Spazi di Indirizzamento IP

Struttura, gerarchia e strumenti pratici

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

March 26, 2026



UniMe
1548

Che cos'è un indirizzo IP?

- Un indirizzo IPv4 è un numero intero a **32 bit**
- Scritto in *dotted-decimal*: 4 ottetti separati da punto
- Ogni ottetto vale da 0 a 255
- $2^{32} \approx 4.3$ miliardi di indirizzi distinti

Esempio

`192.168.1.47`

`11000000.10101000.00000001.00101111`

Perché 32 bit?

IPv4 nasce negli anni '70.
All'epoca 2^{32} indirizzi sembravano *infiniti*. Oggi sono **esauriti**.

Network part e Host part

Ogni indirizzo si divide in due parti:

Network part	Host part
Identifica la **rete**	Identifica il **nodo** nella rete
Uguale per tutti i nodi della stessa subnet	Unico all'interno della rete

Il confine tra le due parti è determinato dalla **subnet mask**.

Notazione CIDR — Classless Inter-Domain Routing (RFC 1519, 1993)

`192.168.1.47/24` $\Rightarrow$ i primi **24 bit** sono la network part

Subnet mask — come funziona

La subnet mask è una maschera binaria:

- **1** nei bit della network part
- **0** nei bit della host part

Per calcolare il **network address**:

addr `AND` mask

```
192.168.1.47     11000000.10101000.00000001.00101111
255.255.255.0    11111111.11111111.11111111.00000000
-----------------------------------------------------
192.168.1.0      11000000.10101000.00000001.00000000
```

Equivalenze CIDR comuni

Prefix	Mask	Host max
/8	255.0.0.0	∼16M
/16	255.255.0.0	∼65K
/24	255.255.255.0	254
/25	255.255.255.128	126
/30	255.255.255.252	2
/32	255.255.255.255	1 (singolo host)

I tre ruoli della subnet mask

La subnet mask **non nasce** per separare indirizzi pubblici da privati. Quella distinzione arriva solo nel 1996 con RFC 1918. La mask ha tre ruoli distinti, tutti indipendenti da pubblico/privato:

1 — Decisione locale: LAN diretta o gateway?

Ogni host, prima di inviare un pacchetto, calcola:

`dst AND mask == mio_ip AND mask` ?

Sì ⇒ consegna diretta in LAN (via ARP + Ethernet). No ⇒ manda al default gateway. Funziona uguale con tutti gli indirizzi pubblici o tutti privati.

2 — Subnetting amministrativo (su blocchi pubblici)

Chi riceveva una classe B poteva suddividerla autonomamente in sottoreti interne senza tornare dall'autorità centrale. Es.: `150.146.0.0/16` → `150.146.1.0/24`, `150.146.2.0/24` ... Tutti indirizzi **pubblici**, organizzati gerarchicamente.

Consegna diretta e indiretta

Prima di inviare ogni pacchetto, ogni host si pone una domanda:

La destinazione è nella mia stessa rete, oppure no?

Consegna diretta (direct delivery)

La destinazione è nella stessa subnet. Il test con la mask dà lo stesso risultato per entrambi gli indirizzi:

```
192.168.1.47 AND mask  = 192.168.1.0
192.168.1.5 AND mask   = 192.168.1.0
```

Risultati **uguali** $\Rightarrow$ stessa rete.
Il mittente risolve il MAC della destinazione via **ARP** e consegna direttamente su Ethernet. Nessun router coinvolto.

Consegna indiretta (indirect delivery)

La destinazione è fuori subnet. Il test AND dà risultati diversi.

Il pacchetto viene inviato al **default gateway**. Il gateway consulta la sua tabella di routing e decide il passo successivo — diretto o indiretto a sua volta.

Subnetting amministrativo — esempio concreto

Siamo nel 1990. L'Università di Messina riceve dalla IANA il blocco di classe B `150.146.0.0/16` — 65.534 indirizzi pubblici, tutti suoi.

L'università ha però dipartimenti fisicamente separati. L'amministratore di rete **suddivide autonomamente** il blocco in subnet `/24`, senza chiedere nulla a nessuno:

Subnet (pubblica)	Assegnata a
`150.146.1.0/24`	Dip. Informatica
`150.146.2.0/24`	Dip. Fisica
`150.146.3.0/24`	Dip. Medicina
`150.146.4.0/24`	Biblioteca
`150.146.5.0/24`	Amministrazione
	altri 250 slot liberi

Tutti pubblici

`150.146.1.47` (un PC di Informatica) è raggiungibile direttamente da qualsiasi punto di Internet, esattamente come `150.146.2.15` (un PC di Fisica). Nessun NAT, nessuna traduzione.

Il router interno

`150.146.1.0/24`	`eth0` → Informatica
`150.146.3.0/24`	`eth2` → Medicina
`default`	`eth3` → Internet

Per instradare verso `150.146.2.33`: AND con la maschera, match su `eth1`, consegnato.

Verifica oggi: `whois 150.146.1.1`

Router di backbone e route aggregation

I **router di backbone** sono i grandi router che formano la "spina dorsale" di Internet — gestiti da operatori come Level3, NTT, Telia. Ogni router di backbone deve sapere come raggiungere **qualsiasi** rete del mondo.

Il problema prima del CIDR

Con le classi rigide ogni rete assegnata generava una voce separata nella tabella di routing globale. Negli anni '80 le tabelle crescevano senza controllo — centinaia di migliaia di rotte, impossibili da gestire in RAM.

CIDR (1993) — route aggregation

Senza CIDR, un ISP con 256 reti /24 pubblica 256 rotte:

`200.10.0.0/24`, `200.10.1.0/24`, ...

Con CIDR, pubblica **una sola rotta** aggregata:

`200.10.0.0/16`

Il backbone vede 1 voce invece di 256.

Gerarchia di visibilità

Livello	Cosa vede
Host nella LAN	subnet /24
Router università	tutte le /24 interne
Router backbone	solo `150.146.0.0/16`
IANA	solo i blocchi dei RIR

Indirizzi speciali nella subnet

Data una subnet `192.168.1.0/24`:

Indirizzo	Valore	Significato
Network address	`192.168.1.0`	Identifica la rete, non assegnabile
Primo host	`192.168.1.1`	Primo indirizzo usabile
Ultimo host	`192.168.1.254`	Ultimo indirizzo usabile
Broadcast	`192.168.1.255`	Raggiunge tutti i nodi, non assegnabile

Verifica rapida da terminale

```
ipcalc 192.168.1.47/24
```

Classi A, B, C — storia

Prima del CIDR (fino al 1993) gli indirizzi erano divisi in **classi rigide**:

Classe	Primo ottetto	Subnet mask	Reti / Host per rete
A	1 – 126	/8	128 reti, ~16M host
B	128 – 191	/16	16K reti, ~65K host
C	192 – 223	/24	2M reti, 254 host
D	224 – 239	—	Multicast
E	240 – 255	—	Riservato/sperimentale

Problema: spreco enorme

Un'azienda con 300 host *doveva* richiedere una rete di classe B (65534 host disponibili) — sprecando oltre 65000 indirizzi. Il CIDR risolve questo con le subnet mask variabili (**VLSM**).

Perché si "richiedeva" una rete?

Nel modello originale, **tutti gli indirizzi IPv4 erano pubblici**. Non esistevano indirizzi privati — ogni dispositivo connesso a Internet aveva un indirizzo globalmente unico.

Esisteva un'autorità centrale (prima SRI, poi IANA) che teneva un **registro manuale** di tutti gli indirizzi assegnati nel mondo. Per connettersi a Internet bisognava fare domanda e ricevere un blocco.

Con le classi rigide, le opzioni erano solo tre:

- Hai $>$ 65534 host? Classe A
- Hai $>$ 254 host? Classe B
- Altrimenti? Classe C

Non esisteva via di mezzo.

Lo spreco reale — ancora oggi visibile

Alcuni blocchi di classe A assegnati negli anni '80:

`17.0.0.0/8`	Apple Inc.
`18.0.0.0/8`	MIT
`19.0.0.0/8`	Ford Motor Co.
`48.0.0.0/8`	Prudential Sec.

Ognuno di questi blocchi contiene **16 milioni** di indirizzi. Molti sono rimasti inutilizzati per decenni.

Verifica: `whois 17.0.0.1`

Internet era tutta pubblica — il modello end-to-end

Il progetto originale di Internet (ARPANET, anni '70) assumeva che ogni dispositivo avesse un indirizzo **globalmente unico e direttamente raggiungibile**.

Gli indirizzi **RFC 1918 non esistevano**. Vengono inventati solo nel **1996** — vent'anni dopo la nascita di IP — con l'unico scopo di rallentare l'esaurimento degli indirizzi pubblici.

Il principio end-to-end

Qualsiasi host può parlare direttamente con qualsiasi altro host. La rete è "stupida" e trasparente — l'intelligenza sta agli estremi.

Il NAT rompe questo modello

Con il NAT il tuo PC ha `192.168.1.47` — un indirizzo che esiste in milioni di altre reti contemporaneamente.

Non sei direttamente raggiungibile dall'esterno. Il router deve tenere una tabella di stato per smistare i pacchetti in arrivo.

Questo crea tutti i problemi noti: NAT traversal, port forwarding, difficoltà con VoIP e P2P, impossibilità di fare un server senza configurazione esplicita.

Indirizzi privati — RFC 1918

Tre blocchi sono riservati per reti **private** (non instradabili su Internet):

Blocco	Range	Prefix	Uso tipico
10.0.0.0/8	10.0.0.0 – 10.255.255.255	/8	Reti aziendali grandi
172.16.0.0/12	172.16.0.0 – 172.31.255.255	/12	Reti medie
192.168.0.0/16	192.168.0.0 – 192.168.255.255	/16	LAN domestiche / lab

Perché esistono?

Nascono dal problema dell'esaurimento IPv4. Milioni di dispositivi usano gli *stessi* indirizzi privati senza conflitti — perché non sono visibili fuori dalla loro rete locale.

Conseguenza: NAT

Per uscire su Internet, i pacchetti con IP privato vengono *tradotti* dal router in un IP pubblico. Questo è il **Network Address Translation (NAT)**.

Altri indirizzi speciali importanti

Indirizzo / Blocco	Tipo	Uso
`127.0.0.1` (o `127.0.0.0/8`)	Loopback	Traffico interno al proprio host
`0.0.0.0`	Unspecified	Qualsiasi interfaccia locale (`INADDR_ANY`)
`169.254.0.0/16`	Link-local	Auto-assegnazione (APIPA) quando manca DH
`224.0.0.0/4`	Multicast	Gruppi di destinazione multipla
`255.255.255.255`	Broadcast	Tutti i nodi della rete locale

Connessione con i socket BSD

`INADDR_ANY` corrisponde esattamente a `0.0.0.0`: quando fai `bind()` su `INADDR_ANY`, il kernel accetta connessioni su **tutte** le interfacce disponibili.

Scope degli indirizzi IP

Lo **scope** definisce il dominio di validità di un indirizzo: fino a dove un pacchetto con quella destinazione può essere inoltrato.

Host scope

Valido solo sul singolo host

`127.0.0.1`
`::1`

Il kernel consegna internamente. Non esce **mai** dalla macchina, nemmeno verso la scheda di rete.

Link scope

Valido solo sul segmento fisico diretto

`169.254.0.0/16`
`fe80::/10`

I router **scartano** questi pacchetti — non li inoltrano mai. In IPv6 è obbligatorio e strutturale.

Site scope

Valido dentro una organizzazione

`10.0.0.0/8`
`172.16.0.0/12`
`192.168.0.0/16`

Instradabile internamente, filtrato sui router di bordo verso Internet.

Global scope

Valido ovunque su Internet

IP pubblici IPv4
`2000::/3` (IPv6)

Nessun router lo scarta per ragioni di scope. Instradabile globalmente.

```
ip addr show
# inet   127.0.0.1/8          scope host     lo
# inet   192.168.1.47/24      scope global   eth0
# inet6  fe80::1c2b:3c4d:5e6f/64  scope link  eth0
```

Gerarchia di assegnazione — chi assegna gli IP?

IANA

Internet Assigned Numbers Authority *(ICANN)*

Assegna blocchi enormi ai RIR

$\Rightarrow$

RIR

Regional Internet Registries

5 nel mondo **RIPE NCC** per Europa/Asia-Ovest

$\Rightarrow$

ISP / LIR

Local Internet Registries

Operatori telefonici, cloud provider, ecc.

$\Rightarrow$

Utente

Aziende, università, utenti finali

I cinque RIR nel mondo

RIR	Area geografica	Sito
ARIN	Nord America	`arin.net`
RIPE NCC	Europa, Medio Oriente, Asia Ovest	`ripe.net`
APNIC	Asia-Pacifico	`apnic.net`
LACNIC	America Latina e Caraibi	`lacnic.net`
AFRINIC	Africa	`afrinic.net`

Per noi: RIPE NCC

Università di Messina ha prefissi IP assegnati attraverso RIPE NCC. Puoi cercare qualsiasi prefisso su `https://apps.db.ripe.net` oppure con `whois`.

Esaurimento IPv4 e nascita del NAT

Cronologia dell'esaurimento:

- **2011**: IANA assegna gli ultimi blocchi ai RIR
- **2012**: APNIC (Asia-Pacifico) — primo RIR esaurito
- **2015**: ARIN (Nord America)
- **2019**: RIPE NCC — Europa esaurita

Oggi gli indirizzi IPv4 si comprano e vendono sul mercato secondario a circa \$40–50 per indirizzo.

Il NAT come "soluzione"

Con il NAT, un singolo IP pubblico può servire centinaia di dispositivi privati. Questo ha *rallentato* la transizione a IPv6.

Il problema fondamentale del NAT

Il modello TCP/IP originale è **end-to-end**: ogni host ha un indirizzo globalmente unico. Il NAT rompe questo modello — un host interno non è direttamente raggiungibile dall'esterno.

NAT — quando nasce e come funziona

RFC 1631 — maggio 1994
È il documento che formalizza il NAT. La motivazione esplicita: rallentare l'esaurimento IPv4 *"mentre soluzioni più complesse vengono elaborate"*. Quella soluzione "temporanea" è ancora in uso trent'anni dopo.

Due varianti

Basic NAT — traduzione 1:1 di indirizzi IP. Serve tanti IP pubblici quanti sono i dispositivi interni attivi. Usato in ambito enterprise per motivi di sicurezza.

NAPT (Network Address *Port* Translation) — traduzione molti:1. Un solo IP pubblico per tutti i dispositivi interni, differenziati dalla porta sorgente. È quello che abbiamo a casa.

Come funziona NAPT

In uscita	PC `192.168.1.5:4321` → Router traduce in `1.2.3.4:9000`
In entrata	Pacchetto arriva su `1.2.3.4:9000` Router consulta tabella di stato Gira a `192.168.1.5:4321`

Sequenza storica

1993 — RFC 1519: CIDR
1994 — RFC 1631: NAT
1995 — RFC 1883: IPv6
1996 — RFC 1918: indirizzi privati

Basic NAT — traduzione 1:1

Il Basic NAT traduce **un IP privato in un IP pubblico dedicato**. Per ogni host interno attivo serve un indirizzo pubblico distinto.

Tabella di traduzione (solo IP)

IP privato		IP pubblico
192.168.1.5	↔	203.0.113.1
192.168.1.6	↔	203.0.113.2
192.168.1.7	↔	203.0.113.3

Il router sostituisce **solo l'indirizzo IP** nell'header. Non guarda porte né protocollo di trasporto.

Flusso in uscita

Prima (LAN)	Dopo (Internet)
src: 192.168.1.5	src: 203.0.113.1
dst: 8.8.8.8	dst: 8.8.8.8
porta src: 4321	porta src: 4321
porta dst: 80	porta dst: 80

Solo l'IP sorgente viene riscritto. Le porte rimangono **invariate**.

Basic NAT — implicazioni

Non risparmia indirizzi pubblici

Il Basic NAT consuma tanti IP pubblici quanti sono gli host interni attivi — **nessun risparmio** rispetto all'assegnazione diretta di IP pubblici.

Il risparmio di indirizzi è una caratteristica **esclusiva di NAPT**, non del Basic NAT.

Uso tipico

Ambito enterprise: server interni esposti con IP pubblici dedicati, separazione amministrativa tra reti, o ambienti che richiedono mappatura stabile e

Connessioni dall'esterno: possibili

La mappatura è **1:1 e statica** — un host esterno può raggiungere `192.168.1.5` inviando pacchetti a `203.0.113.1`.

Il router sa sempre a chi consegnare: non c'è ambiguità, non serve port forwarding.

Questo è il principale vantaggio del Basic NAT rispetto a NAPT per i server.

NAPT — traduzione molti:1

Il NAPT traduce **molti IP privati in un solo IP pubblico**, usando la **porta sorgente** come discriminante. È quello che hai a casa e che usa il tuo ISP.

Tabella di stato (IP + porta)

IP:porta privata		IP:porta pubblica	Destinazione
`192.168.1.5:4321`	→	`203.0.113.1:9001`	`8.8.8.8:80`
`192.168.1.5:4322`	→	`203.0.113.1:9002`	`8.8.8.8:80`
`192.168.1.6:5500`	→	`203.0.113.1:9003`	`8.8.8.8:443`
`192.168.1.7:6600`	→	`203.0.113.1:9004`	`1.1.1.1:53`

Tutti e quattro i flussi escono dallo **stesso IP pubblico** `203.0.113.1`. La porta pubblica è assegnata dal router al momento della connessione.

NAPT — implicazioni

Risposta in entrata

Arriva pacchetto su `203.0.113.1:9003`

↓ router consulta tabella di stato

Consegna a `192.168.1.6:5500`

Funziona solo se la connessione è stata **iniziata dall'interno**.

Problema: connessioni dall'esterno

Se nessun host interno ha *iniziato* una connessione, non c'è voce in tabella.

Un pacchetto in arrivo su `203.0.113.1` viene **scartato** — il router non sa a chi consegnarlo.

ICMP senza porte

`ping` non ha porte TCP/UDP. Il NAPT usa il campo **ICMP Query ID** come discriminante al posto della porta — per questo `ping` funziona anche dietro NAT.

Soluzione: port forwarding

Voce **statica** nella tabella NAPT:

`203.0.113.1:80` → `192.168.1.5:80`

Permette di esporre un server interno su una porta specifica dell'IP pubblico.

IPv6 — la risposta a lungo termine

- Indirizzi a **128 bit**
- $2^{128} \approx 3.4 \times 10^{38}$ indirizzi
- Notazione esadecimale con :
  es. `2001:db8::1`
- Nessun bisogno di NAT
- Meccanismi di autoconfigurazione (SLAAC)

Prefissi speciali IPv6

`::1/128`	Loopback
`fe80::/10`	Link-local
`fc00::/7`	Unique local (privato)
`2000::/3`	Unicast globale (pubblico)

Adozione

Circa il 40–45% del traffico globale è già IPv6 (2024). Su Linux: `ip -6 addr`

NAPT — cosa succede passo per passo

Il PC `192.168.1.5` apre una connessione TCP verso `8.8.8.8:80`. Il kernel sceglie la porta effimera `4321`.

1 — Pacchetto in uscita (LAN → router)

IP src:	`192.168.1.5`
IP dst:	`8.8.8.8`
TCP src:	`4321`
TCP dst:	`80`

2 — Router riscrive e salva in tabella

IP src:	`203.0.113.1` ← riscritto
IP dst:	`8.8.8.8`
TCP src:	`9001` ← riscritto
TCP dst:	`80`

Ricalcola anche il **checksum TCP** (che copre IP src,

3 — Risposta del server (Internet → router)

Il server vede solo `203.0.113.1` — non sa che esiste `192.168.1.5`.

IP src:	`8.8.8.8`
IP dst:	`203.0.113.1`
TCP src:	`80`
TCP dst:	`9001`

4 — Router ripristina e consegna

Consulta tabella: `203.0.113.1:9001` → `192.168.1.5:4321`

IP dst:	`192.168.1.5` ← ripristinato
TCP dst:	`4321` ← ripristinato

Il PC riceve il pacchetto sulla porta 4321 — esattamente quella attesa. **TCP non si accorge**

Mostra interfacce e indirizzi

```
ip addr show
ip addr show eth0
```

Mostra la tabella di routing

```
ip route show
# Output tipico:
# default via 192.168.1.1 dev eth0
# 192.168.1.0/24 dev eth0
```

Come leggere l'output di `ip addr`:

- `inet 192.168.1.47/24` → indirizzo IPv4 con prefix
- `brd 192.168.1.255` → broadcast address
- `scope global` → indirizzo globalmente usabile
- `scope link` → solo link-local

ping — test di raggiungibilità

```
ping -c 4 8.8.8.8
ping -c 4 google.com
# Invia pacchetti ICMP Echo Request
# Misura RTT (round-trip time)
```

traceroute — percorso dei pacchetti

```
traceroute google.com
# Mostra ogni router (hop)
    attraversato
# Usa TTL incrementale
```

Connessione con i raw socket

`ping` usa **ICMP** — esattamente il protocollo che abbiamo gestito con `AF_INET / SOCK_RAW / IPPROTO_ICMP` nelle prossime esercitazioni.

Ogni hop di traceroute è un router che decrementa il TTL. Quando TTL = 0, il router risponde con `ICMP Time Exceeded`.

whois e curl — chi possiede un IP?

Trova il proprietario di un IP pubblico

```
whois 8.8.8.8
# Risponde il RIR di competenza
# Mostra: organizzazione, paese,
#         ASN, range assegnato
```

Scopri il tuo IP pubblico

```
curl ifconfig.me
curl -s https://api.ipify.org
```

Output tipico di whois 8.8.8.8

OrgName:	Google LLC
Country:	US
NetRange:	8.8.8.0 – 8.8.8.255
CIDR:	8.8.8.0/24
OrgTechRef:	ARIN

Il tuo IP pubblico è l'IP del **router NAT** della tua rete, non del tuo PC — a meno che tu non sia direttamente connesso senza NAT.

ipcalc — calcolatore di subnet

Analisi di una subnet

```
ipcalc 192.168.1.47/24
```

Output

```
Address:   192.168.1.47
Netmask:   255.255.255.0 = 24
Network:   192.168.1.0/24
HostMin:   192.168.1.1
HostMax:   192.168.1.254
Broadcast:  192.168.1.255
Hosts/Net:  254
```

Subnetting manuale

```
# Dividi 192.168.1.0/24
# in 4 sottoreti uguali:
ipcalc 192.168.1.0/24 /26
# /26 = 64 indirizzi, 62 host
```

Host discovery sulla LAN

```
nmap -sn 192.168.1.0/24
# Invia ping a ogni IP della
    subnet
# Elenca gli host attivi
```

Risoluzione DNS con dig

```
dig google.com
dig google.com A       # solo IPv4
dig google.com AAAA    # solo IPv6
dig +short google.com  # solo IP
```

Reverse lookup (IP → nome)

```
dig -x 8.8.8.8
# Risposta: dns.google.
```

DNS e i socket BSD

Nella programmazione di rete usi `getaddrinfo()` — che internamente interroga il DNS — per tradurre un hostname in un `struct sockaddr_in` pronto per `connect()`. Lo vedremo subito dopo.

`nslookup` è il predecessore di `dig`, disponibile su tutti i sistemi (incluso Windows). Output meno leggibile ma utile per debug rapido.

Come gli indirizzi IP vivono nel codice C

La struttura `struct sockaddr_in` mappa direttamente il modello IP:

Definizione (POSIX)

```
1 struct sockaddr_in {
2     sa_family_t sin_family; /*
          AF_INET */
3     in_port_t   sin_port;   /*
          porta (network order) */
4     struct in_addr sin_addr;/*
          indirizzo IPv4 */
5 };
6
7 struct in_addr {
8     uint32_t s_addr; /* 32 bit,
          network order */
9 };
```

Campi chiave

- `sin_family`: sempre `AF_INET` per IPv4
- `sin_port`: in **network byte order** (`htons`)
- `sin_addr.s_addr`: IP in **network byte order**

Ricorda

I 4 byte dell'IP e i 2 byte della porta viaggiano sempre in **big-endian** sul cavo (network byte order).

Conversione tra testo e binario

Testo → binario: `inet_pton`

```c
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port   = htons(8080);

/* "presentation to network" */
inet_pton(AF_INET,
          "192.168.1.47",
          &addr.sin_addr);
```

Binario → testo: `inet_ntop`

```c
char ip_str[INET_ADDRSTRLEN];

/* "network to presentation"
   */
inet_ntop(AF_INET,
          &addr.sin_addr,
          ip_str,
          sizeof(ip_str));
printf("IP: %s\n", ip_str);
```

Non usare `inet_ntoa()` — è deprecata

`inet_ntoa()` usa un buffer interno statico: sovrascrive il risultato alla chiamata successiva, ed è **non thread-safe**. Usa sempre `inet_ntop()`.

Leggere gli indirizzi delle proprie interfacce: `getifaddrs()`

Lista indirizzi IP locali in C

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

struct ifaddrs *ifa_list, *ifa;
getifaddrs(&ifa_list);

for (ifa = ifa_list; ifa; ifa = ifa->ifa_next) {
    if (!ifa->ifa_addr) continue;
    if (ifa->ifa_addr->sa_family != AF_INET)
        continue;

    struct sockaddr_in *sa =
        (struct sockaddr_in *)ifa->ifa_addr;
    char buf[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &sa->sin_addr,
              buf, sizeof(buf));
    printf("%-12s %s\n", ifa->ifa_name, buf);
}
freeifaddrs(ifa_list);
```

Output atteso:

- `lo` `127.0.0.1`
- `eth0` `192.168.1.47`
- `wlan0` `10.0.0.5`

Collegamento con `ip addr`

`getifaddrs()` fa esattamente quello che fa `ip addr show`: interroga il kernel per la lista delle interfacce e dei loro indirizzi. È la base per costruire tool di rete diagnostici.

Risolvere un hostname: `getaddrinfo()`

DNS lookup in C

```c
#include <netdb.h>

struct addrinfo hints = {0};
struct addrinfo *res, *p;
hints.ai_family   = AF_INET;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("google.com", "80",
            &hints, &res);

for (p = res; p; p = p->ai_next) {
    struct sockaddr_in *sa =
        (struct sockaddr_in *)p->ai_addr;
    char buf[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &sa->sin_addr,
              buf, sizeof(buf));
    printf("google.com -> %s\n", buf);
}
freeaddrinfo(res);
```

Perché `getaddrinfo()` è moderna

- Supporta sia IPv4 che IPv6
- Restituisce una **lista** di indirizzi (un hostname può avere più IP)
- Thread-safe
- Rimpiazza `gethostbyname()` (obsoleta)

Riepilogo — mappa concettuale

Concetto	Tool da terminale	API C corrispondente
IP e interfacce locali	`ip addr`	`getifaddrs()`
Calcolo subnet / mask	`ipcalc`	operatori bitwise su `uint32_t`
Proprietario di un IP	`whois`	— (query WHOIS via TCP)
Risoluzione DNS	`dig`, `nslookup`	`getaddrinfo()`
Raggiungibilità	`ping`	socket ICMP (`IPPROTO_ICMP`)
Percorso di rete	`traceroute`	socket raw + TTL
IP pubblico	`curl ifconfig.me`	—