceva 8 byte di dati, se prova a stampare vengono fuori valori casuali.

QUALE È IL PROBLEMA?

Mandare un indirizzo è **sbagliatissimo**: tale indirizzo è della **propria memoria**; quindi, il puntatore nell'altro host assume tutt'altro significato, e chissà a quale zona di memoria dell'altro host punta.

SOLUZIONE

I dati binari dovrebbero sempre inviati in modo da:

1. Gestire le differenti **endianness**.

2. Gestire i differenti **padding**.
3. Gestire le differenze nelle **dimensioni intrinseche dei tipi**.

Non si dovrebbe mai scrivere un'intera struttura in maniera binaria, né su un file, né su un socket.

Si devono sempre scrivere i vari campi separatamente, e leggerli separatamente.

Si devono quindi avere funzioni del tipo:

```
unsigned char * serialize_int(unsigned char *buffer, int value) {
    /* Write big-endian int value into buffer; assumes 32-bit int and
     * 8-bit char. */
    buffer[0] = value >> 24;
    buffer[1] = value >> 16;
    buffer[2] = value >> 8;   | Altrimenti usare
    buffer[3] = value;        | la htonl()
    return buffer + 4;       //indirizzo del prossimo byte libero
}

unsigned char * serialize_char(unsigned char *buffer, char value) {
    buffer[0] = value;
    return buffer + 1;
}

unsigned char * serialize_temp(unsigned char *buffer,
                             struct temp *value) {
    buffer = serialize_int(buffer, value->a);
    buffer = serialize_char(buffer, value->b);
    return buffer;
}
```

- ! Quando ho una struttura da inviare, devo fare la serializzazione del messaggio: devo farla transitare come flusso di byte.

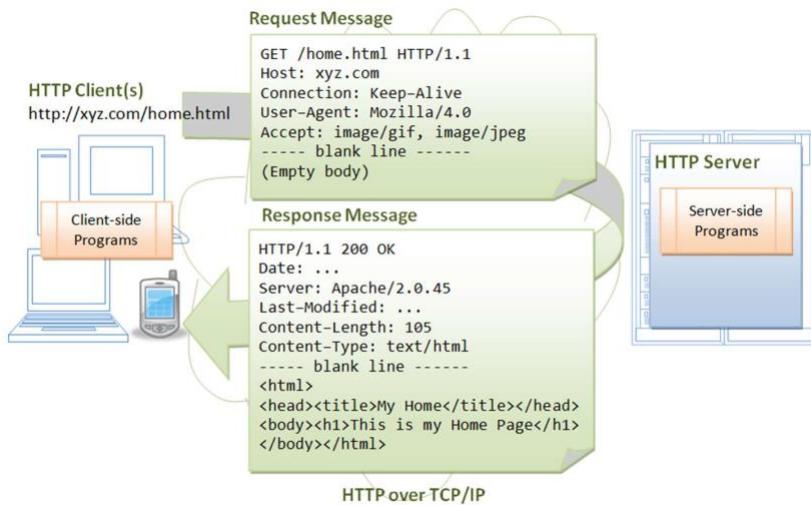
6.3.2 ESEMPIO: CODIFICA DEL NUMERO 1234 IN FORMATO BINARY E TEXT

binary	byte 0	byte 1	byte 2	byte 3
1234	00000000	00000000	00000100	11010010

Text (ASCII)	byte 0	byte 1	byte 2	byte 3
"1234"	011 0001	011 0010	011 0011	011 0100

6.3.3 TEXT PROTOCOLS

ESEMPIO



DA STRUTTURA A STRINGA

La struttura può essere convertita in testo:

```
char buffer[1024];

// Dichiarazione
struct temp{
    int a;
    char b;
};

// Istanziazione
struct temp t;

// Conversione a stringa
sprintf(buffer, "%d %c", t.a, t.b);
```

DA STRINGA A STRUTTURA

Il ricevente deve fare parsing del messaggio:

```
...
// Istanziazione
struct temp t;

// Ricezione
...

// Parsing e memorizzazione nei campi
sscanf(buffer, "%d %c",&t.a, &t.b);
```

6.3.4 BINARY PROTOCOL

Definire messaggi con **struttura fissata**, con campi che rappresentano l'informazione da scambiare.

- Ogni campo ha una **lunghezza** e un **tipo che possa essere trasferito**.
- Alcuni protocolli di tipo binario possono contenere campi di lunghezza variabile, in quel caso il protocollo definisce comunque una **lunghezza massima dei messaggi**.

INVIO

```
...
struct temp{
    uint32_t a;
    uint8_t b;
};

struct temp t;
...

// Convertire in network order prima dell'invio
t.a = htonl(t.a);

// Spedire i campi sul socket 'new_sd'
ret = send(new_sd, (void*)&t.a, sizeof(uint32_t), 0);
...
ret = send(new_sd, (void*)&t.b, sizeof(uint8_t), 0);
...
```

RICEZIONE

```
...
struct temp t;
...

ret = recv(new_sd, (void *)&t.a, sizeof(uint32_t), 0);
if (ret < sizeof(uint32_t)){
    // Gestione errore
}

// Convertire in host order il campo 'a'
t.a = ntohl(t.a);
ret = recv(new_sd, (void *)&t.b, sizeof(uint8_t), 0);
if (ret < sizeof(uint8_t)){
    // Gestione errore
}
```

6.3.5 VANTAGGI E SVANTAGGI DEI DUE PROTOCOLLI

TEXT PROTOCOL	BINARY PROTOCOL
<ol style="list-style-type: none">1. Se lavora con le stringhe, ho già le stringhe convertite.2. Con numeri grandi, ho più occupazione di memoria.3. Vedo meglio tutti i dati: poca sicurezza.	<ol style="list-style-type: none">1. Non devo fare parsing.2. Con numeri grandi, ho meno occupazione di memoria.3. Migliore sicurezza.

6.4 ESERCIZIO 1

Implementare un **server TCP iterativo** che fornisce un messaggio "Hello!" ai client che si collegano.

Implementare il relativo **client**.

- Il client si connette, riceve il messaggio, lo stampa ed esce.

Note:

- Gestire gli errori.
- Implementare una terminazione non forzata del server (non si deve ricorrere a CTRL+C per fermarlo!).
- Server e client **conoscono la dimensione del messaggio**.

6.4.1 CLIENT

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define LEN_REPLY 6 // Lunghezza del messaggio di risposta

int main(int argc, char* argv[]){
    int ret, sd, len;
    struct sockaddr_in srv_addr;
    char buffer[1024];

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del server */
    memset(&srv_addr, 0, sizeof(srv_addr)); // Pulizia
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "127.0.0.1", &srv_addr.sin_addr);
```

```

    ret = connect(sd, (struct sockaddr*)&srv_addr, sizeof(srv_addr));

    if(ret < 0){
        perror("Errore in fase di connessione: \n");
        exit(-1);
    }

    // Attendo risposta
    len = LEN_REPLY;
    ret = recv(sd, (void*)buffer, len, 0);

    if(ret < 0){
        perror("Errore in fase di ricezione: \n");
        exit(-1);
    }

    // Aggiungo il terminatore di stringa che non viene inviato
    buffer[len]='\0';

    printf("%s\n", buffer);

    close(sd);
}

```

6.4.2 SERVER

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[]){

    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, cl_addr;
    char buffer[1024];

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo */
    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "127.0.0.1", &my_addr.sin_addr);

    /* Aggancio del socket all'indirizzo */
    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr) );

    /* Inizio dell'ascolto, coda da 10 connessioni */
}

```

```

    ret = listen(sd, 10);
    if(ret < 0){
        perror("Errore in fase di bind: \n");
        exit(-1);
    }

    while(1){

        // Dimensione dell'indirizzo del client
        len = sizeof(cl_addr);

        // Accetto nuove connessioni
        //*** ATTENZIONE: BLOCCANTE!!! ***
        new_sd = accept(sd, (struct sockaddr*) &cl_addr, &len);

        // Creo la risposta
        strcpy(buffer, "Hello!");
        len = strlen(buffer);

        // Invio risposta (senza '\0' perche' strlen non lo include
        // nella lunghezza)
        ret = send(new_sd, (void*) buffer, len, 0);

        if(ret < 0){
            perror("Errore in fase di invio: \n");
        }

        close(new_sd);
    }
}

```

6.5 ESERCIZIO 2

Implementare un **server TCP mono-processo** che reinvia al mittente un messaggio ricevuto.

Implementare un **client** che, di continuo:

- Legge una stringa da tastiera.
- Se la stringa è "Bye" interrompe la connessione ed esce.
- Altrimenti invia la stringa, riceve la risposta e la stampa.

Il server continua a fare echo al client finché la connessione non si interrompe

Server e client leggono/scrivono **sempre 20 byte**.

- Attenzione al terminatore di stringa!

6.5.1 CLIENT

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

```

```

#define MESSAGE_LEN 20 // Lunghezza del messaggio dal client

int main(int argc, char* argv[]){
    int ret, sd, len;
    struct sockaddr_in srv_addr;
    char buffer[1024];

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del server */
    memset(&srv_addr, 0, sizeof(srv_addr)); // Pulizia
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "127.0.0.1", &srv_addr.sin_addr);

    ret = connect(sd, (struct sockaddr*)&srv_addr, sizeof(srv_addr));

    if(ret < 0){
        perror("Errore in fase di connessione: \n");
        exit(-1);
    }

    len = MESSAGE_LEN;

    while(1){

        // Attendo input da tastiera
        scanf("%s", buffer);

        // Invio al server
        ret = send(sd, (void*) buffer, len, 0);

        if(ret < 0){
            perror("Errore in fase di invio: \n");
            exit(-1);
        }

        // Attendo risposta
        ret = recv(sd, (void*)buffer, len, 0);

        if(ret < 0){
            perror("Errore in fase di ricezione: \n");
            exit(-1);
        }

        printf("%s\n", buffer);

        if(strcmp(buffer,"bye\0") == 0 )
            break;
    }

    close(sd);

}

```

6.5.2 SERVER

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MESSAGE_LEN 20 // Lunghezza del messaggio dal client

int main(int argc, char* argv[]){
    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, cl_addr;
    char buffer[1024];

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    /* Creazione indirizzo di bind */
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4242);
    my_addr.sin_addr.s_addr = INADDR_ANY;

    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
    ret = listen(sd, 10);
    if(ret < 0){
        perror("Errore in fase di bind: \n");
        exit(-1);
    }

    while(1){

        len = sizeof(cl_addr);

        // Accetto nuove connessioni
        new_sd = accept(sd, (struct sockaddr*) &cl_addr, &len);

        while(1){

            // Attendo risposta
            len = MESSAGE_LEN;
            ret = recv(new_sd, (void*)buffer, len, 0);

            if(ret < 0){
                perror("Errore in fase di ricezione: \n");
                continue;
            }

            // Posso fare strcasecmp perché invio anche il fine stringa
            '\0';
            // Invio sempre sul socket 20 byte!
            if( strcmp(buffer, "bye" ) == 0 ){
                close(new_sd);
                break;
            }
        }
    }
}
```

```
    }

    // Invio risposta
    ret = send(new_sd, (void*) buffer, len, 0);

    if(ret < 0){
        perror("Errore in fase di invio: \n");
    }
}

}
```

7 PROGRAMMAZIONE CONCORRENTE

7.1 SERVER CONCORRENTI

7.1.1 TIPI DI PROCESSI SERVER

SERVER ITERATIVO: serve una richiesta alla volta.

- Per ogni richiesta accettata (`accept()`), il processo server la elabora e accoda (tramite la `listen()`) le richieste che sopraggiungono.

SERVER CONCORRENTE: serve più richieste alla volta.

- Per ogni richiesta accettata (`accept()`), il processo server crea un processo che la elabora (detto processo *figlio*).

7.1.2 PRIMITIVA `fork()`

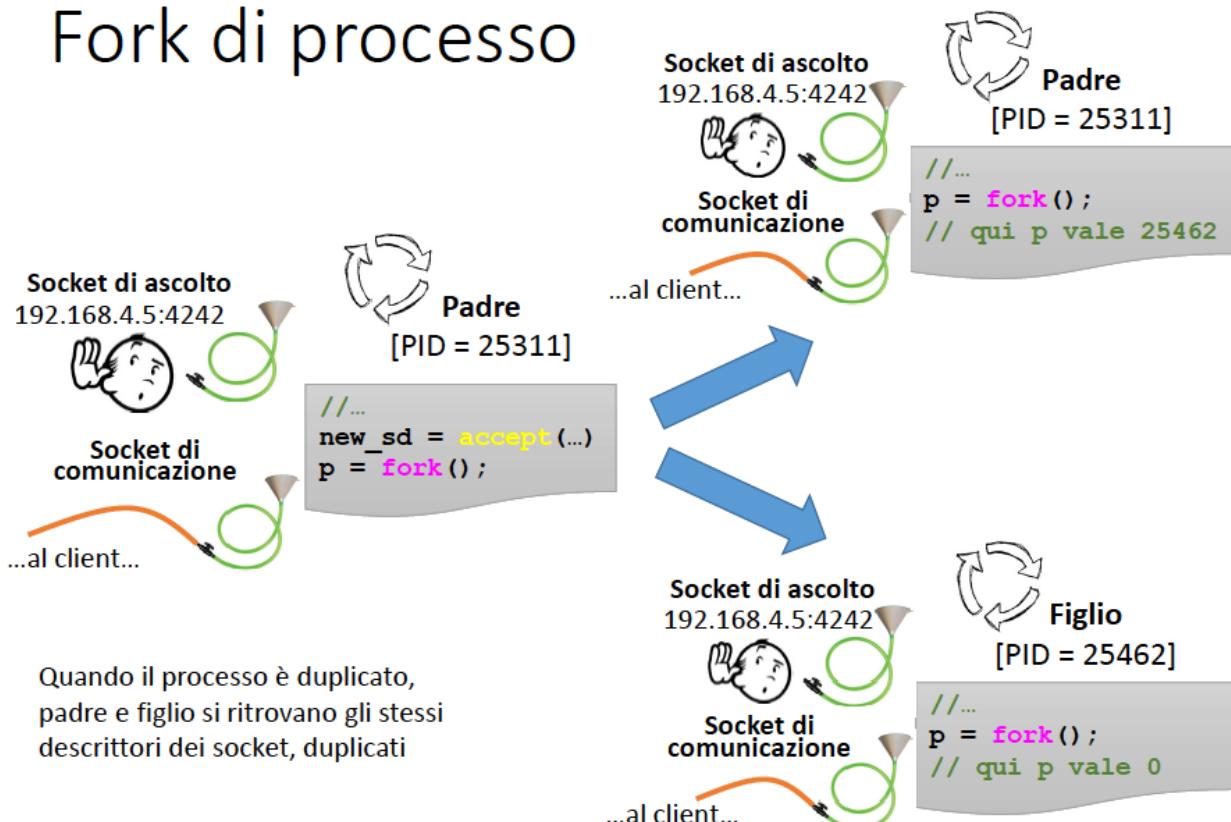
Clona il processo: il processo clone (*figlio*) esegue lo stesso codice del chiamante (*padre*). man 2 fork

```
#include <unistd.h>
pid_t fork(void);
```

- Nel processo padre, restituisce il **process identifier (PID) del processo figlio creato**.
- Nel processo figlio, restituisce 0.
- Restituisce -1 in caso di errore.

Il tipo `pid_t` è un intero con segno. Nella GNU C Library è un `int`.

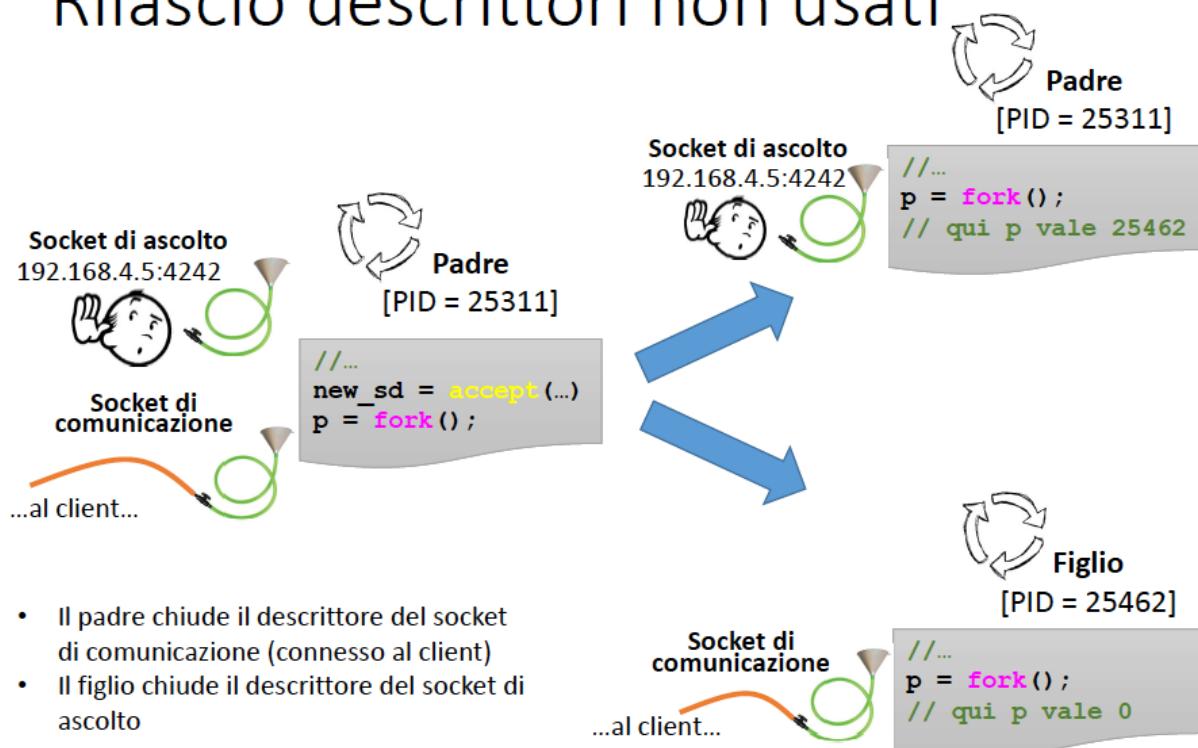
Fork di processo



In questo contesto, il figlio non usa il socket di ascolto, così come il padre non usa quello di comunicazione: è buona norma allora chiudere quelli non necessari nelle due parti.

- ! Non bisogna dimenticare di sfruttare la variabile `errno` per capire se e come si è verificato un errore.

Rilascio descrittori non usati



- Il padre chiude il descrittore del socket di comunicazione (connesso al client)
- Il figlio chiude il descrittore del socket di ascolto

7.1.3 USO DI `fork()`

```
pid_t pid;
//...
while(1) {
    new_sd = accept(sd, ...);
    pid = fork();

    // se pid vale 0 siamo nel FIGLIO
    if (pid == 0) {
        // chiusura del socket sd (il figlio non lo usa)
        close(sd);

        // elaborazione della richiesta (usando il socket new_sd)

        // chiusura del socket new_sd
        close(new_sd);

        // il figlio termina
        exit(0);
    }

    // qui pid!=0, quindi siamo nel PADRE
}
```

FIGLIO

```

    // chiusura del socket new_sd (il padre non lo usa)
    close(new_sd);

    // e fu sera e fu mattina...
}

```

7.1.4 ESEMPIO DI SERVER CONCORRENTE

```

#include ...
int main () {
    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, cl_addr;

    //...

    pid_t pid;
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo */
    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));

    ret = listen(sd, 10);
    len = sizeof(cl_addr);

    while(1) {
        new_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
        pid = fork();

        if (pid == -1) {/* Gestione errore */}
        if (pid == 0) { /* Sono nel processo figlio */

            close(sd);

            /* Gestione richiesta (send, recv, ...) */
            close(new_sd);

            exit(0);
        }
        // Sono nel processo padre
        close(new_sd);
    }
}

```

7.2 ESERCIZIO: ECHO SERVER CONCORRENTE

- Rendere multi-processo il server dell'esercizio 2 della volta precedente, con la primitiva `fork()`.
- Rimuovere il limite dei 20 byte:
 - il client invia la dimensione esatta della stringa
 - il server legge la dimensione esatta di byte.
 - Come fa il server a sapere in anticipo quanti byte leggere?

```

/* CLIENT */
uint16_t lmsg;
...
// Determino la dimensione dei dati che invierò
len = strlen(buffer);
lmsg = htons(len);
// Invio la dimensione dei dati che invierò
ret = send(sd, (void*) &lmsg, sizeof(uint16_t), 0);
// Invio i dati
ret = send(sd, (void*) buffer, len, 0);

```

```

/* SERVER */
...
// Ricevo la quantità di dati
ret = recv(new_sd, (void*)&lmsg, sizeof(uint16_t), // 
Rinconverto la dimensione in formato host
len = ntohs(lmsg);
// Ricevo i dati
ret = recv(new_sd, (void*)buffer, len, 0);

```

7.2.1 CLIENT

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(int argc, char* argv[]){

    int ret, sd, len;
    uint16_t lmsg;
    struct sockaddr_in srv_addr;
    char buffer[BUFFER_SIZE];

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del server */
    memset(&srv_addr, 0, sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "127.0.0.1", &srv_addr.sin_addr);

    /* Connessione */
    ret = connect(sd, (struct sockaddr*)&srv_addr, sizeof(srv_addr));
    if(ret < 0){
        perror("Errore in fase di connessione: \n");
        exit(-1);
    }
}

```

```

while(1) {

    // Attendo input da tastiera
    // scanf("%s", buffer); scanf recupera solo una parola
    fgets(buffer, BUFFER_SIZE, stdin);

    // Calcolo la dimensione del messaggio (compreso '\0')
    len = strlen(buffer) + 1;

    // Gestione endianness (convertito in 'network order')
    lmsg = htons(len);

    // Invio al server la dimensione del messaggio
    ret = send(sd, (void*)&lmsg, sizeof(uint16_t), 0);

    // Invio il messaggio
    ret = send(sd, (void*)buffer, len, 0);

    // Attendo la risposta (è un echo, quindi so già la
    // dimensione)
    ret = recv(sd, (void*)buffer, len, 0);
    if(ret < 0){
        perror("Errore in fase di ricezione: \n");
        exit(-1);
    }

    printf("%s", buffer);

    // Rompo il ciclo se viene digitato 'bye'
    if( strcmp(buffer, "bye\n") == 0 )
        break;
}

// Chiusura del socket connesso
close(sd);
}
}

```

7.2.2 SERVER

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(int argc, char* argv[]){

    int ret, sd, new_sd, len;
    uint16_t lmsg;
    pid_t pid;
    struct sockaddr_in my_addr, cl_addr;

```

```

char buffer[BUFFER_SIZE];

/* Creazione socket */
sd = socket(AF_INET, SOCK_STREAM, 0);

/* Creazione indirizzo di bind */
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(4242);
my_addr.sin_addr.s_addr = INADDR_ANY;

/* Aggancio del socket */
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr) );
ret = listen(sd, 10);
if(ret < 0){
    perror("Errore in fase di bind: \n");
    exit(-1);
}

while(1){

    len = sizeof(cl_addr);

    // Accetto nuove connessioni
    new_sd = accept(sd, (struct sockaddr*) &cl_addr, &len);

    // Creazione child process
    pid = fork();

    if( pid == 0 ){
        // Siamo nel child process (pid è pari a 0)

        // Chiusura del listening socket
        close(sd);

        while(1){

            // Ricezione della dimensione del messaggio
            ret = recv(new_sd, (void*)&lmsg, sizeof(uint16_t), 0);

            // Conversione in formato 'host'
            len = ntohs(lmsg);

            // Ricezione del messaggio
            ret = recv(new_sd, (void*)buffer, len, 0);
            if(ret < 0){
                perror("Errore in fase di ricezione: \n");
                continue;
            }

            // Posso fare strcmp() perché il client invia anche
            // il carattere '\0'
            if( strcmp( buffer, "bye\n" ) == 0 ){
                break;
            }
        }
    }
}

```

```
// Invio risposta
ret = send(new_sd, (void*) buffer, len, 0);
if(ret < 0){
    perror("Errore in fase di invio: \n");
    continue;
}
}

// Qui se è stato riconosciuto il 'bye'
close(new_sd);

exit(1);
}
else{
    // Processo padre

    // Chiusura del socket connesso
    close(new_sd);
}
}
```

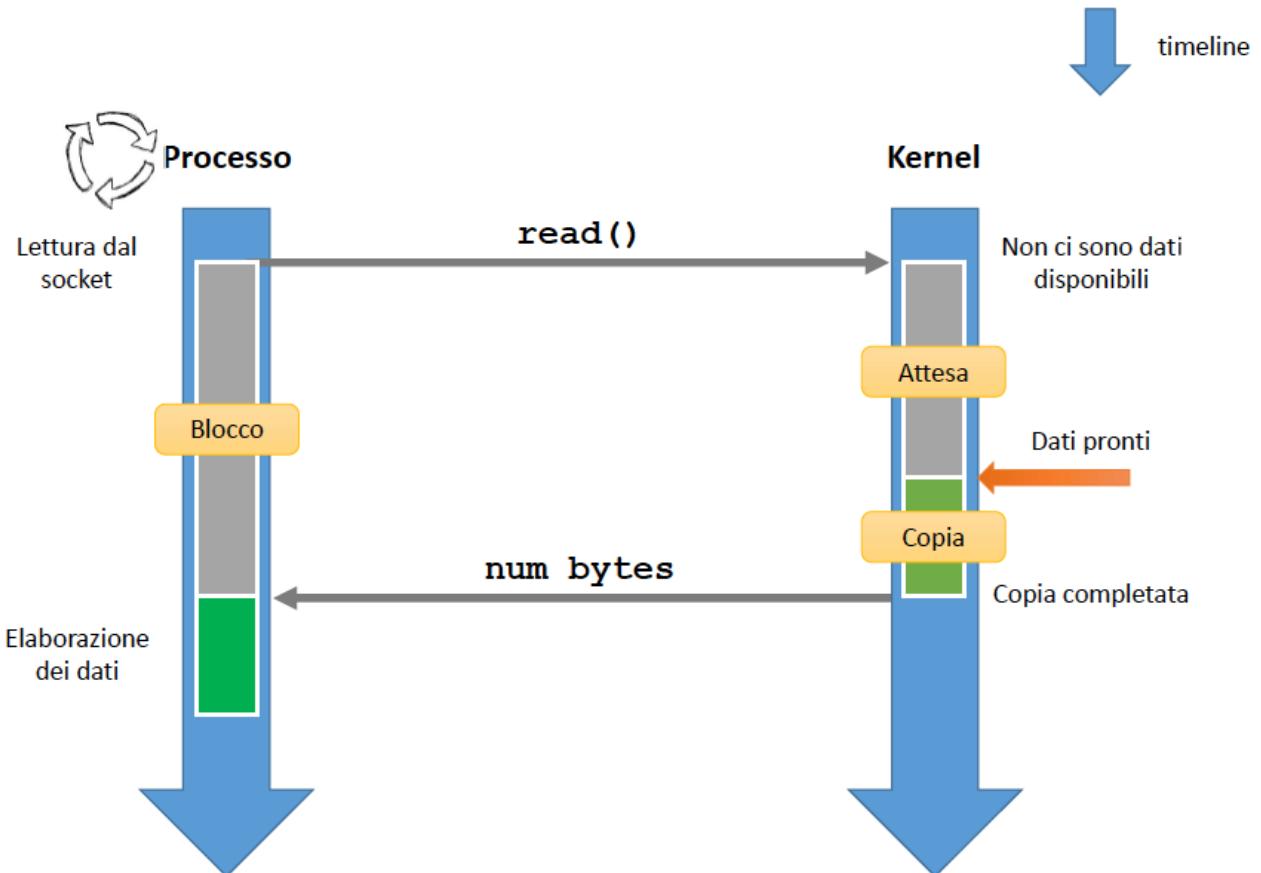
8 SOCKET NON BLOCCANTI, I/O MULTIPLEXING

8.1 MODELLI DI I/O

8.1.1 SOCKET BLOCCANTI

Di default, un socket è **bloccante**.

<code>connect()</code>	Blocca il processo finché il socket non è connesso.
<code>accept()</code>	Blocca il processo finché non arriva una richiesta di connessione.
<code>send()</code>	Blocca il processo finché tutto il messaggio non è stato inviato (il buffer di invio potrebbe essere pieno).
<code>recv()</code>	Blocca il processo finché non ci sono dati disponibili o finché tutto il messaggio richiesto non è disponibile (flag <code>MSG_WAITALL</code>).



- ! Negli esempi che seguiranno, `read()` e `write()` corrispondono a `receive()` e `send()`, rispettivamente.

8.1.2 SOCKET NON BLOCCANTE

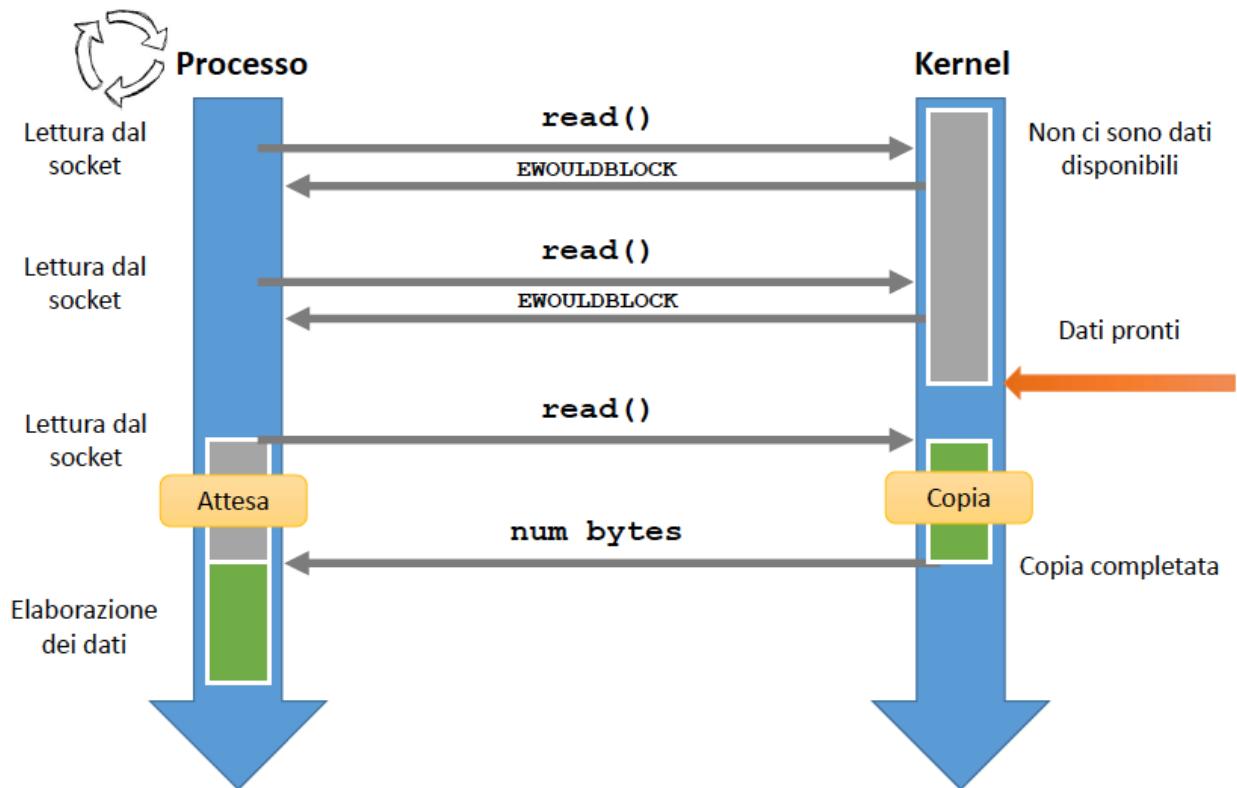
Un socket può essere settato come **non bloccante**: fa altro mentre aspetta la risposta.

- ! Può fare solamente azioni che non coinvolgono i dati richiesti o l'interazione con un altro client nella rete.

```
socket(AF_INET, SOCK_STREAM|SOCK_NONBLOCK, 0);
```

Il comportamento delle diverse primitive è modificato.

<code>connect()</code>	Se non può connettersi, restituisce -1 e imposta <code>errno</code> a EINPROGRESS.
<code>accept()</code>	Se non ci sono richieste, restituisce -1 e imposta <code>errno</code> a EWOULDBLOCK (significa "sarebbe bloccante, ma non si blocca e va alla successiva istruzione").
<code>send()</code>	Se non può inviare tutto il messaggio (buffer pieno) restituisce -1 e imposta <code>errno</code> a EWOULDBLOCK.
<code>recv()</code>	Se non ci sono messaggi, restituisce -1 e imposta <code>errno</code> a EWOULDBLOCK.



! L'attesa del processo che ha chiamato la receive si riduce solamente ad attendere che i dati vengano copiati nel buffer e trasferiti: nel frattempo fa altro.

8.2 I/O MULTIPLEXING: GESTIRE PIU' DESCRITTORI/SOCKET CONTEMPORANEAMENTE

8.2.1 MULTIPLEXING I/O SINCRONO

Problema: se faccio operazioni su un socket bloccante, non posso controllarne altri.

DESCRITTORI PRONTI

Un socket è pronto in **lettura** se:

1. C'è almeno un byte da leggere.
2. Il socket è stato chiuso: `read()` restituirà 0.
3. È un socket in ascolto, e ci sono connessioni effettuate.
4. C'è un errore: `read()` restituirà -1.

Un socket è pronto in **scrittura** se:

1. C'è spazio nel buffer per scrivere.
2. C'è un errore: `write()` restituirà -1 → Se il socket è chiuso, `errno` vale EPIPE.

INSIEMI DI DESCRITTORI

Un **descrittore** è un `int` da 0 a `FD_SETSIZE` (di solito 1024).

Un **insieme di descrittori** (detto **set**) si rappresenta con una variabile di tipo `fd_set`. Si manipola con delle macro.

```
/* Aggiungere un descrittore "fd" all'insieme "set" */
void FD_SET(int fd, fd_set* set);

/* Controllare se un descrittore "fd" è nell'insieme "set" */
int FD_ISSET(int fd, fd_set* set);

/* Rimuovere un descrittore "fd" dall'insieme "set" */
void FD_CLR(int fd, fd_set* set);

/* Svuotare l'insieme "set" */
void FD_ZERO(fd_set* set);
```

8.2.2 PRIMITIVA `select()`

Controlla **più socket**, rilevando quelli pronti: capisce quali descrittori sono pronti in un insieme di descrittori, e toglie da questo insieme tutti i descrittori che non sono pronti, lasciando solo quelli pronti.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set* readfds, fd_set* writefds,
           fd_set* exceptfds, struct timeval* timeout);
```

nfds	Numero del descrittore più alto tra quelli da controllare, +1.
readfds/writefds	Lista di descrittori da controllare per la lettura/scrittura.
exceptfds	Lista di descrittori da controllare per le eccezioni (non ci interessa).
timeout	Intervallo di timeout.

Restituisce il **numero di descrittori pronti** (-1 in caso di errore).

- ! Non seleziona alcun descrittore pronto: non restituisce il numero di un qualche descrittore.

È bloccante: si blocca finché uno dei descrittori controllati non è pronto, oppure finché non scade il timeout.

STRUTTURA PER IL TIMEOUT

```
#include <sys/socket.h>
#include <netinet/in.h>

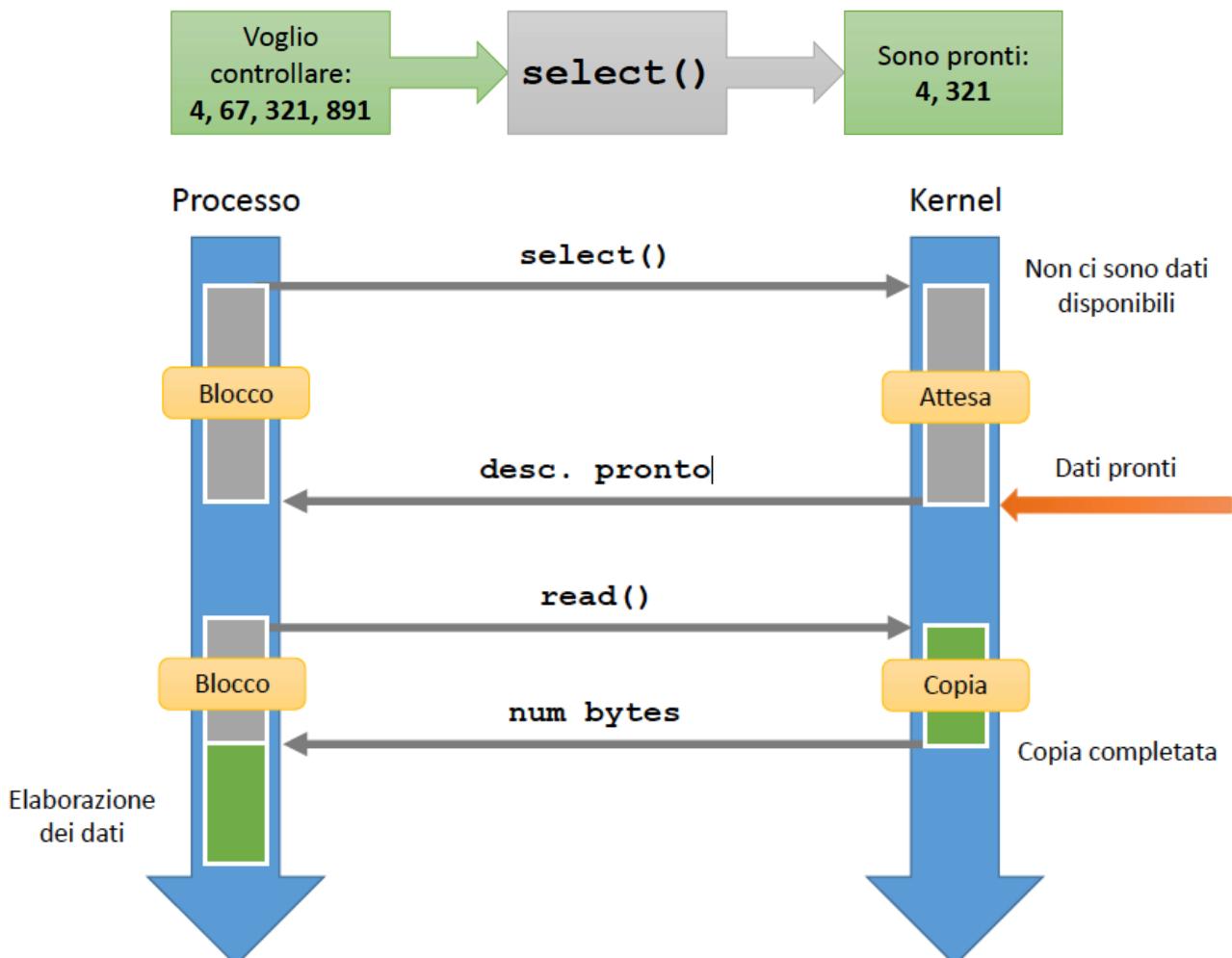
struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
```

- `timeout = NULL` → Attesa indefinita, fino a quando un descrittore è pronto.
- `timeout = { 10; 5; }` → Attesa massima di 10 secondi e 5 microsecondi.
- `timeout = { 0; 0; }` → Attesa nulla, controlla i descrittori ed esce immediatamente (*polling*).

COMPORTAMENTO DI SELECT()

Modifica gli insiemi di descrittori:

- **Prima** di chiamare `select()`, occorre inserire i descrittori da monitorare nei set di lettura e di scrittura.
- **Dopo** l'esecuzione di `select()`, i set di lettura e scrittura contengono i descrittori pronti.



UTILIZZO DI SELECT ()

```
int main(int argc, char* argv[]){
    fd_set master; /* Set principale gestito dal programmatore con le
                     macro */

    fd_set read_fds; /* Set di lettura gestito dalla select */
```

Perché ci sono due fd_set?

- Nel master ci sono i descrittori dei socket che voglio monitorare.
- La select agisce su una copia di questi, perché elimina dal set quelli non pronti, e quindi non sarei in grado di monitorarli in futuro se non avessi una copia.

```
int fdmax; // Numero max di descrittori

struct sockaddr_in sv_addr; // Indirizzo server
struct sockaddr_in cl_addr; // Indirizzo client
int listener; // Socket per l'ascolto
int newfd; // Socket di comunicazione
char buf[1024]; // Buffer
int nbytes;
int addrlen;
int i;

/* Azzero i set */
FD_ZERO(&master);
FD_ZERO(&read_fds);

listener = socket(AF_INET, SOCK_STREAM, 0);

sv_addr.sin_family = AF_INET;
// INADDR_ANY mette il server in ascolto su tutte le
// interfacce (indirizzi IP) disponibili sul server
sv_addr.sin_addr.s_addr = INADDR_ANY;
sv_addr.sin_port = htons(20000);

bind(listener, (struct sockaddr*)& sv_addr, sizeof(sv_addr));
listen(listener, 10);

// Aggiungo il listener al set
FD_SET(listener, &master);

// Tengo traccia del maggiore (ora è il listener)
fdmax = listener;

for(;){
    read_fds = master; // read_fds sarà modificato dalla select
    select(fdmax + 1, &read_fds, NULL, NULL, NULL);

    for(i=0; i<=fdmax; i++) { // f1) Scorro il set
        if(FD_ISSET(i, &read_fds)) { // i1) Trovato desc. pronto
            if(i == listener) { // i2) È il listener
                addrlen = sizeof(cl_addr);
                newfd = accept(listener,
                    (struct sockaddr *)&cl_addr, &addrlen)
```

```

        // Aggiungo il nuovo socket
        FD_SET(newfd, &master);

        //Aggiorno fdmax
        if(newfd > fdmax){ fdmax = newfd; }

    }
    else { // È un altro socket (non il listener)
        nbytes = recv(i, buf, sizeof(buf));
        //... Uso i dati
        // Chiudo il socket connesso
        close(i);
        // Tolgo il descrittore del socket connesso
        //dal set dei monitorati
        FD_CLR(i, &master);
    }
}
} // Fine if i1
} // Fine for f1
} // Fine for(;;)
return 0;
}

```

8.3 ESERCIZIO: TIME SERVER TCP

Implementare un server TCP che periodicamente invia l'ora ai client che si registrano al servizio.

- Il server, periodicamente, controlla se c'è una richiesta di registrazione da parte di un client; se c'è, registra il client.
- I client rimangono connessi e inviano, a intervalli regolari, richieste per ricevere l'ora e aspettano la risposta, stampandola ogni volta che arriva.
- L'ora è inviata in formato hh:mm:ss.

Usare la primitiva select() per gestire le richieste da parte dei client.

8.3.1 STAMPARE L'ORA

```

time_t rawtime;
/* Ora in formato POSIX
esempio: Fri Nov 29 15:24:29 2019*/
time(&rawtime);
/* Stampare l'ora
dove ctime() trasforma l'ora in stringa
esempio: 'Fri Nov 29 15:24:29 2019\0'
*/
printf("%s\n", ctime(&rawtime));

```

8.3.2 CLIENT

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

```

```

#define BUFFER_SIZE 1024
#define RESPONSE_LEN 9 // HH:MM:SS\0

int main(int argc, char* argv[]){
    int ret, sd;
    struct sockaddr_in srv_addr;
    char buffer[BUFFER_SIZE];

    char* cmd = "REQ\0"; // Comando da inviare al server

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del server */
    memset(&srv_addr, 0, sizeof(srv_addr)); // Pulizia
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "127.0.0.1", &srv_addr.sin_addr);

    ret = connect(sd, (struct sockaddr*)&srv_addr, sizeof(srv_addr));

    if(ret < 0){
        perror("Errore in fase di connessione: \n");
        exit(-1);
    }

    // Send the command
    ret = send(sd, cmd, sizeof(cmd), 0);
    if(ret < 0){
        perror("Errore in fase di invio comando: \n");
        exit(-1);
    }

    // Attendo risposta
    ret = recv(sd, (void*)buffer, RESPONSE_LEN, 0);
    if(ret < 0){
        perror("Errore in fase di ricezione: \n");
        exit(-1);
    }

    // Stampo
    printf("%s\n", buffer);

    // Chiudo il socket
    close(sd);
}

```

8.3.3 SERVER

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define BUF_LEN 1024
#define REQUEST_LEN 4 // REQ\0

```

```

int main(int argc, char* argv[]){
    int ret, newfd, listener, addrlen, i, len;

    // Set di descrittori da monitorare
    fd_set master;

    // Set dei descrittori pronti
    fd_set read_fds;

    // Descrittore max
    int fdmax;

    struct sockaddr_in my_addr, cl_addr;
    char buffer[BUF_LEN];

    // Uso la struttura time_t per l'orario
    time_t rawtime;
    struct tm* timeinfo;

    /* Creazione socket */
    listener = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo di bind */
    memset(&my_addr, 0, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4242);
    my_addr.sin_addr.s_addr = INADDR_ANY;

    /* Aggancio */
    ret = bind(listener, (struct sockaddr*)&my_addr, sizeof(my_addr));
    if( ret < 0 ){
        perror("Bind non riuscita\n");
        exit(0);
    }

    /* Apro la coda */
    listen(listener, 10);

    // Reset dei descrittori
    FD_ZERO(&master);
    FD_ZERO(&read_fds);

    // Aggiungo il socket di ascolto 'listener' ai socket monitorati
    FD_SET(listener, &master);
    // Tengo traccia del nuovo fdmax
    fdmax = listener;

    while(1){

        // Imposto il set di socket da monitorare in lettura per la
        // select()
        // NOTA: select() modifica il set 'read_fds' lasciando solo i
        // descrittori pronti
        // ma non modifica il set 'master' dei descrittori monitorati!
    }
}

```

```

read_fds = master;

// Mi blocco (potenzialmente) in attesa di descrittori pronti
// Attesa ***senza timeout*** (ultimo parametro attuale
// 'NULL')
select(fdmax+1, &read_fds, NULL, NULL, NULL);

// Scorro ogni descrittore 'i'
for(i=0; i<=fdmax; i++) {

    // Se il descrittore 'i' è rimasto nel set 'read_fds',
    // cioè se la select() ce lo ha lasciato, allora 'i' è
    // pronto
    if(FD_ISSET(i, &read_fds)) {

        // Se il descrittore pronto 'i' è il listening
        // socket 'listener' ho ricevuto una richiesta di
        // connessione
        if(i == listener) {

            // Calcolo la lunghezza dell'indirizzo del
            // client
            addrlen = sizeof(cl_addr);

            // Accetto la connessione e creo il socket
            // connesso ('newfd')
            newfd = accept(listener, (struct sockaddr *)
                &cl_addr, &addrlen);

            // Aggiungo il socket connesso al set dei
            // descrittori monitorati
            FD_SET(newfd, &master);

            // Aggiorno l'ID del massimo descrittore
            if(newfd > fdmax){
                fdmax = newfd;
            }
        }

        // Altrimenti, ho ricevuto una richiesta di
        // servizio (orario)
        else {

            // Ricevo il messaggio di richiesta (so che è
            // di 4 // byte)
            ret = recv(i, (void*)buffer, REQUEST_LEN, 0);

            // Recupero l'ora corrente
            time(&rawtime);

            // Converto l'ora
            timeinfo = localtime(&rawtime);

            // Creo la risposta nel formato 'hh:mm:ss'
            sprintf(buffer, "%d:%d:%d",
                timeinfo->tm_hour, timeinfo->tm_min, timeinfo-
                >tm_sec);
        }
    }
}

```

```

        // Invio la risposta, compreso '\0'
        ret = send(i, (void*) buffer,
                   strlen(buffer)+1, 0);
        if(ret < 0){
            perror("Errore in fase di comunicazione
                    con il client: \n");
        }

        // Chiudo il socket connesso 'i', che ho
        // servito
        close(i);

        // Rimuovo il socket 'i' dal set dei socket
        // monitorati
        FD_CLR(i, &master);

    }

}

} // fine for che scorre i descrittori (torno alla linea 75)
} // fine while(1)

close(listener);

}

```

8.3.4 CLIENT CON SELECT

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
#define RESPONSE_LEN 9 // HH:MM:SS\0

int main(int argc, char* argv[]){

    int ret, sd, j;

    struct sockaddr_in srv_addr;
    char buffer[BUFFER_SIZE];

    char* cmd = "REQ\0"; // Comando da inviare al server

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del server */
    memset(&srv_addr, 0, sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "127.0.0.1", &srv_addr.sin_addr);

    /* connessione */
    ret = connect(sd, (struct sockaddr*)&srv_addr, sizeof(srv_addr));
}

```

```

    if(ret < 0){
        perror("Errore in fase di connessione: \n");
        exit(1);
    }

    for(;;){

        // invio della richiesta
        ret = send(sd, cmd, strlen(cmd)+1, 0);

        if(ret < 0){
            perror("Errore in fase di invio comando: \n");
            exit(1);
        }

        // Attendo risposta
        ret = recv(sd, (void*)buffer, RESPONSE_LEN, 0);

        if(ret < 0){
            perror("Errore in fase di ricezione: \n");
            exit(1);
        }

        // Stampo
        printf("%s\n", buffer);
        sleep(5);

    } //chiudo il "for sempre" (linea 40)
}

```

8.3.5 SERVER CON SELECT

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define BUF_LEN 1024
#define REQUEST_LEN 4 // REQ\0

int main(int argc, char* argv[]){
    int ret, newfd, listener, addrlen, i, len, k;

    fd_set master;
    fd_set read_fds;
    int fdmax;

    struct sockaddr_in my_addr, cl_addr;
    char buffer[BUF_LEN];

    time_t rawtime;
    struct tm * timeinfo;

    /* Creazione socket */

```

```

listener = socket(AF_INET, SOCK_STREAM, 0);

/* Creazione indirizzo di bind */
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(4242);
my_addr.sin_addr.s_addr = INADDR_ANY;

ret = bind(listener, (struct sockaddr*)&my_addr, sizeof(my_addr) );

if( ret < 0 ){
    perror("Bind non riuscita\n");
    exit(0);
}

listen(listener, 10);

// Reset FDs
FD_ZERO(&master);
FD_ZERO(&read_fds);

// Aggiungo il socket di ascolto (listener), creato dalla socket()
// all'insieme dei descrittori da monitorare (master)
FD_SET(listener, &master);

// Aggiorno il massimo
fdmax = listener;

//main loop
while(1){

    // Inizializzo il set read_fds, manipolato dalla select()
    read_fds = master;

    // Mi blocco in attesa di descrittori pronti in lettura
    // imposto il timeout a infinito
    // Quando select() si sblocca, in &read_fds ci sono solo
    // i descrittori pronti in lettura!
    ret = select(fdmax+1, &read_fds, NULL, NULL, NULL);
    if(ret<0){
        perror("ERRORE SELECT:");
        exit(1);
    }

    // Spazzolo i descrittori
    for(i = 0; i <= fdmax; i++) {

        // controllo se i è pronto
        if(FD_ISSET(i, &read_fds)) {

            // se i è il listener, ho ricevuto una richiesta di
            // connessione
            // (un client ha invocato connect())
            if(i == listener) {

                printf("Nuovo client rilevato!\n");
                fflush(stdout);
                addrlen = sizeof(cl_addr);
                // faccio accept() e creo il socket connesso
                // 'newfd'
                newfd = accept(listener,

```

```

        (<struct sockaddr *>)&cl_addr, &addrlen);

        // Aggiungo il descrittore al set dei socket
        // monitorati
        FD_SET(newfd, &master);

        // Aggiorno l'ID del massimo descrittore
        if(newfd > fdmax){ fdmax = newfd; }

    }

    // se non è il listener, 'i' è un descrittore di socket
    // connesso che ha fatto la richiesta di orario, e va
    // servito ***senza poi chiudere il socket*** perché
    // l'orario potrebbe essere chiesto nuovamente al server
    else {
        // Metto la richiesta nel buffer (pacchetto
        // "REQ\0")
        ret = recv(i, (void*)buffer, REQUEST_LEN, 0);

        if(ret == 0){
            printf("CHIUSURA client rilevata!\n");
            fflush(stdout);
            // il client ha chiuso il socket, quindi
            // chiudo il socket connesso sul server
            close(i);
            // rimuovo il descrittore newfd da quelli da
            // monitorare
            FD_CLR(i, &master);
        }
        else if(ret < 0){
            perror("ERRORE! \n");
            // si è verificato un errore
            close(i);
            // rimuovo il descrittore newfd da quelli da
            // monitorare
            FD_CLR(i, &master);
        }
        else{
            printf("REQ client rilevata!\n");
            fflush(stdout);

            // Recupero l'ora corrente
            time(&rawtime);

            // Converto l'ora
            timeinfo = localtime(&rawtime);

            // Creo la risposta mettendola in "buffer"
            sprintf(buffer, "%d:%d:%d",
                    timeinfo->tm_hour, timeinfo->tm_min, timeinfo-
                    >tm_sec );

            // Invio la risposta (e il terminatore di stringa)
            len = send(i, (void*) buffer, strlen(buffer)+1, 0);

            if(ret < 0){
                perror("Errore in fase di comunicazione \n");
            }
        }
    }
}

```

```
        }

    }
    //break;
}
// ci arrivo solo se monitoro stdin (descrittore 0)
// -> rompo il while e passo a chiudere il listener
}
printf("CHIUDO IL LISTENER!\n");
fflush(stdout);
close(listener);

}
```

9 SOCKET UDP

9.1 SOCKET UDP

9.1.1 TCP VS UDP

TCP: instaura una **connessione** → Prevede delle operazioni preliminari per instaurare un canale virtuale.

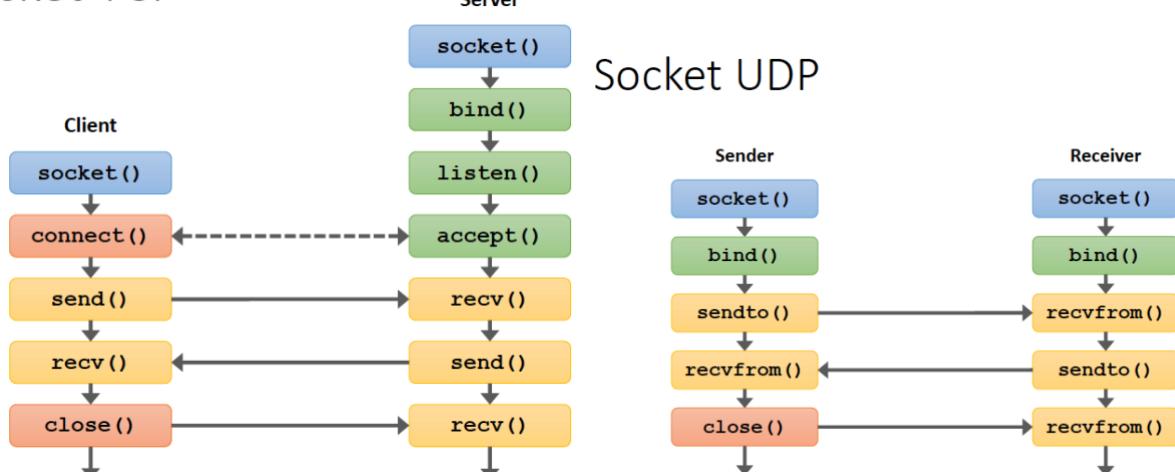
- È **affidabile**: i pacchetti inviati arrivano tutti, nell'ordine in cui sono stati inviati.
- Comporta latenza maggiore per il riordino ed eventuali ritrasmissioni.

UDP: è **connectionless** → Non si stabilisce alcuna connessione e non si fanno operazioni preliminari per instaurare la connessione.

- È rapido: non c'è nessun recupero e nessun riordino.

I pacchetti si possono perdere e/o corrompere. Rivedendo il protocollo TCP, le operazioni preliminari sono quelle della `listen` e della `accept` per il server, `connect` per il client.

Socket TCP



Non c'è più la differenza tra socket di ascolto e socket di comunicazione. Questo perché il protocollo è di tipo connectionless: sul receiver non c'è la connessione con `listen` e `accept`, si fa solo un canale "rosso" per inviare dati senza alcun tipo di garanzia.

Nelle due system call si deve quindi specificare ogni volta chi sia il destinatario e quale sia il socket dall'altro capo della connessione.

- ! In TCP questo non accadeva perché il preambolo univa inesorabilmente le due parti.

9.1.2 PRIMITIVA `sendto()`

Invia un **messaggio** attraverso un socket all'indirizzo specificato.

```
ssize_t sendto(int sockfd, const void* buf, size_t len,
               int flags, const struct sockaddr* dest_addr,
               socklen_t addrlen);
```

sockfd	Descrittore del socket creato sull'host locale, dichiarato con <code>SOCK_DGRAM</code> .
buf	Puntatore al buffer contenente il messaggio da inviare.
len	Dimensione in byte del messaggio.
flags	Per settare le opzioni (lasciamolo a 0).
dest_addr	Puntatore alla struttura contenente l'indirizzo del destinatario.
addrlen	Lunghezza di <code>dest_addr</code> .

Restituisce il **numero di byte inviati** (o -1 in caso di errore).

- ! Si suppone che con una chiamata si riesca ad inviare tutto il messaggio, ovvero trasferire il messaggio dal buffer dell'applicazione al buffer di invio del socket. Volendo inviare un file, dobbiamo dividerlo in chunk e caricarli sul buffer di invio del socket: la funzione restituisce un valore inferiore di quello inserito. Inviare qualcosa significa riversare il contenuto del buffer applicazione nel buffer kernel. Se si verifica un errore, a livello applicazione non ci interessa, perché ci pensa il protocollo di trasporto.

È **bloccante**: il programma si ferma finché non ha scritto tutto il messaggio.

9.1.3 PRIMITIVA recvfrom()

Riceve un messaggio attraverso un socket.

```
ssize_t recvfrom(int sockfd, const void* buf, size_t len,
                 int flags, struct sockaddr* src_addr,
                 socklen_t addrlen);
```

sockfd	Descrittore del socket.
buf	Puntatore al buffer contenente il messaggio da ricevere.
len	Dimensione in byte del messaggio.
flags	Per settare le opzioni (lasciamolo a 0).
src_addr	Puntatore a una struttura vuota per salvare l'indirizzo del mittente.
addrlen	Lunghezza di <code>src_addr</code> .

Restituisce il **numero di byte ricevuti**, -1 in caso di errore, 0 se il socket remoto si è chiuso.

È **bloccante**: il programma si ferma finché non ha letto qualcosa.

9.1.4 CODICE DEL SERVER

```
int main () {
    int ret, sd, len;
    char buf[BUFSIZE];
    struct sockaddr_in my_addr, cl_addr;
    int addrlen = sizeof(cl_addr);

    /* Creazione socket UDP */
    sd = socket(AF_INET, SOCK_DGRAM, 0);

    /* Creazione indirizzo */
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
```

```

my_addr.sin_family = AF_INET ;
my_addr.sin_port = htons(4242) ;
my_addr.sin_addr.s_addr = INADDR_ANY ;
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr)) ;

while(1) {
    len = recvfrom(sd, buf, BUFSIZE, 0,
    (struct sockaddr*)&cl_addr, &addrlen) ;
    //fai cose ...
}
}

```

9.1.5 CODICE DEL CLIENT

```

int main () {
    int ret, sd, len;
    char buf[BUFSIZE];
    struct sockaddr_in sv_addr; // Struttura per il server

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_DGRAM, 0);

    /* Creazione indirizzo del server */
    memset(&sv_addr, 0, sizeof(sv_addr)); // Pulizia
    sv_addr.sin_family = AF_INET ;
    sv_addr.sin_port = htons(4242) ;
    inet_pton(AF_INET, "192.168.4.5", &sv_addr.sin_addr);

    while(1) {
        len = sendto(sd, buf, BUFSIZE, 0,
        (struct sockaddr*)&sv_addr, sizeof(sv_addr));
        // fai cose...
    }
}

```

Il client deve ovviamente conoscere autonomamente l'indirizzo del server.

9.2 SOCKET UDP “CONNESSO”

Usando `connect()` su un socket UDP gli si può associare un indirizzo remoto: il socket riceverà/invierà pacchetti **solo** da/a quell'indirizzo.

! Non è una connessione!

Con un socket connesso, si possono usare `send()` e `recv()` senza specificare ogni volta l'indirizzo.

10 FIREWALL, PACKET FILTERING, IPTABLES

10.1 FIREWALL

PROBLEMA: le reti e i computer connessi a Internet vanno **protetti da accessi indesiderati e malware.**

Un firewall è quindi un meccanismo di protezione che un host della rete può utilizzare. Queste tipologie di protezione servono per vietare determinati accessi o permettere altri: nell'ambito dei filtraggi dei pacchetti, significa consentire che una certa macchina possa, ad esempio, raggiungere tutti gli host eccetto quelli appartenenti a una certa sottorete.

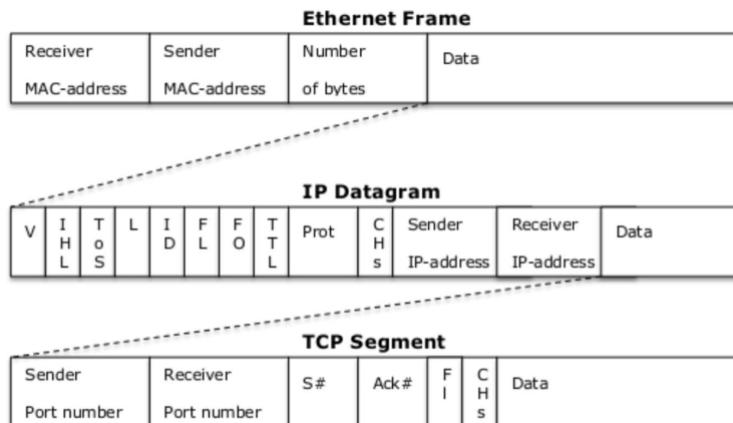
Possiamo altrimenti vietare tutto il traffico HTTP in uscita da una certa sottorete, o consentire solamente il traffico UDP in uscita.

DEFINIZIONE FIREWALL: sistema hardware o software che controlla le connessioni in ingresso e in uscita, analizzando delle **regole** che il pacchetto deve soddisfare per passare.

- Opera a livello rete (network firewall) o di macchina (host-based firewall).

10.1.1 TIPI DI FIREWALL

- **Network layer** (packet filter): operano a livello di TCP/IP, analizzando gli header IP, TCP e UDP.



- **Application layer:** operano a livello applicazione, facendo deep packet inspection. Più saliamo con il firewall, più abbiamo consapevolezza del contesto del pacchetto: a livello applicazione potrei tenere in considerazione un thread e-mail e rimuoverla se appartiene ad un certo tipo semantico.
 - Analizzano tutto il pacchetto (header e payload) e forniscono un controllo fino al livello 7 (vedere tutto il traffico che proviene da un host, ricostruire una pagina web, una conversazione e-mail...).
 - Più efficaci, ma usano maggiori risorse computazionali (efficaci contro malware, vulnerabilità note, comportamenti dannosi delle applicazioni...).

Level	Shallow Packet Inspection	Medium Packet Inspection	Deep Packet Inspection	ISO/OSI
7				Application
6				Presentation
5				Session
4				Transport
3				Network
2				Data Link
1				Physical

10.2 PACKET FILTERING (FIREWALL DI LIVELLO NETWORK)

10.2.1 TIPI DI PACKET FILTERING

- **Stateless:** ogni pacchetto viene analizzato in base a campi statici come indirizzo sorgente o destinazione. La decisione se far passare o meno un pacchetto è quindi presa in base al suo contenuto.
- **Stateful:** tiene traccia delle connessioni TCP e degli scambi UDP in corso, e discrimina le connessioni legittime da quelle sospette. Più efficace, ma più complesso e pesante rispetto al filtraggio stateless.

10.2.2 FUNZIONAMENTO

Il firewall contiene una **tabella di regole**. Ogni regola sta su una riga, ha un indice progressivo che parte da uno e contiene:

1. **Criteria:** caratteristiche del pacchetto (IP Sorgente, Porta Sorgente, IP Destinatario, Porta Destinatario).
2. **Target:** azione da intraprendere, ovvero scarta (DROP) o accetta (ACCEPT).

Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	131.114.0.0/16		131.114.54.4	80	SCARTA
2	0.0.0.0	23	112.143.2.2		ACCETTA

10.2.3 FUNZIONAMENTO PER OGNI PACCHETTO

Scorre le regole (in ordine crescente di indice) e quando ne trova una i cui criteri corrispondono ai criteri del pacchetto, applica **quella sola regola**, poi analizza altri pacchetti.

Nell'immagine vista al paragrafo precedente:

- Regola 1: scarta tutti i pacchetti provenienti dalla sottorete 113.114.0.0/16 destinati alla porta 80 del destinatario 131.114.54.4.
- Regola 2: accetta tutti i pacchetti diretti al destinatario 112.143.2.2 provenienti dalla porta 23 (Telnet) di qualsiasi sorgente.

PSEUDOCODICE

```
for each pacchetto p:  
    1. estrai l'header di p  
    2. for i := 1 to n_regole do  
        2.2 considera la regola r_i (cioè quella con Index == i)  
        2.3 if caratteristiche(p) == caratteristiche(r_i) then  
            2.3.1 Esegui l'azione della regola r_i  
            2.3.2 break  
        2.4 end if  
    2.5 i := i + 1  
3. end for  
end for
```

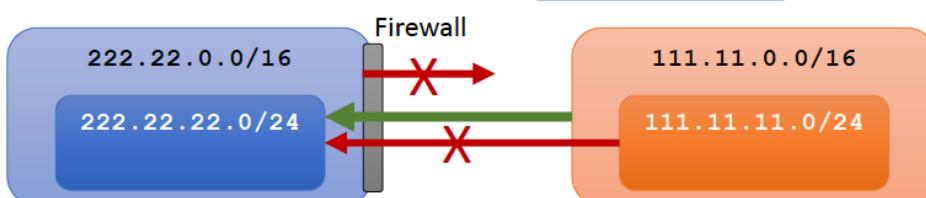
ORDINE DELLE REGOLE

Nell'inserire le regole nella tabella, bisogna tenere conto dell'ordine con cui si presentano, che corrisponde all'ordine con cui il firewall analizza le regole.

Abbiamo una rete locale con indirizzo

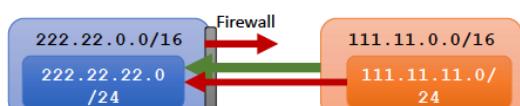
222.22.0.0/16 e vogliamo:

- **Impedire** l'accesso a Internet dall'interno della rete
- **Consentire** alla rete esterna 111.11.0.0/16 di accedere alla sottorete locale 222.22.22.0/24, ma:
- **Impedire** alla sottorete esterna 111.11.11.0/24 di accedere alla sottorete locale 222.22.22.0/24



Se usassimo l'ordine riportato di seguito, non sarebbe rispettata la terza condizione:

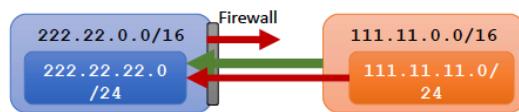
Ordine 1



Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	111.11.0.0/16		222.22.22.0/24		ACCETTA
2	111.11.11.0/24		222.22.22.0/24		BLOCCA
3	0.0.0.0		0.0.0.0		BLOCCA

Di conseguenza, l'ordine giusto è il seguente:

Ordine 2



Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	111.11.11.0/24		222.22.0.0/24		BLOCCA
2	111.11.0.0/16		222.22.22.0/24		ACCETTA
3	0.0.0.0		0.0.0.0		BLOCCA

REGOLA DI DEFAULT

Se il pacchetto non soddisfa nessuna regola, si applica la **default rule** (ACCETTA o BLOCCA).

A seconda di questa regola di default, il firewall può essere:

- **Inclusivo:** l'ultima regola **blocca tutto**.
 - Sicuro ma scomodo, senza definire regole non si può accedere a nulla.
- **Esclusivo:** l'ultima regola **consente tutto**.
 - Comodo ma non sicuro, devo prevedere e inserire manualmente tutte le regole che ritengo utili.

Si imposta con l'opzione -P del comando **iptables**.

10.3 NETFILTER E IPTABLES (PACKET FILTERING SU LINUX)

netfilter	Componente del kernel di Linux che si occupa del firewall. Offre le funzionalità di: <ol style="list-style-type: none"> 1. Stateless/statefull packet filtering. 2. NAT/PAT. 3. Packet mangling – manipolazione generica di pacchetto.
iptables	Programma (linea di comando) per configurare le tabelle di regole.

10.3.1 IPTABLES

Lavora su diverse tabelle (table), ognuna dedicata a una funzionalità.

- Vedremo solo le tabelle **filter** (identica a quella descritta prima per il filtraggio) e **nat** (si occupa del network address translation, che non ha nulla a che vedere con il firewall).

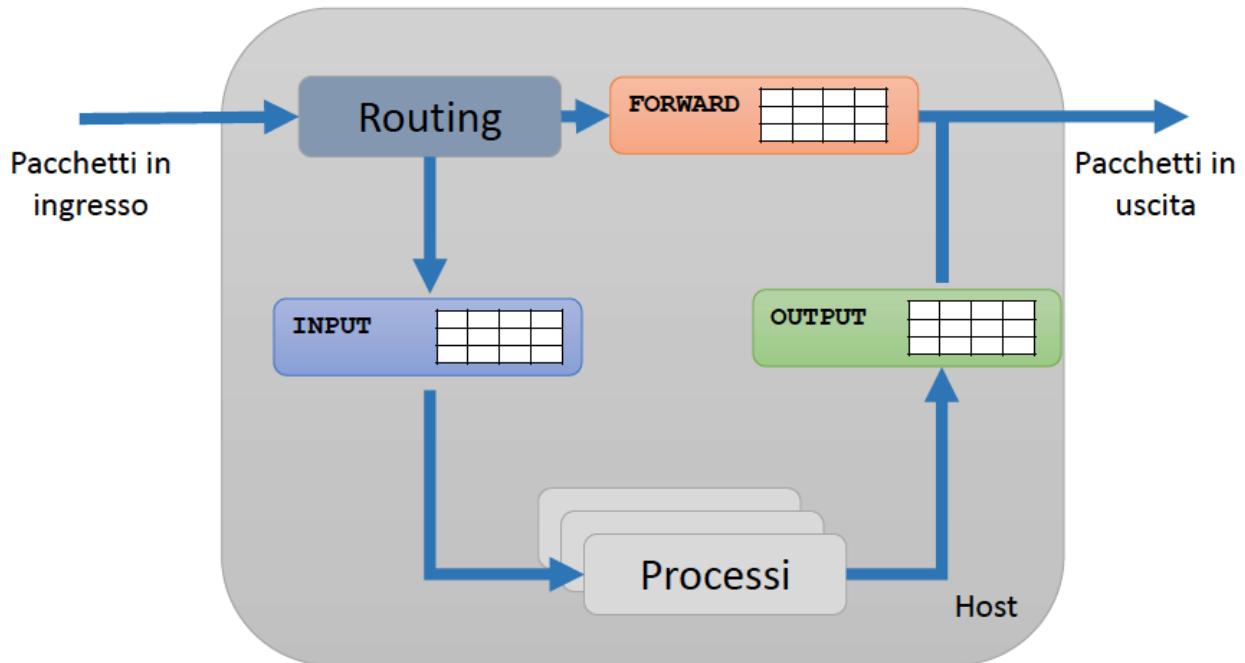
Ogni tabella contiene diverse catene (**chain**). Ogni catena contiene una lista di regole da applicare a una categoria di pacchetti, in modo da raggruppare le regole in dipendenza di cosa il pacchetto sta facendo.

TABELLA FILTER

La tabella filter ha 3 chain:

1. **INPUT:** per i pacchetti in ingresso destinati ai processi locali.
2. **OUTPUT:** per i pacchetti in uscita dai processi locali.

3. **FORWARD**: per i pacchetti in transito, cioè da inoltrare ad altri host.



Resta quindi solo da capire come impostare le regole. Per visualizzare le regole abbiamo il comando:

```
# iptables [-t table] -L [chain]
```

- Se la tabella non è specificata, viene selezionata **filter**.
- Se la catena non è specificata, vengono elencate tutte le catene.

-L serve per visualizzare tutte le regole rispetto alle catene scelte.

```
# iptables -L INPUT
Chain INPUT (policy ACCEPT)
target      prot opt source          destination
...
# iptables -t nat -L
Chain REROUTING (policy ACCEPT)
target      prot opt source          destination
...
```

AGGIUNTA DI REGOLE

```
# iptables [-t table] -A chain rule-specification
```

Aggiungere una regola in fondo alla catena.

- **rule-specification** descrive la regola, poi vedremo il formato.

```
# iptables [-t table] -I chain [num] rule-specification
```

Aggiungere una regola in una posizione specifica.

- Se **num** non è specificato, assume 1 e la regola è posta per prima nella catena.

RIMOZIONE DI REGOLE

```
# iptables [-t table] -D chain rule-specification  
# iptables [-t table] -D chain num
```

Per rimuovere una regola dalla catena.

```
# iptables [-t table] -F [chain]
```

Per rimuovere tutte le regole da una o più catene.

```
# iptables [-t table] -P target
```

Per cambiare la regola di default (policy) DROP/ACCEPT.

FORMATO DELLE REGOLE

rule-specification è una stringa, dove specificare:

-p <protocollo>	Protocollo (TCP, UDP, ICMP...).
-s <address>	Indirizzo IP sorgente.
-d <address>	Indirizzo IP destinazione.
--sport <port>	Porta sorgente.
--dport <port>	Porta destinazione.
-i <interface>	Interfaccia di ingresso.
-o <interface>	Interfaccia di uscita.
-j <target>	Azione (DROP/ACCEPT).

ESEMPI

```
# iptables -A OUTPUT -p tcp -d 10.0.5.4 --dport 80 -j DROP
```

Aggiungi in fondo alla catena OUTPUT della tabella filter una regola che scarti tutti i pacchetti TCP destinati alla porta 80 (HTTP) dell'host 10.0.5.4.

```
# iptables -A INPUT -p udp -s 121.0.0.0/16 -j ACCEPT
```

Aggiungi in fondo alla catena INPUT della tabella filter una regola che lasci passare tutti i pacchetti UDP provenienti dalla sottorete 121.0.0.0/16.

```
# iptables -A INPUT -p icmp -i eth0 -j DROP
```

Aggiungi in fondo alla catena INPUT della tabella filter una regola che scarti tutti i pacchetti ICMP provenienti dall'interfaccia d'ingresso eth0.

RIVEDERE DELLE REGOLE INSERITE

```
# iptables -L  
Chain INPUT (policy ACCEPT)  
target prot opt source destination  
ACCEPT udp -- 121.0.0.0/16 anywhere  
DROP icmp -- anywhere anywhere  
Chain OUTPUT (policy ACCEPT)  
target prot opt source destination  
DROP tcp -- anywhere 10.0.5.4 tcp dpt:http
```

Rimozione della prima regola della catena INPUT e nuova visualizzazione delle regole:

```
# iptables -D INPUT 1  
# iptables -L  
Chain INPUT (policy ACCEPT)  
target prot opt source destination  
DROP icmp -- anywhere anywhere
```

SALVARE E CARICARE LE REGOLE

Le regole **non vengono salvate**, è quindi necessario reimpostarle all'avvio.

# iptables-save > file	Salvare le regole su file.
# iptables-restore < file	Caricare le regole da file.

10.4 NAT E PAT/NAPT

10.4.1 NETWORK ADDRESS TRANSLATION (NAT)

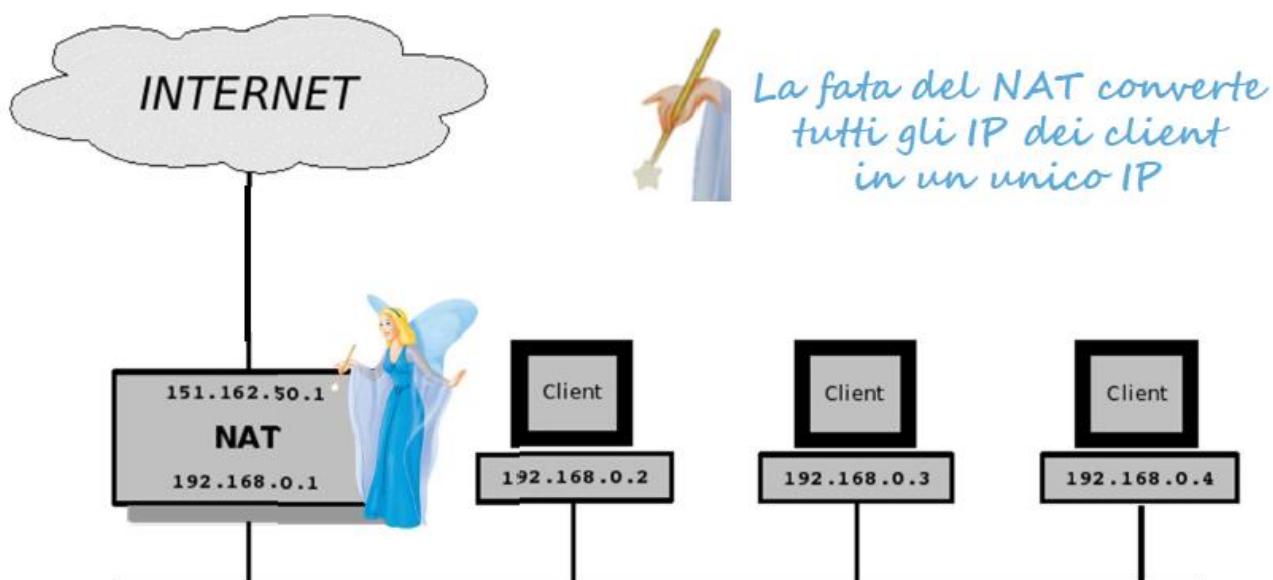
Gli indirizzi IP sono **scarsi** (un ISP potrebbe avere indirizzi '/16' capaci di gestire 65534 host: e se il numero di host supera questo valore?).

La scarsità di IP ha portato all'implementazione di **tecniche per fare economia**:

- Assegnamento e recupero dinamico degli indirizzi.
- NAT

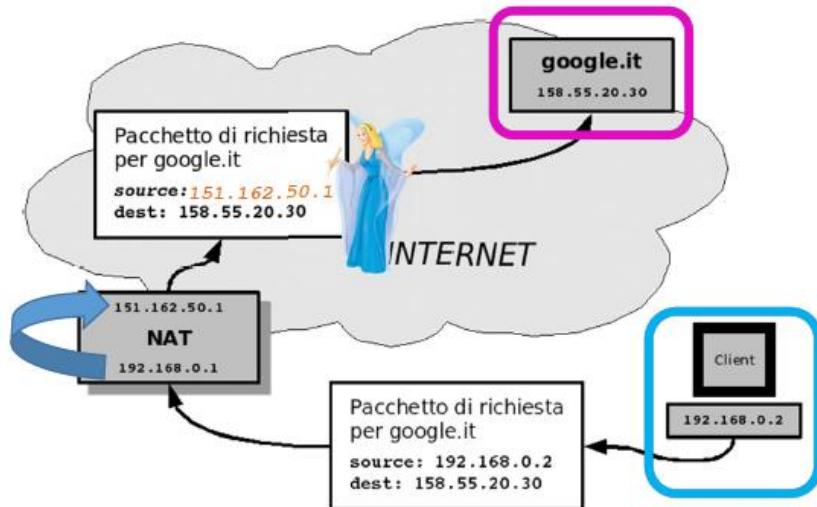
COME FUNZIONA

- Assegnare a un cliente (aziendale o residenziale) **un unico IP** per il traffico Internet.
- Nella rete del cliente, ogni host riceve un IP unico, usato per instradare il traffico interno.
- Quando un pacchetto sta per lasciare la rete locale per dirigersi verso l'ISP, **si esegue la traduzione** dall'indirizzo interno a quello pubblico.



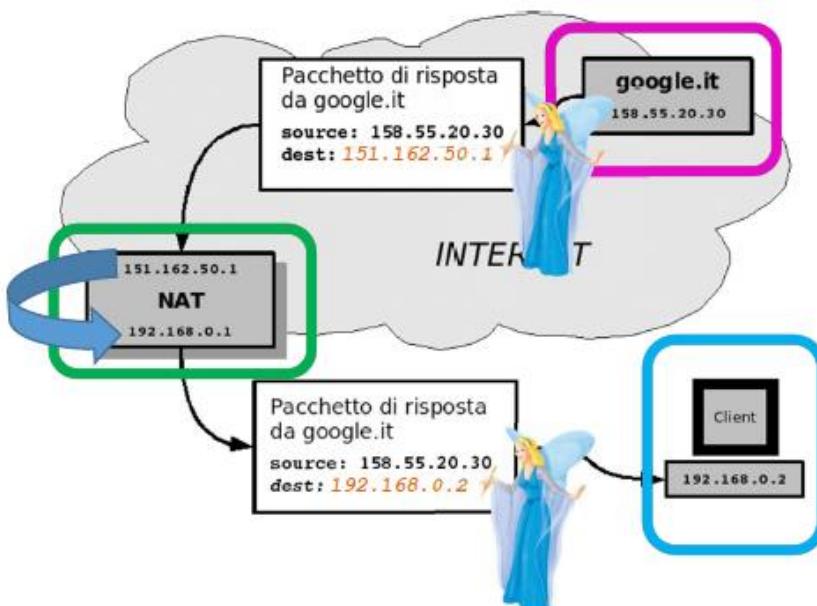
USCITA (SOURCE NAT)

Le connessioni effettuate da un host (in basso a destra) sono alterate per **mostrare all'esterno un IP diverso** da quello originale. Chi riceve le connessioni (in alto a destra) le vede provenire da un IP diverso da quello utilizzato da chi le genera.



INGRESSO (DESTINATION NAT)

Le connessioni effettuate da uno o più host sono alterate in modo da **essere redirette verso indirizzi IP diversi da quelli originali**. Chi effettua le connessioni (in alto a destra) alla fine si collega in realtà ad un indirizzo diverso (in basso a destra) da quello che seleziona (in basso a sinistra).



... e se nella rete ci sono più host?

PIU' HOST NELLA LAN

Il mapping è di tipo **molti-a-uno**, cioè più IP interni vanno mappati in un solo IP esterno.

Ogni host invia messaggi verso il router con il proprio IP interno, e il router converte questo IP con un IP esterno: lo stesso IP esterno per tutti gli host.

Quando arriva la risposta, come fa il router a sapere qual è l'host (l'IP interno) a cui inviarla?

- Se le porte sorgente dei due IP interni fossero diverse, basterebbe tener traccia di queste due porte per smistare il traffico.
- Ma se due processi identificati dalla stessa porta (per esempio la porta 2000), in esecuzione su due host della LAN (IP1 e IP2), inviano un messaggio HTTP (porta di destinazione 80) per chiedere due pagine (pag_A e pag_B) allo stesso server Web, quando il server risponde, il router deve sapere se mandare pag_A a IP1 e pag_B a IP2, oppure il contrario.

10.4.2 PORT ADDRESS TRANSLATION (PAT)

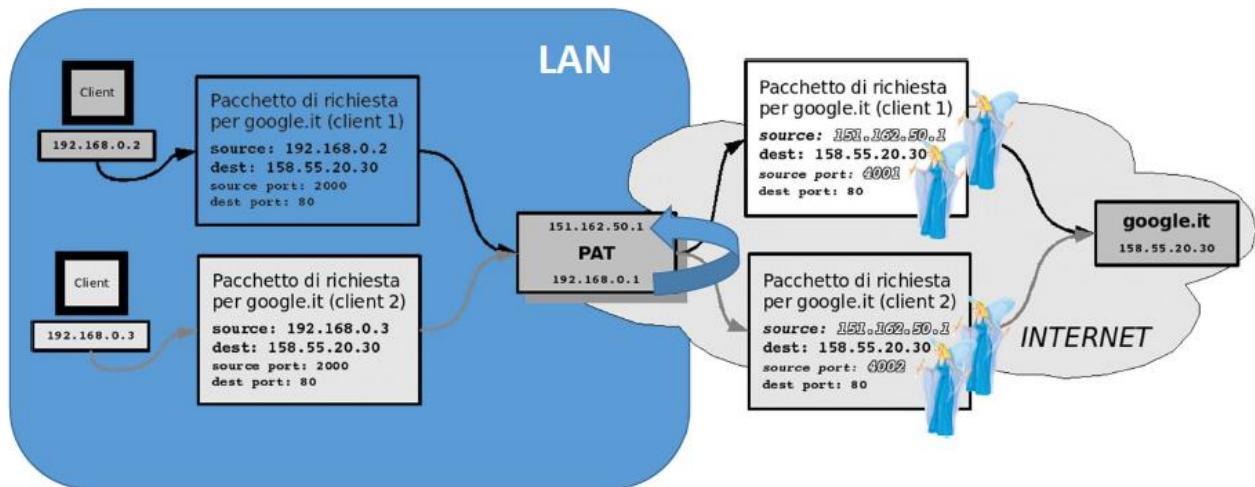
Estensione di NAT: **mappa più host della LAN in un solo IP**.

Con PAT, un pacchetto entrante avente una porta di destinazione (detta **porta esterna**) è tradotto in un pacchetto avente una porta differente (la **porta interna**).

L'ISP assegna un indirizzo IP al router; quando un host della rete si connette a Internet, il router assegna a questo host un numero di porta che viene abbinato all'IP interno, dando quindi all'host un IP unico.

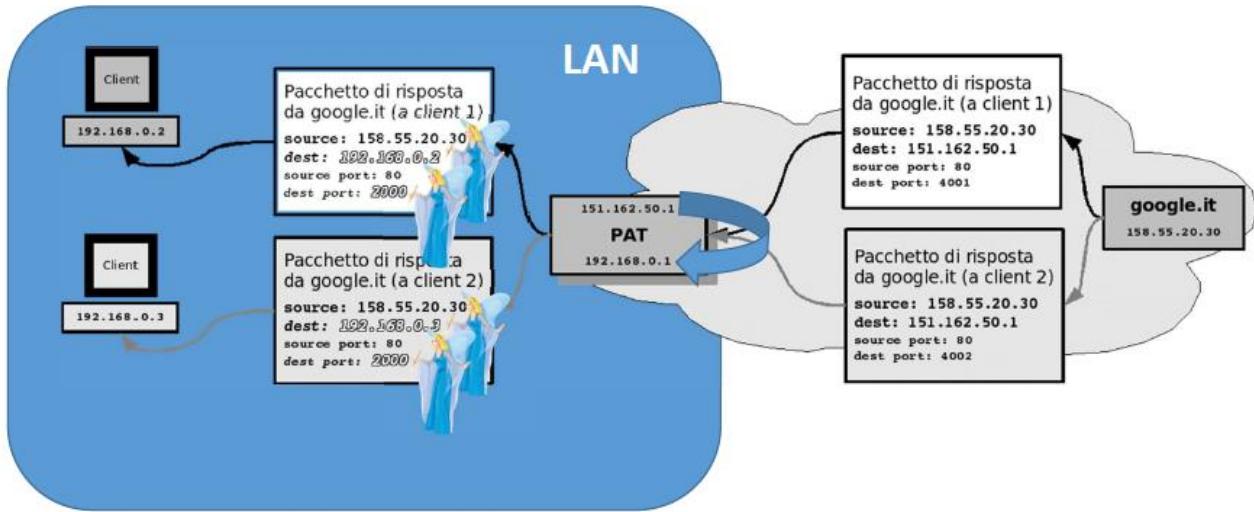
- ! **Attenzione:** il numero di porta che assegna il router serve per identificare un host (accezione diversa dal solito, dove il numero di porta identifica un processo/servizio).

10.4.3 NETWORK AND PORT TRANSLATION



La porta sorgente 2000 (interna) **viene cambiata con due nuove porte esterne**. La 4001 identifica il client 192.168.0.2, mentre la 4002 identifica il client 192.168.0.3.

Quando Google invierà le due risposte all'unico IP pubblico da cui vede provenire le richieste (151.162.50.1) il router potrà capire a quale il client della LAN deve inoltrare ciascuna risposta.

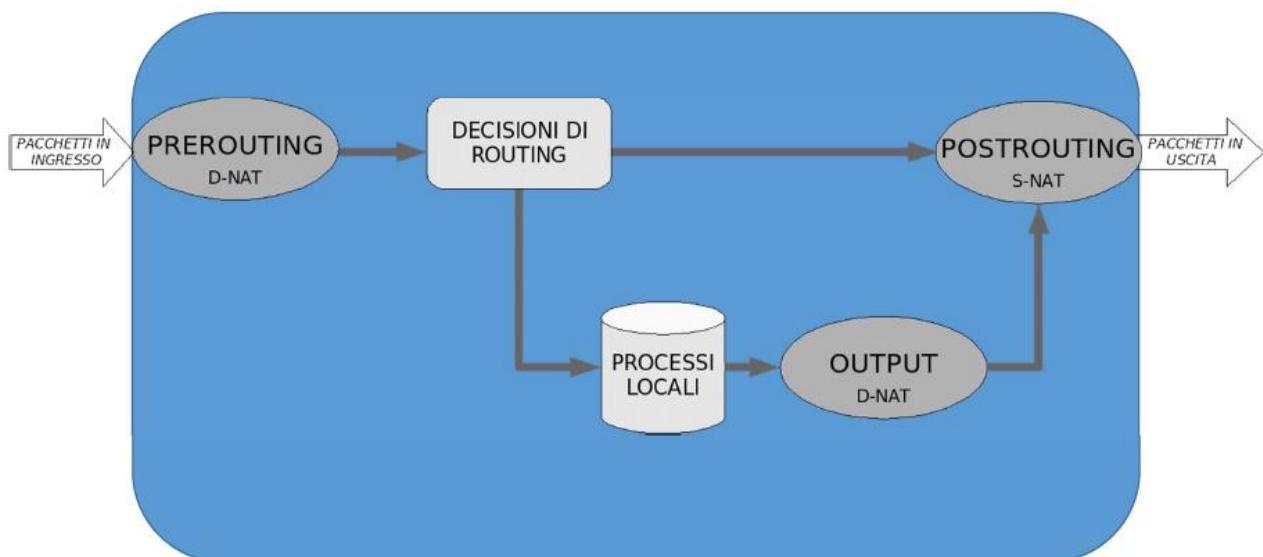


10.4.4 IPTABLES E NA[P]T

`iptables` gestisce il NA[P]T nella tabella nat. La tabella nat ha 3 catene:

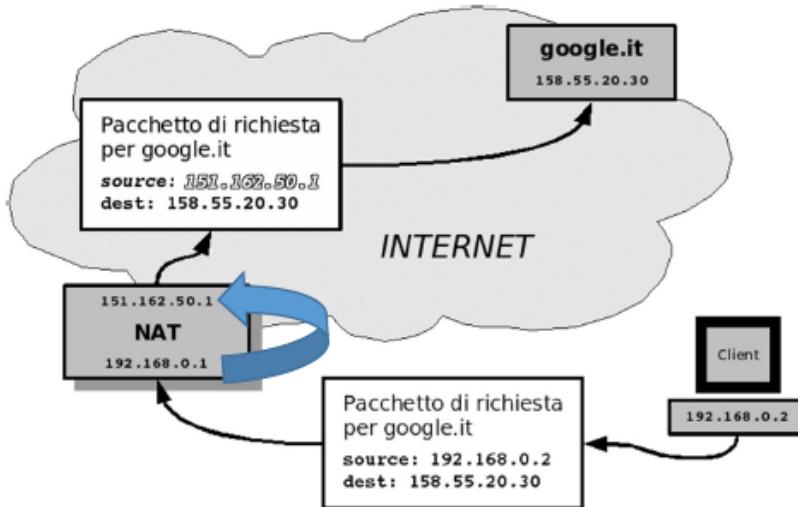
1. **PREROUTING:** fa destination NAT (D-NAT), cioè altera indirizzo/porta di destinazione dei pacchetti in arrivo.
2. **OUTPUT:** fa destination NAT (D-NAT) dei pacchetti in uscita dai processi locali, prima dei routing.
3. **POSTROUTING:** fa source NAT (S-NAT), cioè altera indirizzo/porta sorgente dei pacchetti in uscita.

I pacchetti in ingresso prima sono manipolati dal PREROUTING, che altera le destinazioni. Decisioni di routing effettua l'algoritmo di routing che consente di stabilire il next hop del pacchetto. Il POSTROUTING si fa quando il pacchetto è in uscita. Eventualmente c'è una modifica interna tramite la catena OUTPUT, prima che il pacchetto esca dall'host.



S-NAT (PACCHETTI IN USCITA)

Per tutti i pacchetti in uscita provenienti dalla sorgente 192.168.0.2, fai S-NAT impostando l'IP a 151.162.50.2. Aggiungi questa regola nella catena POSTROUTING della tabella nat.



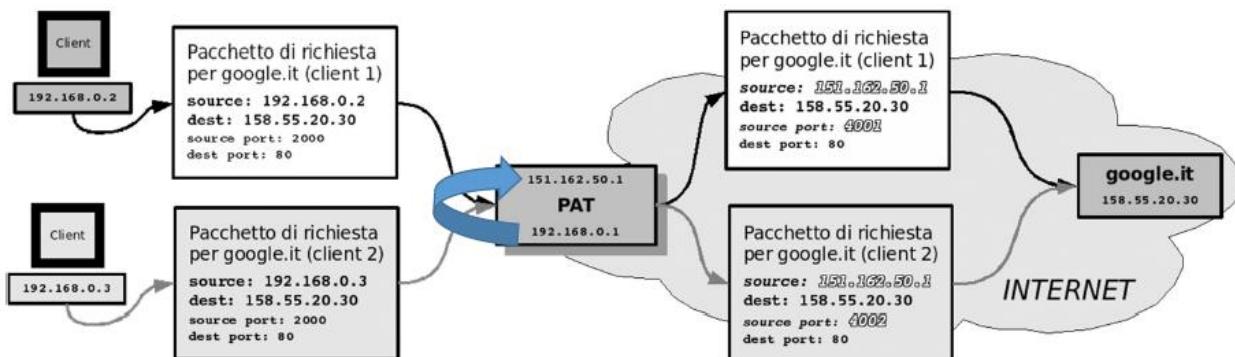
```
# iptables -t nat -A POSTROUTING -s 192.168.0.2
-j SNAT --to-source 151.162.50.1
```

Nel default gateway c'è un sistema operativo in esecuzione che gestirà i pacchetti come sopra. Se collegato al gateway c'è solo un host, allora basta modificare l'indirizzo sorgente, senza necessità di modificare anche la porta. Per tutti i pacchetti in uscita, si modifica l'indirizzo sorgente da quello specificato all'altro.

-t	Selezionare la tabella.
-A	Aggiungere una regola.
-s	Indicare il source.
-j	Indica cosa fare.

S-NAT

Per tutti i pacchetti in uscita provenienti dalla sottorete 192.168.0.0/24, fai S-NAT impostando l'IP a 151.162.50.1 e fai PAT usando il pool di porte esterne (4001,...,4100). Aggiungi questa regola nella catena POSTROUTING della tabella nat.

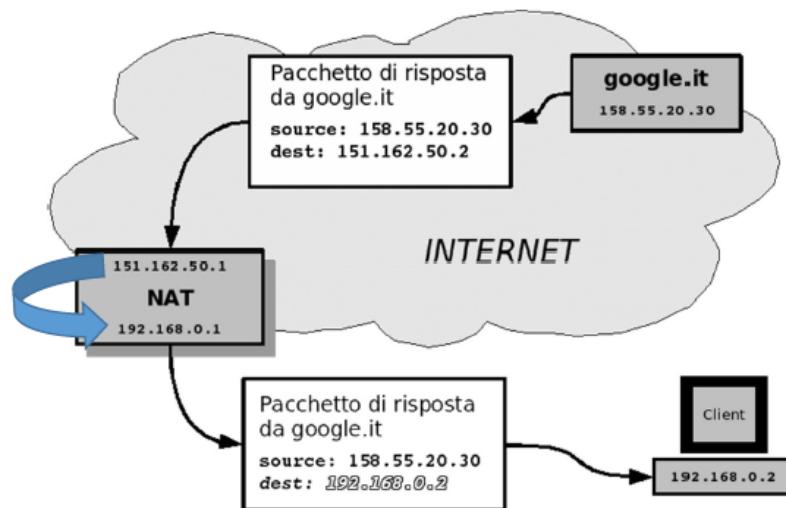


```
# iptables -t nat -A POSTROUTING -s 192.168.0.0/24
-j SNAT --to-source 151.162.50.1:4001-4100
```

Quando arriva un ingresso dalla sottorete indicata, si modifica sia l'indirizzo sorgente che il numero di porta, scegliendoli nel range 4001-4100. Il PAT mantiene la lista delle porte già utilizzate. Viene fatto questo perché la destinazione e il numero di porta è lo stesso; quindi, non si può sostituire solamente l'indirizzo IP. Il PAT a ritorno deve disambiguare, usando le informazioni scritte nella tabella.

D-NAT (PACCHETTI IN INGRESSO)

Per tutti i pacchetti in ingresso destinati a 151.162.50.1, fai D-NAT impostando l'IP a 192.168.0.2. Aggiungi la regola nella catena PREROUTING della tabella nat.

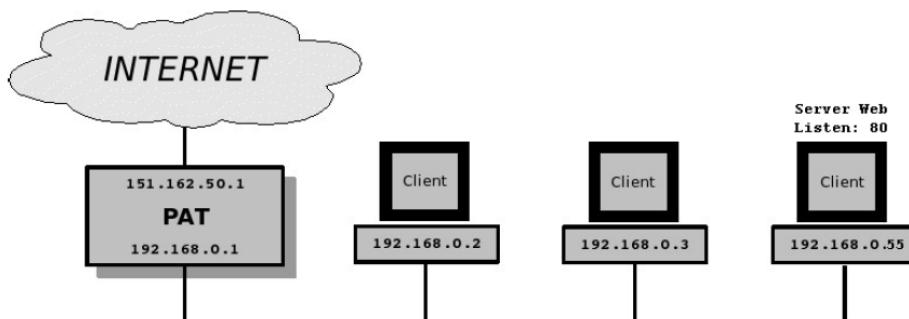


```
# iptables -t nat -A PREROUTING -d 151.162.50.1
-j DNAT --to 192.168.0.2
```

Il D-NAT ha a che fare con i processi locali. Nel caso in cui si abbia un solo client, ragioniamo in termini di NAT. Prima il NAT ha modificato un indirizzo interno in un altro. Qua facciamo un prerouting (il postrouting ha a che fare con il S-NAT): modifichiamo il valore dell'indirizzo destinatario. La differenza rispetto al caso precedente è che non si fa riferimento alle porte: vuol dire che questa regola si applica a tutti, indipendentemente dalle porte.

D-NAT

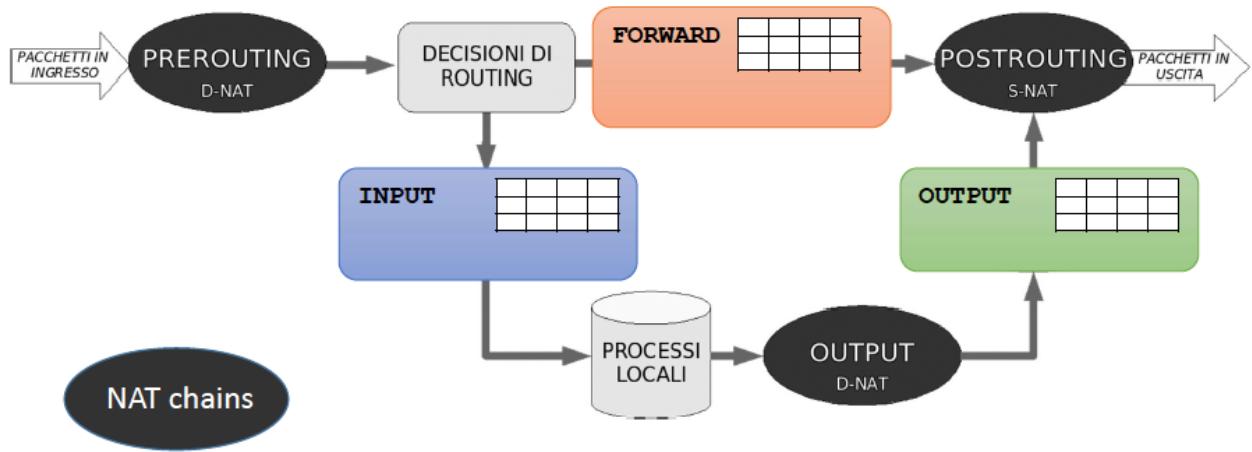
Per tutti i pacchetti TCP in ingresso destinati alla porta 80, fai D-NAT impostando l'IP a 192.168.0.55 e imposta la porta (interna) a 80. Aggiungi la regola nella catena PREROUTING della tabella nat.



```
# iptables -t nat -A PREROUTING -p tcp --dport 80
      -j DNAT --to 192.168.0.55:80
```

CATENE FILTER E NAT

Le catene di **filter** e **nat** sono disposte in modo che quelle di **filter** vedano indirizzi e porte “reali”.

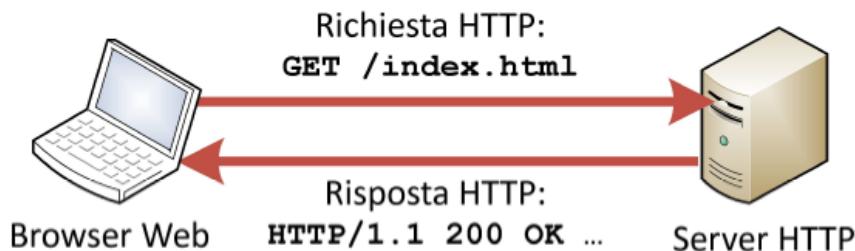


11 RICHIAMI HTTP, APACHE HTTP SERVER

11.1 HTTP

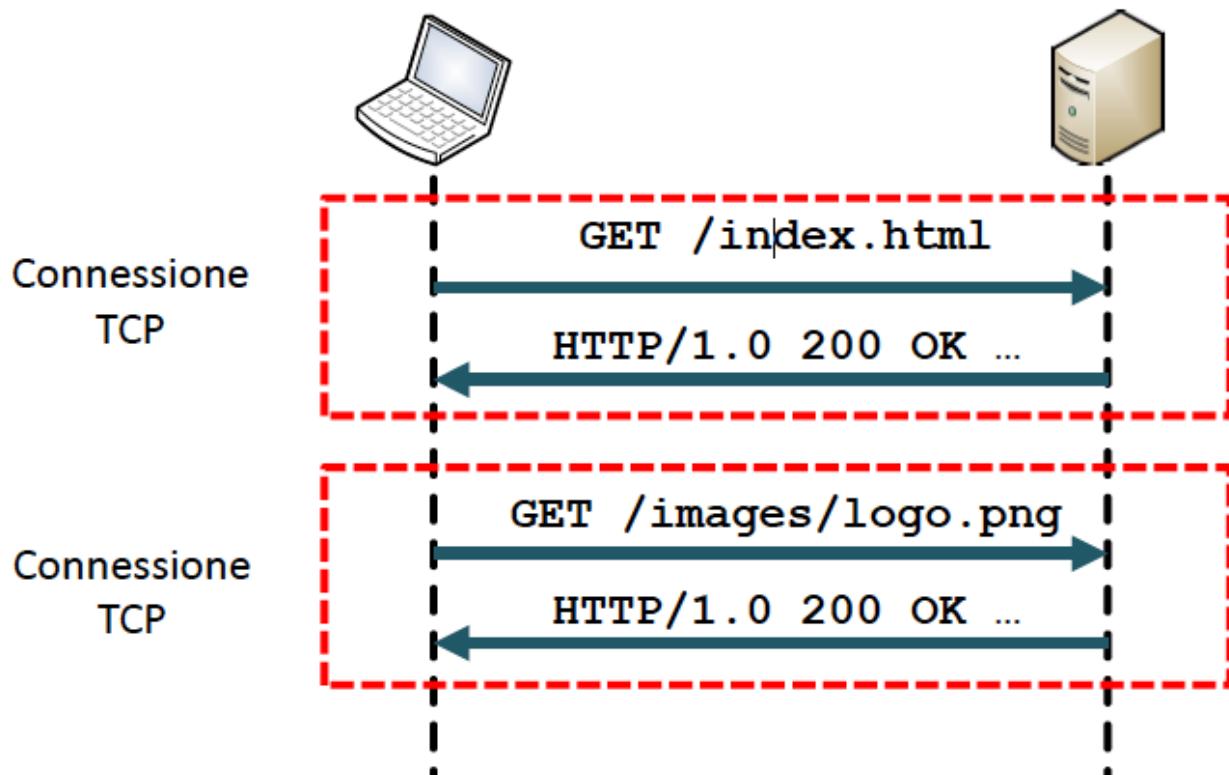
Hypertext Transfer Protocol (HTTP) è un protocollo di livello applicazione per lo scambio di ipermmedia (Il World Wide Web si basa su HTTP).

- Architettura **client/server**: il client (browser) chiede un documento e il server HTTP risponde inviandolo.
- Protocollo **stateless**: il server non tiene traccia delle precedenti connessioni. Ogni scambio richiesta/risposta è indipendente dai precedenti.



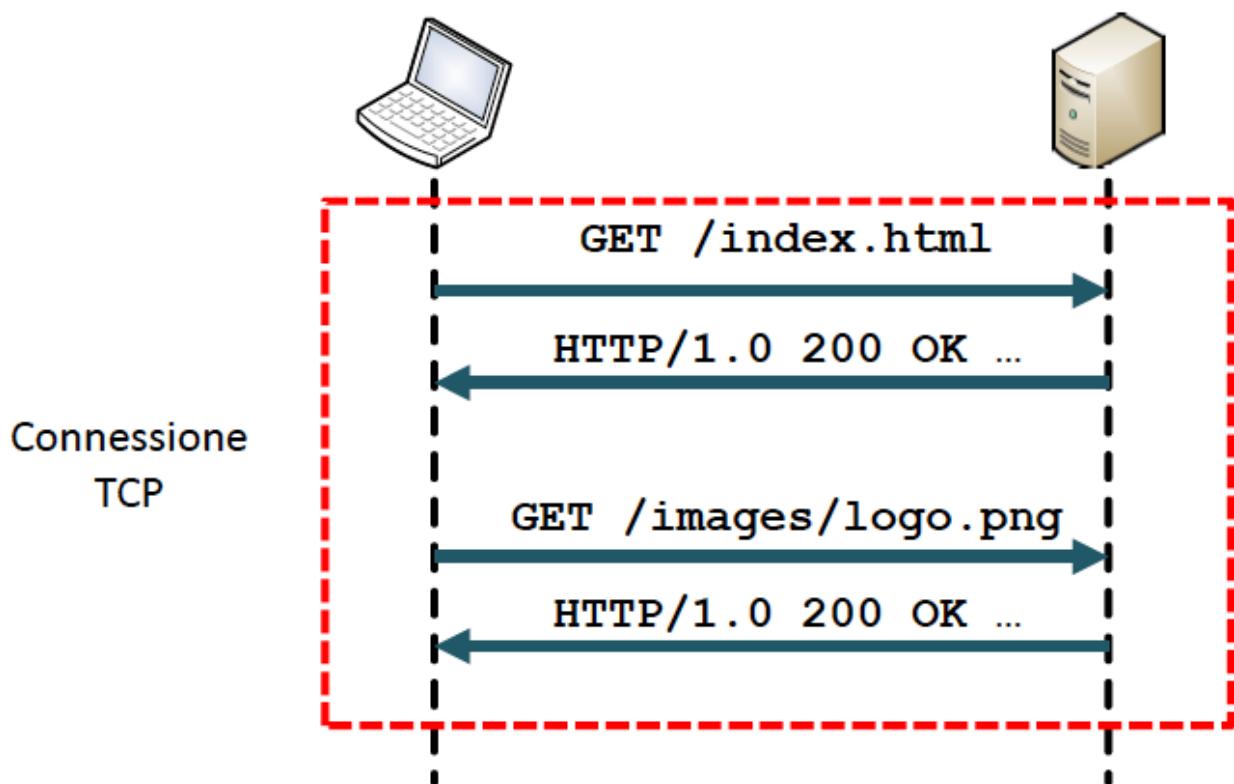
11.1.1 HTTP 1.0

Comunicazioni **non persistenti**: una connessione TCP per ogni scambio richiesta/risposta.



11.1.2 HTTP 1.1

Comunicazioni **persistenti**: una connessione TCP per tutte le richieste consecutive da un client.



11.1.3 MESSAGGI HTTP

RICHIESTA:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 ...
Connection: Keep-Alive
```

RISPOSTA:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
<qui ci sono i dati>
```

11.2 APACHE HTTP SERVER

Su Debian 8 è presente un server WEB, Apache HTTP Server, nella **versione 2.4**. È implementato tramite il programma **apache2** (solitamente, in distribuzioni diverse, è **httpd**).

Non si invoca direttamente, ma si deve usare il comando **apache2ctl**:

```
# apache2ctl <comando>
```

man 8 apache2
man 8 apache2ctl

Che permette di utilizzare comandi come **start**, **stop**, **restart**, **status**, **configtest**.

Per manipolare il server apache sono necessari i privilegi di root.

! In realtà, il privilegio di root viene lasciato quando entra in esecuzione.

In alternativa a `apache2ctl` si usa `service`, che permette di utilizzare comandi come `start`, `stop`, `restart`, `reload`.

```
# service apache2 <comando>
```

Quando Apache è in esecuzione, possiamo aprire un browser e accedere al sito

`http://localhost/`

La pagina che ci verrà mostrata sarà

`/var/www/html/index.html`

Apache può accettare e servire **più richieste** contemporaneamente. Si può impostare il modello di I/O da usare per modificare il modo con cui sono gestite le richieste (per ora supponiamo che il server web ospiti una sola pagina web).

11.2.1 FILE DI CONFIGURAZIONE

<code>/etc/apache2/</code>	Directory principale.
<code>/etc/apache2/apache2.conf</code>	File di configurazione principale.

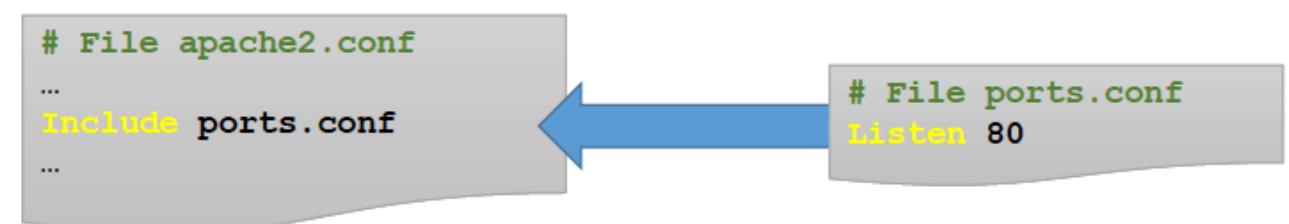
Si tratta di un file modulare, quindi la configurazione si fa tramite **direttive**, eventualmente raggruppate in **direttive contenitore**.

```
# Esempio di commento
Direttiva1 valore
Direttiva2 valore

# Inizio contenitore
<Contenitore valore>
    Direttiva3 valore
    Direttiva4 valore
</Contenitore>
# Fine contenitore
```

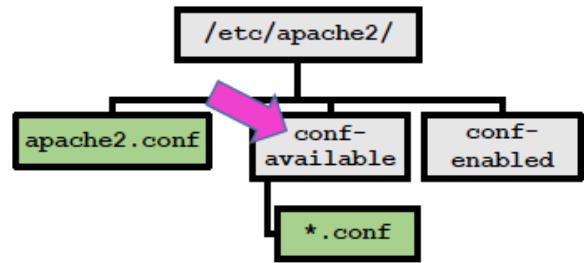
Su Debian, Apache usa un **sistema modulare**: il file di configurazione recupera le varie **parti di configurazione** (in formato `.conf`) da altri file, tramite la direttiva `Include`.

Esempio: il file `/etc/apache2/ports.conf` contiene le direttive per specificare le porte da usare.



Le **parti di configurazione** si mettono nella directory

/etc/apache2/conf-available



I file vengono **abilitati** con

```
# a2enconf <nome_file>
```

Il comando crea un **soft link** nella directory

/etc/apache2/conf-enabled

Il file di configurazione principale è impostato per includere tutti i file in conf-enabled.

Ogni volta che si modifica il file di configurazione di apache, **per rendere le modifiche effettive bisogna riavviare il server** (il firewall invece diventa immediatamente attivo).

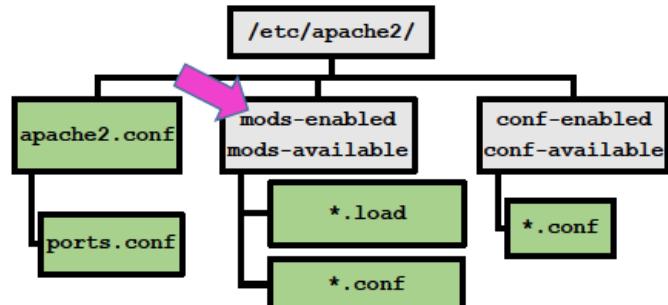
Un file si disabilita con

```
# a2disconf <nome_file>
```

11.2.2 MODULI

Le parti di configurazione che forniscono funzionalità complesse sono i **moduli**, e si trovano in

/etc/apache2/mods-available/



Si abilitano e disabilitano con

```
# a2enmod <nome_file>
# a2dismod <nome_file>
```

- Un modulo abilitato ha un soft link in mods-enabled.
- **Per ricaricare la configurazione, riavviare il server.**

11.3 DIRETTIVE GLOBALI

Specificano **opzioni globali**, cioè valide per l'intero server.

11.3.1 DIRETTIVA SERVERROOT

```
ServerRoot /etc/apache2
```

Specifica la directory principale dei file di configurazione di Apache.

! I path relativi specificati nelle altre direttive sono risolti partendo da questa directory.

Su Debian, la direttiva ServerRoot è configurata automaticamente all'avvio del servizio dal comando apache2ctl (nel file apache2.conf è infatti commentata).

11.3.2 DIRETTIVE KEEPALIVE E KEEPALIVETIMEOUT

```
KeepAlive on
```

```
KeepAliveTimeout 5
```

- `KeepAlive` specifica se offrire o meno le connessioni persistenti di HTTP 1.1.
- `KeepAliveTimeout` specifica quanti secondi attendere la successiva richiesta dal client, su una stessa connessione, prima di chiuderla.
 - Valori troppo elevati potrebbero bloccare inutilmente un processo che sta servendo un client lento o disconnesso.

11.3.3 DIRETTIVA LISTEN

```
Listen 80
```

```
Listen 8080
```

Specifica le porte su cui Apache si mette in ascolto di connessioni.

- È **obbligatoria**: se la direttiva non c'è il server non parte.
- È possibile specificare **più porte**.

Su Debian, questa direttiva è presente in `/etc/apache2/ports.conf` (incluso automaticamente da `apache2.conf`).

11.3.4 DIRETTIVA ERRORLOG

```
ErrorLog /var/log/apache2/error.log
```

Specifica il file di log degli errori. Formato e livello di dettaglio dei messaggi di errore possono essere rispettivamente definiti con `ErrorLogFormat` e `LogLevel`.

11.4 DIRETTIVE PER I SITI WEB (VIRTUAL HOST)

11.4.1 HOSTING

Nel caso più semplice, chiamiamolo 1-1-1-1, **un server Web** è in esecuzione su **una macchina** con **un indirizzo IP** e ospita **un solo sito Web**.

Ma non è realistico avere un web server per ciascun sito web: c'è un'elevata richiesta di risorse, e una configurazione del genere non scala. Basterebbe rendere apache capace di gestire più siti web contemporaneamente.

11.4.2 VIRTUAL HOST

Con il (name based) **Virtual Hosting**, si possono configurare **più siti sullo stesso server Web**, sulla stessa macchina, con lo stesso indirizzo IP.

Il server discrimina le richieste dei client in base al campo '**Host**' della richiesta HTTP.



Come accade per parti di configurazione e moduli, i siti (detti virtual host) sono in una **directory dedicata**:

`/etc/apache2/sites-available`

Si abilitano e disabilitano con

```
# a2ensite <nome_file>
# a2dissite <nome_file>
```

- Un sito abilitato ha un soft link in sites-enabled.
- **Bisogna riavviare il server per ricaricare la configurazione.**

Nel caso più semplice, l'1-1-1-1, Apache ha un **default Virtual Host** abilitato in

`/etc/apache2/sites-available/000-default.conf`

Al suo interno si trovano delle direttive contenitori:

```
<VirtualHost *:80>
    ServerName www.example.com
    ...
    DocumentRoot /var/www/html
</VirtualHost>
```

11.4.3 DIRETTIVA VIRTUALHOST

```
<VirtualHost ip:porta>
...
</VirtualHost>
```

Serve per **definire un virtual host**: è una direttiva contenitore. Come valore, necessita indirizzo IP e porta.

11.4.4 DIRETTIVA SERVERNAME

```
ServerName www.example.com
```

Specifica il **nome simbolico del sito**. È indispensabile per discriminare le richieste fatte dai client ai vari virtual host “hostati” dal server.

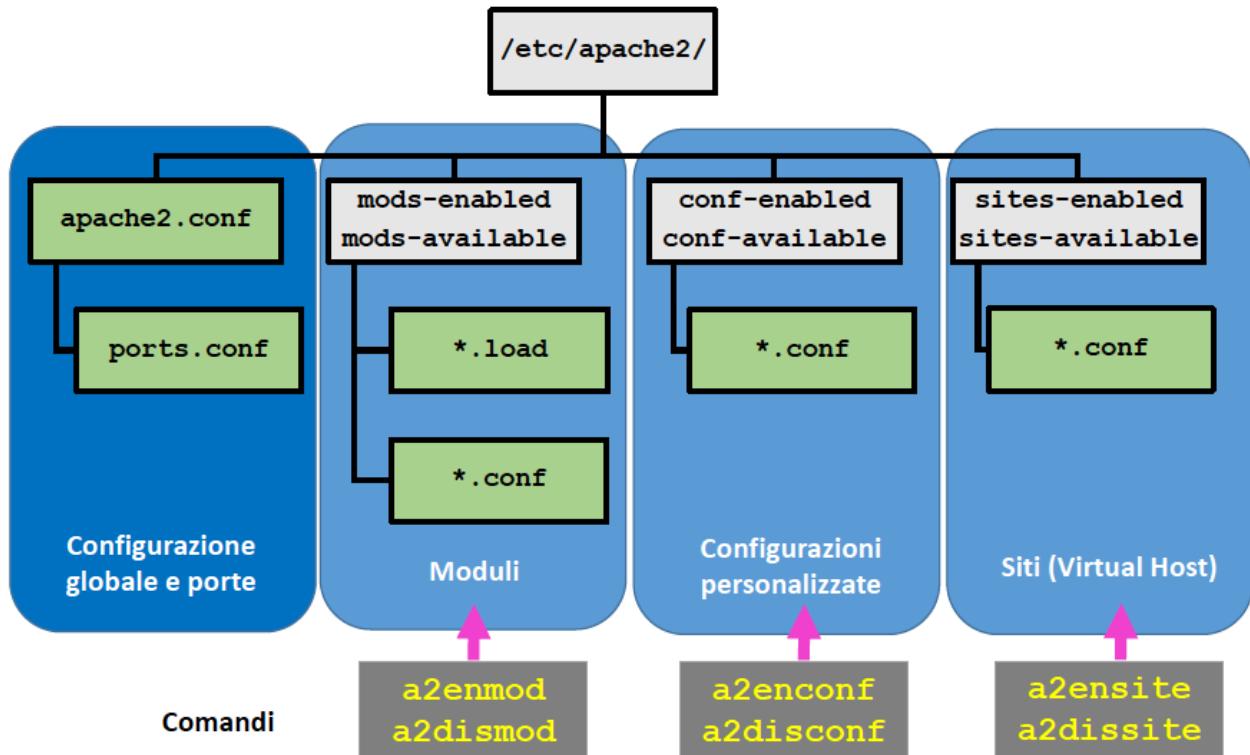
11.4.5 DIRETTIVA DOCUMENTROOT

```
DocumentRoot /var/www/html
```

Specifica la **directory dei file del sito**.

ServerRoot ≠ DocumentRoot

11.4.6 FILE DI CONFIGURAZIONE



11.5 MULTI-PROCESSING MODULES

Apache accetta e serve **più richieste contemporaneamente** tramite i moduli **Multi-Processing Module (MPM)**. Questo viene fatto tramite:

1. Gestione dei **socket**.
2. Binding delle **porte**.
3. **Processazione delle richieste** usando processi figli e thread.

Su Unix, si può scegliere tra gli MPM `prefork`, `worker` e `event`.

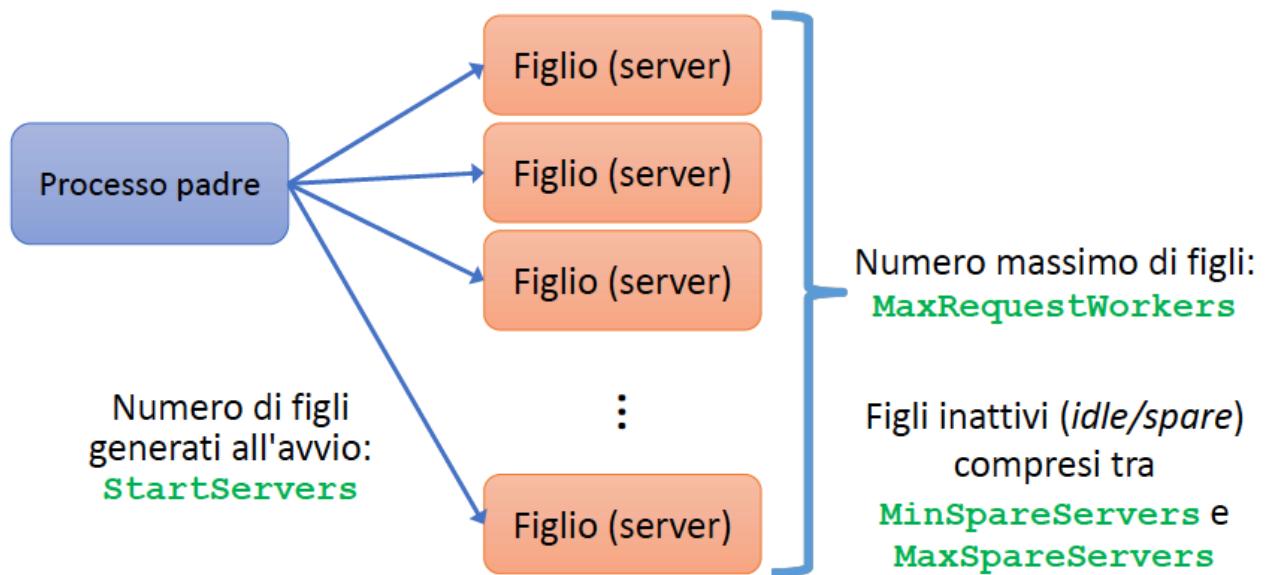
11.5.1 MPM PREFORK

Il modulo `prefork` implementa un server **multi-processo senza thread**.

All'avvio, un processo padre lancia un certo numero di processi figli (questo procedimento si chiama preforking):

1. I figli (worker o server) restano in ascolto, accettano connessioni e le servono.
2. Dopo aver
3. Dopo aver servito una connessione, il worker torna disponibile.
4. Il padre gestisce il pool dei figli, cercando di mantenerne sempre alcuni disponibili.

Il preforking all'avvio e il riuso dei vari worker evitano l'overhead della `fork()` a ogni connessione.



Ogni figlio viene riciclato per `MaxConnectionsPerChild` connessioni, poi viene terminato, per evitare memory leak accidentali.

VANTAGGI	SVANTAGGI
<ul style="list-style-type: none"> Massima compatibilità: alcuni moduli Apache o librerie potrebbero non supportare il multithreading. Massima stabilità: un processo che crasha interrompe una sola connessione. 	<ul style="list-style-type: none"> Occupazione di memoria: i processi ne occupano di più rispetto ai thread. Complessità nel tuning (occorre regolare bene le impostazioni): troppi processi inattivi occupano inutilmente memoria, troppo pochi causano più overhead da <code>fork()</code> in caso di picchi di richieste.

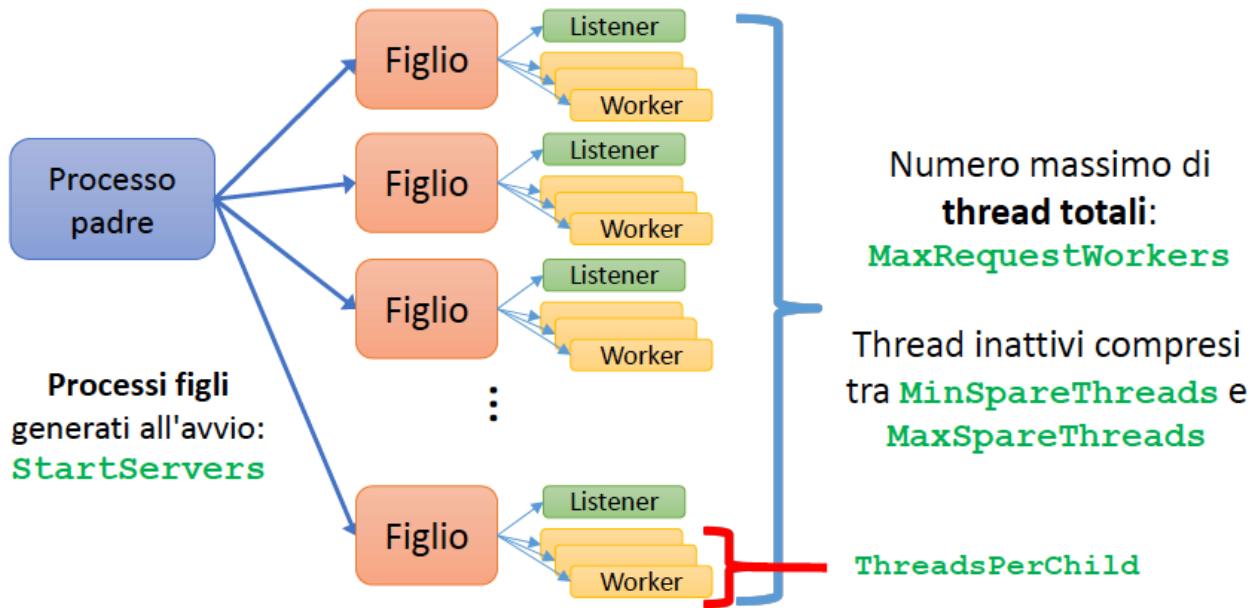
11.5.2 MPM WORKER

Server **multi-processo** e **multi-thread**.

Il processo padre genera un certo numero di processi figli. **Ogni processo figlio genera:**

1. Un thread listener che accetta/smista le connessioni.
2. Un certo numero di thread worker che servono le richieste.

Overhead ridotto grazie al preforking e risparmio di memoria grazie ai thread.



- Ogni processo figlio viene riciclato per `MaxConnectionsPerChild` connessioni.
- Il numero massimo di figli è necessariamente `MaxRequestWorkers / ThreadsPerChild`.

11.5.3 MPM EVENT

Versione migliorata di **worker**.

! **MPM di default** dalla versione 2.4.

Oltre ad accettare connessioni, il listener gestisce le connessioni **temporaneamente inattive**:

- Esempio 1: un worker è connesso a un client che **tarda ad inviare una richiesta**. Invece di attendere, restituisce il controllo del socket al *listener* e passa a seguire un altro client. Quando il primo client invierà la richiesta, il *listener* la assegnerà a un altro worker libero.
- Esempio 2: un worker sta servendo un client con una **connessione lenta** e il buffer di invio del socket si riempie. Invece di attendere, restituisce il controllo del socket al *listener* che lo assegnerà ad un altro worker non appena sarà di nuovo scrivibile.

Aumenta il numero di connessioni servibili contemporaneamente a parità di numero di thread, eliminando i tempi morti.

Dal punto di vista delle direttive, vale quanto detto per **worker**: si può settare il numero di figli da generare all'avvio, il numero massimo di thread totali, il numero di thread inattivi in un range di valori e il numero di connessioni per cui un processo figlio viene generato.

ThreadLimit è il limite massimo configurabile per il numero di thread attivi per processo.

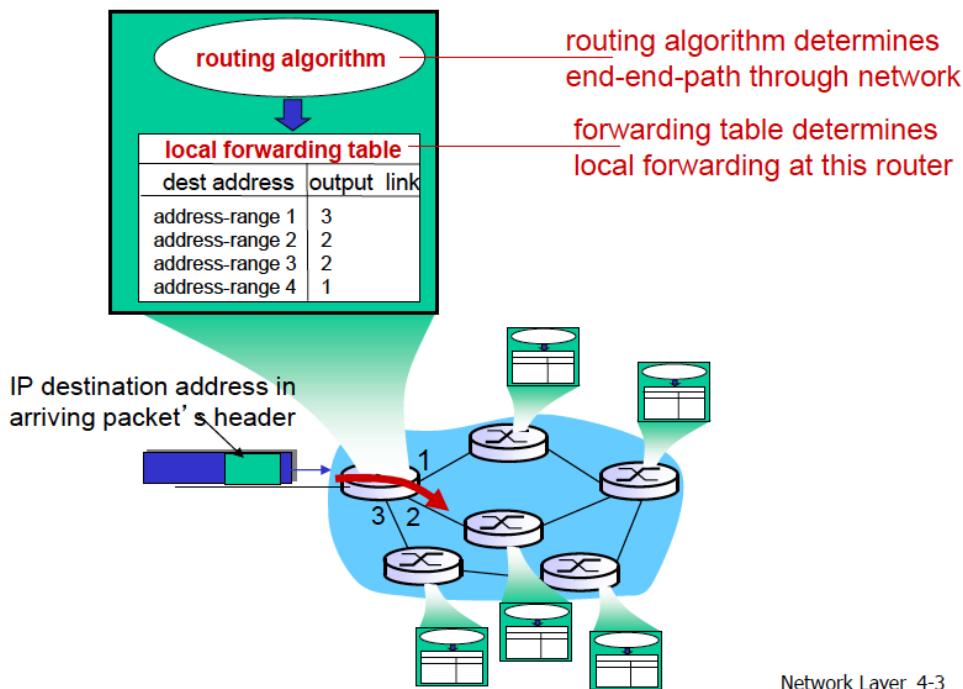
ServerLimit è il limite massimo configurabile di processi figli attivi.

12 ALGORITMI DI ROUTING

12.1 INTERAZIONE TRA ROUTING E FORWARDING

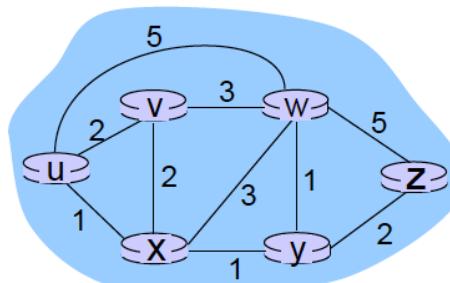
Quando un pacchetto raggiunge un router, occorre determinare l'hop successivo. I router collaborano tra di loro e si coordinano affinché un pacchetto possa arrivare a destinazione seguendo un percorso ottimo.

Un router ha una tabella di forwarding contenente i link e le direzioni: rispetto ad un destination address di un certo tipo, si instrada una data interfaccia (un router sarà connesso a più router).



12.2 GRAFO

L'astrazione che prendiamo in considerazione è quella di **grafo**, un insieme di coppie di nodi e archi che li collegano.



graph: $G = (N, E)$

N: set of routers = { u, v, w, x, y, z }

E: set of links = { (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }

Ogni arco avrà un costo per il passaggio su di esso, eventualmente variabile nel tempo a causa della congestione e della larghezza di banda.

$$c(x, x') = \text{costo del link } (x, x') \rightarrow \text{Esempio: } c(w, z) = 5$$

$$\text{costo del percorso } (x_1, x_2, x_3, \dots, x_p) = c(x_1, x_2) + c(x_2, x_3) + \dots + c(x_{p-1}, x_p)$$

12.3 CLASSIFICAZIONE DEGLI ALGORITMI DI ROUTING

Per poter classificare gli algoritmi di routing, dobbiamo porci due domande:

1. L'informazione è globale o decentralizzata?
2. L'algoritmo è statico o dinamico?

INFORMAZIONE GLOBALE: prendono il nome di **algoritmi Link State**, i router conoscono la topologia completa e tutte le informazioni sui costi dei link.

INFORMAZIONE DECENTRALIZZATA: prendono il nome di **algoritmi Distance Vector**, i router conoscono i vicini fisicamente connessi, e i costi dei link che li collegano ai vicini.

- Computazione iterativa, scambio di informazioni solo con i vicini.

ALGORITMO STATICO: i percorsi cambiano lentamente nel tempo.

ALGORITMO DINAMICO: i percorsi cambiano rapidamente.

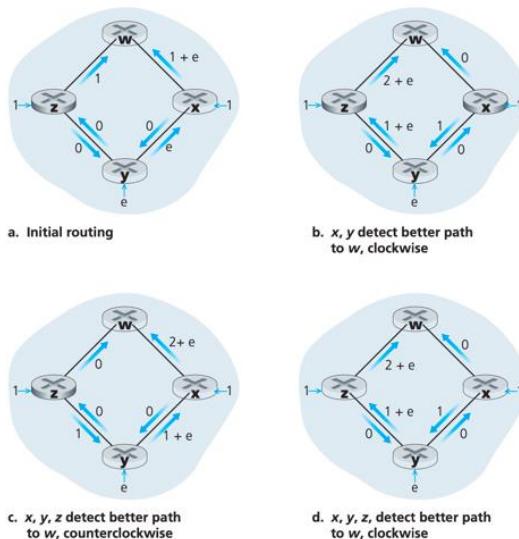
- Aggiornamenti periodici.
- I costi dei link cambiano.

12.4 ALGORITMO DI DIJKSTRA: ALGORITMO LINK STATE

L'algoritmo di Dijkstra consente di determinare lo **shortest path** tra due nodi. I nodi si scambiano la topologia completa della rete, in una fase di broadcast.

- È link-state con complessità quadratica.

L'uso di Dijkstra diventa problematico nel momento in cui si hanno delle modifiche dei costi degli archi dovuti alla congestione.



1. Z vuole inoltrare un'unità di flusso a W.
2. X vuole inoltrare un'unità di flusso a W.
3. Y vuole inoltrare un'unità di flusso a W.

Lo shortest path è quello indicato in figura a, in cui i router tendono a usare i link scarichi. Tuttavia, facendo passare i pacchetti in questo modo, si modifica il livello di congestione. Quindi, si effettua nuovamente Dijkstra per modificare i percorsi e le destinazioni di pacchetti: si parla in tal senso di oscillazioni dell'algoritmo.

- È iterativo: dopo k iterazioni si conosce il path migliore.

Il problema si limita applicando l'algoritmo a reti statiche, in cui la presenza dei pacchetti sulla linea non è rilevante.

12.5 ALGORITMI DISTANCE VECTOR

L'altro gruppo di algoritmi è il Distance Vector, relativi a una conoscenza basata solo sui router vicini, quelli a cui il router è direttamente connesso.

Si sfrutta la **programmazione dinamica**, tramite l'**equazione di Bellman-Ford**, in cui si divide il problema in sotto problemi e si trova una sequenza di risoluzione dei sotto problemi che porti in ultimo all'ottimo.

Sia quindi

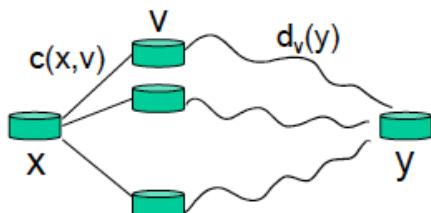
$$d_x(y) = \text{costo del percorso di costo minimo da } x \text{ a } y$$

Abbiamo che

$$d_x(y) = \min_v \{c(x, v) + d_v(y)\}$$

Dove:

1. \min_v è il minimo preso tra tutti i vicini v di x.
2. $c(x, v)$ è il costo per raggiungere il vicino v.
3. $d_v(y)$ è il minimo costo da v alla destinazione y.



Per ogni router, si considera il costo per andare a ciascun suo nodo vicino: il cammino di costo minimo da x a y è il **minimo tra tutti i costi**, calcolato come il costo dal nodo x al suo vicino v, più il costo minimo dal vicino v a y.

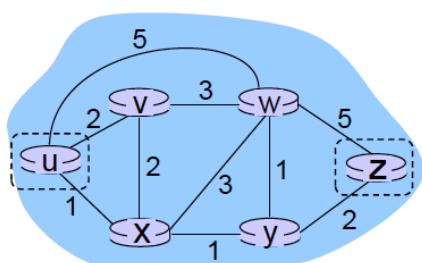
- La ricerca del minimo viene quindi fatta sui vicini.

12.5.1 ESEMPIO BELLMAN-FORD

Vogliamo calcolare il percorso minimo tra u e z.

Abbiamo $d_v(z) = 5$, $d_x(z) = 3$, $d_w(z) = 3$. L'equazione di Bellman-Ford ci dice che

$$\begin{aligned} d_u(z) &= \min\{c(u, v) + d_v(z), c(u, x) + d_x(z), c(u, w) + d_w(z)\} \\ &= \min\{2 + 5, 1 + 3, 5 + 3\} = 4 \end{aligned}$$



Il nodo n^* che raggiunge il minimo sarà il **prossimo hop** nel percorso più breve, usato nel forwarding table.

$$n^* = \operatorname{argmin}_v \{c(x, v) + d_v(y)\}$$

12.5.2 RITORNANDO ALL'ALGORITMO GENERALE

La soluzione del problema non è shortest path, perché ognuno ha il “paraocchi” sul percorso complessivo: ogni router sa solo dove deve instradare il pacchetto per ottenere il percorso di costo minimo. Essendoci coordinazione tra i vicini, ci deve essere una propagazione delle informazioni.

Chiamiamo $D_x(y)$ la **stima del costo minimo tra x e y**. Il vettore

$$D_x = [D_x(y) : y \in N]$$

Contiene le stime di costo minimo per andare da x a tutti gli altri.

Ogni nodo x :

1. Conosce il costo per raggiungere ogni vicino v : $c(x, v)$.
2. Per ogni vicino v , ha un Distance Vector $D_v = [D_v(y) : y \in N]$

IDEA CHIAVE:

- Di volta in volta, ogni nodo invia il proprio Distance Vector (DV) ai vicini.
- Quando x riceve un nuovo DV da un vicino, x aggiorna il suo DV utilizzando l’equazione di Bellman-Ford:

$$D_x(y) \leftarrow \min_v \{c(x, v) + D_v(y)\} \quad \forall y \in N$$

- In condizioni naturali, la stima $D_x(y)$ **converge al minimo costo effettivo $d_x(y)$** .

Questo algoritmo è:

1. **ITERATIVO E ASINCRONO:** ogni iterazione locale è causata da cambiamenti dei costi dei link locali e messaggi di aggiornamento dei DV dai vicini.
2. **DISTRIBUITO:** ogni nodo manda una notifica ai vicini soltanto quando il suo DV cambia. I vicini manderanno a loro volta una notifica ai loro vicini se necessario.

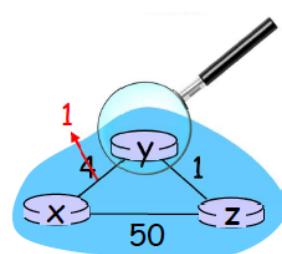
Ogni nodo:

- **Aspetta** per cambiamenti nel costo dei link locali o messaggi dai vicini.
- **Ricalcola** le stime.
- Se il DV di qualsiasi destinazione è cambiato, allora **lo notifica** ai vicini e ritorna al primo punto.

12.5.3 CAMBIAMENTI NEI COSTI DEI LINK

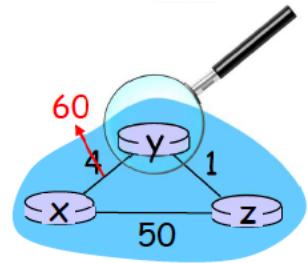
Supponiamo che ad un certo punto il costo di un arco passi da 4 a 1.

- $t_0 - y$ si accorge del cambiamento del costo del link, aggiorna il suo DV e lo comunica ai vicini.



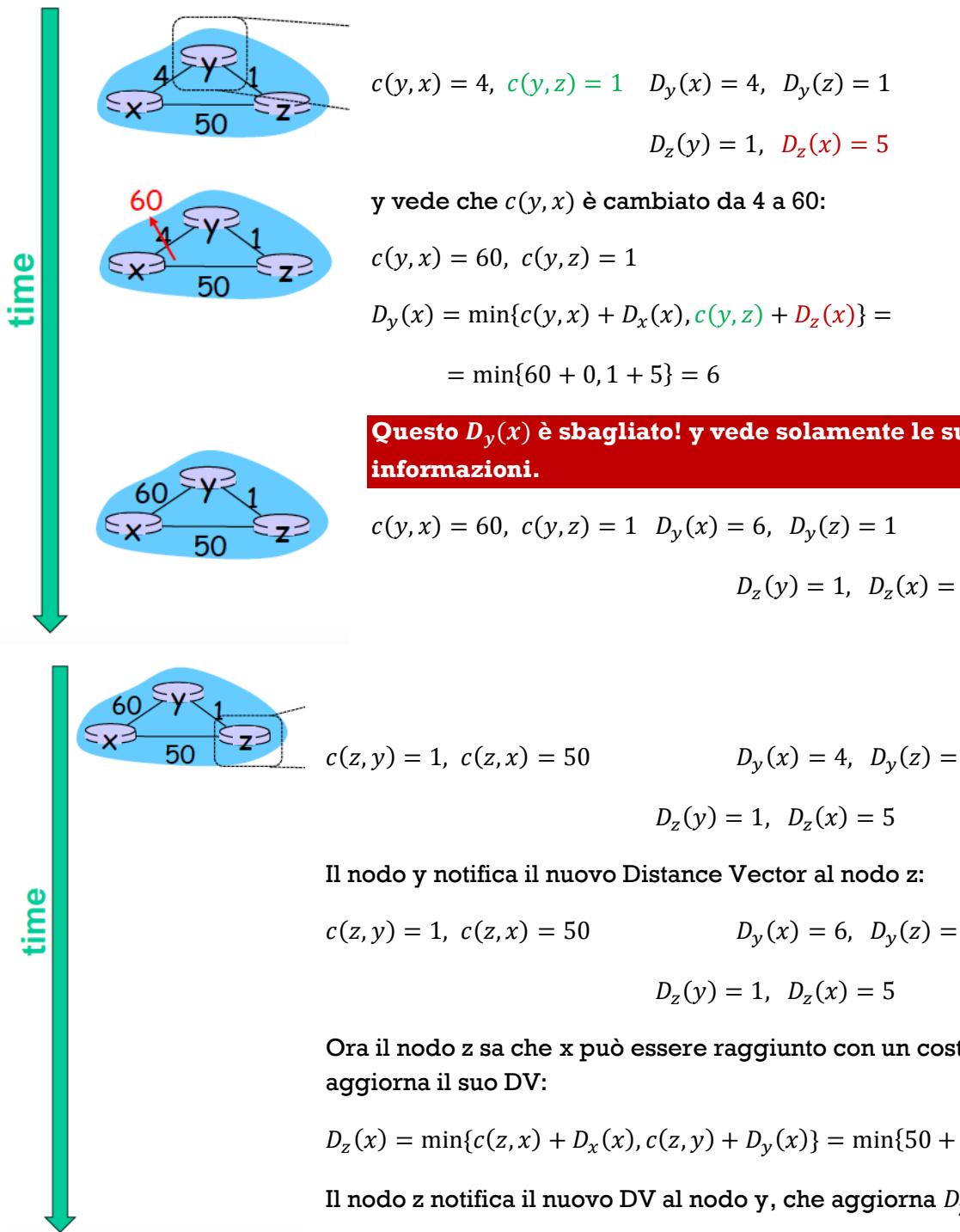
- t_1 – z riceve l'aggiornamento da y, aggiorna la sua tabella delle distanze, ricalcola il costo minore per arrivare a x e invia il suo DV ai vicini.
- t_2 – y riceve l'aggiornamento di z, aggiorna la sua tabella delle distanze. Il costo minimo di y **non cambia**, quindi y **non invia** un messaggio a z.

Cosa succede se invece il costo aumenta? Il comportamento non è lineare come prima, e servono 44 iterazioni prima che l'algoritmo si stabilizzi. Questo problema prende il nome di **conteggio all'infinito**.



CONTEGGIO ALL'INFINITO

Andiamo ad analizzare l'evoluzione nel tempo del sistema:



Questo procedimento va avanti per 44 iterazioni.

POISONED REVERSE

Per risolvere questa situazione, si usa il **poisoned reverse**: se il router z, per raggiungere x, passa per y, allora z dice a y che la sua distanza per x è infinito. In questo modo, y si rende conto che non è conveniente inviare ad x tramite z, perché abbiamo costo infinito, rompendo il meccanismo delle 44 iterazioni.

12.5.4 CONFRONTO TRA ALGORITMO LS E DV

ALGORITMO LS	ALGORITMO DV
COMPLESSITA' DEI MESSAGGI	
Con n nodi, E link, abbiamo $O(nE)$ messaggi inviati.	Lo scambio avviene solo tramite vicini, quindi è variabile.
VELOCITA' DI CONVERGENZA	
Algoritmi $O(n^2)$ necessitano $O(nE)$ messaggi. <ul style="list-style-type: none">• Ci possono essere oscillazioni.	Variabile: <ul style="list-style-type: none">1. Ci possono essere loop.2. Problema del conteggio all'infinito.
ROBUSTEZZA	
In caso di malfunzionamenti del router: <ul style="list-style-type: none">1. I nodi possono inviare il costo sbagliato del link.2. Ogni nodo calcola solamente la propria tabella.	In caso di malfunzionamenti del router: <ul style="list-style-type: none">1. Il DV può inviare il costo sbagliato del percorso.2. Ogni tabella di ogni nodo è usata anche dagli altri: l'errore si può propagare.

12.6 ROUTING GERARCHICO

Il routing che abbiamo visto fino ad ora è un'idealizzazione basata su alcune ipotesi:

1. Tutti i router sono identici.
2. Il network è “piatto”.

Questa idealizzazione **non è realizzabile in pratica**. Con oltre 600 milioni di destinazioni:

- Non è possibile memorizzare tutte le destinazioni nelle routing tables.
- Uno scambio di tabelle di routing “sommergebbe” i link.

Inoltre, Internet è un **network di network**, e ogni amministratore di network vorrebbe controllare il routing interno al proprio network.

Ogni insieme di router si comporta in modo indipendente dagli altri (**Autonomous System**). All'interno di uno stesso AS c'è lo **stesso protocollo di routing**, il cosiddetto intra-AS routing protocol.

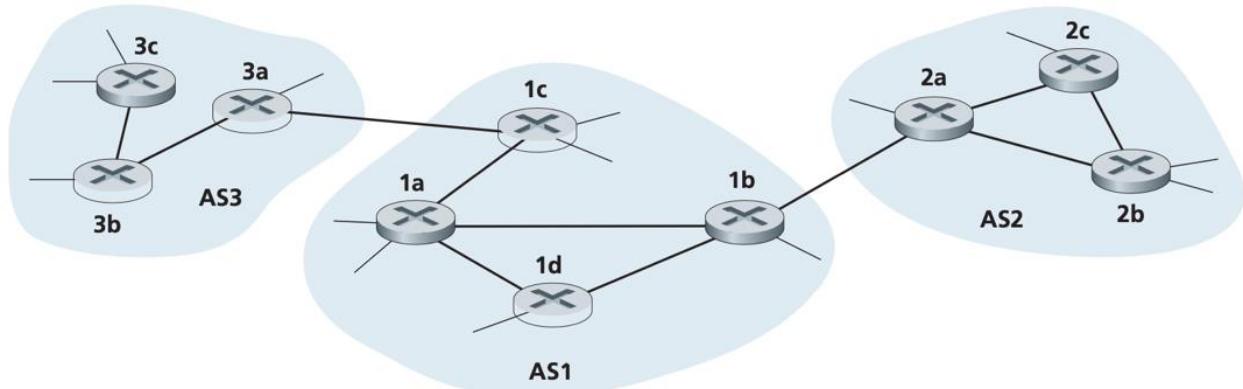
- Router in diversi AS possono eseguire diversi intra-AS routing protocol.

A collegare due AS ci sono i **gateway di frontiera**.

12.6.1 AS INTERCONNESSI

Le tabelle di forwarding sono configurate sia da algoritmi intra-AS che da algoritmi inter-AS:

- **Intra-AS:** setta le entrate per le destinazioni interne. Contengono i next-hop per inoltrare a destinazioni nel proprio AS.
- **Inter-AS:** setta le entrate per le destinazioni esterne. Contengono i next-hop per inoltrare a destinazioni fuori dal proprio AS.
 - La politica che si adotta è quella di uscire dal proprio AS nel modo più veloce possibile.



Supponiamo che AS1 riceva dei datagram che devono essere inoltrati all'esterno di AS1: i router dovrebbero inoltrare i pacchetti al gateway di frontiera, ma a quale? AS1 deve:

1. Capire quali destinazioni sono raggiungibili tramite AS2 e quali tramite AS3, in modo da capire se conviene propagare un pacchetto verso 1c o 1b.
2. Propagare questa informazione a tutti i router in AS1.

È conosciuto anche come **interior gateway protocol (IGP)**. Il protocollo di routing intra-AS più famoso è l'**OSPF: Open Shortest Path First**.

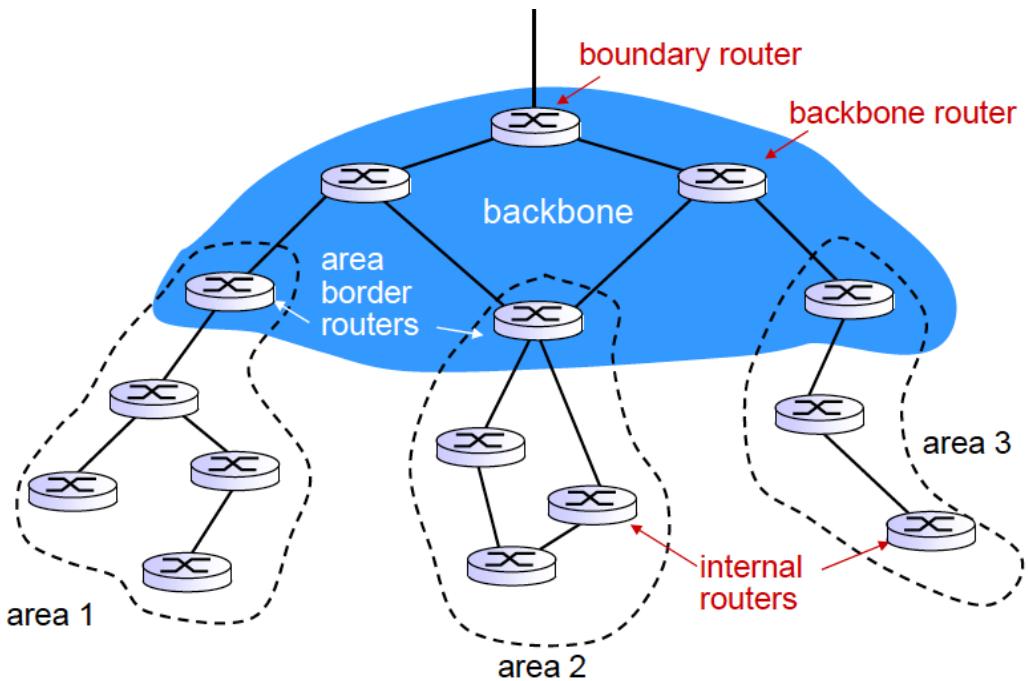
OSPF (OPEN SHORTEST PATH FIRST): "open" perché è pubblico.

È un **algoritmo di tipo link-state**:

- Dissemina pacchetti sulla topologia della rete.
- Tutti i router in un AS diventano consci della topologia di rete.
- Il calcolo del percorso è fatto usando l'algoritmo di Dijkstra.

Gli annunci di OSPF contengono una sola entrata per ciascun vicino.

OSPF GERARCHICO: visto che le AS potrebbero essere piuttosto grandi, si utilizza il routing gerarchico, basato sul dividere i router appartenenti ad un AS in più aree.



Gerarchia a due livelli: area locale e backbone.

- Solo annunci con link-state nella stessa area.
- Ogni nodo ha una topologia dettagliata dell'area, tuttavia conosce solamente la direzione (percorso più breve) alle reti in altre aree.

L'intera figura è un AS, le aree OSPF sono partizioni dell'AS. In questo caso abbiamo tre aree, all'interno delle quali si utilizzeranno degli algoritmi intra-AS, con il vantaggio che, dividendo i router in tre aree più piccole, il flooding è più leggero, coinvolgendo meno router.

- **Router di bordo area:** “riassumono” le distanze verso i net della propria area e inviano annunci agli altri router di bordo area.
- **Router backbone:** esegue l'instradamento OSPF limitato alla backbone.
- **Router di confine:** connettono ad altri AS.

Se i router interni vogliono comunicare con prefissi di rete esterni dalla propria area, si rivolgono ai router di bordo area, che tramite la backbone inoltrano i pacchetti in un'altra area o in un'altra AS (l'area che comprende la backbone, i router di bordo area e l'area zero).

Ogni nodo ha informazioni sullo shortest path relativamente all'area a cui afferisce, e sa quale è il router di bordo per inoltrare un pacchetto al di fuori della propria area OSPF.

I router di bordo contengono anch'essi delle informazioni sulla topologia dell'area OSPF e della backbone, all'interno della quale si usa ancora OSPF.

BGP (BORDER GATEWAY PROTOCOL): il protocollo di routing inter-domain *de facto*.

- È il collante per l'intero Internet.

BGP fornisce a ogni AS un mezzo per:

- **eBGP:** ottiene le informazioni di raggiungibilità dai vicini. Si preoccupa di capire quali AS vicini portano a quali prefissi di rete esterni al mio AS.
- **iBGP:** propaga le informazioni di raggiungibilità ai router degli AS.

BGP determina una “buona” rotta verso altre reti sulla base delle informazioni e delle politiche di raggiungibilità.

Permette alle sottoreti di avvisare il resto di Internet riguardo la loro esistenza.

PROTOCOLLI PER GATEWAY ESTERNI: si occupano di **aspetti politici**, che non hanno necessariamente a che fare con l’ottimizzazione.

- Esempio: un AS aziendale potrebbe non essere disposto a trasportare pacchetti generati da un AS estraneo e diretti ad un altro AS estraneo (anche se si trova sullo shortest path). Potrebbe essere disposto a farlo per AS specifici che **hanno pagato per il servizio** → Le aziende telefoniche sono felici di farlo per i loro clienti, ma non per quelli di altre aziende.

CRITERI PER PROTOCOLLI INTERDOMINIO

- Nessun traffico commerciale su reti di ricerca.
- Non usare AT&T in Australia perché ha basse prestazioni.
- Il traffico da/verso Apple non deve passare attraverso Google.

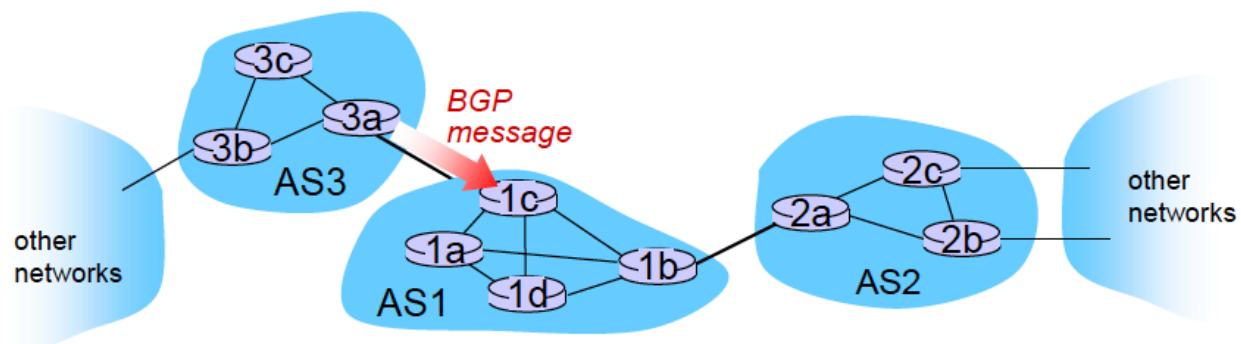
POLITICHE DI ROUTING: decidono quale traffico può fluire su quali linee fra AS.

- Sono proprietarie e individuali.
- Politica comune: un cliente di un ISP paga un altro ISP per consegnare pacchetti a qualunque altra destinazione di Internet e ricevere pacchetti inviati da qualunque altra destinazione (l’ISP cliente compra un **servizio di transito** da un ISP fornitore).

12.6.2 BPG BASICS

SESSIONE BGP: due router BGP (peer) si scambiano messaggi BGP.

- Protocollo di tipo **path vector**: i router mandano advertisement per i **path** a diversi prefissi di rete destinatari.



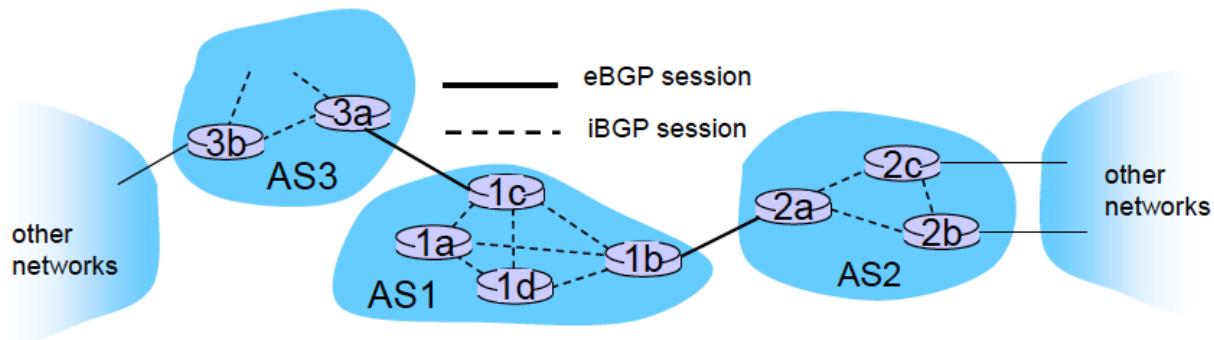
Quando AS3 pubblicizza un prefisso ad AS1:

- AS3 **promette** che inoltrerà i datagram attraverso quel prefisso.
- AS3 può aggregare i prefissi nel suo avviso.

DISTRIBUIRE LE INFORMAZIONI DEL PATH: usando una sessione eBGP tra 3a e 1c, AS3 invia il prefisso delle informazioni di raggiungibilità ad AS1.

- 1c può usare iBGP per distribuire nuovi prefissi a tutti i router di AS1.
 - 1b può pubblicizzare nuove informazioni di raggiungibilità ad AS2 tramite la sessione eBGP 1b-to-2a.

Quando un router impara il nuovo prefisso, **crea un'entrata per il prefisso nella sua tabella di forwarding**.



ATTRIBUTI DEL PATH E PERCORSI BGP: gli advertisement di BGP includono degli attributi BGP.

prefisso + attributi = percorso

Due attributi fondamentali sono:

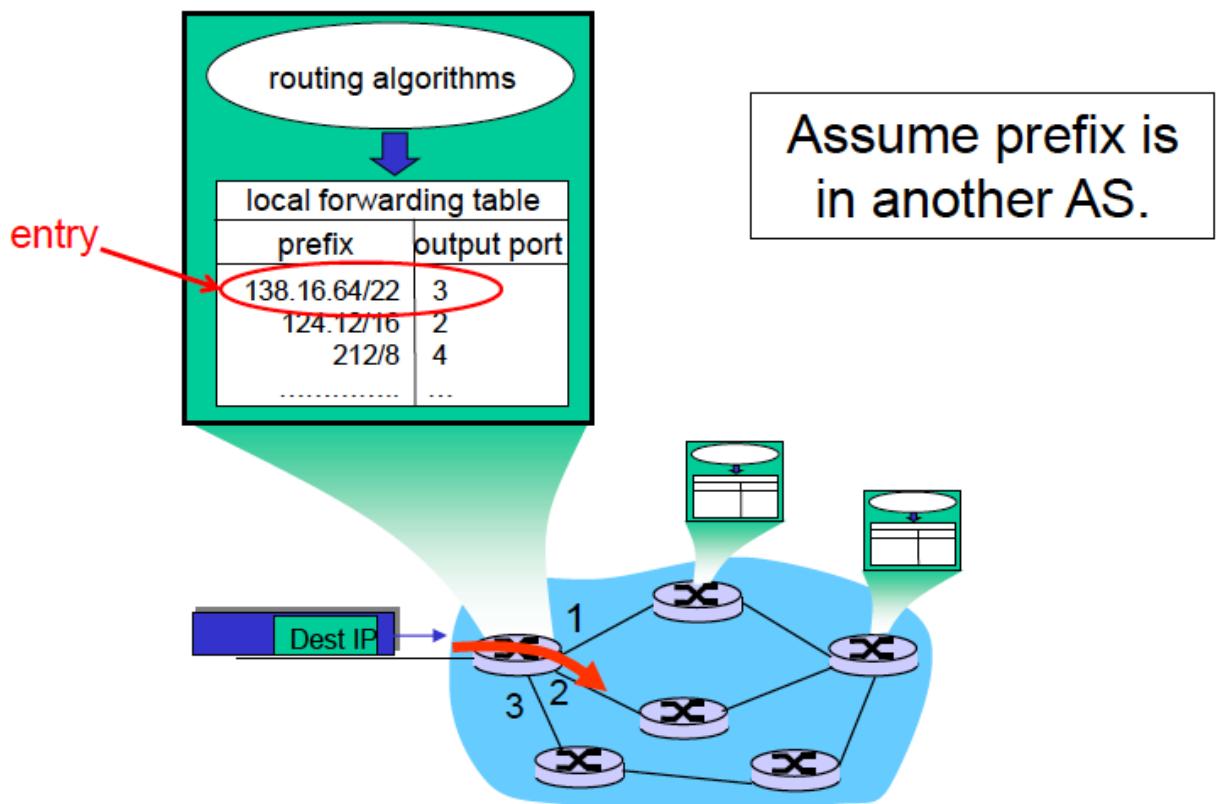
1. **AS-PATH**: contiene tutti gli AS tramite i quali l'advertisement è passato, in modo da sapere il path da percorrere per ricondursi all'indirizzo.
 2. **NEXT-HOP**: indica il router interno all'AS al quale bisogna inoltrare per raggiungere il next-hop.

Il gateway router che riceve questi advertisement usa delle **import policy** per scegliere un determinato percorso.

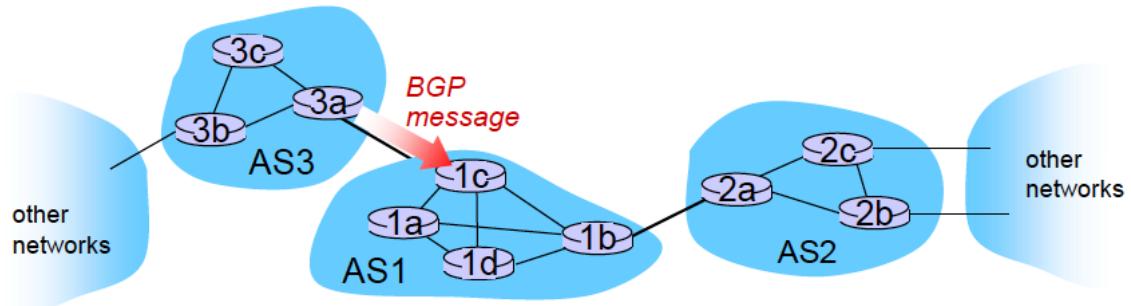
BGP ROUTE SELECTION: ogni router può conoscere più di un percorso verso l'AS destinatario → La selezione di quale percorso prendere viene fatta in base a:

- Attributi di preferenze locali.
 - AS-PATH più breve.
 - Altri criteri.

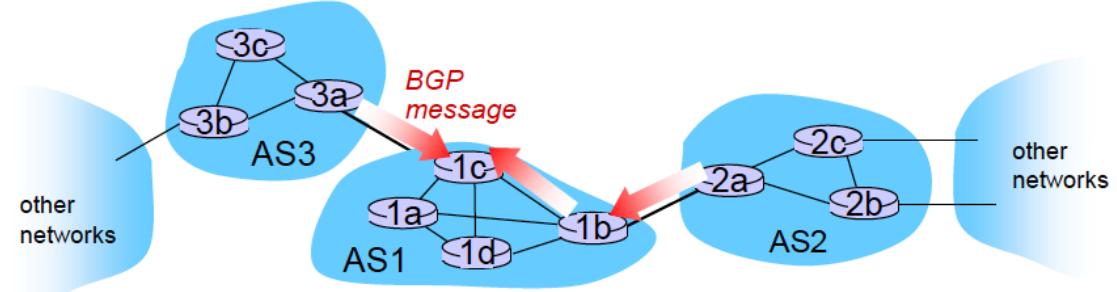
COME VIENE AGGIUNTA L'ENTRATA NELLA TABELLA DI FORWARDING?



- 1. IL ROUTER VIENE A CONOSCENZA DEL PREFISSO:** il messaggio BGP contiene delle rotte, ciascuna delle quali è una coppia con prefisso e attributi [AS-PATH; NEXT-HOP].



- 2. IL ROUTER DETERMINA LA PORTA DI OUTPUT PER IL PREFISSO:** il router può ricevere più percorsi per lo stesso prefisso.



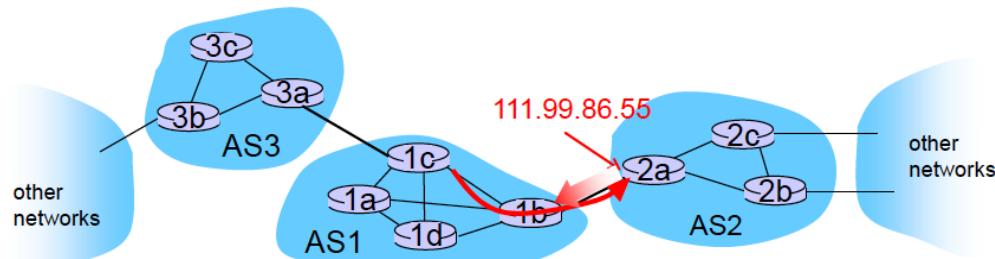
Selezione il percorso basandosi sull'**AS-PATH più breve**.

A questo punto si trova la miglior intra-route per la rotta BGP: si usa l'**attributo NEXT-HOP** del percorso selezionato → L'attributo NEXT-HOP della route è l'indirizzo IP dell'interfaccia del router che inizia l'AS-PATH.

- Esempio: AS2 invia un advertisement per il prefisso 138.16.64/22 come segue

AS-PATH: AS2 AS17 ; NEXT-HOP: 111.99.86.55

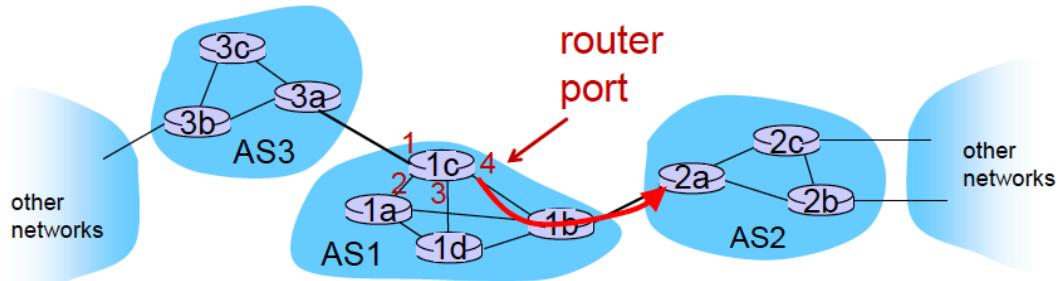
Il router usa OSPF per trovare il percorso più breve da 1c a 111.99.86.55.



3. IL ROUTER INSERISCE LA PORTA PER IL PREFISSO NELLA FORWARDING TABLE

TABLE: viene quindi identificata la porta lungo il percorso più breve OSPF e viene aggiunta l'entrata per la porta del prefisso nella forwarding table.

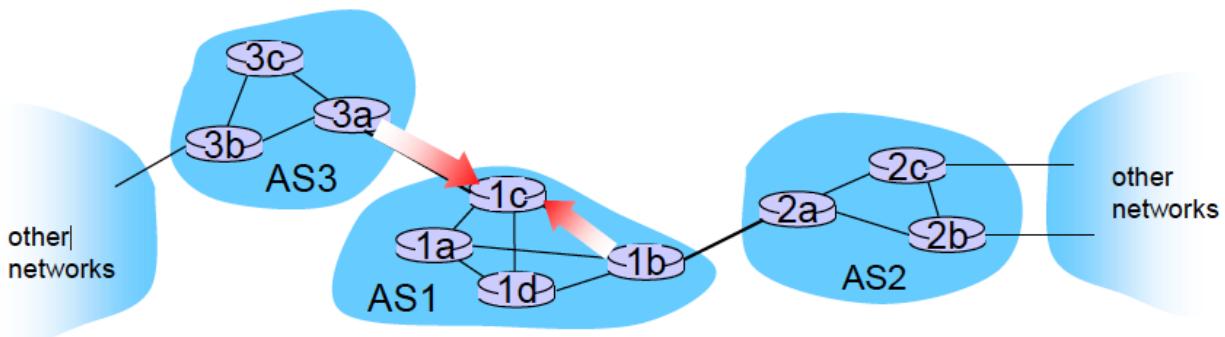
(138.16.64/22, port 4)



HOT POTATO ROUTING: supponiamo che ci siano due o più “best inter-routes”. Sceglio il percorso con il NEXT-HOP più vicino.

- Usiamo OSPF per determinare quale gateway è il più vicino.

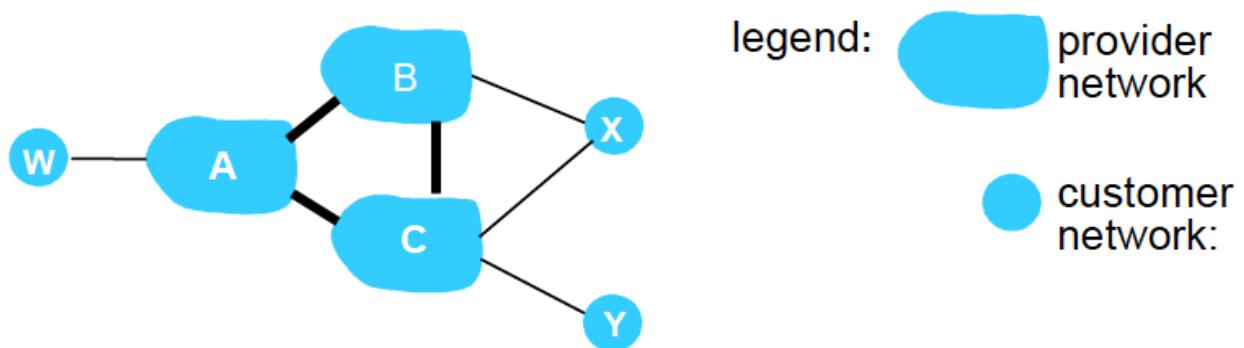
Considerato un router x in AS1, se ci sono più AS-PATH per raggiungere un prefisso esterno ad AS1, il router x sceglie l'AS-PATH che include il router di gateway più vicino a x.



Quindi, riassumendo:

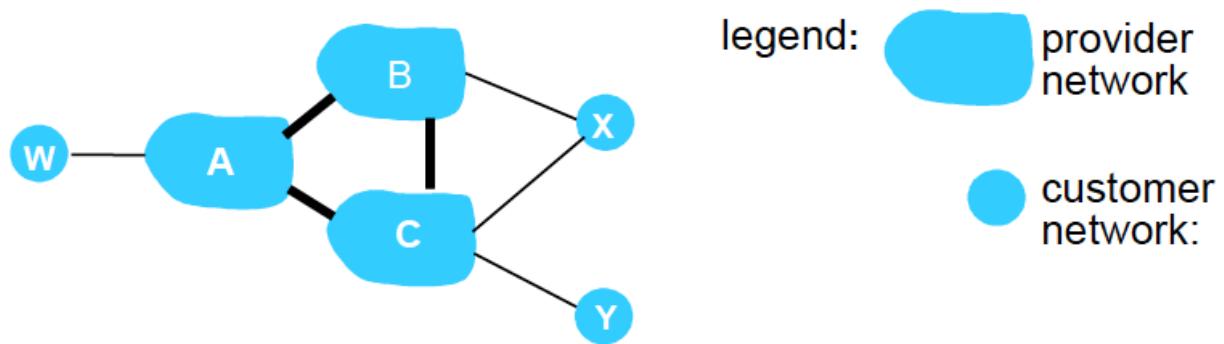
1. Il router viene a conoscenza del prefisso tramite gli advertisement BGP da altri router.
2. Determina la porta di output per il prefisso.
 - a. Usa la BGP route selection per trovare il percorso inter-AS migliore.
 - b. Usa OSPF per trovare il miglior percorso intra-AS che porta al miglior percorso inter-AS.
 - c. Il router identifica la porta per questo percorso migliore.
3. Inserisce l'entrata per la porta del prefisso nella forwarding table.

ESEMPIO DI BGP ROUTING POLICY



- A, B, C sono **provider network**.
- X, W, Y sono customer dei provider network.
- X è **dual-homed**: collegato a due network.
 - X non vuole il percorso da B, tramite X, fino a C.
 - Quindi X non pubblicherà a B un percorso per C.

ESEMPIO DI BGP ROUTING POLICY (2)



- A manda a B un advertisement sul percorso AW.
- B manda a X un advertisement sul percorso BAW.
- B dovrebbe mandare a C un advertisement sul percorso BAW?
 - No! B non ottiene alcun “ricavo” per il routing CBAW poiché né W né C sono clienti di B.
 - B vuole costringere C a instradare verso W passando per A.
 - B vuole instradare **soltamente** da/verso i suoi clienti.

12.6.3 PERCHE' ABBIAMO DIVERSI ROUTING INTRA-AS E INTER-AS?

POLICY:

- **Inter-AS:** l'admin vuole controllare come è instradato il traffico, chi instrada tramite la sua rete.
- **Intra-AS:** non è necessaria nessuna politica.

SCALE: il routing gerarchico consente di risparmiare dimensioni della tabella e riduce il traffico di aggiornamento.

PERFORMANCE:

- **Intra-AS:** si può focalizzare sulle performance.
- **Inter-AS:** la policy può predominare rispetto alla performance.