



NoiStudiamo

DISPENSA DI LABORATORIO

# Reti di Calcolatori Laboratorio

Configurazione di reti con Linux e iptables, programmazione dei socket in C e in Python, con esercizi svolti

---

A CURA DI

**Damiano La Pusata**

Ottobre 2026

# Indice dei capitoli

I capitoli della dispensa, dalla configurazione delle reti alla programmazione dei socket.

|                                                                 |    |
|-----------------------------------------------------------------|----|
| 1. Reti, nodi e dispositivi . . . . .                           | 3  |
| 2. Lo stack di rete . . . . .                                   | 5  |
| 3. Indirizzi fisici e logici . . . . .                          | 8  |
| 4. Routing: come viaggiano i messaggi . . . . .                 | 10 |
| 5. Virtualizzazione e laboratorio virtuale . . . . .            | 12 |
| 6. I comandi di rete in Linux . . . . .                         | 14 |
| 7. Configurazione persistente e accesso remoto . . . . .        | 18 |
| 8. Esercizio svolto: la rete cittadina . . . . .                | 20 |
| 9. Il problema dell'ISP e il NAT . . . . .                      | 24 |
| 10. iptables . . . . .                                          | 25 |
| 11. Programmazione di rete ed endianness . . . . .              | 29 |
| 12. I socket BSD . . . . .                                      | 32 |
| 13. Il primo client e il primo server TCP . . . . .             | 35 |
| 14. Messaggistica TCP con i thread . . . . .                    | 39 |
| 15. Protocolli testuali e messaggi strutturati . . . . .        | 45 |
| 16. Socket UDP e messaggistica a datagrammi . . . . .           | 47 |
| 17. Comunicazione fra linguaggi diversi . . . . .               | 53 |
| 18. Esercizi proposti . . . . .                                 | 59 |
| 19. Esercizio svolto: sensori e nodo centrale . . . . .         | 61 |
| 20. Esercizio svolto: rete di messaggistica ad anello . . . . . | 67 |
| 21. Esercizio svolto: chat con registrazione e login . . . . .  | 77 |
| 22. Esercizio svolto: chat peer to peer . . . . .               | 83 |
| 23. Compilazione, prove ed errori frequenti . . . . .           | 90 |

# Reti, nodi e dispositivi

Il capitolo definisce che cos'è una rete, chi sono i nodi che la compongono e quali dispositivi permettono di collegarli: hub, bridge, switch e router, con la distinzione fondamentale fra switching e routing.

## 1.1 Che cos'è una rete

Una **rete** è un insieme di due o più dispositivi collegati fra loro allo scopo di **condividere risorse**. I dispositivi possono essere computer, smartphone, stampanti di rete o televisori; il collegamento può avvenire via cavo, via onde radio, via cavo coassiale o via luce, cioè attraverso un **mezzo** (*medium*); le risorse condivise sono, in ultima analisi, **informazione**: file, pagine web, brani musicali, transazioni.

La rete più semplice è composta da un **mittente** e da un **destinatario**, che da qui in avanti chiameremo **nodi**. Il mittente invia un **messaggio** sotto forma di **segnale**, e il segnale attraversa il mezzo. Perché il messaggio venga compreso, i dispositivi devono seguire un insieme di regole condivise, cioè un **protocollo**, che stabilisce la struttura del messaggio e il modo di scambiarlo. Poiché ogni dispositivo può essere a turno mittente e destinatario, occorre anche definire una **topologia**, cioè il modo in cui i nodi sono collegati.

### DEFINIZIONE

Un **nodo** (o **host**) è qualunque dispositivo in grado di inviare e ricevere traffico di rete: computer, smartphone, server, dispositivi IoT. Per comunicare con gli altri nodi deve usare il loro stesso protocollo, così come due persone devono parlare la stessa lingua per capirsi.

## 1.2 Client e server

I nodi si dividono tipicamente in due categorie. I **client** avviano le richieste, i **server** rispondono. Un server non è altro che un computer su cui gira un software che risponde a richieste specifiche, e il ruolo di client o di server è **relativo a una specifica comunicazione**: un web server che, per rispondere a un browser, interroga un database server agisce come client nei confronti di quest'ultimo, mentre il database diventa la risorsa del web server.

## 1.3 Come collegare i nodi

Il modo più banale di creare una rete è collegare due nodi con un cavo, inserito nella scheda di rete di ciascuno. La scheda di rete si chiama **NIC** (*Network Interface Controller*) e il punto in cui si inserisce il cavo è una **porta di rete**. La soluzione funziona per due nodi, ma non scala: un cavo occupa una sola porta, e collegare direttamente ciascun nodo a tutti gli altri richiederebbe un numero di schede di rete che cresce con il numero dei nodi. Serve allora un dispositivo da porre al centro della rete, a cui collegare tutti gli host.

L'**hub** è il dispositivo più semplice: un **ripetitore multiporta**. Rigenera il segnale ricevuto su una porta e lo ritrasmette su tutte le altre, senza preoccuparsi del destinatario. Oggi non si usa più, per tre ragioni: ogni messaggio raggiunge tutti i nodi, con evidenti problemi di riservatezza; la banda è condivisa fra tutti i nodi collegati, perché uno solo alla volta può trasmettere senza collisioni; e il traffico inutile aumenta i disturbi e le collisioni.

Il **bridge** nasce per collegare due hub. Ha due porte e **impara quali host si trovano da quale lato**, così da inoltrare un messaggio soltanto verso il lato necessario e contenere il traffico.

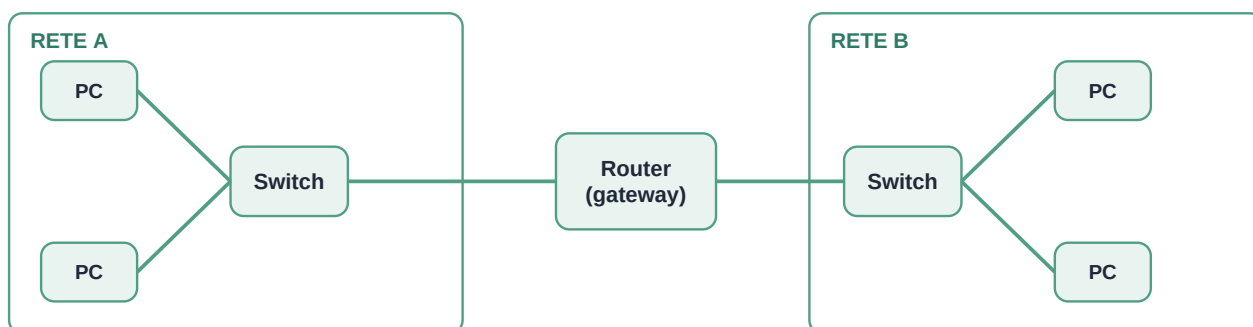
Lo **switch** è di fatto la fusione di hub e bridge: come l'hub ha molte porte, come il bridge impara quale nodo è collegato a quale porta. Un messaggio non viene duplicato su tutte le porte, ma inviato **solo sulla porta del**

**destinatario**; ogni collegamento dispone così di tutta la banda. Lo switch è il dispositivo che **facilita la comunicazione all'interno di una rete**.

## 1.4 Come collegare le reti

Una **rete**, in senso logico, è un raggruppamento di nodi che hanno esigenze di connettività simili e usano lo stesso insieme di protocolli, collegati fra loro, per esempio, tramite uno switch. Resta da capire come **uscire** dalla propria rete e raggiungerne un'altra.

Il **router** è il dispositivo il cui scopo principale è **facilitare la comunicazione fra reti diverse**. I router sono anche punti di controllo del traffico, dove applicare politiche di sicurezza, filtrare o reindirizzare i pacchetti; imparano a quali reti sono collegati e, dal punto di vista dei nodi, fungono da **gateway** verso le altre reti. I router creano la **gerarchia** delle reti: ogni rete usa il proprio gateway per raggiungere le altre, e Internet stessa non è altro che un enorme insieme di router interconnessi.



*Due reti locali, ciascuna organizzata attorno a uno switch, collegate da un router che fa da gateway*

## 1.5 Switching e routing

La distinzione fra i due dispositivi si riassume in due definizioni.

| FUNZIONE  | CHE COSA FA                                        | DISPOSITIVO |
|-----------|----------------------------------------------------|-------------|
| Switching | Sposta i dati <b>all'interno</b> della stessa rete | Switch      |
| Routing   | Sposta i dati <b>fra reti</b> diverse              | Router      |

Esistono molti altri dispositivi di rete, come access point, firewall, bilanciatori di carico e proxy, ma tutti svolgono, in ultima analisi, funzioni di switching, di routing o entrambe.

# Lo stack di rete

Per comunicare, due nodi devono condividere un modello. Il capitolo presenta il modello OSI, il meccanismo di incapsulamento con le unità di dati di ciascun livello, e la sua versione semplificata e realmente implementata, il modello TCP/IP.

## 2.1 Perché serve un modello

Quando parliamo diamo per scontato che l'interlocutore conosca la nostra lingua. Un computer non può fare supposizioni di questo tipo: le regole della comunicazione devono essere definite in anticipo e in modo preciso. Per ridurre la complessità, queste regole sono divise in **livelli**, ciascuno con una responsabilità precisa. Il modello di riferimento è il **modello OSI** (*Open Systems Interconnection*), definito dall'**ISO** (*International Organization for Standardization*).

L'idea alla base di OSI è favorire la concorrenza senza perdere la compatibilità, e porta con sé la distinzione fra **protocolli proprietari** e **protocolli aperti**. Un protocollo proprietario appartiene a un'azienda, che è la sola autorizzata a modificarlo e che ne concede l'uso dietro licenza. Un protocollo aperto è invece uno standard pubblico, che chiunque può consultare e implementare: i protocolli di Internet sono descritti nelle **RFC**, pubblicate liberamente dall'**IETF**.

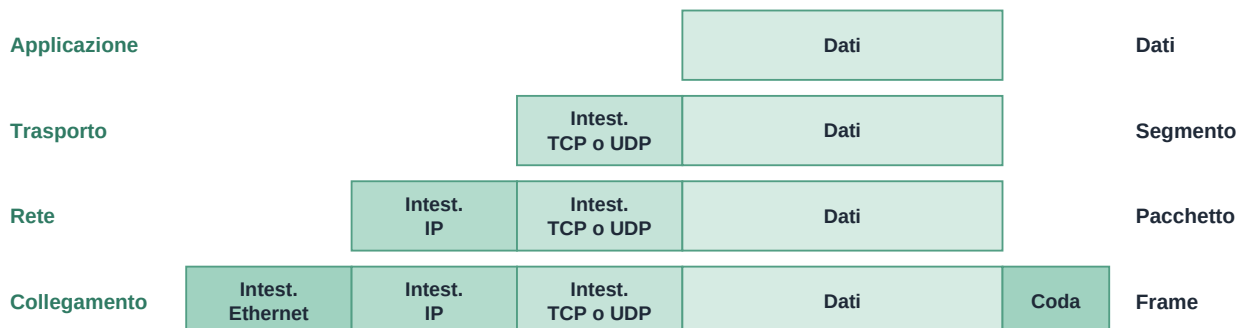
## 2.2 I sette livelli del modello OSI

| LIVELLO | NOME          | RESPONSABILITÀ                                                                                                                      |
|---------|---------------|-------------------------------------------------------------------------------------------------------------------------------------|
| 7       | Applicazione  | Dove lavorano gli sviluppatori: interfaccia fra l'applicazione e la rete. Un browser, per esempio, usa il protocollo HTTP.          |
| 6       | Presentazione | Codifica e decodifica dei dati (per esempio la codifica dei caratteri), cifratura come in TLS e compressione.                       |
| 5       | Sessione      | Gestione del dialogo fra le parti, per mantenere attive le connessioni.                                                             |
| 4       | Trasporto     | Consegna i dati all'applicazione o al servizio giusto, identificato dalla porta; con TCP garantisce consegna affidabile e ordinata. |
| 3       | Rete          | Instrada i pacchetti lungo il percorso fino alla destinazione, identificando mittente e destinatario con indirizzi logici.          |
| 2       | Collegamento  | Collega due nodi adiacenti; si divide nei sottolivelli LLC ( <i>Logical Link Control</i> ) e MAC ( <i>Media Access Control</i> ).   |
| 1       | Fisico        | Trasmissione dei bit sul mezzo: segnali elettrici, ottici o radio.                                                                  |

Un **servizio**, di cui si parla al livello di trasporto, è un'applicazione progettata per girare in background in attesa di richieste; in Linux i servizi si chiamano **demoni**. Su una stessa macchina possono esserne in ascolto molti, ed è compito del livello di trasporto consegnare ciascun messaggio a quello giusto.

## 2.3 Incapsulamento e PDU

Quando i nodi comunicano, i livelli di uno stesso nodo dialogano con quello superiore e con quello inferiore. Scendere lungo lo stack si chiama **incapsulamento**, risalirlo **decapsulamento**. Ogni livello produce una propria unità di dati, la **PDU** (*Protocol Data Unit*), formata dai dati ricevuti dal livello superiore più un'intestazione (*header*) con le informazioni di quel livello.



*L'incapsulamento: ogni livello aggiunge la propria intestazione alla PDU del livello superiore*

| LIVELLO      | PDU              | INDIRIZZAMENTO                                   | ESEMPIO           |
|--------------|------------------|--------------------------------------------------|-------------------|
| Applicazione | Dati o messaggio | Formato del contenuto (estensione, magic number) | .PNG, .PDF        |
| Trasporto    | Segmento         | Porte di origine e destinazione                  | 80, 443, 8080     |
| Rete         | Pacchetto        | Indirizzi IP di origine e destinazione           | 192.168.1.10      |
| Collegamento | Frame            | Indirizzi MAC di origine e destinazione          | 31:31:ab:ab:e6:e6 |

I livelli dal settimo al quinto possono essere raggruppati in uno solo: il loro unico compito è produrre i dati da consegnare al trasporto. Questi dati sono una sequenza binaria, il cui formato si riconosce dall'estensione del file o dal **magic number**, cioè dai primi byte che ne identificano il tipo.

Il **livello di trasporto** identifica l'applicazione che effettua la richiesta e il servizio che la riceve tramite i **numeri di porta**, e suddivide i dati in **segmenti**. La segmentazione serve a rispettare la dimensione massima dei segmenti, a rendere efficiente la ritrasmissione delle sole parti perse e a permettere più comunicazioni contemporanee sulla stessa connessione di rete (**multiplexing**). Il protocollo scelto determina il comportamento: **TCP** sacrifica tempo in cambio di affidabilità e ordine, **UDP** ha un overhead minimo ma nessuna garanzia. La porta di destinazione è definita dal servizio (80 per un web server), mentre quella di origine viene scelta dal sistema operativo del client.

### DA RICORDARE

I numeri di porta sono interi a 16 bit, quindi vanno **da 0 a 65 535**. Le porte **da 0 a 1023** (*well known*) sono riservate ai servizi standard e, su Linux, solo l'amministratore può mettersi in ascolto su di esse; le porte da 1024 in su sono liberamente utilizzabili dai programmi.

Il **livello di rete** trasforma i segmenti in **pacchetti**, la cui intestazione contiene gli **indirizzi IP** di origine e di destinazione. IP non è l'unico protocollo di rete mai esistito (AppleTalk e IPX ne sono esempi storici), ma è di gran lunga il più diffuso perché aperto e gratuito.

Il **livello di collegamento** prende i dati logici e li lega al mondo fisico: la sua PDU è il **frame**, composto da un'intestazione e da una coda (*trailer*). È un livello che consegna **da un salto al successivo** (*hop to hop*). La traduzione dal virtuale al reale dipende dalla scheda di rete, che usa protocolli standard come Ethernet o WiFi. Il sottolivello MAC usa gli **indirizzi MAC**, cioè indirizzi fisici a 48 bit, per identificare le connessioni fisiche. Il

riconoscimento dell'inizio di un frame nel flusso di bit è affidato a sequenze di sincronizzazione (in Ethernet il preambolo e il delimitatore di inizio frame), mentre il trailer contiene il codice di controllo con cui il ricevitore rileva gli errori.

## 2.4 Il modello TCP/IP

Il modello OSI è soprattutto una guida concettuale. Il modello realmente implementato è il più semplice **modello TCP/IP**, che prende il nome dai suoi due protocolli principali e conta quattro livelli.

| MODELLO TCP/IP | LIVELLI OSI CORRISPONDENTI            | CHE COSA GESTISCE                                                                                |
|----------------|---------------------------------------|--------------------------------------------------------------------------------------------------|
| Applicazione   | Applicazione, presentazione, sessione | Logica dell'applicazione, comprensibilità dei dati, sicurezza: responsabilità degli sviluppatori |
| Trasporto      | Trasporto                             | Consegna al processo giusto tramite le porte                                                     |
| Internetwork   | Rete                                  | Instradamento fra reti interconnesse                                                             |
| Network Access | Collegamento e fisico                 | I driver traducono le informazioni fra mondo logico e mondo reale                                |

### DA RICORDARE

La divisione in livelli ha una conseguenza pratica preziosa: i **problemi di connettività** riguardano i livelli bassi (indirizzi, rotte, interfacce) e non dipendono dai bug del programma, mentre i **bug** del software applicativo non dipendono dalla connettività. Quando qualcosa non funziona, conviene prima verificare che i nodi si raggiungano (per esempio con ping) e solo dopo cercare l'errore nel codice.

# Indirizzi fisici e logici

Prima di capire come un messaggio raggiunge la destinazione occorre distinguere l'indirizzo fisico, che dice dove si trova un nodo, dall'indirizzo logico, che dice chi è. Il capitolo segue passo per passo l'invio di un messaggio, fino alla scoperta del protocollo ARP.

## 3.1 Un'analogia: l'aula universitaria

Immaginiamo un'aula di un dipartimento, piena di sedie disposte in file. Per consegnare un biglietto a uno studente dobbiamo poterlo identificare. Un primo modo è indicare il suo **posto**: la stanza (per esempio ROOM 24), la fila (R1, R2 e così via) e il numero della sedia. La sequenza ROOM 24, fila 1, sedia 4 identifica in modo univoco un posto: è il suo **indirizzo fisico**, e le regole con cui lo costruiamo sono, a tutti gli effetti, un protocollo.

Un secondo modo è usare il **nome** dello studente: è il suo **indirizzo logico**, assegnato secondo un'altra regola. Anche qui può nascere un problema: nella stessa aula possono esserci due studenti con lo stesso nome, così come nella stessa rete non devono esserci due dispositivi con lo stesso indirizzo logico, pena l'impossibilità di stabilire chi debba ricevere il messaggio.

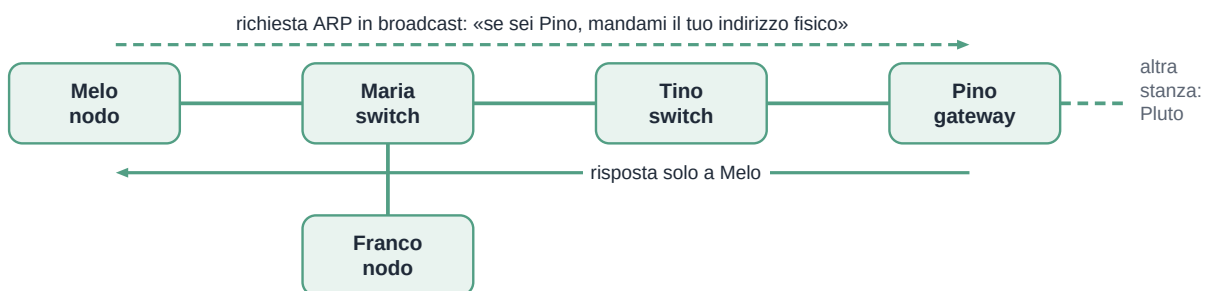
Nelle reti reali gli **indirizzi logici** sono gli **indirizzi IP** e gli **indirizzi fisici** sono gli **indirizzi MAC**.

## 3.2 Topologia fisica e topologia logica

La **topologia fisica** descrive ciò che è reale: dove passano i cavi, quali interfacce vengono usate, dove si trovano fisicamente i dispositivi. La **topologia logica** descrive invece il **comportamento**, cioè come i dati fluiscono fra i nodi, indipendentemente da come i cavi sono stesi.

## 3.3 L'invio di un messaggio, passo per passo

Riprendiamo l'aula e diamo un nome ai protagonisti: **Melo** vuole inviare il messaggio «Ciao» a **Pluto**, che si trova in un'altra stanza. Fra i due ci sono **Maria** e **Tino**, che fanno da switch, **Franco**, un altro nodo della stanza di Melo, e **Pino**, il gateway della stanza, cioè l'unico punto collegato a un'altra rete.



*La stanza di Melo: due switch in cascata, un altro nodo e il gateway Pino*

Melo deve **incapsulare** i dati scendendo lungo lo stack.

1. **Trasporto.** Melo crea un segmento che contiene «Ciao»; nell'intestazione mette la porta di destinazione, decisa dal servizio di Pluto (per esempio 80), e una porta di origine scelta a caso (per esempio 46254).

2. **Rete.** Melo crea un pacchetto che contiene il segmento; nell'intestazione mette gli **indirizzi logici**: origine Melo, destinazione Pluto.
3. **Collegamento.** Melo deve creare un frame con gli **indirizzi fisici**, e qui sorge la domanda decisiva: **la destinazione si trova nella mia stessa rete?**

Melo confronta gli indirizzi: Pluto è in un'altra stanza, quindi il frame non va spedito a Pluto ma al **gateway** Pino, l'unico che può farlo uscire. L'indirizzo fisico di origine è quello di Melo (ROOM 24, R1, 1); quello di destinazione dovrebbe essere quello di Pino, che però Melo **non conosce**.

### 3.4 Il protocollo ARP

Per scoprirlo, Melo «grida» a tutti i nodi della stanza una richiesta speciale: *se sei Pino, mandami il tuo indirizzo fisico*. Per raggiungere tutti usa un indirizzo fisico di destinazione particolare, l'indirizzo di **broadcast**.

La richiesta arriva a Maria, che è uno switch. Maria non sa dove si trovi Pino e quindi **replica la richiesta su tutte le altre porte**; nel frattempo **impara** che Melo si trova sulla sua porta di sinistra e lo annota nella propria rubrica fisica. Franco riceve la richiesta, ma la scarta perché non è Pino. Tino, anch'esso switch, si comporta come Maria: replica la richiesta e impara da che parte si trova Melo.

Pino riconosce il proprio nome e prepara una **risposta** indirizzata a Melo. Questa volta il percorso è diverso: Tino non replica la risposta su tutte le porte, ma la inoltra **solo verso Maria**, perché ha già imparato che Melo sta da quella parte; nello stesso momento impara che Pino si trova sulla destra. Maria fa lo stesso, e la risposta raggiunge Melo.

Melo ora conosce l'indirizzo fisico di Pino, lo memorizza associandolo al suo indirizzo logico e può finalmente spedire il frame con il messaggio originale. Questo procedimento si chiama **ARP** (*Address Resolution Protocol*).

| TABELLA   | CHI LA MANTIENE | CHE COSA ASSOCIA                    | ESEMPIO                  |
|-----------|-----------------|-------------------------------------|--------------------------|
| MAC table | Switch          | Porta e indirizzo fisico            | Sinistra: ROOM 24, R1, 1 |
| ARP table | Nodi e router   | Indirizzo logico e indirizzo fisico | Pino: ROOM 24, R1, 4     |

#### DEFINIZIONE

L'**ARP** (*Address Resolution Protocol*) permette a un nodo di scoprire l'indirizzo fisico (MAC) corrispondente a un indirizzo logico (IP) della propria rete: la richiesta viaggia in **broadcast** e la risposta arriva **solo** al richiedente, che la memorizza nella propria **ARP table**.

Poiché lavorano soltanto con indirizzi fisici, gli switch sono dispositivi di **livello 2**.

# Routing: come viaggiano i messaggi

Il capitolo riassume i comportamenti di nodi, switch e router, applica il ragionamento a reti reali con indirizzi IP e MAC, e presenta la tabella di routing con i tre modi per popolarla: rotte direttamente connesse, statiche e dinamiche.

## 4.1 Che cosa fanno nodi, switch e router

Pino ha ricevuto il messaggio, ma come lo consegnerà a un'altra rete? Un router conosce le reti a cui è collegato direttamente e, per le altre, sa a quale router vicino affidare il pacchetto: o perché qualcuno glielo ha detto (**rotta statica**), o perché i router si sono scambiati le informazioni (**rotta dinamica**). Se non esiste alcuna strada verso la destinazione, il pacchetto viene scartato. Queste informazioni stanno nella **tabella di routing**. Quando il messaggio arriva al router della rete di Pluto, che non ne conosce l'indirizzo fisico, comincia un nuovo procedimento ARP.

| DISPOSITIVO | COMPORTAMENTO                                                                                                                                                                                                                                                                                                                                                                      |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Nodo        | Crea messaggi per incapsulamento e li riceve per decapsulamento. Comunica con i nodi della stessa rete tramite indirizzi fisici e, se non conosce quello di un vicino, lo scopre con ARP e lo salva nella ARP table. Per sapere se una destinazione è nella propria rete confronta l'indirizzo con la <b>netmask</b> ; se non lo è, invia il messaggio al proprio <b>gateway</b> . |
| Switch      | Riceve frame e li inoltra sull'interfaccia giusta. Se la destinazione è nota la usa; se è ignota o broadcast duplica il frame sulle altre interfacce. Impara la posizione dei mittenti nella MAC table. È trasparente: può avere indirizzi propri, ma solo per essere gestito, non per il routing. Lavora al livello 2.                                                            |
| Router      | È un nodo che, se riceve un messaggio non destinato a lui, prova a recapitarlo prima di scartarlo. Decapsula il frame per leggere l'indirizzo IP, conosce i vicini tramite ARP e sa dove inoltrare grazie alla tabella di routing, popolata per connessione diretta, a mano o con protocolli come RIP. Ha un indirizzo IP e un MAC per ogni interfaccia. Lavora al livello 3.      |

## 4.2 Un esempio con indirizzi reali

Consideriamo quattro nodi collegati a uno switch, nella rete 10.1.1.0/24. Il nodo 10.1.1.10, con MAC a1a1, vuole scrivere a 10.1.1.20 ma non ne conosce il MAC. Invia allora una richiesta ARP in broadcast. Lo switch, che lavora al livello 2, considera tutto ciò che sta sopra il livello 2 come semplici dati; impara che a1a1 è collegato alla sua porta 1 e, poiché la destinazione è il broadcast, replica il frame su tutte le altre porte.

Gli altri nodi decapsulano il frame, vedono che l'IP richiesto non è il loro e lo ignorano; il destinatario giusto risponde. Nel frattempo **tutti** i nodi hanno appreso che l'IP 10.1.1.10 corrisponde al MAC a1a1 e hanno aggiornato la propria ARP table.

Tutto ciò funziona perché i due nodi stanno nella stessa rete. Per capirlo i nodi usano la **netmask**: se la destinazione è in un'altra rete, il messaggio va inviato al gateway. Per questo i router hanno, **per ogni interfaccia**, un indirizzo IP, una netmask e un indirizzo MAC.

## 4.3 La tabella di routing

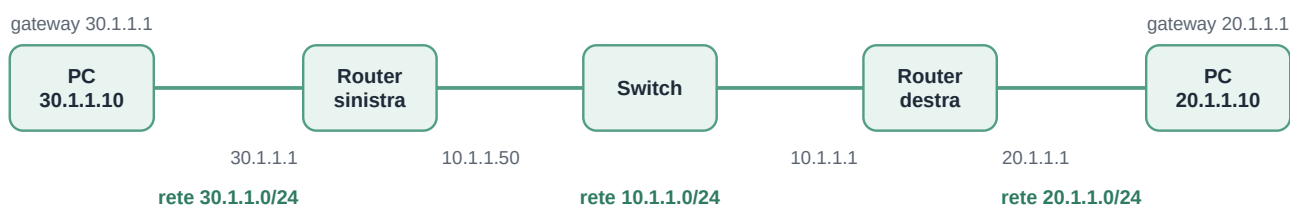
Immaginiamo due reti, 10.1.1.0/24 e 20.1.1.0/24, collegate da un solo router: servono due interfacce, una per ciascuna rete. Il router mantiene una mappa di tutte le reti che conosce, la **tabella di routing**, composta da **rotte**. Una rotta indica come raggiungere una rete (indirizzo di rete e netmask) attraverso una certa interfaccia o un certo router successivo, così come la MAC table associa un indirizzo MAC a una porta.

### DEFINIZIONE

Una **rotta** è un'istruzione che indica come raggiungere una specifica rete (indirizzo IP di rete più netmask) attraverso una specifica interfaccia o un router successivo (*next hop*).

Le tabelle di routing si popolano in tre modi.

- **Direttamente connesse** (*Directly Connected*, DC): esiste automaticamente una rotta per ogni rete a cui il router è collegato con una propria interfaccia.
- **Statiche** (*Static Route*): inserite a mano dall'amministratore; anziché un'interfaccia indicano l'indirizzo IP del router attraverso cui passare.
- **Dinamiche** (*Dynamic Route*): i router si scambiano le tabelle tramite protocolli di routing dinamico come RIP, OSPF e BGP.



Due router collegati da uno switch: ciascuno conosce le proprie reti per connessione diretta e raggiunge l'altra con una rotta statica

Nell'esempio il router di sinistra conosce direttamente le reti 30.1.1.0/24 e 10.1.1.0/24; per raggiungere 20.1.1.0/24 deve passare dal router di destra, il cui indirizzo sulla rete comune è 10.1.1.1. Simmetricamente, il router di destra raggiunge 30.1.1.0/24 attraverso 10.1.1.50. Senza la seconda rotta i pacchetti arriverebbero a destinazione, ma le risposte non saprebbero tornare indietro.

| ROUTER   | TIPO    | RETE        | INTERFACCIA O NEXT HOP |
|----------|---------|-------------|------------------------|
| Sinistra | DC      | 30.1.1.0/24 | Porta 1                |
| Sinistra | DC      | 10.1.1.0/24 | Porta 2                |
| Sinistra | Statica | 20.1.1.0/24 | via 10.1.1.1           |
| Destra   | DC      | 10.1.1.0/24 | Porta 1                |
| Destra   | DC      | 20.1.1.0/24 | Porta 2                |
| Destra   | Statica | 30.1.1.0/24 | via 10.1.1.50          |

### DA RICORDARE

Le rotte servono **in entrambe le direzioni**: per ogni rotta che porta i pacchetti verso una rete remota, deve esserne una che riporti le risposte indietro. È l'errore più frequente nella configurazione di una rete.

# Virtualizzazione e laboratorio virtuale

Per costruire reti senza disporre di decine di computer si ricorre alla virtualizzazione. Il capitolo spiega che cos'è un hypervisor, come creare una macchina virtuale con VirtualBox e Alpine Linux e come configurarne le schede di rete.

## 5.1 Virtualizzazione e containerizzazione

Per sperimentare le reti viste finora servirebbero molti computer e dispositivi di rete. La soluzione è la **virtualizzazione**, cioè l'emulazione di hardware tramite altro hardware o software. A differenza della **containerizzazione**, in cui i container condividono il kernel del sistema ospite e usano le risorse solo quando servono, una macchina virtuale **occupa effettivamente le risorse che le vengono assegnate** ed esegue un proprio kernel: la RAM data a una macchina virtuale non è più disponibile per il sistema ospite.

Il software che rende possibile tutto ciò è l'**hypervisor**, che mette a disposizione degli ambienti virtuali le risorse della macchina fisica (l'**host**).

| TIPO                         | DOVE SI INSTALLA                                           | CARATTERISTICHE                                | ESEMPIO    |
|------------------------------|------------------------------------------------------------|------------------------------------------------|------------|
| Tipo 1 ( <i>bare metal</i> ) | Direttamente sull'hardware, al posto del sistema operativo | Prestazioni e sicurezza migliori               | KVM        |
| Tipo 2 ( <i>hosted</i> )     | Sopra un sistema operativo già installato                  | Più semplice da usare su un computer personale | VirtualBox |

Una **macchina virtuale** (VM) è un computer realizzato in software: è isolata dalle altre, può avere hardware e sistema operativo diversi da quelli dell'host e può essere spostata su un altro hypervisor.

## 5.2 VirtualBox e Alpine Linux

VirtualBox è una buona scelta, ma va bene qualunque hypervisor. Si scarica dal sito ufficiale e si installa seguendo le istruzioni; su alcuni computer occorre prima abilitare la virtualizzazione dal BIOS. Dalla sezione *Tools* si può creare una rete **Host only**, cioè un'interfaccia di rete virtuale sull'host che permette di comunicare con le VM, ed eventualmente simulare un server DHCP che assegni loro gli indirizzi.

Quando si parla di reti e di server si parla quasi sempre di sistemi Linux o Unix. Le principali famiglie di distribuzioni sono Arch, Red Hat (RHEL) e Debian, e la famiglia determina come il sistema gestisce i dispositivi e soprattutto la rete, che dipende dal kernel, dal software di gestione (per esempio il servizio NetworkManager) e dai file di configurazione. In laboratorio si usano **Alpine Linux**, molto leggera, oppure Debian: dal sito di Alpine si scarica l'immagine ISO della *Standard Edition*, per la stessa architettura dell'host.

## 5.3 Creare una macchina virtuale

La creazione richiede pochi passi.

1. Si assegnano un **nome** alla VM, l'**immagine ISO** scaricata e la versione di Linux.
2. Nella scheda *Hardware* si scelgono le risorse: per Alpine bastano circa **512 MB di RAM** e una **CPU** (la RAM si può ridurre, per tentativi).
3. Nella scheda *Hard Disk* si sceglie la dimensione del disco virtuale: per Alpine è sufficiente circa **1 GB**.
4. Prima di accendere la macchina si configurano le **schede di rete**.

## 5.4 I tipi di scheda di rete

Dalle impostazioni della VM si gestiscono le schede di rete. Dall'interfaccia grafica se ne possono virtualizzare al massimo quattro, ma da riga di comando se ne possono aggiungere altre; si possono anche cambiare gli indirizzi MAC.

| TIPO DI SCHEDA    | USO                                                                                                                                 |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| Internal Network  | Emula il <b>cavo</b> fra due nodi. Il campo nome identifica il cavo: due schede con lo stesso nome sono collegate allo stesso cavo. |
| Host only Adapter | Collega la VM a una scheda virtuale dell'host, così che dall'host si possa raggiungere la VM.                                       |
| NAT               | Simula una scheda dietro un NAT, come un computer di casa: dà alla VM l'accesso a Internet.                                         |

### DA RICORDARE

Nel laboratorio virtuale ogni **cavo** è una scheda di tipo *Internal Network* con un nome, e ogni **porta** di un dispositivo è una scheda di rete della VM. Uno switch con quattro porte è quindi una VM con quattro schede.

# I comandi di rete in Linux

Con il comando `ip` si configurano indirizzi, interfacce, rotte e tabelle ARP. Il capitolo lo introduce attraverso tre esercizi di complessità crescente: due nodi collegati da un cavo, una rete con uno switch, due reti collegate da un router.

## 6.1 Il comando `ip`

Il comando più usato per gestire il livello IP è, ironicamente, **ip**: una suite di utilità che sostituisce i vecchi comandi `ifconfig` e `route` e permette di assegnare indirizzi, gestire interfacce, aggiungere rotte e molto altro. I comandi che modificano la configurazione richiedono i privilegi di **amministratore**: nella documentazione e nelle slide sono preceduti dal prompt `#` (root) oppure da `sudo`, mentre il prompt `$` indica un utente normale. Nei riquadri di codice che seguono il prompt è omissso.

Il comando **ip addr show** (abbreviato **ip a**) elenca tutte le interfacce. La prima è **lo**, l'interfaccia di **loopback**: un'interfaccia virtuale con cui il sistema comunica con sé stesso, per esempio fra processi della stessa macchina. Seguono le schede vere, come **eth0** (la Internal Network della VM) ed **eth1** (la Host only, disattivata); per ciascuna sono indicati lo stato, l'indirizzo MAC e gli eventuali indirizzi IP.

```
ip addr show          # tutte le interfacce
ip addr show dev eth0 # una sola interfaccia
ip link set eth0 down # disattiva la scheda (livello 2)
ip link set eth0 up   # attiva la scheda
ip addr add 192.168.56.2/24 dev eth0 # assegna indirizzo e netmask
ip addr del 192.168.56.2/24 dev eth0 # rimuove l'indirizzo
ping 192.168.56.3      # verifica la raggiungibilità
```

*Riepilogo dei comandi per indirizzi e interfacce*

## 6.2 Esercizio 1: due nodi collegati da un cavo

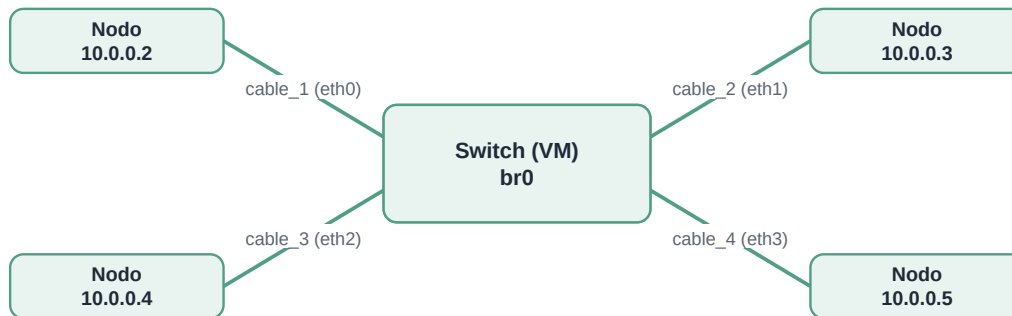
Il primo esercizio collega due nodi con un cavo (una Internal Network chiamata, per esempio, `cable_1`), disattivando per ora le schede NAT e Host only. Ogni nodo deve avere un indirizzo IP e una netmask uguale o compatibile.

```
# nodo A
ip addr add 192.168.56.2/24 dev eth0
# nodo B
ip addr add 192.168.56.3/24 dev eth0
# dal nodo A
ping 192.168.56.3
```

*Configurazione dei due nodi e verifica con ping*

## 6.3 Esercizio 2: una rete con uno switch

Per una rete con più nodi serve uno switch, che realizziamo con una VM dotata di quattro schede di tipo Internal Network, una per cavo. Ricordiamo che uno switch è la fusione di hub e bridge: in Linux lo si costruisce creando un'interfaccia di tipo **bridge** e rendendola «padrona» (*master*) di tutte le schede, che diventano così le sue porte. In VirtualBox le schede dello switch vanno impostate con la **modalità promiscua** su *Allow All*, così che accettino anche i frame destinati ad altri.



Esercizio 2: quattro nodi collegati a uno switch realizzato con una VM e un bridge

```
# sulla VM che fa da switch (schede eth0, eth1, eth2, eth3)
ip link set eth0 up           # attivo tutte le schede
ip link set eth1 up
ip link set eth2 up
ip link set eth3 up
ip link add br0 type bridge    # creo il bridge
ip link set eth0 master br0    # ogni scheda diventa una porta di br0
ip link set eth1 master br0
ip link set eth2 master br0
ip link set eth3 master br0
ip link set eth0 promisc on    # modalità promiscua (anche da riga di comando)
ip link set br0 up            # attivo il bridge
```

Configurazione dello switch

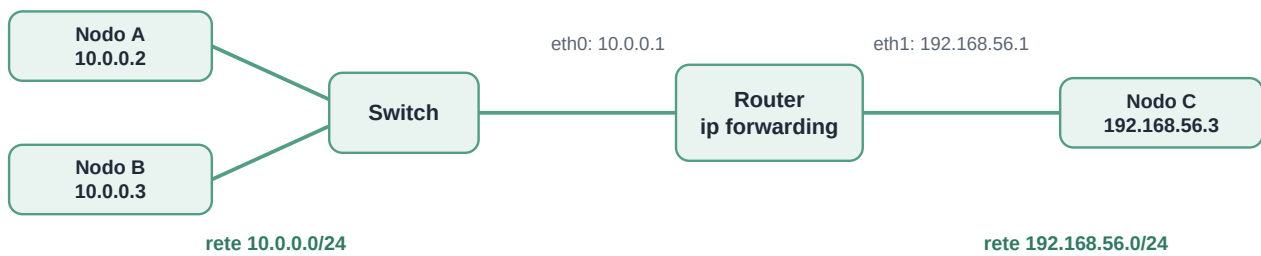
```
# nodi
ip addr add 10.0.0.2/24 dev eth0    # nodo A, cavo cable_1
ip addr add 10.0.0.3/24 dev eth0    # nodo B, cavo cable_2
# dal nodo A
ping 10.0.0.3
ip neigh                            # tabella ARP del nodo A
```

Configurazione dei nodi e verifica

Dopo il ping, la tabella ARP del nodo A contiene l'associazione fra l'indirizzo 10.0.0.3 e il MAC del nodo B. Il comando storico **arp** mostra la stessa tabella.

## 6.4 Esercizio 3: due reti collegate da un router

Aggiungiamo una seconda rete, 192.168.56.0/24, con il nodo C, e colleghiamo le due reti con un router: una VM con due schede, una per rete.



Esercizio 3: la rete 10.0.0.0/24 e la rete 192.168.56.0/24 collegate da un router

```

# nodo C
ip addr add 192.168.56.3/24 dev eth0
# router: un indirizzo per ogni rete
ip addr add 10.0.0.1/24 dev eth0
ip addr add 192.168.56.1/24 dev eth1
ip link set eth1 up
  
```

Indirizzi del nodo C e del router

A questo punto il router raggiunge entrambe le reti, ma i nodi delle due reti non si parlano ancora, per due motivi. Il primo è che un nodo Linux, per impostazione predefinita, **scarta** i pacchetti che non sono destinati a lui: per trasformarlo in router bisogna abilitare l'**inoltro IP** (*ip forwarding*). La sua tabella di routing, invece, è già pronta, perché contiene le due reti direttamente connesse.

```

echo 1 > /proc/sys/net/ipv4/ip_forward      # abilita l'inoltro (fino al riavvio)
# in modo permanente: aggiungere al file /etc/sysctl.conf la riga
#   net.ipv4.ip_forward = 1
# e ricaricare la configurazione
sysctl -p
  
```

Abilitazione dell'inoltro IP

Il secondo motivo è che i nodi non sanno di dover passare dal router: servono delle **rotte**. Si può aggiungere una rotta verso la rete remota oppure, più semplicemente, una **rotta predefinita** (*default route*) verso il router, che viene usata per ogni destinazione per cui non esiste una regola più specifica. Il router diventa così il **gateway** del nodo.

```

ip route show                                # tabella di routing
# rotta verso una rete
ip route add 192.168.56.0/24 via 10.0.0.1 dev eth0
# rotta verso un singolo host (maschera /32)
ip route add 192.168.56.3/32 via 10.0.0.1 dev eth0
# rotta predefinita: le due forme sono equivalenti
ip route add default via 10.0.0.1 dev eth0
ip route add 0.0.0.0/0 via 10.0.0.1 dev eth0
# rimozione di una rotta
ip route del 192.168.56.0/24 via 10.0.0.1 dev eth0
  
```

Gestione della tabella di routing

Nell'esercizio basta una rotta predefinita su ciascun nodo: su A e B verso 10.0.0.1, su C verso 192.168.56.1, cioè l'indirizzo del router nella rete del nodo. Dopodiché il ping fra le due reti funziona.

#### DA RICORDARE

La notazione **0.0.0.0/0** non è un indirizzo «non valido»: è la rete con netmask nulla, che contiene **tutti** gli indirizzi. Per questo coincide con la rotta predefinita, che si applica quando nessuna rotta più specifica corrisponde alla destinazione.

## 6.5 Nascondere un nodo al ping

Per nascondere nodi o router, si può disattivare la risposta alle richieste di ping. La modifica va nello stesso file usato per l'inoltro IP.

```
# riga da aggiungere a /etc/sysctl.conf
net.ipv4.icmp_echo_ignore_all = 1
# poi ricaricare la configurazione
sysctl -p
```

*Disattivazione della risposta al ping*

## 6.6 La tabella dei vicini: ip neigh

Il comando **ip neigh** gestisce la **tabella dei vicini**, che associa gli indirizzi IP agli indirizzi MAC dei dispositivi sullo stesso collegamento: per IPv4 è la tabella **ARP**, per IPv6 quella del **Neighbor Discovery**. La tabella MAC di uno switch realizzato con un bridge Linux, cioè l'associazione fra MAC e porte, si visualizza invece con il comando **bridge fdb show**.

```
ip neigh                                # mostra la tabella dei vicini
ip neigh flush all                      # la svuota
ip neigh add 10.0.0.3 lladdr 08:00:27:aa:bb:cc dev eth0
ip neigh del 10.0.0.3 dev eth0
ip neigh get 10.0.0.3 dev eth0
```

*Gestione della tabella ARP*

# Configurazione persistente e accesso remoto

I comandi visti finora valgono fino al riavvio. Il capitolo mostra come rendere permanente la configurazione con i file di sistema, come impostare DNS e nomi, e come collegarsi e copiare file su un nodo remoto con SSH e SCP.

## 7.1 Il file `/etc/network/interfaces`

Tutti i comandi introdotti in precedenza sono **temporanei**: al riavvio del nodo la configurazione va persa. Per mantenerla occorre scriverla nei file di configurazione di rete, che però cambiano da una famiglia di distribuzioni all'altra. Le distribuzioni della famiglia Debian, e Alpine, usano il file `/etc/network/interfaces`, che contiene la configurazione di tutte le interfacce. La configurazione del loopback va lasciata com'è; sotto si aggiungono le proprie.

```
# scheda configurata automaticamente con DHCP
auto eth0
iface eth0 inet dhcp

# scheda con indirizzo statico
auto eth1
iface eth1 inet static
    address 10.0.0.2
    netmask 255.255.255.0
    gateway 10.0.0.1
```

*Configurazione dinamica e statica di un'interfaccia*

La parola **auto** attiva l'interfaccia all'avvio; l'indirizzo può essere scritto anche nella forma `10.0.0.2/24`, omettendo la riga `netmask`. Nello stesso file si può definire anche un **bridge**: la modalità **manual** attiva l'interfaccia senza assegnarle un indirizzo.

```
auto br0
iface br0 inet manual
    bridge_ports eth0 eth1 eth2 eth3
    bridge_stp off
    bridge_fd 0
```

*Definizione di un bridge nel file delle interfacce*

Le rotte statiche si possono legare agli eventi di attivazione e disattivazione dell'interfaccia, con le direttive `up` e `down`, che eseguono un comando quando la scheda viene attivata o disattivata.

```
auto eth0
iface eth0 inet static
    address 10.0.0.1/24
    up ip route add 20.0.0.0/24 via 10.0.0.254
    down ip route del 20.0.0.0/24 via 10.0.0.254
```

*Rotte statiche legate all'interfaccia*

Dopo ogni modifica si riavvia il servizio di rete, oppure l'intera macchina.

```
rc-service networking restart    # Alpine (OpenRC)
service networking restart      # Debian
```

*Riavvio del servizio di rete*

## 7.2 DNS, nome della macchina e nomi locali

Per usare nomi di dominio come google.it occorre indicare almeno un server DNS nel file **/etc/resolv.conf**, una riga per server. Il nome leggibile della macchina (*hostname*) sta nel file **/etc/hostname**. Nel file **/etc/hosts**, infine, si possono associare nomi a indirizzi IP direttamente sulla macchina: queste associazioni hanno la precedenza sul DNS, e funzionano come un piccolo DNS locale.

```
# /etc/resolv.conf
nameserver 1.1.1.1
nameserver 8.8.8.8

# /etc/hosts: un indirizzo seguito da uno o più nomi
192.168.56.2  node_1  youtube.com
```

*Server DNS e nomi locali*

## 7.3 Accesso remoto e copia di file

Per lavorare su un nodo remoto si usa **SSH**: sul nodo deve essere attivo il server SSH (il demone sshd). Il nome utente è facoltativo, perché in sua assenza viene richiesto; per autenticarsi si può usare anche una chiave pubblica al posto della password. Per copiare file si usa **SCP**, che lavora sopra SSH, oppure un client grafico come FileZilla tramite SFTP.

```
ssh 192.168.1.2                # accesso con l'utente corrente
ssh fedyfausto@192.168.56.2    # accesso con un utente specifico
scp server_exec usernode@192.168.1.2:/home/usernode/  # copia verso il nodo remoto
```

*Accesso remoto e copia di un file*

# Esercizio svolto: la rete cittadina

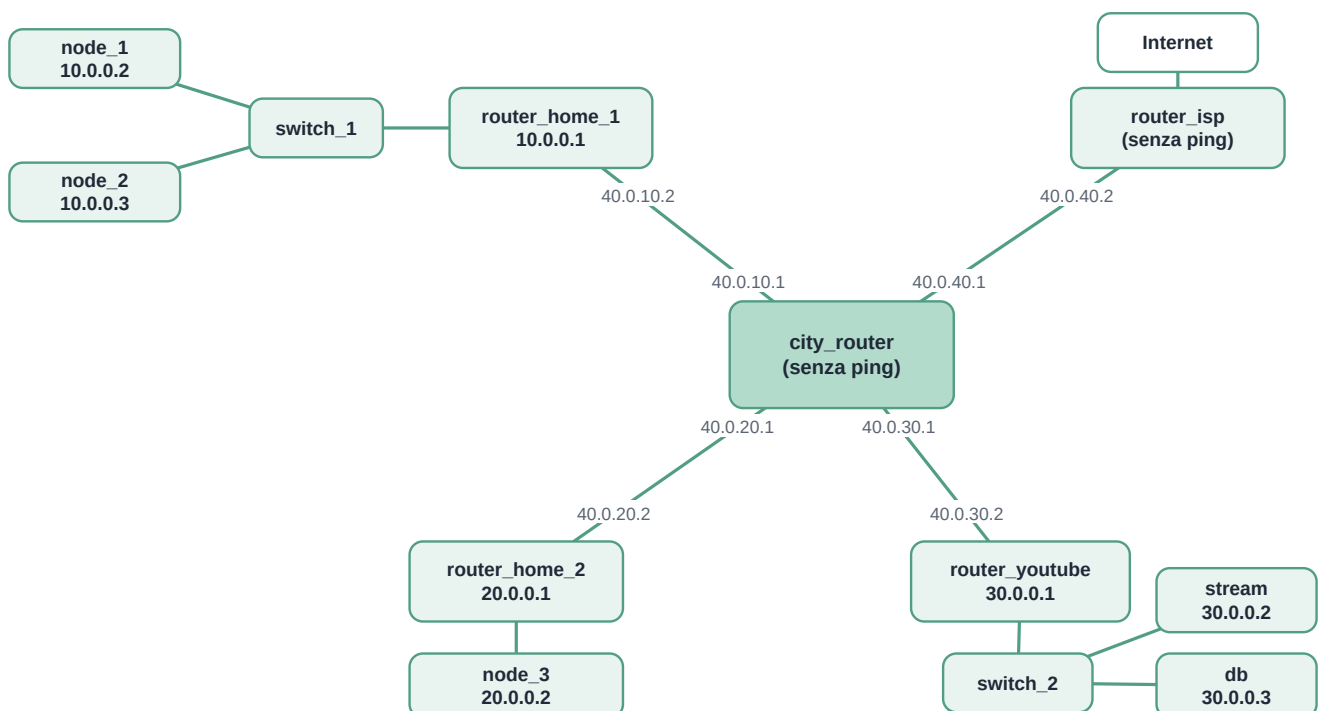
Il capitolo mette insieme tutto ciò che si è visto: due reti domestiche, la rete di un servizio di streaming e il router dell'ISP, collegati da un router cittadino. Per ogni nodo è riportata la configurazione completa.

## 8.1 La traccia

Si vuole progettare e configurare la seguente infrastruttura.

- Una **rete domestica** con due nodi, collegata alla rete cittadina.
- Nella rete cittadina c'è un'altra **rete domestica**, con un solo nodo.
- Attraverso un gateway, la rete cittadina è collegata a **Internet**, tramite il router dell'ISP.
- A un certo punto, alla rete cittadina si collega anche la rete di **YouTube**, con due nodi: il server di streaming e il database.

Ogni linea del disegno è un cavo diverso e ogni punto di attacco è una scheda di rete. Il router cittadino e quello dell'ISP **non devono rispondere al ping**, una volta verificato il funzionamento. L'accesso a Internet si simula con una scheda NAT sul router dell'ISP, e il sito di YouTube si può «falsificare» associandone il nome a un indirizzo interno nel file `/etc/hosts`. Conviene procedere per fasi, verificando ogni rete prima di passare alla successiva.



La rete cittadina: due reti domestiche, la rete di YouTube e il router dell'ISP, collegati dal router cittadino

| RETE                                 | INDIRIZZI                                              | GATEWAY                     |
|--------------------------------------|--------------------------------------------------------|-----------------------------|
| Casa 1                               | 10.0.0.0/24 (node_1 .2, node_2 .3)                     | router_home_1, 10.0.0.1     |
| Casa 2                               | 20.0.0.0/24 (node_3 .2)                                | router_home_2, 20.0.0.1     |
| YouTube                              | 30.0.0.0/24 (youtube_stream .2, youtube_db .3)         | router_youtube, 30.0.0.1    |
| Collegamenti con il router cittadino | 40.0.10.0/24, 40.0.20.0/24, 40.0.30.0/24, 40.0.40.0/24 | city_router, .1 su ciascuna |

## 8.2 La rete domestica 1

I due nodi hanno un indirizzo nella rete 10.0.0.0/24 e una rotta predefinita verso il proprio router; lo switch è una VM con un bridge; il router domestico ha un'interfaccia verso la casa e una verso la città, l'inoltro IP attivo e una rotta predefinita verso il router cittadino.

```
# node_1
ip addr add 10.0.0.2/24 dev eth0
ip route add 0.0.0.0/0 via 10.0.0.1 dev eth0

# node_2
ip addr add 10.0.0.3/24 dev eth0
ip route add 0.0.0.0/0 via 10.0.0.1 dev eth0

# switch_1
ip link set eth1 up
ip link set eth2 up
ip link add br0 type bridge
ip link set br0 up
ip link set eth0 master br0
ip link set eth1 master br0
ip link set eth2 master br0          # ora node_1 e node_2 si parlano

# router_home_1
ip link set eth1 up
ip addr add 10.0.0.1/24 dev eth0     # lato casa
ip addr add 40.0.10.2/24 dev eth1    # lato città
echo 1 > /proc/sys/net/ipv4/ip_forward
ip route add 0.0.0.0/0 via 40.0.10.1 dev eth1
```

*Configurazione della rete domestica 1*

## 8.3 La rete domestica 2 e la rete di YouTube

La seconda casa ha un solo nodo, collegato direttamente al router; la rete di YouTube ha la stessa struttura della prima casa.

```
# node_3
ip addr add 20.0.0.2/24 dev eth0
ip route add 0.0.0.0/0 via 20.0.0.1 dev eth0

# router_home_2
ip link set eth1 up
ip addr add 20.0.0.1/24 dev eth0
ip addr add 40.0.20.2/24 dev eth1
echo 1 > /proc/sys/net/ipv4/ip_forward
ip route add 0.0.0.0/0 via 40.0.20.1 dev eth1
```

*Configurazione della rete domestica 2*

```
# youtube_stream
ip addr add 30.0.0.2/24 dev eth0
ip route add 0.0.0.0/0 via 30.0.0.1 dev eth0

# youtube_db
ip addr add 30.0.0.3/24 dev eth0
ip route add 0.0.0.0/0 via 30.0.0.1 dev eth0

# switch_2: come switch_1
ip link set eth1 up
ip link set eth2 up
ip link add br0 type bridge
ip link set br0 up
ip link set eth0 master br0
ip link set eth1 master br0
ip link set eth2 master br0

# router_youtube
ip link set eth1 up
ip addr add 30.0.0.1/24 dev eth0
ip addr add 40.0.30.2/24 dev eth1
echo 1 > /proc/sys/net/ipv4/ip_forward
ip route add 0.0.0.0/0 via 40.0.30.1 dev eth1
```

*Configurazione della rete di YouTube*

## 8.4 Il router cittadino e il router dell'ISP

Il router cittadino è il centro della topologia: ha quattro interfacce, conosce le reti dei collegamenti per connessione diretta, raggiunge le tre reti interne con rotte statiche e manda tutto il resto verso l'ISP. Il router dell'ISP ha una scheda NAT verso Internet (eth0), dalla quale ottiene l'indirizzo via DHCP, e una verso la città; deve sapere come raggiungere le tre reti interne.

```
# city_router
ip link set eth1 up
ip link set eth2 up
ip link set eth3 up
ip addr add 40.0.10.1/24 dev eth0
ip addr add 40.0.20.1/24 dev eth1
ip addr add 40.0.30.1/24 dev eth2
ip addr add 40.0.40.1/24 dev eth3
ip route add 10.0.0.0/24 via 40.0.10.2 dev eth0
ip route add 20.0.0.0/24 via 40.0.20.2 dev eth1
```

```
ip route add 30.0.0.0/24 via 40.0.30.2 dev eth2
ip route add 0.0.0.0/0 via 40.0.40.2 dev eth3
echo 1 > /proc/sys/net/ipv4/ip_forward
# in /etc/sysctl.conf: net.ipv4.icmp_echo_ignore_all = 1    (niente ping)

# router_isp (eth0 è la scheda NAT di VirtualBox)
udhcpd -i eth0          # indirizzo e rotta predefinita via DHCP
ip link set eth1 up
ip addr add 40.0.40.2/24 dev eth1
ip route add 10.0.0.0/24 via 40.0.40.1 dev eth1
ip route add 20.0.0.0/24 via 40.0.40.1 dev eth1
ip route add 30.0.0.0/24 via 40.0.40.1 dev eth1
echo 1 > /proc/sys/net/ipv4/ip_forward
# i pacchetti diretti a Internet escono con l'indirizzo dell'ISP
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
# in /etc/sysctl.conf: net.ipv4.icmp_echo_ignore_all = 1    (niente ping)
```

*Configurazione del router cittadino e del router dell'ISP*

#### DA RICORDARE

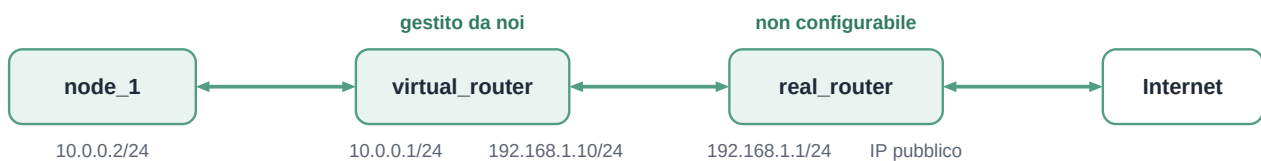
Il router dell'ISP raggiunge Internet attraverso il NAT di VirtualBox, che conosce soltanto l'indirizzo assegnato alla scheda eth0. Perché le risposte da Internet tornino ai nodi interni, il router dell'ISP deve a sua volta riscrivere l'indirizzo di origine dei pacchetti in uscita: è la regola **MASQUERADE** dell'ultima riga, spiegata nei due capitoli che seguono.

# Il problema dell'ISP e il NAT

Quando un router che non controlliamo non conosce le nostre reti, le risposte da Internet non possono tornare indietro. Il capitolo mostra perché l'inoltro IP non basta e perché serve la traduzione degli indirizzi (NAT).

## 9.1 Il problema

L'obiettivo è permettere a qualunque nodo di raggiungere Internet attraverso il router dell'ISP, o comunque attraverso un router che **non possiamo amministrare**. Senza aggiungere una rotta statica su quel router esterno, il traffico di ritorno non saprebbe come raggiungere la nostra rete.



*Il nodo raggiunge Internet attraverso il nostro router virtuale e il router reale, che non possiamo configurare*

Quando node\_1 invia un pacchetto verso Internet (per esempio a 1.1.1.1), l'indirizzo di origine è 10.0.0.2. Dentro la nostra rete tutto funziona, perché i nodi e il virtual\_router li gestiamo noi e hanno le rotte giuste. Ma quando il virtual\_router consegna il pacchetto al real\_router, l'origine resta 10.0.0.2: alla risposta, il real\_router non sa come raggiungere 10.0.0.2 e il pacchetto viene scartato.

Si potrebbe pensare: «abilito l'inoltro IP e imposto il virtual\_router come gateway di node\_1, non basta?». Basta **solo se** il real\_router sa come raggiungere la sottorete di node\_1, cioè se qualcuno vi ha aggiunto una rotta. L'inoltro IP permette al virtual\_router di agire da router, spostando i pacchetti fra le interfacce, ma **non riscrive le intestazioni**.

## 9.2 La soluzione: il NAT

Se il real\_router non è configurabile, il nostro virtual\_router deve eseguire una **traduzione degli indirizzi** (NAT, *Network Address Translation*): sostituisce l'indirizzo di origine dei pacchetti in uscita con il **proprio** indirizzo (192.168.1.10) e ricorda la sostituzione, così che quando arriva la risposta possa rimettere al suo posto l'indirizzo di node\_1 e inoltrargliela. Per il real\_router tutto il traffico proviene dal virtual\_router, che è nella sua rete e che sa raggiungere.

### DEFINIZIONE

Il **NAT** (*Network Address Translation*) riscrive gli indirizzi IP, e spesso le porte, dei pacchetti che attraversano un router, tenendo traccia delle traduzioni così da applicare la trasformazione inversa alle risposte. Il caso più comune, il **masquerading**, nasconde un'intera rete privata dietro l'unico indirizzo del router.

Per realizzare il NAT, e molti altri meccanismi, serve lo strumento **iptables**.

# iptables

iptables è lo strumento con cui si configura il firewall del kernel Linux. Il capitolo ne descrive tabelle, catene, regole e target, il percorso dei pacchetti attraverso le catene e i comandi per gli scenari più comuni, dal blocco di un indirizzo al NAT verso Internet.

## 10.1 Che cos'è iptables

**iptables** è il comando con cui si configura **netfilter**, il firewall del kernel Linux. Permette di definire regole concatenate che controllano il traffico in ingresso e in uscita, e gestisce il **filtraggio dei pacchetti** (consentire o bloccare), il **NAT** (riscrivere indirizzi e porte), l'inoltro di porte (*port forwarding*) e la modellazione del traffico (*traffic shaping*). Su Alpine si installa con il gestore di pacchetti apk.

```
apk add iptables
iptables --version
```

*Installazione su Alpine Linux*

Lo strumento si basa su quattro concetti.

- **Tabelle:** raggruppano regole con uno scopo simile; ogni tabella contiene più catene.
- **Catene** (*chains*): elenchi ordinati di regole, ciascuno legato a un punto del percorso del pacchetto. Quando un pacchetto arriva, attiva un aggancio (*hook*) di netfilter che corrisponde a una catena, e le regole vengono esaminate in ordine finché una corrisponde.
- **Regole:** condizioni che un pacchetto deve soddisfare (origine, destinazione, protocollo, porta...), seguite da un target.
- **Target:** la decisione da prendere sul pacchetto.

## 10.2 Tabelle, catene e target

| TABELLA         | SCOPO                                                                                                                                                      |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>filter</b>   | La tabella predefinita: le regole che decidono se un pacchetto può raggiungere la sua destinazione. È il firewall vero e proprio.                          |
| <b>nat</b>      | Regole che modificano indirizzi di origine e destinazione, per esempio per far uscire una rete privata o per raggiungere servizi che stanno dietro un NAT. |
| <b>mangle</b>   | Regole che alterano altri campi delle intestazioni.                                                                                                        |
| <b>raw</b>      | Permette ai pacchetti di saltare il tracciamento delle connessioni; si usa raramente.                                                                      |
| <b>security</b> | Controlli di accesso obbligatori (per esempio SELinux); presente solo in alcune distribuzioni.                                                             |

| CATENA      | SI APPLICA AI PACCHETTI...                                                       |
|-------------|----------------------------------------------------------------------------------|
| PREROUTING  | appena arrivati sull'interfaccia di rete, prima della decisione di instradamento |
| INPUT       | destinati a un processo locale, subito prima che gli vengano consegnati          |
| OUTPUT      | prodotti da un processo locale, subito dopo la loro creazione                    |
| FORWARD     | che attraversano l'host perché vanno instradati altrove                          |
| POSTROUTING | in procinto di lasciare l'interfaccia di rete                                    |

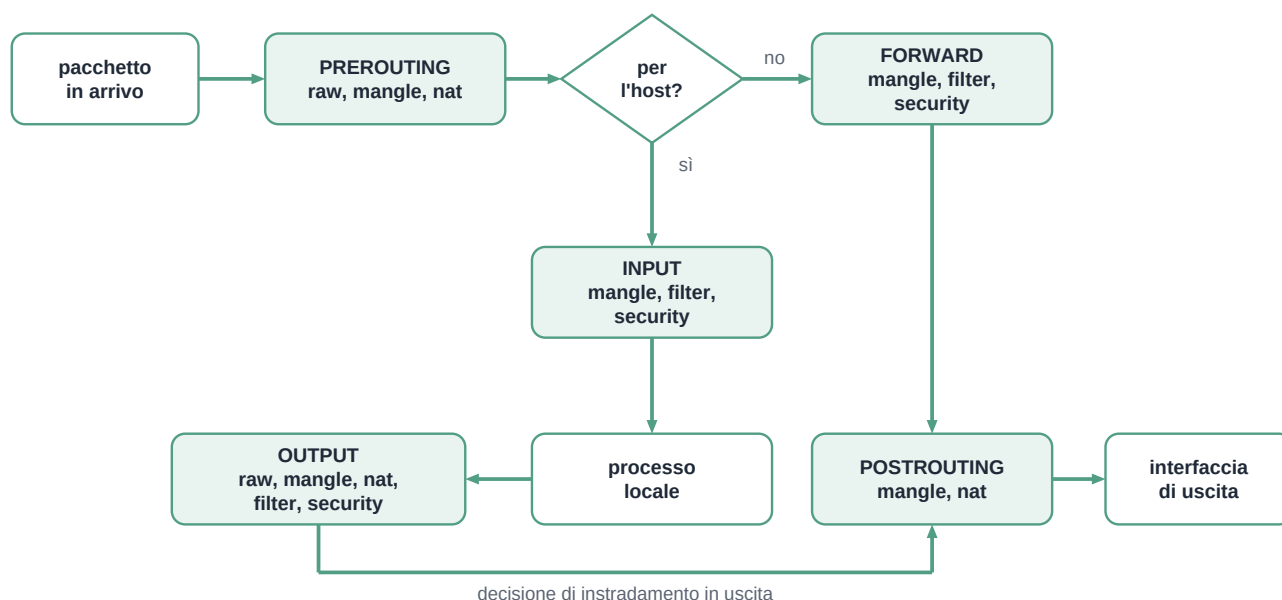
  

| TABELLA | CATENE DISPONIBILI                                           |
|---------|--------------------------------------------------------------|
| filter  | INPUT, OUTPUT, FORWARD                                       |
| raw     | PREROUTING, OUTPUT                                           |
| nat     | PREROUTING, OUTPUT, POSTROUTING (e INPUT nei kernel recenti) |
| mangle  | PREROUTING, INPUT, OUTPUT, FORWARD, POSTROUTING              |

I target principali sono tre. **ACCEPT** accetta il pacchetto. **DROP** lo scarta in silenzio: a chi tenta di collegarsi sembra che il sistema non esista nemmeno. **REJECT** lo rifiuta inviando un errore al mittente: per impostazione predefinita un messaggio ICMP di porta irraggiungibile, mentre con l'apposita opzione si può rispondere a una connessione TCP con un segmento di reset.

## 10.3 Il percorso dei pacchetti

Le catene vanno lette come tappe del percorso di un pacchetto attraverso la macchina.



*Il percorso di un pacchetto attraverso le catene di iptables*

- **Instradamento in ingresso.** Un pacchetto appena arrivato attraversa le catene PREROUTING delle tabelle raw, mangle e nat, in quest'ordine: raw decide se saltare il tracciamento, mangle se modificare le intestazioni, nat se tradurre la destinazione di una nuova connessione. Poi si decide se il pacchetto è destinato all'host: se sì va al firewall in ingresso, altrimenti alla catena FORWARD.

- **Firewall in ingresso.** Il pacchetto attraversa le catene INPUT di mangle, filter e security. La catena INPUT della tabella filter è dove si trova la maggior parte delle regole di firewall; superati i controlli, il pacchetto viene consegnato all'applicazione.
- **Firewall in uscita.** I pacchetti prodotti localmente attraversano le catene OUTPUT di raw, mangle, nat, filter e security, e passano poi all'instradamento in uscita.
- **Instradamento in uscita.** I pacchetti da inoltrare attraversano le catene FORWARD di mangle, filter e security; se il pacchetto non è instradabile viene scartato, altrimenti attraversa le catene POSTROUTING di mangle e nat (dove si applica il masquerading) e lascia l'interfaccia di uscita.

## 10.4 La sintassi del comando

La sintassi generale è *iptables, opzioni, catena, target*. Ogni opzione ha una forma breve e una lunga: per esempio la forma breve di append è A, quella di source è s, quella di jump è j. Le regole create da riga di comando, come quelle del comando ip, valgono fino al riavvio: per conservarle si salva la configurazione con il servizio iptables.

```
iptables -A INPUT -s 192.168.0.28 -j ACCEPT          # forma breve
iptables --append INPUT --source 192.168.0.28 --jump DROP # forma estesa
rc-service iptables save                            # salva le regole (Alpine)
rc-service iptables restart
```

*Forma breve e forma estesa delle opzioni*

## 10.5 Gli scenari più comuni

```
# bloccare un indirizzo: aggiungere, cancellare o sostituire la regola
iptables --table filter --append INPUT --source 59.45.175.62 --jump REJECT
iptables --table filter --delete INPUT --source 59.45.175.62 --jump REJECT
iptables --table filter --replace INPUT 1 --source 59.45.175.62 --jump REJECT

# bloccare una porta (SSH, 22) per una rete
iptables --append INPUT --protocol tcp --match tcp --dport 22 \
    --source 59.45.175.0/24 --jump DROP

# più porte insieme con il modulo multiport
iptables --append INPUT --protocol tcp --match multiport --dports 22,5901 \
    --source 59.45.175.0/24 --jump DROP
```

*Blocco di un indirizzo e di una porta*

```
# rifiutare i ping con un messaggio di errore
iptables --append INPUT --protocol icmp --icmp-type echo-request --jump REJECT
# ignorarli in silenzio, e non rispondere a quelli che arrivano comunque
iptables --append INPUT --protocol icmp --icmp-type echo-request --jump DROP
iptables --append OUTPUT --protocol icmp --icmp-type echo-reply --jump DROP
```

*Blocco di ICMP*

```
# reindirizzare sulla stessa macchina: dalla porta 80 alla 8080
iptables --table nat --append PREROUTING --protocol tcp --dport 80 \
    --jump REDIRECT --to-ports 8080
# inoltrare a un altro nodo (proxy): dalla porta 5001 a 123.123.123.123:110
iptables --table nat --append PREROUTING --protocol tcp --dport 5001 \
    --jump DNAT --to-destination 123.123.123.123:110
# e far apparire come mittente la macchina che reindirizza
iptables -t nat -A POSTROUTING -p tcp --dport 110 -j MASQUERADE
```

#### *Reindirizzamento di porte*

Lo scenario più importante è quello del capitolo precedente: far uscire una rete privata verso Internet. Supponendo che eth0 sia la scheda della rete locale ed eth1 quella verso Internet, servono tre regole: il masquerading dei pacchetti che escono da eth1, il permesso di inoltrare il traffico dalla LAN verso Internet e il permesso di far rientrare soltanto le risposte a connessioni già esistenti.

```
# 1. i pacchetti che escono da eth1 prendono l'indirizzo del router
iptables --table nat --append POSTROUTING --out-interface eth1 --jump MASQUERADE
# 2. inoltrare consentito dalla LAN (eth0) verso Internet (eth1)
iptables --append FORWARD --in-interface eth0 --out-interface eth1 --jump ACCEPT
# 3. da Internet verso la LAN passano solo le risposte a connessioni esistenti
iptables --append FORWARD --in-interface eth1 --out-interface eth0 \
    --match state --state RELATED,ESTABLISHED --jump ACCEPT
```

#### *Dalla rete locale a Internet*

#### **DA RICORDARE**

Il masquerading da solo non basta: l'inoltro IP deve essere attivo, e la catena FORWARD deve consentire il passaggio. Il modulo **state** permette di distinguere le risposte (ESTABLISHED) e le connessioni collegate (RELATED) dalle nuove connessioni, così che dall'esterno non si possano aprire connessioni verso la rete privata.

# Programmazione di rete ed endianness

Dalla connettività si passa ai programmi che comunicano. Il capitolo distingue programmazione concorrente e distribuita e affronta il primo problema concreto dello scambio di dati fra macchine diverse: l'ordine dei byte.

## 11.1 Che cosa serve

Per affrontare la programmazione di rete servono la conoscenza di base del linguaggio C, del sistema operativo (in particolare Linux), del funzionamento dei thread, della gestione del file system e, soprattutto, una solida conoscenza teorica dei protocolli di trasporto. Per la parte sulla comunicazione fra linguaggi diversi serve anche qualche nozione di Python. L'obiettivo è creare **processi che hanno bisogno di comunicare** e che sanno farlo.

## 11.2 Programmazione concorrente e distribuita

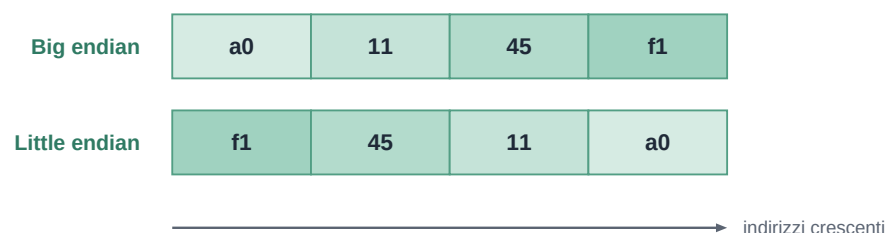
Finora la comunicazione fra flussi di esecuzione si è vista in termini di **thread o processi** che condividono informazioni attraverso una **memoria condivisa**, protetta da meccanismi come mutex e semafori. Questa è la **programmazione concorrente**, e richiede che tutto giri sulla stessa macchina.

Se due processi si scambiano informazioni **attraverso la rete**, possono invece girare su macchine diverse: è la **programmazione distribuita**. Quando si usano entrambe le tecniche, come in un server che serve ogni client con un thread, si parla di programmazione **concorrente e distribuita**.

## 11.3 L'endianness

Il termine viene dai *Viaggi di Gulliver*: gli abitanti di Lilliput si dividono fra chi rompe l'uovo sodo dalla parte piccola (*little endian*) e chi dalla parte grande (*big endian*). In informatica la scelta riguarda i dati di **più byte**, come i numeri interi: in quale ordine disporre i byte in memoria?

- **Big endian**: prima il byte **più** significativo. È l'ordine in cui gli esseri umani, leggendo da sinistra a destra, scrivono i numeri.
- **Little endian**: prima il byte **meno** significativo.



*Lo stesso valore 0xa01145f1 rappresentato in big endian e in little endian*

È utile separare il **valore** dalla sua **rappresentazione**: le due sequenze di byte rappresentano lo stesso valore. Ma se non si conosce l'endianness di una rappresentazione non si sa quale valore rappresenti: la sequenza aa bb vale 0xaabb o 0xbbaa? Per questo, quando comunicano, i nodi devono accordarsi. Gli standard stabiliscono che **tutti i campi delle intestazioni di rete sono big endian**: un nodo little endian deve convertire i valori prima di interpretarli e prima di inviarli. Questo ordine standard si chiama **Network Byte Order**.

L'ordine usato da una macchina dipende dall'architettura: i processori Intel e AMD (x86) sono little endian; storicamente PowerPC e le architetture IBM sono big endian; ARM può lavorare in entrambi i modi (*bi endian*), anche se nei sistemi diffusi è usato in little endian.

#### DA RICORDARE

L'endianness riguarda l'**ordine dei byte**, non dei bit, e si applica solo ai valori di più byte: il byte 0xff resta 0xff in ogni caso. Una stringa ASCII come «mario» è una sequenza di byte singoli, e quindi l'endianness non la riguarda: inviare i dati come **testo** è un modo per aggirare il problema, ed è il motivo per cui molti protocolli applicativi sono testuali.

## 11.4 Le funzioni di conversione

Per convertire i valori fra l'ordine della macchina (*host*) e quello della rete si usano quattro funzioni, dichiarate in `arpa/inet.h`. Sulle macchine big endian non fanno nulla; sulle little endian invertono i byte.

| FUNZIONE     | SIGNIFICATO           | DIMENSIONE              |
|--------------|-----------------------|-------------------------|
| <b>htons</b> | da host a rete, short | 16 bit (porte)          |
| <b>htonl</b> | da host a rete, long  | 32 bit (indirizzi IPv4) |
| <b>ntohs</b> | da rete a host, short | 16 bit                  |
| <b>ntohl</b> | da rete a host, long  | 32 bit                  |

Il programma seguente mostra in pratica i byte di un intero in memoria, prima e dopo la conversione, e i byte di una stringa.

```
/* endianness.c: mostra come la macchina dispone in memoria i byte di
   un intero, prima e dopo la conversione nel Network Byte Order.
   compilare: gcc endianness.c -o endianness */
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <arpa/inet.h>      /* htonl */

int main() {
    uint32_t x = 0x01234567;      /* valore su 4 byte */
    uint8_t buffer[4];

    /* copio i byte dell'intero così come sono in memoria */
    memcpy(buffer, &x, sizeof(x));
    for (int i = 0; i < 4; i++) {
        printf("%02x ", buffer[i]);      /* su Intel: 67 45 23 01 */
    }
    printf("\n");

    /* stesso valore convertito nell'ordine di rete (big endian) */
    uint32_t rete = htonl(x);
    memcpy(buffer, &rete, sizeof(rete));
    for (int i = 0; i < 4; i++) {
        printf("%02x ", buffer[i]);      /* sempre: 01 23 45 67 */
    }
    printf("\n");

    /* una stringa è una sequenza di singoli byte:
```

```

    l'endianness non la riguarda                */
char testo[] = "Hello";
for (int i = 0; i < (int)strlen(testo); i++) {
    printf("%02x (%c) ", (uint8_t)testo[i], testo[i]);
}
printf("\n");
return 0;
}

```

*Disposizione dei byte in memoria e Network Byte Order*

Su un processore Intel la prima riga stampa 67 45 23 01 (il byte meno significativo per primo), la seconda 01 23 45 67 in qualunque macchina, e la terza i codici dei caratteri nell'ordine in cui compaiono nella stringa.

Lo stesso accade con le porte. Il numero 8080 vale 0x1F90: in memoria una macchina little endian lo conserva come 90 1F, mentre in rete deve viaggiare come 1F 90. Se il programmatore dimentica htons, il server si mette in ascolto sulla porta che corrisponde ai byte invertiti, cioè 0x901F = 36895, e il client lo cerca invano sulla 8080.

```

addr.sin_port = 8080;           /* sbagliato: su x86 si ascolta sulla porta 36895 */
addr.sin_port = htons(8080);    /* corretto: 8080 in ordine di rete          */

```

*L'errore più comune: dimenticare htons*

# I socket BSD

Il socket è il punto di accesso che il sistema operativo offre ai programmi per comunicare in rete. Il capitolo presenta i due tipi principali, l'interfaccia BSD con le sue funzioni e le strutture che descrivono gli indirizzi.

## 12.1 Che cos'è un socket

I **socket** sono gli estremi di basso livello attraverso cui si elaborano le informazioni scambiate in rete: con essi si crea una vera connessione fra due nodi. Si collocano fra il livello di trasporto e quello di applicazione, perché permettono alle applicazioni di usare i protocolli di trasporto. Si dicono di basso livello perché TCP e UDP lo sono, mentre HTTP o FTP, che si appoggiano ai socket, sono protocolli di alto livello. Dal punto di vista del sistema operativo un socket si comporta come un **file**, da cui leggere e su cui scrivere (con attenzione alla concorrenza).

| TIPO                   | COMPORTAMENTO                                                                                                                                                                             | PROTOCOLLO TIPICO |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|
| <b>Stream socket</b>   | Flusso di byte affidabile e ordinato: un grande blocco di dati viene spezzato in segmenti in modo trasparente, e il sistema ritrasmette i segmenti persi e riordina quelli fuori sequenza | TCP               |
| <b>Datagram socket</b> | Invio di singoli datagrammi, senza connessione, senza controllo di congestione, senza garanzie di consegna né di ordine: l'applicazione deve gestire da sé ciò che le serve               | UDP               |

I datagram socket si scelgono quando l'applicazione non ha bisogno delle funzioni del flusso, per esempio nello streaming video, dove un dato arrivato in ritardo è inutile. Esistono anche altri tipi di socket, come quelli locali di Unix: il manuale di sistema li descrive tutti.

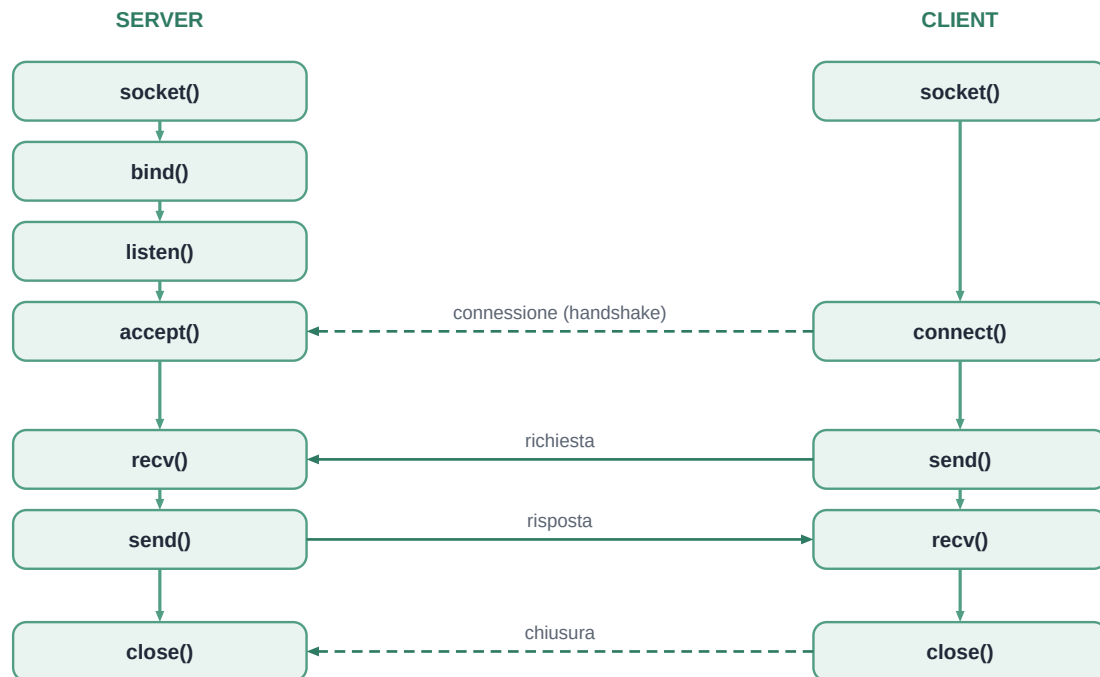
Un socket si può immaginare come un **tunnel** fra due processi su macchine diverse, identificato dalla combinazione di protocollo, indirizzo IP e porta, per esempio tcp://192.168.1.2:8080.

## 12.2 L'interfaccia BSD

I **socket Berkeley** (o BSD) sono un'interfaccia di programmazione (API) per i socket di dominio Internet e per quelli di dominio Unix, usata per la comunicazione fra processi. Nati come standard di fatto, sono entrati con poche modifiche nella specifica **POSIX**, tanto che «socket POSIX» e «socket BSD» sono sinonimi. Tutti i sistemi operativi moderni ne implementano una versione, e anche Winsock di Windows la segue da vicino. L'API è scritta in C, e la maggior parte degli altri linguaggi offre interfacce simili, costruite sopra di essa.

| FUNZIONE       | RUOLO                                                             | CHI LA USA      |
|----------------|-------------------------------------------------------------------|-----------------|
| <b>socket</b>  | Crea il socket                                                    | Client e server |
| <b>bind</b>    | Associa il socket a una porta, come un numero di telefono         | Server          |
| <b>listen</b>  | Si prepara a ricevere connessioni                                 | Server TCP      |
| <b>connect</b> | Si collega a un server                                            | Client TCP      |
| <b>accept</b>  | Accetta una connessione in arrivo, come rispondere a una chiamata | Server TCP      |

| FUNZIONE                 | RUOLO                                                              | CHI LA USA          |
|--------------------------|--------------------------------------------------------------------|---------------------|
| <b>send e recv</b>       | Inviano e ricevono dati su una connessione                         | Client e server TCP |
| <b>sendto e recvfrom</b> | Inviano e ricevono datagrammi indicando ogni volta l'altro estremo | UDP                 |
| <b>close</b>             | Chiude il socket                                                   | Client e server     |



La sequenza delle chiamate in un client e in un server TCP

## 12.3 La funzione socket

La sola funzione usata sia dal client sia dal server è socket.

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

Prototipo della funzione socket

Il valore restituito è dello stesso tipo di quello di open, un intero (un descrittore di file), ed è ciò che permette di trattare i socket come file; in caso di errore vale meno 1. L'argomento **domain** indica la famiglia di protocolli: PF\_INET per IPv4. L'argomento **type** indica il tipo di servizio: SOCK\_STREAM chiede un flusso affidabile (con PF\_INET è TCP), SOCK\_DGRAM un servizio a datagrammi senza connessione (UDP). L'argomento **protocol** dipende dai primi due e spesso non serve: in quel caso vale 0. Si noti che la funzione non dice **a chi** collegarsi: il socket appena creato non è collegato a nulla.

## 12.4 Le strutture degli indirizzi

Diverse funzioni della famiglia ricevono il puntatore a una piccola area di memoria che descrive un indirizzo, la **struct sockaddr**. La sua dichiarazione è volutamente vaga, con un campo sa\_data di 14 byte: i socket sono un'interfaccia generale, usata per molti tipi di comunicazione fra processi, e IP è solo uno di questi. Il campo sa\_family, la **famiglia di indirizzi**, stabilisce come interpretare i byte successivi. Per IPv4 la famiglia è **AF\_INET**, che vale 2, e al posto della struttura generica si usa la **struct sockaddr\_in**, definita in netinet/in.h.

```

/* versione Linux (glibc) */
struct sockaddr_in {
    sa_family_t    sin_family; /* famiglia: AF_INET          */
    in_port_t      sin_port;   /* porta, in ordine di rete  */
    struct in_addr sin_addr;    /* indirizzo IPv4            */
    unsigned char  sin_zero[8]; /* riempimento fino a 16 byte */
};

struct in_addr {
    in_addr_t s_addr; /* intero a 32 bit, in ordine di rete */
};

```

*Le strutture per gli indirizzi IPv4*

Nei sistemi BSD e in macOS le strutture hanno in testa un ulteriore campo di lunghezza (`sa_len` e `sin_len`, di un byte), e anche i valori di alcune famiglie di indirizzi cambiano: `AF_INET6`, per esempio, vale 10 in Linux e 28 nei BSD. Per questo nel codice si usano sempre le costanti simboliche, mai i numeri.

La notazione 193.204.114.105 non è altro che un modo comodo di scrivere un intero a 32 bit elencandone i quattro byte, dal più significativo al meno significativo. Poiché la macchina memorizza gli interi secondo la propria endianness, scrivere direttamente la porta 13 o l'indirizzo come intero produrrebbe un risultato diverso su architetture diverse: i campi vanno convertiti nel Network Byte Order.

```

sa.sin_family = AF_INET;
sa.sin_port = htons(13); /* porta */
sa.sin_addr.s_addr = htonl((((193 << 8 | 204) << 8 | 114) << 8) | 105);
/* oppure, più semplicemente, a partire dalla stringa */
sa.sin_addr.s_addr = inet_addr("193.204.114.105");
inet_pton(AF_INET, "193.204.114.105", &sa.sin_addr); /* versione moderna */

```

*Compilazione di una `sockaddr_in` nel Network Byte Order*

`inet_addr` e `inet_pton` convertono un indirizzo da testo a binario, già nell'ordine di rete; `inet_ntop` e `inet_ntoa` fanno l'operazione inversa, utile per stampare l'indirizzo di un client.

# Il primo client e il primo server TCP

Un client per il protocollo daytime e un server che restituisce data e ora, gestendo ogni client con un processo figlio: i due programmi mostrano tutte le chiamate fondamentali dei socket TCP.

## 13.1 La funzione connect

Creato il socket, il client deve collegarlo a una porta di un sistema remoto con **connect**, che riceve il socket, il puntatore alla struttura con l'indirizzo e la sua dimensione. Restituisce 0 se la connessione riesce; altrimenti restituisce meno 1 e memorizza il codice di errore nella variabile `errno`. Le cause di errore sono molte: l'indirizzo può non esistere o non essere raggiungibile, la macchina può essere spenta o sovraccarica, oppure nessun server è in ascolto su quella porta.

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

*Prototipo della funzione connect*

## 13.2 Il client daytime

Il protocollo **daytime** prevede che il server, sulla porta 13, invii una riga di testo con data e ora correnti e chiuda la connessione. Il client usa TCP, quindi la famiglia `AF_INET` e un socket di tipo stream; come server si usa quello indicato nelle slide, all'indirizzo 193.204.114.105, attribuito all'INRiM (Istituto Nazionale di Ricerca Metrologica), l'ente che custodisce l'ora ufficiale italiana.

```
/* client_daytime.c: primo client TCP. Si collega a un server del
   protocollo daytime (porta 13), che risponde con una riga di testo
   contenente data e ora, e la stampa a video.
   compilare: gcc client_daytime.c -o client_daytime */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>      /* inet_addr: mancava nella versione delle slide */

int main() {
    int s, numBytes;
    struct sockaddr_in sa;
    char buffer[BUFSIZ + 1]; /* BUFSIZ: dimensione di I/O del sistema */

    /* 1. creo la socket: IPv4 (PF_INET) e flusso affidabile (TCP) */
    if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return 1;
    }

    /* 2. preparo l'indirizzo del server: azzero e riempio i campi */
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
```

```

sa.sin_port = htons(13); /* porta in ordine di rete */
sa.sin_addr.s_addr = inet_addr("193.204.114.105"); /* time server INRiM */

/* 3. mi collego: qui avviene l'handshake a tre vie di TCP */
if (connect(s, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
    perror("connect");
    close(s);
    return 2;
}

/* 4. leggo finché il server non chiude (read restituisce 0) */
while ((numBytes = read(s, buffer, BUFSIZ)) > 0) {
    write(1, buffer, numBytes); /* il descrittore 1 è lo stdout */
}

close(s);
return 0;
}

```

*Il primo client TCP*

Il programma si compila con gcc (o con il compilatore cc) indicando il file sorgente e, con l'opzione o, il nome dell'eseguibile. La lettura avviene con read, la stessa funzione usata per i file, e la scrittura con write sul descrittore 1, lo standard output: un'ulteriore prova che il sistema tratta i socket come file.

### 13.3 Le funzioni del server

Un server non avvia la connessione: aspetta che i client lo chiamino. Non sa quando lo faranno, né quanti saranno: in un momento può restare in paziente attesa, in quello successivo ritrovarsi sommerso di richieste contemporanee. Dei 65 536 numeri di porta, un server di solito ne serve uno solo, e con **bind** dice al sistema quale.

```

int bind(int s, const struct sockaddr *addr, socklen_t addrlen);
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

```

*Prototipi delle funzioni del server*

Oltre alla porta, nella struttura il server può indicare il proprio indirizzo IP, ma di solito usa la costante **INADDR\_ANY** (che vale 0), per servire le richieste che arrivano su quella porta da qualunque interfaccia. Con **listen** il server si mette in ascolto: il parametro **backlog** stabilisce quante connessioni possono restare in coda mentre il server è impegnato, cioè la dimensione massima della coda delle connessioni in sospeso.

Con **accept** il server accetta una connessione. Questa volta la dimensione dell'indirizzo è passata per puntatore, perché è accept a riempire la struttura con l'indirizzo del client. Il valore restituito è un **nuovo socket**, dedicato alla comunicazione con quel client; il socket originale continua ad ascoltare nuove richieste finché non viene chiuso.

#### DEFINIZIONE

In un server TCP convivono due tipi di socket. Il **socket di ascolto**, creato con socket, bind e listen, serve solo ad accettare le connessioni. Ogni chiamata ad **accept** restituisce un **socket di connessione**, dedicato a un singolo client, su cui avvengono send e recv.

## 13.4 Il primo server: un processo per client

Il server seguente restituisce data e ora a ogni client che si collega. Per poter servire più client, per ogni connessione accettata crea un **processo figlio** con **fork**: il figlio serve il client e termina, mentre il padre torna subito ad accettare nuove connessioni.

```
/* server_ora.c: primo server TCP. Per ogni client che si collega crea
   un processo figlio con fork, che invia data e ora e chiude.
   compilare: gcc server_ora.c -o server_ora */
#include <stdio.h>
#include <stdlib.h>          /* exit: mancava nella versione delle slide */
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BACKLOG 4           /* massimo di connessioni in coda */
#define PORT 8080

/* crea la socket del server, la lega alla porta e la mette in ascolto */
int create_server(uint16_t port) {
    int socketServerFD;
    struct sockaddr_in addr;

    if ((socketServerFD = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket failed");
        return -1;
    }

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_ANY); /* tutte le interfacce */

    /* bind: associo la socket alla porta (il "numero di telefono") */
    if (bind(socketServerFD, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind failed");
        return -2;
    }

    /* listen: da qui in poi il sistema accoda le richieste in arrivo */
    if (listen(socketServerFD, BACKLOG) < 0) {
        perror("listen failed");
        return -3;
    }

    return socketServerFD;
}

int main() {
    int server_fd, client_fd;
    struct sockaddr_in addr;
    socklen_t addr_len = sizeof(addr);

    server_fd = create_server(PORT);
    if (server_fd < 0) exit(1);
    printf("Server running on port %d\n", PORT);
```

```

for (;;) {
    /* accept restituisce una NUOVA socket, dedicata a questo client */
    if ((client_fd = accept(server_fd, (struct sockaddr *)&addr, &addr_len)) < 0) {
        perror("accept failed");
        continue;
    }

    pid_t pid = fork();                /* duplico il processo */
    if (pid < 0) {
        perror("fork failed");
        close(client_fd);
    } else if (pid == 0) {             /* processo figlio: serve il client */
        close(server_fd);              /* al figlio la socket di ascolto non serve */

        time_t now = time(NULL);
        struct tm *tm_info = gmtime(&now);
        char buffer[1024];
        strftime(buffer, sizeof(buffer), "%Y %m %d %H:%M:%S UTC\n", tm_info);

        send(client_fd, buffer, strlen(buffer), 0);
        close(client_fd);
        _exit(0);                      /* il figlio termina */
    } else {                          /* processo padre: torna ad ascoltare */
        close(client_fd);              /* al padre la socket del client non serve */
    }
}

close(server_fd);
return 0;
}

```

*Il primo server TCP, con un processo figlio per ogni client*

Dopo la fork entrambi i processi possiedono entrambi i socket, e ciascuno chiude quello che non gli serve: il figlio il socket di ascolto, il padre il socket del client. Si può provare il server con il client daytime, cambiando indirizzo e porta, oppure con un client già pronto come telnet o nc puntati sulla porta 8080.

#### DA RICORDARE

Due dettagli pratici. Un processo figlio terminato resta come processo **zombie** finché il padre non ne raccoglie lo stato: in un server che gira a lungo si evita con `signal(SIGCHLD, SIG_IGN)`. Inoltre, riavviando subito un server, `bind` può fallire con l'errore «Address already in use», perché la porta è ancora occupata dalle connessioni appena chiuse, che restano per un po' nello stato `TIME_WAIT`: si attende qualche secondo, oppure si imposta l'opzione `SO_REUSEADDR` con `setsockopt` prima di `bind`.

# Messaggistica TCP con i thread

Un client che invia e riceve nello stesso momento e un server che inoltra i messaggi di ogni client a tutti gli altri: il capitolo costruisce passo per passo una piccola chat TCP, introducendo thread e mutex.

## 14.1 Il client, un passo alla volta

Vogliamo un software di messaggistica con cui più utenti comunicano attraverso un server. Partiamo dal client, che riceve dalla riga di comando l'indirizzo IP e la porta del server: per prima cosa controlla gli argomenti, ricordando che `argv[0]` è il nome del programma stesso.

```
if (argc < 3) {
    fprintf(stderr, "Usage: %s <IP> <Port>\n", argv[0]);
    exit(1);
}
```

*Controllo degli argomenti*

Servono poi il socket e un buffer per i dati da inviare e ricevere, la cui dimensione dipende dai messaggi previsti. La struttura dell'indirizzo si compila come nel capitolo precedente, usando `inet_pton` per convertire l'indirizzo scritto come testo. Infine ci si collega e si entra nel ciclo principale, che legge una riga dallo standard input, la invia, attende la risposta e la stampa; la parola `exit` interrompe il ciclo.

```
for (;;) {
    printf("> ");
    fflush(stdout); /* stampa subito il prompt */
    if (!fgets(buffer, BUFFER_SIZE, stdin)) break; /* Ctrl+D */
    buffer[strcspn(buffer, "\n")] = '\0'; /* tolgo l'a capo */
    if (strcmp(buffer, "exit") == 0) break;

    if (send(sockfd, buffer, strlen(buffer), 0) < 0) {
        perror("send failed");
        break;
    }
    int bytes_received = recv(sockfd, buffer, BUFFER_SIZE - 1, 0);
    if (bytes_received <= 0) break; /* errore o server chiuso */
    buffer[bytes_received] = '\0';
    printf("%s\n", buffer);
}
```

*Il ciclo principale della prima versione del client*

## 14.2 Il problema della `recv` bloccante

La prima versione ha un limite serio: dopo ogni invio il client può ricevere un solo messaggio, perché **`recv` è un'operazione bloccante**. Se l'utente non scrive, il programma è fermo su `fgets` e non riceve nulla; se il programma attende una risposta, l'utente non può scrivere. La soluzione è separare le due attività in due **thread** che lavorano in parallelo: uno legge la tastiera e invia, l'altro resta bloccato su `recv` e stampa ciò che arriva. Il main crea il socket, si collega, avvia i due thread e ne attende la fine con `pthread_join`. Il socket è una variabile globale, perché deve essere visibile a entrambi i thread.

```

/* client.c: client di messaggistica TCP. Due thread lavorano in
parallelo: uno legge la tastiera e invia, l'altro riceve e stampa.
uso:      ./client <IP> <Porta>
compilare: gcc client.c -o client -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 1024

int sockfd;      /* globale: deve essere visto da entrambi i thread */

/* thread di ricezione: recv è bloccante, per questo ha un thread tutto suo */
void *receive_handler(void *arg) {
    char buffer[BUFFER_SIZE];
    for (;;) {
        ssize_t bytes = recv(sockfd, buffer, BUFFER_SIZE - 1, 0);
        if (bytes <= 0) {
            /* 0: il server ha chiuso */
            if (bytes == 0) printf("\nServer closed connection\n");
            else perror("Receive error");
            exit(1);
        }
        buffer[bytes] = '\0';
        printf("\nReceived: %s\n", buffer);
        fflush(stdout);
    }
    return NULL;
}

/* thread di invio: legge una riga dalla tastiera e la spedisce */
void *send_handler(void *arg) {
    char buffer[BUFFER_SIZE];
    for (;;) {
        printf("> ");
        fflush(stdout);
        /* fine input (Ctrl+D) oppure comando exit: chiudo ed esco */
        if (!fgets(buffer, BUFFER_SIZE, stdin)) {
            close(sockfd);
            exit(0);
        }
        buffer[strcspn(buffer, "\n")] = '\0'; /* tolgo l'a capo */
        if (strcmp(buffer, "exit") == 0) {
            close(sockfd);
            exit(0);
        }
        if (send(sockfd, buffer, strlen(buffer), 0) < 0) {
            perror("Send failed");
            exit(1);
        }
    }
    return NULL;
}

```

```

int main(int argc, char *argv[]) {
    struct sockaddr_in serv_addr;

    if (argc < 3) {
        /* argv[0] è il programma stesso */
        fprintf(stderr, "Usage: %s <IP> <Port>\n", argv[0]);
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket creation failed");
        exit(1);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(argv[2]));
    /* inet_pton converte l'indirizzo da testo a binario */
    if (inet_pton(AF_INET, argv[1], &serv_addr.sin_addr) <= 0) {
        fprintf(stderr, "invalid address\n");
        exit(1);
    }

    if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("connection failed");
        exit(1);
    }
    printf("Connected to %s:%s\n", argv[1], argv[2]);

    pthread_t send_thread, recv_thread;
    pthread_create(&recv_thread, NULL, receive_handler, NULL);
    pthread_create(&send_thread, NULL, send_handler, NULL);

    /* il main aspetta la fine dei due thread */
    pthread_join(send_thread, NULL);
    pthread_join(recv_thread, NULL);

    close(sockfd);
    return 0;
}

```

*Il client TCP con un thread di invio e uno di ricezione*

Si noti che i thread condividono la memoria del processo, ed è proprio ciò che serve qui: entrambi devono usare lo stesso socket. Con fork si avrebbero invece due processi con memoria separata.

## 14.3 Il server: un thread per client

Il server deve gestire più client contemporaneamente e inoltrare a tutti gli altri ciò che scrive ciascuno. Ogni client viene servito da un proprio thread, e i client collegati sono memorizzati in una **lista condivisa**. La lista è una **sezione critica**: un client può aggiungersi o andarsene mentre un altro thread la sta scorrendo per inoltrare un messaggio, e per questo ogni accesso è protetto da un **mutex**.

Il server si compone di quattro parti. La struttura **client\_info** contiene socket e indirizzo di un client, e la struttura **client\_list** l'elenco dei client con il relativo contatore e il mutex. La funzione **broadcast\_message** scorre la lista, sotto mutex, e invia il messaggio a tutti tranne il mittente. La funzione **handle\_client**, eseguita da un thread per ogni client, ne stampa l'indirizzo, lo aggiunge alla lista, inoltra a tutti i suoi messaggi preceduti da indirizzo e porta, e alla disconnessione lo rimuove. Il **main**, infine, crea il socket di ascolto e per ogni connessione accettata avvia

un thread, in modalità *detached* così che le sue risorse vengano liberate automaticamente alla fine. La struttura `client_info` di ogni client è allocata con `malloc` nel `main` e liberata dal suo thread: ogni thread deve ricevere la **propria** copia dei dati, mentre l'indirizzo di una variabile locale del `main` verrebbe sovrascritto alla connessione successiva.

```
/* server.c: server di chat TCP. Un thread per ogni client; ogni
   messaggio ricevuto viene inoltrato a tutti gli altri client.
   compilare: gcc server.c -o server -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 8080
#define MAX_CLIENTS 10
#define BUFFER_SIZE 1024

/* informazioni su un client collegato */
typedef struct {
    int sockfd;
    struct sockaddr_in address;
} client_info;

/* lista condivisa fra i thread: va protetta con un mutex */
typedef struct {
    client_info *clients[MAX_CLIENTS];
    int count;
    pthread_mutex_t mutex;
} client_list;

client_list clients = { .count = 0 };

/* invia il messaggio a tutti i client tranne il mittente */
void broadcast_message(const char *message, int sender_fd) {
    pthread_mutex_lock(&clients.mutex); /* nessuno modifica la lista ora */
    for (int i = 0; i < clients.count; i++) {
        if (clients.clients[i]->sockfd != sender_fd) {
            send(clients.clients[i]->sockfd, message, strlen(message), 0);
        }
    }
    pthread_mutex_unlock(&clients.mutex);
}

/* thread dedicato a un client: lo registra, ne inoltra i messaggi,
   e alla disconnessione lo toglie dalla lista */
void *handle_client(void *arg) {
    client_info *info = (client_info *)arg;
    int client_sock = info->sockfd;
    char buffer[BUFFER_SIZE];
    char client_ip[INET_ADDRSTRLEN];

    /* indirizzo e porta del client, convertiti in forma leggibile */
    inet_ntop(AF_INET, &info->address.sin_addr, client_ip, INET_ADDRSTRLEN);
    int client_port = ntohs(info->address.sin_port);
```

```

printf("New connection from %s:%d\n", client_ip, client_port);

/* aggiungo il client alla lista, solo se c'è posto */
pthread_mutex_lock(&clients.mutex);
if (clients.count == MAX_CLIENTS) {
    pthread_mutex_unlock(&clients.mutex);
    printf("Too many clients, connection refused\n");
    close(client_sock);
    free(info);
    return NULL;
}
clients.clients[clients.count++] = info;
pthread_mutex_unlock(&clients.mutex);

for (;;) {
    ssize_t bytes_received = recv(client_sock, buffer, BUFFER_SIZE - 1, 0);
    if (bytes_received <= 0) {
        /* 0: il client ha chiuso */
        if (bytes_received == 0)
            printf("Client %s:%d disconnected\n", client_ip, client_port);
        else
            perror("recv failed");
        break;
    }
    buffer[bytes_received] = '\0';

    /* formato del messaggio inoltrato: [IP:Porta] testo */
    char formatted_message[BUFFER_SIZE + INET_ADDRSTRLEN + 16];
    snprintf(formatted_message, sizeof(formatted_message),
             "[%s:%d] %s", client_ip, client_port, buffer);
    printf("%s\n", formatted_message);
    broadcast_message(formatted_message, client_sock);
}

/* tolgo il client dalla lista e compatto l'array */
pthread_mutex_lock(&clients.mutex);
for (int i = 0; i < clients.count; i++) {
    if (clients.clients[i]->sockfd == client_sock) {
        free(clients.clients[i]);
        for (int j = i; j < clients.count - 1; j++) {
            clients.clients[j] = clients.clients[j + 1];
        }
        clients.count--;
        break;
    }
}
pthread_mutex_unlock(&clients.mutex);

close(client_sock);
return NULL;
}

int main() {
    int server_fd;
    struct sockaddr_in address;

    pthread_mutex_init(&clients.mutex, NULL);

    /* socket restituisce un valore negativo in caso di errore */

```

```

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket failed");
    exit(1);
}

memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    perror("bind failed");
    exit(1);
}
if (listen(server_fd, MAX_CLIENTS) < 0) {
    perror("listen");
    exit(1);
}
printf("Chat server listening on port %d...\n", PORT);

for (;;) {
    /* ogni client ha la sua struttura, liberata dal suo thread */
    client_info *client = malloc(sizeof(client_info));
    socklen_t addrlen = sizeof(client->address);

    client->sockfd = accept(server_fd, (struct sockaddr *)&client->address, &addrlen);
    if (client->sockfd < 0) {
        perror("accept");
        free(client);
        continue;
    }

    /* un thread per client; detach: si ripulisce da solo alla fine */
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, handle_client, client) != 0) {
        perror("pthread_create");
        close(client->sockfd);
        free(client);
    } else {
        pthread_detach(thread_id);
    }
}

close(server_fd);
pthread_mutex_destroy(&clients.mutex);
return 0;
}

```

*Il server di chat TCP con un thread per client e la lista protetta da mutex*

```

gcc server.c -o server -pthread
gcc client.c -o client -pthread
./server                # nel primo terminale
./client 127.0.0.1 8080  # in un secondo terminale
./client 127.0.0.1 8080  # in un terzo terminale

```

*Compilazione ed esecuzione su una sola macchina*

# Protocolli testuali e messaggi strutturati

Inviare stringhe è semplice ma fragile. Il capitolo confronta i messaggi testuali con i messaggi strutturati, spiega come serializzarli in C e perché servono la direttiva `pragma pack` e il Network Byte Order.

## 15.1 Il problema delle stringhe

Inviare stringhe è il modo più semplice di definire un protocollo: il client invia un testo con `send`, il server lo riceve con `recv` e lo stampa.

```
/* il client invia */
const char *msg = "Hello Server!";
send(sockfd, msg, strlen(msg), 0);

/* il server riceve */
char buffer[1024];
int n = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
buffer[n] = '\0'; /* recv non aggiunge il terminatore */
printf("Received: %s\n", buffer);
```

*Un protocollo testuale*

I vantaggi sono reali: non serve alcuna formattazione complessa, i messaggi sono leggibili da un essere umano e il metodo è flessibile, adatto a qualunque protocollo testuale. Ma ci sono anche svantaggi: mancano i **metadati** (mittente, istante di invio, tipo del messaggio), non c'è alcun controllo sui **tipi** dei dati e l'analisi del testo (*parsing*) è fragile.

## 15.2 I messaggi strutturati

L'alternativa è definire il messaggio come una **struttura** C condivisa da client e server, che diventa così il nostro piccolo protocollo applicativo.

```
#pragma pack(push, 1) /* disattiva il riempimento dell'architettura */
typedef struct {
    char sender[32]; /* nome del mittente, dimensione fissa */
    char message[256]; /* testo, dimensione fissa */
    uint8_t msg_type; /* 0 = testo, 1 = file, 2 = comando */
    uint32_t timestamp; /* istante di invio (Unix time) */
} ChatMessage; /* 32 + 256 + 1 + 4 = 293 byte */
#pragma pack(pop) /* ripristina il comportamento normale */
```

*Un messaggio strutturato*

Per usare la struttura bisogna **serializzarla** (trasformarla in byte da inviare) e **deserializzarla** (ricostruirla dai byte ricevuti). In C, con strutture di dimensione fissa e senza puntatori, le due operazioni si riducono a inviare i byte della struttura e a riceverli dentro una struttura vuota.

```

/* il client compila e invia */
ChatMessage msg;
memset(&msg, 0, sizeof(msg));           /* niente spazzatura nei campi */
strncpy(msg.sender, "Alice", sizeof(msg.sender) - 1);
strncpy(msg.message, "Hello World!", sizeof(msg.message) - 1);
msg.msg_type = 0;
msg.timestamp = htonl(time(NULL));      /* campo di 4 byte: ordine di rete */
send(sockfd, &msg, sizeof(msg), 0);

/* il server riceve direttamente dentro la struttura */
ChatMessage received_msg;
recv(sockfd, &received_msg, sizeof(received_msg), MSG_WAITALL);
printf("[%s][%u] %s\n", received_msg.sender,
        ntohl(received_msg.timestamp), received_msg.message);

```

#### *Serializzazione e deserializzazione della struttura*

Tre accorgimenti rendono il meccanismo affidabile. La direttiva **pragma pack(1)** elimina i byte di riempimento (*padding*) che il compilatore inserirebbe per allineare i campi, così che la struttura abbia la stessa disposizione su qualunque macchina. I campi di più byte vanno convertiti nel **Network Byte Order**, perché mittente e destinatario potrebbero avere endianness diverse. Infine l'opzione **MSG\_WAITALL** fa sì che `recv` attenda l'intera struttura: TCP è un flusso di byte, non di messaggi, e una sola `send` può arrivare spezzata in più `recv`.

#### **DA RICORDARE**

Con TCP i confini dei messaggi non esistono: il protocollo garantisce che i byte arrivino tutti e in ordine, non che una `recv` restituisca esattamente ciò che una `send` ha inviato. Con le strutture di dimensione fissa il problema si risolve leggendo sempre `sizeof` della struttura con `MSG_WAITALL`; con UDP invece ogni datagramma arriva intero oppure non arriva affatto.

# Socket UDP e messaggistica a datagrammi

Con UDP non ci sono connessioni: ogni datagramma porta con sé l'indirizzo di destinazione. Il capitolo presenta `sendto` e `recvfrom` e realizza una chat in cui il server riconosce i client dal loro indirizzo.

## 16.1 Datagrammi invece di connessioni

In UDP il client non stabilisce una connessione con il server, ma gli invia direttamente un **datagramma**; il server, a sua volta, non accetta connessioni, ma attende l'arrivo dei datagrammi. Ogni datagramma ricevuto porta con sé l'**indirizzo del mittente**, che il server usa per rispondere al client giusto.

Anche in UDP si può chiamare `connect`, ma l'effetto è del tutto diverso da TCP: non avviene alcun handshake a tre vie. La funzione si limita a controllare eventuali errori immediati e a **memorizzare l'indirizzo dell'interlocutore**, così che le chiamate successive non debbano più indicarlo.

Come in TCP, il server crea il socket e lo associa a una porta con `bind`, ma con il tipo `SOCK_DGRAM`. Non esiste invece la `listen`, perché non ci sono connessioni da mettere in coda.

```
int create_server(uint16_t port) {
    int socketServerFD = socket(AF_INET, SOCK_DGRAM, 0);    /* UDP */
    struct sockaddr_in addr;
    if (socketServerFD < 0) {
        perror("socket failed");
        return -1;
    }
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(socketServerFD, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind failed");
        return -2;
    }
    return socketServerFD;                                /* niente listen: non ci sono connessioni */
}
```

*Creazione di un server UDP*

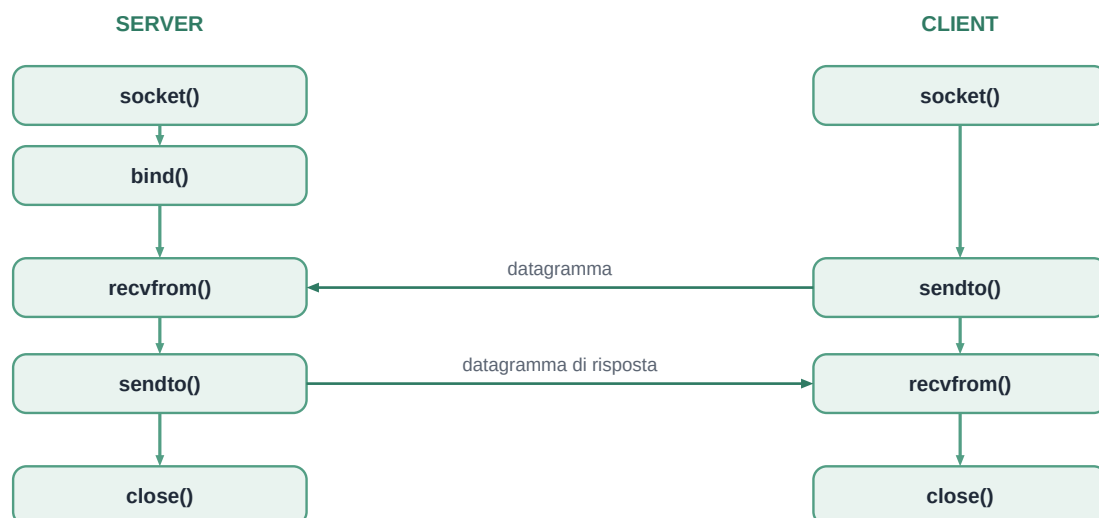
## 16.2 `sendto` e `recvfrom`

Poiché non c'è una connessione, l'altro estremo va indicato **a ogni invio**. Per questo UDP usa `sendto` e `recvfrom`, che hanno due parametri in più rispetto a `send` e `recv`.

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

*Prototipi di `sendto` e `recvfrom`*

Con `sendto` si indicano, nella solita struttura `sockaddr`, indirizzo IP e porta della destinazione. Con `recvfrom`, se serve sapere chi ha scritto, si passa il puntatore a una struttura in cui la funzione salverà indirizzo e porta del mittente (e la sua dimensione, per puntatore); se non serve, si passano due puntatori nulli.



*La sequenza delle chiamate in un client e in un server UDP*

## 16.3 Una chat UDP

Per provare UDP realizziamo un servizio di messaggistica. Il protocollo è una struttura con un comando (messaggio normale, registrazione o uscita), il nickname del mittente e il testo.

```

/* protocol.h: formato dei messaggi della chat UDP, condiviso da
   client e server (entrambi devono vedere la stessa struttura).  */
#include <stdint.h>

#define NICKNAME_LEN 32
#define MSG_LEN 256

#pragma pack(push, 1)          /* niente byte di riempimento fra i campi */
typedef struct {
    uint8_t command;           /* 0 = normale, 1 = registrazione, 2 = uscita */
    char nickname[NICKNAME_LEN];
    char message[MSG_LEN];
} chat_message;               /* 1 + 32 + 256 = 289 byte */
#pragma pack(pop)
  
```

*Il protocollo della chat UDP*

Il server riceve i datagrammi in un unico ciclo. Una **registrazione** aggiunge il mittente all'elenco dei client, se non c'è già; un **uscita** lo rimuove; un **messaggio normale** viene inoltrato a tutti gli altri. I client sono riconosciuti confrontando con `memcmp` l'intera struttura dell'indirizzo, cioè IP e porta. Poiché il server ha un solo flusso di esecuzione, non servono mutex.

```

/* server.c: server della chat UDP. Non ci sono connessioni: il server
   riconosce i client dall'indirizzo (IP e porta) da cui scrivono.
   compilare: gcc server.c -o server */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "protocol.h"

#define PORT 8080
#define MAX_CLIENTS 10

/* un client registrato: il suo indirizzo e il suo nickname */
typedef struct {
    struct sockaddr_in addr;
    char nickname[NICKNAME_LEN];
} client_info;

client_info clients[MAX_CLIENTS];
int client_count = 0;

/* inoltra il messaggio a tutti i registrati tranne il mittente,
   usando la stessa socket del server */
void broadcast_message(int sockfd, chat_message *msg, struct sockaddr_in *sender) {
    for (int i = 0; i < client_count; i++) {
        if (memcmp(&clients[i].addr, sender, sizeof(struct sockaddr_in)) != 0) {
            sendto(sockfd, msg, sizeof(chat_message), 0,
                (struct sockaddr *)&clients[i].addr, sizeof(clients[i].addr));
        }
    }
}

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_len;
    chat_message msg;

    /* socket UDP: SOCK_DGRAM */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket failed");
        return 1;
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    /* bind sì, listen no: in UDP non ci sono connessioni da accettare */
    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind failed");
        return 2;
    }
    printf("UDP Chat Server running on port %d\n", PORT);
}

```

```

while (1) {
    addr_len = sizeof(client_addr);
    /* recvfrom riempie client_addr con l'indirizzo del mittente */
    ssize_t bytes = recvfrom(sockfd, &msg, sizeof(msg), 0,
                             (struct sockaddr *)&client_addr, &addr_len);
    if (bytes != sizeof(msg)) continue; /* scarto i messaggi incompleti */

    if (msg.command == 1) { /* REGISTRAZIONE */
        int exists = 0;
        for (int i = 0; i < client_count; i++) {
            if (memcmp(&clients[i].addr, &client_addr, sizeof(client_addr)) == 0) {
                exists = 1;
                break;
            }
        }
        if (!exists && client_count < MAX_CLIENTS) {
            clients[client_count].addr = client_addr;
            strncpy(clients[client_count].nickname, msg.nickname, NICKNAME_LEN);
            printf("%s joined from %s:%d\n", msg.nickname,
                  inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
            client_count++;
        }
    } else if (msg.command == 2) { /* USCITA */
        for (int i = 0; i < client_count; i++) {
            if (memcmp(&clients[i].addr, &client_addr, sizeof(client_addr)) == 0) {
                printf("%s left\n", clients[i].nickname);
                /* rimuovo spostando indietro gli elementi successivi */
                for (int j = i; j < client_count - 1; j++) {
                    clients[j] = clients[j + 1];
                }
                client_count--;
                break;
            }
        }
    } else { /* MESSAGGIO NORMALE */
        printf("Broadcasting: [%s] %s\n", msg.nickname, msg.message);
        broadcast_message(sockfd, &msg, &client_addr);
    }
}

close(sockfd);
return 0;
}

```

*Il server della chat UDP*

Il client chiede il nickname, si registra con un messaggio di comando 1 e avvia un thread che riceve con `recvfrom`; il main legge la tastiera e invia ogni riga con `sendto`, finché l'utente non scrive `/exit`. Il client non chiama mai `bind`: alla prima `sendto` il sistema gli assegna automaticamente una porta libera, ed è su quella che il server gli risponderà.

```

/* client.c: client della chat UDP. Un thread riceve, il main invia.
   compilare: gcc client.c -o client -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "protocol.h"

#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 8080

int sockfd;
struct sockaddr_in server_addr;
char nickname[NICKNAME_LEN];

/* thread di ricezione: stampa i messaggi inoltrati dal server */
void *receive_handler(void *arg) {
    chat_message msg;
    struct sockaddr_in from_addr;
    socklen_t addr_len;

    while (1) {
        addr_len = sizeof(from_addr);
        ssize_t bytes = recvfrom(sockfd, &msg, sizeof(msg), 0,
                                (struct sockaddr *)&from_addr, &addr_len);
        if (bytes == sizeof(msg)) {
            printf("\n[%s] %s\n> ", msg.nickname, msg.message);
            fflush(stdout);
        }
    }
    return NULL;
}

/* costruisce un messaggio e lo spedisce al server */
void send_message(const char *text, uint8_t cmd) {
    chat_message msg;
    memset(&msg, 0, sizeof(msg)); /* niente spazzatura nei campi */
    strncpy(msg.nickname, nickname, NICKNAME_LEN);
    strncpy(msg.message, text, MSG_LEN);
    msg.command = cmd;
    sendto(sockfd, &msg, sizeof(msg), 0,
           (struct sockaddr *)&server_addr, sizeof(server_addr));
}

int main() {
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket failed");
        return 1;
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr);

```

```

printf("Enter nickname: ");
fgets(nickname, NICKNAME_LEN, stdin);
nickname[strcspn(nickname, "\n")] = '\0';

send_message("", 1);                                /* registrazione */

pthread_t recv_thread;
pthread_create(&recv_thread, NULL, receive_handler, NULL);

printf("Connected! Type messages (/exit to quit)\n> ");
fflush(stdout);
char input[MSG_LEN];
while (fgets(input, MSG_LEN, stdin) != NULL) {
    input[strcspn(input, "\n")] = '\0';
    if (strcmp(input, "/exit") == 0) break;
    send_message(input, 0);                          /* messaggio normale */
}
send_message("", 2);                                /* uscita */

close(sockfd);
return 0;
}

```

*Il client della chat UDP*

#### DA RICORDARE

In UDP nessuno garantisce che un datagramma arrivi: un messaggio di registrazione perso lascia il client fuori dalla chat senza che nessuno se ne accorga. Se l'applicazione ha bisogno di affidabilità deve costruirsela da sé, per esempio con conferme e ritrasmissioni, oppure usare TCP.

# Comunicazione fra linguaggi diversi

Due programmi che rispettano lo stesso protocollo possono comunicare anche se sono scritti in linguaggi diversi. Il capitolo introduce i socket in Python e il modulo struct, e mostra lo scambio di dati binari fra un programma Python e uno C.

## 17.1 Processi, non linguaggi

Client e server, mittente e destinatario sono **processi**: ciò che conta è che parlino lo stesso protocollo attraverso lo stack TCP/IP, non il linguaggio in cui sono scritti. Qualunque linguaggio o tecnologia che supporti i socket BSD può quindi comunicare con un programma C. Python è una scelta comoda per la semplicità del linguaggio, ma lo stesso esercizio si può fare con qualunque altro.

## 17.2 Socket TCP in Python

In Python si importa il modulo socket e si eseguono, nello stesso ordine, le stesse operazioni viste in C. Le differenze sono di forma: bind e connect ricevono una **tupla** con indirizzo e porta, accept restituisce la coppia formata dal nuovo socket e dall'indirizzo del client, e i dati viaggiano come **bytes**, per cui le stringhe vanno codificate con encode prima dell'invio e decodificate con decode dopo la ricezione.

```
# client.py: client TCP in Python, equivalente al client C
import socket

def client_program():
    host = "127.0.0.1"
    port = 8080

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # socket TCP
    client_socket.connect((host, port)) # connessione al server

    message = input("> ") # testo da tastiera
    while message.lower().strip() != "exit":
        client_socket.send(message.encode()) # str diventa bytes e parte
        data = client_socket.recv(1024).decode() # risposta, da bytes a str
        print("Received from server: " + data)
        message = input("> ")

    client_socket.close() # chiusura della connessione

if __name__ == "__main__":
    client_program()
```

*Client TCP in Python*

Come in C, il server può servire più client contemporaneamente con un thread per ciascuno, usando il modulo threading.

```
# server.py: server TCP in Python; un thread per ogni client
import socket
import threading

MAX_CLIENTS = 24

def handle_client(client_socket, address):
    # risponde "ok" a ogni messaggio finché il client non chiude
    while True:
        data = client_socket.recv(1024).decode()
        if not data:
            # stringa vuota: client chiuso
            break
        print("from " + str(address) + ": " + data)
        client_socket.send("ok".encode())
        client_socket.close()

def server_program():
    host = "0.0.0.0"
    port = 8080
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(MAX_CLIENTS)

    while True:
        client_socket, address = server_socket.accept()
        print("Connection from: " + str(address))
        # ogni nuovo client viene servito da un thread dedicato
        t = threading.Thread(target=handle_client, args=(client_socket, address))
        t.start()

if __name__ == "__main__":
    server_program()
```

*Server TCP in Python, con un thread per client*

## 17.3 Il modulo struct

Come in C, anche in Python si può definire il protocollo con messaggi strutturati. Il modulo **struct** converte valori Python in sequenze di byte e viceversa, usando una **stringa di formato** come descrizione compatta della struttura C. Le funzioni principali sono tre.

| FUNZIONE                                | CHE COSA FA                                          |
|-----------------------------------------|------------------------------------------------------|
| <code>pack(formato, v1, v2, ...)</code> | Converte i valori in byte secondo il formato         |
| <code>unpack(formato, buffer)</code>    | Converte i byte in una tupla di valori Python        |
| <code>calcsize(formato)</code>          | Calcola la dimensione in byte dei dati impacchettati |

Esistono anche `pack_into` e `unpack_from`, che lavorano su un buffer già esistente. La stringa di formato è composta da un carattere per l'**ordine dei byte** seguito dai caratteri dei **tipi** dei campi.

| CARATTERE | ORDINE DEI BYTE | NOTE                                             |
|-----------|-----------------|--------------------------------------------------|
| @         | Nativo          | Dipende dalla piattaforma, con allineamento      |
| <         | Little endian   | Come i processori x86                            |
| >         | Big endian      | Ordine di rete                                   |
| !         | Rete            | Equivale a big endian: è quello da usare in rete |

| CARATTERE | TIPO C                                              | BYTE |
|-----------|-----------------------------------------------------|------|
| B         | uint8_t                                             | 1    |
| H         | uint16_t                                            | 2    |
| I         | uint32_t                                            | 4    |
| f         | float                                               | 4    |
| s         | char[] (preceduto dalla lunghezza, per esempio 32s) | 32   |

Con il formato **!H I B f B B** si descrive una struttura di  $2 + 4 + 1 + 4 + 1 + 1 = \mathbf{13 \text{ byte}}$  in ordine di rete: con i caratteri **!** e **>** il modulo non inserisce alcun riempimento, esattamente come `pragma pack(1)` in C.

## 17.4 Dal Python al C e dal C al Python

Nel primo esempio un mittente Python invia in UDP una lettura di un sensore, impacchettata con il formato appena visto, a un ricevitore C che usa la struttura equivalente.

```
# sensore.py: mittente Python che invia una lettura a un ricevitore C (UDP)
import socket
import struct

# ordine di rete (!) e campi: uint16 uint32 uint8 float uint8 uint8
FORMATO = "!H I B f B B"          # 2 + 4 + 1 + 4 + 1 + 1 = 13 byte

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
dati = struct.pack(FORMATO,
                    1234,           # ID
                    1689424245,     # timestamp
                    1,             # stato
                    23.5,          # temperatura
                    65,            # umidità
                    95)            # qualità dell'aria
sock.sendto(dati, ("127.0.0.1", 8080))
print("inviati", struct.calcsize(FORMATO), "byte")
```

*Mittente Python*

```

/* ricevitore.c: riceve in UDP la lettura inviata da sensore.py.
   compilare: gcc ricevitore.c -o ricevitore */
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#pragma pack(push, 1)          /* stessa disposizione del formato Python */
typedef struct {
    uint16_t ID;
    uint32_t timestamp;
    uint8_t stato;
    float temperature;
    uint8_t humidity;
    uint8_t quality;
} SensorRead;                  /* 13 byte */
#pragma pack(pop)

/* porta i campi dall'ordine di rete all'ordine della macchina */
void handle_packet(SensorRead *data) {
    data->ID = ntohs(data->ID);
    data->timestamp = ntohl(data->timestamp);

    /* il float non ha una funzione dedicata: lo tratto come 4 byte interi */
    uint32_t temp;
    memcpy(&temp, &data->temperature, 4);
    temp = ntohl(temp);
    memcpy(&data->temperature, &temp, 4);

    printf("ID %u, temperatura %.2f, umidità %u, qualità %u\n",
           data->ID, data->temperature, data->humidity, data->quality);
}

int main() {
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in addr;
    SensorRead lettura;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        return 1;
    }

    /* accetto solo datagrammi della dimensione giusta */
    if (recvfrom(sockfd, &lettura, sizeof(lettura), 0, NULL, NULL) == sizeof(lettura)) {
        handle_packet(&lettura);
    }
    close(sockfd);
    return 0;
}

```

*Ricevitore C: la struttura deve avere la stessa disposizione del formato Python*

Il campo float richiede un trattamento particolare: non esiste una funzione di conversione per i numeri in virgola mobile, ma poiché un float occupa 4 byte lo si copia in un intero a 32 bit, si converte l'intero con `ntohl` e lo si ricopia nel float.

Nel secondo esempio i ruoli si invertono: un client C converte i campi nell'ordine di rete e invia la struttura a un server Python, che la spacchetta con `unpack`.

```
/* client_sensore.c: invia una lettura al server Python in UDP.
   compilare: gcc client_sensore.c -o client_sensore */
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <time.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#pragma pack(push, 1)
typedef struct {
    uint16_t ID; /* 2 byte */
    uint32_t last_timestamp; /* 4 byte */
    uint8_t stato; /* 1 byte */
    float temperature; /* 4 byte */
    uint8_t humidity; /* 1 byte */
    uint8_t quality; /* 1 byte */
} SensorRead; /* totale: 13 byte */
#pragma pack(pop)

int main() {
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in server_addr;

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr);

    SensorRead data = {
        .ID = 1234,
        .last_timestamp = (uint32_t)time(NULL),
        .stato = 1,
        .temperature = 23.5f,
        .humidity = 65,
        .quality = 95
    };

    /* conversione nell'ordine di rete prima dell'invio */
    data.ID = htons(data.ID);
    data.last_timestamp = htonl(data.last_timestamp);
    uint32_t temp;
    memcpy(&temp, &data.temperature, sizeof(float));
    temp = htonl(temp);
    memcpy(&data.temperature, &temp, sizeof(float));
}
```

```

sendto(sockfd, &data, sizeof(data), 0,
        (struct sockaddr *)&server_addr, sizeof(server_addr));
close(sockfd);
return 0;
}

```

*Client C*

```

# server_sensori.py: ricevitore Python per le letture inviate da un client C
import socket
import struct

FORMATO = "!H I B f B B"          # 13 byte, ordine di rete

def start_server(host="0.0.0.0", port=8080):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((host, port))
    while True:
        data, addr = sock.recvfrom(13)          # attendo esattamente 13 byte
        if len(data) != 13:
            print("dimensione non valida:", len(data))
            continue
        # unpack restituisce una tupla, nell'ordine dei campi
        ident, ts, stato, temp, umid, qual = struct.unpack(FORMATO, data)
        print(f"ID {ident}, temperatura {temp:.2f}")

if __name__ == "__main__":
    start_server()

```

*Server Python*

#### DA RICORDARE

Perché due programmi in linguaggi diversi si capiscano devono concordare tre cose: l'**ordine dei campi**, la **dimensione** di ciascuno (niente riempimento) e l'**ordine dei byte** (sempre quello di rete). Il formato `!H I B f B B` in Python e la struttura con pragma `pack(1)` e conversioni `htons`, `htonl` in C sono due descrizioni dello stesso protocollo.

# Esercizi proposti

Gli esercizi di autovalutazione del corso, con alcune indicazioni di progetto. Conviene svilupparli anche in linguaggi diversi e provarli su un'infrastruttura di rete, anche virtuale.

## 18.1 Progettare una rete

Si progetti una topologia con le seguenti caratteristiche: una rete domestica con cinque dispositivi; la rete domestica collegata alla rete cittadina; nella rete cittadina altre due reti domestiche; la rete cittadina collegata a Internet attraverso un gateway; a un certo punto, la rete di YouTube collegata a Internet, con due nodi, il server di streaming e il database. Per ogni rete si indichino indirizzi, netmask e gateway, e si costruiscano le tabelle di routing, le tabelle ARP e le tabelle MAC dei dispositivi. Il capitolo 8 ne svolge una versione completa.

## 18.2 Implementare lo stack di rete

Con un linguaggio a oggetti si progetti un software che emuli lo stack TCP/IP: due nodi, come processi o thread separati, ciascuno con la propria istanza dello stack, si scambiano una stringa.

- Una classe **Application**, estendibile o astratta.
- Una classe **Transport** che prende i dati dall'applicazione e li divide in oggetti **Segment**, per esempio in base a una dimensione massima.
- Una classe **Internetwork** che crea oggetti **Packet** a partire dai segmenti.
- Una classe **NetworkAccess** che crea oggetti **Frame** con i pacchetti all'interno e li deposita in una memoria condivisa, che fa le veci del mezzo fisico; ogni istanza controlla se nella memoria sono arrivati nuovi frame.
- Una classe **Device** con un'istanza di ciascun livello, usate per incapsulare e decapsulare le PDU; i livelli sono collegati fra loro, per esempio tramite riferimenti o puntatori.

## 18.3 Esercizio 1: sensori

Si sviluppi un sistema in cui un **nodo sensore** invia dati a un **nodo centrale**, che al verificarsi di una condizione invia un allarme a un **nodo di controllo**.

- **Sensore**: invia dati ogni 3 secondi; la comunicazione può non essere affidabile; i dati sono temperatura, umidità e qualità dell'aria; ogni sensore ha un identificativo univoco.
- **Nodo centrale**: registra ogni nuovo sensore al primo messaggio ricevuto; se la temperatura supera 30 gradi o la qualità dell'aria è scarsa invia un messaggio al nodo di controllo; questa comunicazione deve essere affidabile.
- **Nodo di controllo**: stampa gli allarmi e li registra in un file di log.

Il capitolo 19 ne presenta una soluzione completa.

## 18.4 Esercizio 2: sasso, carta, forbici

Si crei un gioco in rete. Il **giocatore** si collega a un server, si registra con email e password e deve effettuare il login per qualunque azione; può cercare un avversario mettendosi, per sua scelta, in stato di attesa; quando il server trova un avversario la partita comincia, e una partita è composta da tre turni. Il **server di gioco** gestisce registrazione e autenticazione, si occupa dell'abbinamento dei giocatori (prendendo i primi due in attesa oppure due a caso fra quelli in attesa), è l'autorità che controlla lo stato della partita e la validità delle mosse e, a fine partita, riporta i giocatori nella lobby (non in attesa).

### DA RICORDARE

**Indicazioni di progetto.** Le comunicazioni richiedono affidabilità, quindi TCP, con un thread per giocatore come nella chat del capitolo 21. Lo stato di ogni giocatore (non autenticato, lobby, in attesa, in partita) va conservato nella struttura dell'utente e protetto da mutex; l'abbinamento si fa quando un secondo giocatore entra in attesa. Il server confronta le due mosse di ogni turno e comunica l'esito a entrambi.

## 18.5 Esercizio 3: sensori affidabili

Come l'esercizio 1, con alcune differenze. La comunicazione del sensore **deve** essere affidabile; se la temperatura supera 30 gradi o la qualità dell'aria è scarsa il sensore entra in **modalità allarme**, in cui non può inviare dati, e torna alla modalità normale quando riceve un comando di stop. Il nodo centrale, oltre a quanto già visto, inoltra al sensore interessato il comando di stop inviato dal nodo di controllo. Il nodo di controllo, oltre a stampare e registrare gli allarmi, dopo 5 secondi dall'ingresso di un sensore in modalità allarme lo fa uscire inviando il comando di stop.

### DA RICORDARE

**Indicazioni di progetto.** Tutte le comunicazioni diventano TCP, e il nodo centrale deve poter scrivere a ogni sensore: conserva quindi il socket di ciascuno, associato al suo identificativo. Il nodo di controllo, alla ricezione di un allarme, può avviare un thread che attende 5 secondi con sleep e poi invia il comando di stop, che il centrale inoltra al sensore giusto.

## 18.6 Esercizio 4: servizio di chat

Si crei un servizio di chat in cui gli utenti si registrano con email e password, effettuano il login e scrivono a uno specifico utente, se collegato. Il server gestisce registrazione e autenticazione e il sistema di comunicazione fra gli utenti collegati; i dati degli utenti sono salvati in un file, che il server ricarica all'avvio. Il capitolo 21 ne presenta una soluzione completa.

## 18.7 Esercizio 5: chat peer to peer

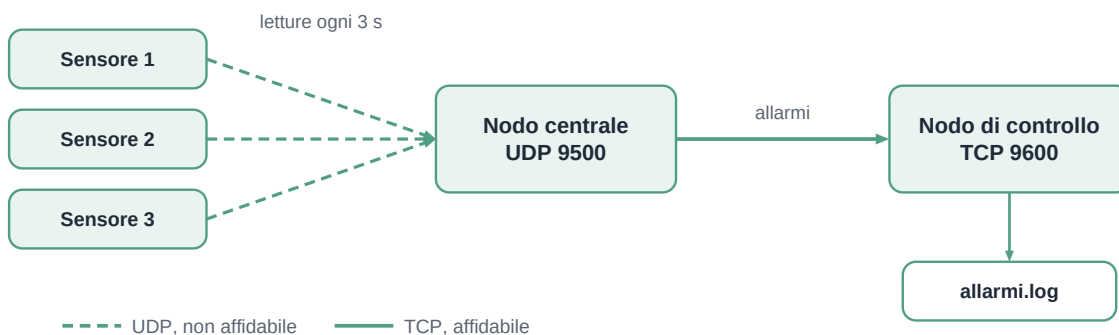
Come l'esercizio 4, ma i messaggi fra utenti viaggiano in **P2P**, senza passare dal server, e con una comunicazione **non affidabile**. Per farlo i client comunicano al server la propria porta di comunicazione, e il server scambia fra i client le informazioni necessarie a parlarsi direttamente. Il capitolo 22 ne presenta una soluzione completa.

# Esercizio svolto: sensori e nodo centrale

Una soluzione completa dell'esercizio 1: i sensori inviano letture in UDP, il nodo centrale le valuta e inoltra gli allarmi in TCP al nodo di controllo, che li stampa e li registra su file.

## 19.1 Le scelte di progetto

La traccia indica già quale trasporto usare. Le letture dei sensori possono andare perse senza danno, perché ne arriva una nuova ogni 3 secondi: si usa **UDP**. Gli allarmi invece non devono andare persi: fra nodo centrale e nodo di controllo si usa **TCP**. Ne derivano i ruoli: il nodo centrale è un server UDP per i sensori e un client TCP per il controllo, mentre il nodo di controllo è un server TCP.



*I sensori inviano letture in UDP al nodo centrale, che inoltra gli allarmi in TCP al nodo di controllo*

Per evitare la conversione dei numeri in virgola mobile, la temperatura viaggia come intero in **decimi di grado** (315 significa 31,5 gradi): basta così htons. La qualità dell'aria è un codice: buona, media o scarsa.

```

/* sensori.h: protocollo comune a sensore, nodo centrale e nodo di controllo */
#ifndef SENSORI_H
#define SENSORI_H

#include <stdint.h>

#define IP_CENTRALE      "127.0.0.1"
#define PORTA_CENTRALE  9500      /* UDP: letture dai sensori      */
#define IP_CONTROLLO    "127.0.0.1"
#define PORTA_CONTROLLO 9600      /* TCP: allarmi verso il controllo */

#define QUALITA_BUONA    0
#define QUALITA_MEDIA    1
#define QUALITA_SCARSA   2

#define MAX_SENSORI      50
#define DESCR_LEN        100

#pragma pack(push, 1)

/* lettura inviata dal sensore (UDP, non affidabile).
   La temperatura viaggia in decimi di grado come intero:
   così basta htons, senza trucchi per i float.          */
typedef struct {

```

```

    uint16_t id;           /* identificativo univoco del sensore */
    int16_t  temperatura; /* decimi di grado: 315 vale 31,5 */
    uint8_t  umidita;      /* percentuale */
    uint8_t  qualita;      /* QUALITA_BUONA, MEDIA o SCARSA */
} lettura;                /* 6 byte */

/* allarme inviato dal nodo centrale al controllo (TCP, affidabile) */
typedef struct {
    uint16_t id;
    int16_t  temperatura;
    uint8_t  qualita;
    char      descrizione[DESCR_LEN];
} allarme;                /* 105 byte */

#pragma pack(pop)

#endif

```

*Il protocollo comune ai tre programmi*

## 19.2 Il sensore

Il sensore riceve il proprio identificativo dalla riga di comando, genera letture casuali e le invia ogni 3 secondi al nodo centrale con sendto. Non deve fare bind: non riceve nulla.

```

/* sensore.c: nodo sensore. Ogni 3 secondi invia una lettura al nodo
   centrale in UDP: la comunicazione può non essere affidabile.
   uso:          ./sensore <id>
   compilare:    gcc sensore.c -o sensore */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "sensori.h"

int main(int argc, char *argv[]) {
    int sockfd;
    struct sockaddr_in centrale;
    lettura l;
    int id;

    if (argc != 2) {
        printf("uso: %s <id>\n", argv[0]);
        exit(1);
    }
    id = atoi(argv[1]);
    srand(time(NULL) + id); /* valori casuali diversi per ogni sensore */

    /* in UDP basta la socket: nessuna connessione da aprire */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("socket");
    }
}

```

```

        exit(1);
    }

    memset(&centrale, 0, sizeof(centrale));
    centrale.sin_family = AF_INET;
    centrale.sin_port = htons(PORTA_CENTRALE);
    centrale.sin_addr.s_addr = inet_addr(IP_CENTRALE);

    while (1) {
        /* simulo una lettura: temperatura da 20,0 a 34,9 gradi */
        l.id = htons(id); /* campi a 16 bit: htons */
        l.temperatura = htons(200 + rand() % 150);
        l.umidita = rand() % 101;
        l.qualita = rand() % 3;

        sendto(sockfd, &l, sizeof(l), 0,
                (struct sockaddr *)&centrale, sizeof(centrale));
        printf("inviata lettura: %.1f gradi, qualità %d\n",
                (short)ntohs(l.temperatura) / 10.0, l.qualita);
        fflush(stdout);

        sleep(3); /* una lettura ogni 3 secondi */
    }

    close(sockfd);
    return 0;
}

```

*Il nodo sensore*

## 19.3 Il nodo centrale

All'avvio il nodo centrale si collega al nodo di controllo, che deve quindi essere già in esecuzione, e apre il socket UDP per le letture. In un unico ciclo riceve le letture, registra i sensori nuovi e, se la temperatura supera 300 decimi o la qualità è scarsa, compone un allarme e lo invia sulla connessione TCP.

```

/* centrale.c: nodo centrale. Riceve le letture dei sensori in UDP,
registra i sensori nuovi e, se la temperatura supera 30 gradi o
la qualità dell'aria è scarsa, invia un allarme al nodo di
controllo su una connessione TCP (affidabile).
compilare: gcc centrale.c -o centrale */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "sensori.h"

int sensori[MAX_SENSORI]; /* ID dei sensori già conosciuti */
int numero_sensori = 0;

/* registra il sensore se è la prima volta che scrive */
void registra_sensore(int id) {
    for (int i = 0; i < numero_sensori; i++) {

```

```

        if (sensori[i] == id) return;          /* già noto */
    }
    if (numero_sensori < MAX_SENSORI) {
        sensori[numero_sensori] = id;
        numero_sensori++;
        printf("nuovo sensore registrato: %d\n", id);
    }
}

int main() {
    int sock_udp, sock_tcp;
    struct sockaddr_in mio_addr, controllo;
    lettura l;
    allarme a;

    /* 1. connessione TCP verso il controllo: deve essere già avviato */
    sock_tcp = socket(AF_INET, SOCK_STREAM, 0);
    memset(&controllo, 0, sizeof(controllo));
    controllo.sin_family = AF_INET;
    controllo.sin_port = htons(PORTA_CONTROLLO);
    controllo.sin_addr.s_addr = inet_addr(IP_CONTROLLO);
    if (connect(sock_tcp, (struct sockaddr *)&controllo, sizeof(controllo)) < 0) {
        perror("connect al controllo");
        exit(1);
    }

    /* 2. socket UDP su cui arrivano le letture */
    sock_udp = socket(AF_INET, SOCK_DGRAM, 0);
    memset(&mio_addr, 0, sizeof(mio_addr));
    mio_addr.sin_family = AF_INET;
    mio_addr.sin_addr.s_addr = INADDR_ANY;
    mio_addr.sin_port = htons(PORTA_CENTRALE);
    if (bind(sock_udp, (struct sockaddr *)&mio_addr, sizeof(mio_addr)) < 0) {
        perror("bind");
        exit(1);
    }
    printf("nodo centrale in ascolto sulla porta %d\n", PORTA_CENTRALE);
    fflush(stdout);

    while (1) {
        /* scarto i datagrammi che non hanno la dimensione di una lettura */
        if (recvfrom(sock_udp, &l, sizeof(l), 0, NULL, NULL) != sizeof(l)) {
            continue;
        }
        int id = ntohs(l.id);
        int temp = (short)ntohs(l.temperatura);    /* decimi di grado */
        registra_sensore(id);

        /* condizione di allarme: più di 30 gradi o aria scarsa */
        if (temp > 300 || l.qualita == QUALITA_SCARSA) {
            memset(&a, 0, sizeof(a));
            a.id = l.id;                          /* già in ordine di rete */
            a.temperatura = l.temperatura;
            a.qualita = l.qualita;
            if (temp > 300)
                strcpy(a.descrizione, "temperatura oltre 30 gradi");
            else
                strcpy(a.descrizione, "qualità dell'aria scarsa");
        }
    }
}

```

```

        send(sock_tcp, &a, sizeof(a), 0);          /* TCP: consegna garantita */
        printf("allarme inviato per il sensore %d\n", id);
        fflush(stdout);
    }
}

close(sock_udp);
close(sock_tcp);
return 0;
}

```

*Il nodo centrale*

## 19.4 Il nodo di controllo

Il nodo di controllo è un server TCP con un solo interlocutore. Riceve gli allarmi con MSG\_WAITALL, perché sulla connessione TCP un allarme potrebbe arrivare spezzato, li stampa e li aggiunge al file allarmi.log, aperto in modalità append così che il file cresca a ogni allarme senza essere sovrascritto.

```

/* controllo.c: nodo di controllo. Attende la connessione TCP del nodo
   centrale, stampa ogni allarme e lo aggiunge al file allarmi.log.
   compilare: gcc controllo.c -o controllo */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "sensori.h"

int main() {
    int server_fd, centrale_fd;
    struct sockaddr_in addr;
    allarme a;
    FILE *log;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(PORTA_CONTROLLO);

    if (bind(server_fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        exit(1);
    }
    listen(server_fd, 1);
    printf("controllo in attesa del nodo centrale sulla porta %d\n", PORTA_CONTROLLO);
    fflush(stdout);

    /* un solo interlocutore: il nodo centrale */
    centrale_fd = accept(server_fd, NULL, NULL);
    printf("nodo centrale collegato\n");
}

```

```

/* MSG_WAITALL: recv attende l'allarme intero, anche se TCP lo
   consegna a pezzi (TCP è un flusso di byte, non di messaggi) */
while (recv(centrale_fd, &a, sizeof(a), MSG_WAITALL) == sizeof(a)) {
    printf("ALLARME sensore %d: %s (%.1f gradi)\n",
           ntohs(a.id), a.descrizione, (short)ntohs(a.temperatura) / 10.0);
    fflush(stdout);

    /* "a" = append: aggiungo in fondo senza cancellare il file */
    log = fopen("allarmi.log", "a");
    if (log != NULL) {
        fprintf(log, "sensore %d: %s (%.1f gradi)\n",
               ntohs(a.id), a.descrizione, (short)ntohs(a.temperatura) / 10.0);
        fclose(log);
    }
}

printf("nodo centrale disconnesso\n");
close(centrale_fd);
close(server_fd);
return 0;
}

```

*Il nodo di controllo*

```

gcc controllo.c -o controllo
gcc centrale.c -o centrale
gcc sensore.c -o sensore
./controllo          # primo terminale: deve partire per primo
./centrale           # secondo terminale
./sensore 1          # terzo terminale
./sensore 2          # quarto terminale

```

*Compilazione e ordine di avvio*

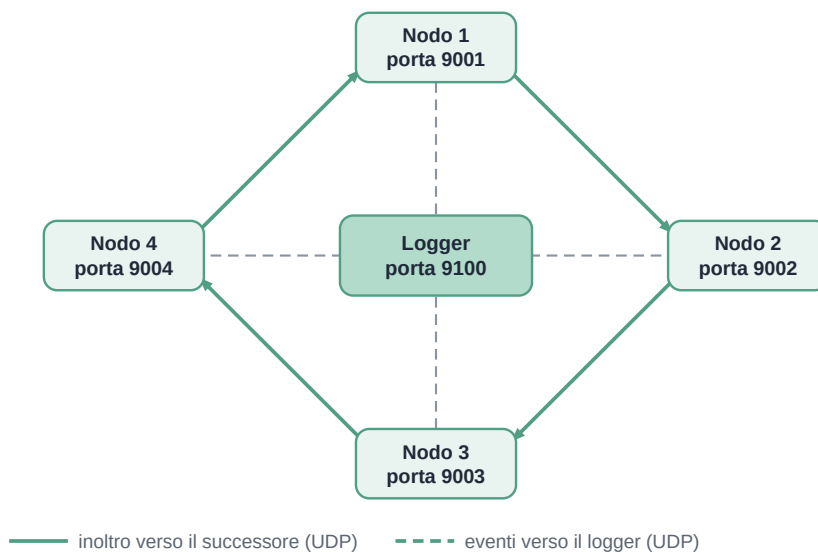
# Esercizio svolto: rete di messaggistica ad anello

Una rete senza server centrale, in cui ogni nodo conosce soltanto il successivo: i messaggi diretti viaggiano di nodo in nodo fino al destinatario, i broadcast fanno il giro completo, e un servizio di logging osserva gli eventi.

## 20.1 La traccia

Si realizzi una rete di messaggistica **decentralizzata ad anello** con le seguenti caratteristiche.

- Ogni nodo è identificato da un **ID** e conosce soltanto il **nodo successivo**, di cui riceve indirizzo IP e porta all'avvio insieme al proprio ID e alla propria porta.
- L'utente di un nodo può inviare un **messaggio diretto** a un altro nodo, che viene inoltrato di nodo in nodo fino al destinatario, oppure un **broadcast**, che deve essere letto da tutti i nodi e fermarsi quando ritorna al mittente.
- Le comunicazioni fra i nodi **non sono affidabili**.
- Un **servizio di logging**, con un indirizzo noto a tutti, riceve dai nodi la segnalazione di ogni evento.
- Un nodo interrotto con Ctrl+C deve chiudersi in modo pulito.



Quattro nodi disposti ad anello, ciascuno collegato al successivo, e il servizio di logging che riceve gli eventi da tutti

## 20.2 Le scelte di progetto

Le comunicazioni non affidabili indicano **UDP**. Con UDP un solo socket basta per tutto: il nodo riceve dal predecessore, invia al successore e invia al logger, indicando ogni volta la destinazione nella sendto. Il nodo deve fare due cose contemporaneamente, leggere la tastiera e ricevere i messaggi, e poiché `recvfrom` è bloccante la ricezione va in un **thread** separato. Si usano i thread e non fork perché i due flussi devono condividere socket, ID e indirizzi.

Il cuore del protocollo è la logica di ricezione, che si riduce a due domande.

| TIPO DI MESSAGGIO | CONDIZIONE                      | AZIONE                                           |
|-------------------|---------------------------------|--------------------------------------------------|
| <b>Broadcast</b>  | Il mittente sono io             | Il messaggio ha fatto il giro completo: lo fermo |
| <b>Broadcast</b>  | Il mittente è un altro nodo     | Lo leggo e lo inoltro al successore              |
| <b>Diretto</b>    | Il destinatario sono io         | Lo leggo: è arrivato                             |
| <b>Diretto</b>    | Il destinatario è un altro nodo | Lo inoltro senza leggerlo: faccio da router      |

Resta un caso da gestire: un messaggio diretto a un ID che non esiste girerebbe per sempre. Come nel campo TTL dell'instestazione IP, ogni messaggio porta un **contatore dei salti**, incrementato a ogni inoltro: oltre il massimo il messaggio viene scartato.

## 20.3 Il protocollo

```
#ifndef PROTOCOLLO_H
#define PROTOCOLLO_H

#include <stdint.h>

/* servizio di logging
   indirizzo noto a tutti i nodi: è l'unico punto fisso della rete */
#define IP_LOG    "127.0.0.1"
#define PORTA_LOG 9100

#define MSG_LEN    256    /* testo del messaggio */
#define LOG_LEN    200    /* descrizione dell'evento */

/* tipi di messaggio che circolano nell'anello */
#define TIPO_DIRETTO 0    /* per un nodo preciso */
#define TIPO_BROADCAST 1 /* per tutti i nodi */

/* numero massimo di inoltri: rete di sicurezza contro un messaggio
   indirizzato a un ID inesistente, che altrimenti girerebbe per
   sempre. È lo stesso principio del campo TTL di IP. */
#define MAX_SALTI 32

/* pragma pack(1): niente byte di riempimento fra i campi.
   Mando in rete i byte grezzi della struct, quindi mittente e
   destinatario devono vederla identica byte per byte.
   uint8_t è garantito 1 byte su qualunque macchina. */
#pragma pack(push, 1)

/* messaggio che viaggia nell'anello: nodo → nodo successivo */
typedef struct {
    uint8_t tipo;           /* TIPO_DIRETTO o TIPO_BROADCAST */
    uint8_t mittente;       /* chi l'ha originato: serve a fermare
                             il broadcast quando torna a casa */
    uint8_t destinatario;   /* usato solo nei messaggi diretti */
    uint8_t salti;          /* quanti inoltri ha già fatto */
    char    testo[MSG_LEN];
} messaggio;               /* 4 + 256 = 260 byte */

/* messaggio di servizio: nodo → logger */
typedef struct {
    uint8_t nodo;           /* quale nodo segnala l'evento */

```

```

    char    evento[LOG_LEN];
} log_msg;                                /* 1 + 200 = 201 byte */

#pragma pack(pop)

#endif

```

*I messaggi dell'anello e del servizio di logging*

## 20.4 Il nodo

```

/* nodo.c: nodo di una rete di messaggistica decentralizzata ad anello.
   Ogni nodo conosce SOLO il proprio successore: un messaggio diretto a
   un nodo lontano viene inoltrato di vicino in vicino finchè non
   arriva. Non esiste un server centrale: tutti i nodi sono uguali ed
   eseguono questo stesso programma (architettura peer to peer).
   Tutte le comunicazioni sono NON affidabili, quindi UDP.

   uso:      ./nodo <mio_id> <mia_porta> <ip_successore> <porta_successore>
   esempio:  ./nodo 1 9001 127.0.0.1 9002
   compilare: gcc nodo.c -o nodo -Wall -pthread */

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

#include "protocollo.h"

/* variabili globali
   globali perchè servono sia al main (che legge la tastiera) sia al
   thread ricevitore, che girano in parallelo. */

int sockfd;    /* UNA SOLA socket per tutto: riceve dal predecessore,
               invia al successore, invia al logger. In UDP la
               socket non è legata a un interlocutore: il
               destinatario si indica in ogni sendto. */
int mio_id;    /* il mio identificativo nell'anello */

struct sockaddr_in succ_addr; /* indirizzo del nodo successivo */
struct sockaddr_in log_addr;  /* indirizzo del servizio di log */

/* invio di un evento al logger
   è una sendto UDP "spara e dimentica": non aspetta risposta e non
   blocca nulla, quindi non intacca il funzionamento principale. */
void logga(const char *evento) {
    log_msg log;

    /* azzero prima di riempire, altrimenti nei byte non scritti
       resterebbe spazzatura di memoria */
    memset(&log, 0, sizeof(log));
    log.nodo = mio_id;

```

```

strcpy(log.evento, evento);

sendto(sockfd, &log, sizeof(log), 0,
        (struct sockaddr *)&log_addr, sizeof(log_addr));
}

/* chiusura pulita
   chiamata sia dal comando di uscita sia da Ctrl+C: segnala l'evento
   al logger e chiude la socket prima di terminare. */
void chiudi(int segnale) {
    logga("nodo in chiusura");
    close(sockfd);
    printf("\nsocket chiusa, nodo %d terminato\n", mio_id);
    exit(0);
}

/* creazione della socket
   socket + bind: in UDP non serve la listen perchè non ci sono
   connessioni da accettare. */
int crea_socket(uint16_t porta) {
    int fd;
    struct sockaddr_in mio_addr;

    fd = socket(AF_INET, SOCK_DGRAM, 0);      /* SOCK_DGRAM = UDP */
    if (fd < 0) {
        perror("socket");
        exit(1);
    }

    memset(&mio_addr, 0, sizeof(mio_addr));
    mio_addr.sin_family = AF_INET;           /* IPv4 */
    mio_addr.sin_addr.s_addr = INADDR_ANY;   /* qualunque interfaccia */
    mio_addr.sin_port = htons(porta);        /* htons: host → rete */

    /* bind = "sto in ascolto qui": mi rende raggiungibile */
    if (bind(fd, (struct sockaddr *)&mio_addr, sizeof(mio_addr)) < 0) {
        perror("bind");
        exit(1);
    }

    return fd;
}

/* inoltra al successore
   unica direzione possibile: nell'anello non esistono scorciatoie.
   Incremento il contatore dei salti e scarto il messaggio se ha
   girato troppo (ID destinatario inesistente). */
void inoltra(messaggio *msg) {
    msg->salti = msg->salti + 1;

    if (msg->salti > MAX_SALTI) {
        printf("\n[messaggio scartato: troppi salti, destinatario inesistente]\n");
        fflush(stdout);
        logga("messaggio scartato: superato il numero massimo di salti");
        return;
    }

    sendto(sockfd, msg, sizeof(messaggio), 0,

```

```

        (struct sockaddr *)&succ_addr, sizeof(succ_addr));
}

/* thread ricevitore: IL CUORE DEL PROTOCOLLO
   gira in parallelo al main. Serve un thread separato perchè la
   recvfrom è bloccante: in un flusso solo il nodo resterebbe fermo
   in attesa e non potrebbe più leggere la tastiera. */
void *ricevitore(void *arg) {
    messaggio msg;
    struct sockaddr_in mittente;
    socklen_t len;

    while (1) {
        len = sizeof(mittente);

        /* bloccato qui finchè il predecessore non manda qualcosa */
        if (recvfrom(sockfd, &msg, sizeof(msg), 0,
                    (struct sockaddr *)&mittente, &len) <= 0) {
            continue;
        }

        /* PRIMA DOMANDA: è un broadcast o un messaggio diretto? */
        if (msg.tipo == TIPO_BROADCAST) {

            /* SECONDA DOMANDA: l'ho mandato io? */
            if (msg.mittente == mio_id) {
                /* ha percorso tutto l'anello ed è tornato a casa:
                   STOP. Senza questa condizione girerebbe sempre. */
                printf("\n[broadcast tornato al mittente: giro completo]\n");
                fflush(stdout);
                logga("broadcast completato, giro chiuso");
            } else {
                /* il broadcast è destinato anche a me: lo LEGGO e
                   poi lo INOLTRO, perchè deve raggiungere gli altri.
                   Unico caso in cui il nodo fa entrambe le cose. */
                printf("\n[broadcast da nodo %d] %s\n", msg.mittente, msg.testo);
                fflush(stdout);
                logga("ricevuto broadcast, inoltrato al successore");
                inoltra(&msg);
            }
        } else { /* TIPO_DIRETTO */

            /* SECONDA DOMANDA: sono io il destinatario? */
            if (msg.destinatario == mio_id) {
                /* arrivato a destinazione: STOP */
                printf("\n[messaggio da nodo %d] %s\n", msg.mittente, msg.testo);
                fflush(stdout);
                logga("ricevuto messaggio diretto");
            } else {
                /* non è per me, sono solo di passaggio: inoltra
                   senza leggerlo. Qui il nodo fa da router. */
                printf("\n[in transito: da %d verso %d]\n",
                    msg.mittente, msg.destinatario);
                fflush(stdout);
                logga("messaggio in transito, inoltrato al successore");
                inoltra(&msg);
            }
        }
    }
}

```

```

    }
}

return NULL;
}

/* lettura di una riga da tastiera
fgets prende tutta la riga spazi compresi (scanf("%s") si fermerebbe
al primo spazio) ma lascia dentro l'a capo: strcspn me ne dà la
posizione e ci scrivo il terminatore. */
void leggi_riga(char *buffer, int dimensione) {
    if (fgets(buffer, dimensione, stdin) == NULL) {
        chiudi(0); /* fine input: esco pulito */
    }
    buffer[strcspn(buffer, "\n")] = 0;
}

int main(int argc, char *argv[]) {
    pthread_t thread_ricevitore;
    messaggio msg;
    char riga[MSG_LEN];
    int mia_porta, porta_succ, destinatario;

    /* CONTROLLO DEGLI INPUT DI AVVIO */

    if (argc != 5) {
        printf("uso: %s <mio_id> <mia_porta> <ip_successore> <porta_successore>\n",
            argv[0]);
        printf("esempio: %s 1 9001 127.0.0.1 9002\n", argv[0]);
        exit(1);
    }

    mio_id = atoi(argv[1]);
    mia_porta = atoi(argv[2]);
    porta_succ = atoi(argv[4]);

    if (mio_id < 1 || mio_id > 255) {
        printf("errore: l'ID deve essere compreso fra 1 e 255\n");
        exit(1);
    }

    if (mia_porta < 1024 || mia_porta > 65535) {
        printf("errore: la mia porta deve essere compresa fra 1024 e 65535\n");
        exit(1);
    }

    if (porta_succ < 1024 || porta_succ > 65535) {
        printf("errore: la porta del successore deve essere fra 1024 e 65535\n");
        exit(1);
    }

    /* inet_addr restituisce INADDR_NONE se la stringa non è un IP valido */
    if (inet_addr(argv[3]) == INADDR_NONE) {
        printf("errore: indirizzo IP del successore non valido\n");
        exit(1);
    }

    /* PREPARAZIONE DELLA RETE */

```

```

/* Ctrl+C → chiusura pulita della socket */
signal(SIGINT, chiudi);

/* mi metto in ascolto sulla mia porta: da qui sono raggiungibile
   dal mio predecessore */
sockfd = crea_socket(mia_porta);

/* indirizzo del successore: preso dai parametri di avvio, non
   calcolato. Così i nodi possono stare anche su macchine
   diverse, basta passare l'IP giusto. */
memset(&succ_addr, 0, sizeof(succ_addr));
succ_addr.sin_family = AF_INET;
succ_addr.sin_port = htons(porta_succ);
succ_addr.sin_addr.s_addr = inet_addr(argv[3]); /* stringa → numero */

/* indirizzo del servizio di logging: fisso e noto a tutti.
   ATTENZIONE: blocco quasi identico al precedente, è facile
   scrivere succ_addr per distrazione. Compila lo stesso, ma i
   messaggi finirebbero tutti al logger. */
memset(&log_addr, 0, sizeof(log_addr));
log_addr.sin_family = AF_INET;
log_addr.sin_port = htons(PORTA_LOG);
log_addr.sin_addr.s_addr = inet_addr(IP_LOG);

printf("nodo %d avviato sulla porta %d\n", mio_id, mia_porta);
printf("successore: %s:%d\n\n", argv[3], porta_succ);
fflush(stdout);

/* avvio il ricevitore in parallelo.
   Thread e non fork perchè i thread CONDIVIDONO la memoria del
   processo: socket, ID e indirizzi devono essere visti da
   entrambi i flussi. Con fork sarebbero copie separate. */
pthread_create(&thread_ricevitore, NULL, ricevitore, NULL);

logga("nodo avviato");

/* CICLO DI INTERAZIONE CON L'UTENTE */

while (1) {

    printf("\nID destinatario (0 = broadcast, -1 = esci): ");
    fflush(stdout);
    leggi_riga(riga, sizeof(riga));

    destinatario = atoi(riga);

    if (destinatario == -1) {
        chiudi(0); /* chiusura pulita richiesta */
    }

    if (destinatario < 0 || destinatario > 255) {
        printf("ID non valido\n");
        continue; /* ricomincio il ciclo */
    }

    printf("messaggio: ");
    fflush(stdout);

```

```

leggi_riga(riga, sizeof(riga));

if (strlen(riga) == 0) {
    printf("messaggio vuoto, annullato\n");
    continue;
}

/* costruisco il messaggio da immettere nell'anello */
memset(&msg, 0, sizeof(msg));
msg.mittente = mio_id;
msg.salti = 0;
strcpy(msg.testo, riga);

if (destinatario == 0) {
    msg.tipo = TIPO_BROADCAST;
    msg.destinatario = 0;          /* non usato nel broadcast */
    logga("inviato broadcast");
} else {
    msg.tipo = TIPO_DIRETTO;
    msg.destinatario = destinatario;
    logga("inviato messaggio diretto");
}

/* parte sempre dal successore, anche se il destinatario è
   lontano: nell'anello non c'è altra strada */
inoltra(&msg);
}

close(sockfd);
return 0;
}

```

*Il nodo dell'anello*

## 20.5 Il servizio di logging

Il logger è un semplice server UDP: resta in ascolto sulla porta nota e stampa gli eventi. Non fa parte dell'anello e non inoltra nulla: se lo si spegne, l'anello continua a funzionare.

```

/* logger.c: servizio di Logging.
   È un semplice server UDP: resta in ascolto su una porta fissa e
   stampa gli eventi che i nodi gli segnalano. Non fa parte
   dell'anello e non inoltra niente: è un osservatore passivo, e se
   lo spengo l'anello continua a funzionare.
   compilare: gcc logger.c -o logger -Wall */

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

#include "protocollo.h"

```

```

int sockfd;      /* globale: serve al gestore di chiusura */

/* chiusura pulita: chiudo la socket prima di uscire */
void chiudi(int segnale) {
    printf("\nchiusura logger, socket chiusa\n");
    close(sockfd);
    exit(0);
}

/* crea la socket UDP e la lega alla porta.
   In UDP non serve la listen: dopo il bind sono già operativo. */
int crea_socket(uint16_t porta) {
    int fd;
    struct sockaddr_in mio_addr;

    fd = socket(AF_INET, SOCK_DGRAM, 0);      /* SOCK_DGRAM = UDP */
    if (fd < 0) {
        perror("socket");
        exit(1);
    }

    memset(&mio_addr, 0, sizeof(mio_addr));
    mio_addr.sin_family = AF_INET;           /* IPv4 */
    mio_addr.sin_addr.s_addr = INADDR_ANY;   /* qualunque interfaccia */
    mio_addr.sin_port = htons(porta);        /* htons: host → rete */

    if (bind(fd, (struct sockaddr *)&mio_addr, sizeof(mio_addr)) < 0) {
        perror("bind");
        exit(1);
    }

    return fd;
}

int main(void) {
    struct sockaddr_in mittente;
    socklen_t len;
    log_msg log;

    /* Ctrl+C → chiusura pulita invece di terminazione brusca */
    signal(SIGINT, chiudi);

    sockfd = crea_socket(PORTA_LOG);
    printf("servizio di logging attivo sulla porta %d\n\n", PORTA_LOG);
    fflush(stdout);

    while (1) {
        len = sizeof(mittente);

        /* resto bloccato qui finchè un nodo non segnala un evento.
           I byte ricevuti finiscono direttamente dentro la struct */
        if (recvfrom(sockfd, &log, sizeof(log), 0,
                     (struct sockaddr *)&mittente, &len) <= 0) {
            continue;
        }

        printf("[nodo %d] %s\n", log.nodo, log.evento);
        fflush(stdout);      /* stampa subito, senza bufferizzare */
    }
}

```

```
}

close(sockfd);
return 0;
}
```

*Il servizio di logging*

## 20.6 Esecuzione

Per provare l'anello su una sola macchina si avvia prima il logger e poi un nodo per terminale, chiudendo il cerchio: ogni nodo indica come successore la porta del nodo seguente, e l'ultimo indica il primo.

```
gcc logger.c -o logger
gcc nodo.c -o nodo -pthread
./logger                                # terminale 1
./nodo 1 9001 127.0.0.1 9002            # terminale 2: il nodo 1 scrive al 2
./nodo 2 9002 127.0.0.1 9003            # terminale 3: il nodo 2 scrive al 3
./nodo 3 9003 127.0.0.1 9004            # terminale 4: il nodo 3 scrive al 4
./nodo 4 9004 127.0.0.1 9001            # terminale 5: il 4 chiude l'anello sull'1
```

*Avvio di un anello di quattro nodi*

Dal nodo 1, un messaggio con destinatario 3 attraversa il nodo 2, che lo segnala come in transito, e viene letto dal nodo 3. Un broadcast (destinatario 0) viene letto dai nodi 2, 3 e 4 e, tornato al nodo 1, viene fermato. Ogni passaggio compare nel terminale del logger.

### DA RICORDARE

Due condizioni tengono in vita l'anello. La prima: il broadcast si ferma **soltanto** quando torna al mittente, quindi il campo mittente non va mai modificato durante gli inoltri. La seconda: l'anello deve essere **chiuso**; se un nodo manca, i messaggi si perdono in silenzio, come è normale con UDP.

# Esercizio svolto: chat con registrazione e login

Una soluzione completa dell'esercizio 4: un server TCP gestisce utenti salvati su file, registrazione, login e inoltra dei messaggi al destinatario scelto.

## 21.1 Le scelte di progetto

La chat richiede comunicazioni affidabili, quindi **TCP**, e un server che serva più utenti contemporaneamente, quindi un **thread per client**. Tutti i messaggi hanno la stessa struttura, con un campo comando che ne indica il significato. Il server mantiene in memoria l'elenco degli utenti, salvato anche nel file utenti.txt (una riga per utente) e ricaricato all'avvio. Ogni utente collegato è associato al proprio socket, così che il server possa inoltrargli i messaggi: è questo a rendere possibile la comunicazione con un utente specifico.

```
/* chat_protocol.h: messaggi scambiati fra client e server della chat */
#ifndef CHAT_PROTOCOL_H
#define CHAT_PROTOCOL_H

#include <stdint.h>

#define PORTA_SERVER 9000
#define IP_SERVER "127.0.0.1"

#define MAX_UTENTI 50
#define EMAIL_LEN 50
#define PASS_LEN 50
#define MSG_LEN 200

/* valori del campo comando */
#define CMD_REGISTRA 1 /* client → server: nuova registrazione */
#define CMD_LOGIN 2 /* client → server: accesso */
#define CMD_MESSAGGIO 3 /* messaggio per un utente */
#define CMD_RISPOSTA 4 /* server → client: esito di un comando */

#pragma pack(push, 1)
typedef struct {
    uint8_t comando;
    char email[EMAIL_LEN]; /* mittente (o chi si registra) */
    char password[PASS_LEN];
    char destinatario[EMAIL_LEN];
    char messaggio[MSG_LEN];
} chat_message; /* 351 byte */
#pragma pack(pop)

#endif
```

*Il protocollo della chat*

## 21.2 Il server

Il thread di ogni client esegue i comandi ricevuti: la **registrazione** aggiunge l'utente all'elenco e al file, il **login** verifica le credenziali e associa all'utente il socket della connessione, il **messaggio** viene inoltrato sul socket del destinatario, se collegato, dopo aver scritto nel campo mittente l'email dell'utente autenticato. L'elenco degli utenti è condiviso fra tutti i thread e ogni accesso è protetto da un mutex.

```
/* server.c: server della chat (esercizio 4). Gestisce registrazione e
login, salva gli utenti su file e inoltra i messaggi al destinatario.
Un thread per ogni client collegato.
compilare: gcc server.c -o server -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "chat_protocol.h"

#define FILE_DB "utenti.txt" /* il "database": una riga per utente */

typedef struct {
    char email[EMAIL_LEN];
    char password[PASS_LEN];
    int socket; /* 0 se l'utente non è collegato */
} utente;

utente utenti[MAX_UTENTI];
int numero_utenti = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* protegge l'array utenti */

/* crea la socket TCP del server, la lega alla porta e ascolta */
int create_server(uint16_t port) {
    int sockfd;
    struct sockaddr_in server_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("socket");
        exit(1);
    }
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind");
        exit(1);
    }
    if (listen(sockfd, 5) < 0) {
        perror("listen");
        exit(1);
    }
    return sockfd;
}
```

```

}

/* all'avvio ricarica gli utenti salvati nel file */
void carica_database(void) {
    FILE *f = fopen(FILE_DB, "r");
    if (f == NULL) return; /* primo avvio: file assente */

    while (numero_utenti < MAX_UTENTI &&
           fscanf(f, "%49s %49s", utenti[numero_utenti].email,
                  utenti[numero_utenti].password) == 2) {
        utenti[numero_utenti].socket = 0;
        numero_utenti++;
    }
    fclose(f);
}

/* aggiunge un utente in fondo al file ("a" = append) */
void salva_utente(utente *u) {
    FILE *f = fopen(FILE_DB, "a");
    if (f == NULL) {
        perror("fopen");
        return;
    }
    fprintf(f, "%s %s\n", u->email, u->password);
    fclose(f);
}

/* restituisce la posizione dell'utente nell'array, oppure un valore negativo */
int trova_utente(char *email) {
    for (int i = 0; i < numero_utenti; i++) {
        if (strcmp(utenti[i].email, email) == 0) return i;
    }
    return -1;
}

/* thread dedicato a un client */
void *gestisci_client(void *arg) {
    int clientfd = *(int *)arg;
    int mio_indice = -1; /* negativo: non ancora autenticato */
    int indice;
    chat_message msg, risposta;

    free(arg);

    /* MSG_WAITALL: aspetto il messaggio intero, TCP può spezzarlo */
    while (recv(clientfd, &msg, sizeof(msg), MSG_WAITALL) == sizeof(msg)) {
        memset(&risposta, 0, sizeof(risposta));
        risposta.comando = CMD_RISPOSTA;

        pthread_mutex_lock(&mutex); /* accesso esclusivo agli utenti */

        if (msg.comando == CMD_REGISTRA) {
            if (trova_utente(msg.email) >= 0) {
                strcpy(risposta.messaggio, "email già registrata");
            } else if (numero_utenti == MAX_UTENTI) {
                strcpy(risposta.messaggio, "numero massimo di utenti raggiunto");
            } else {
                strcpy(utenti[numero_utenti].email, msg.email);
            }
        }
    }
}

```

```

        strcpy(utenti[numero_utenti].password, msg.password);
        utenti[numero_utenti].socket = 0;
        salva_utente(&utenti[numero_utenti]);
        numero_utenti++;
        strcpy(risposta.messaggio, "registrazione completata");
    }
    send(clientfd, &risposta, sizeof(risposta), 0);

} else if (msg.comando == CMD_LOGIN) {
    indice = trova_utente(msg.email);
    if (indice >= 0 && strcmp(utenti[indice].password, msg.password) == 0) {
        utenti[indice].socket = clientfd; /* ora è raggiungibile */
        mio_indice = indice;
        strcpy(risposta.messaggio, "login effettuato");
    } else {
        strcpy(risposta.messaggio, "credenziali errate");
    }
    send(clientfd, &risposta, sizeof(risposta), 0);

} else if (msg.comando == CMD_MESSAGGIO) {
    indice = trova_utente(msg.destinatario);
    if (mio_indice < 0) {
        strcpy(risposta.messaggio, "devi prima fare il login");
        send(clientfd, &risposta, sizeof(risposta), 0);
    } else if (indice < 0 || utenti[indice].socket == 0) {
        strcpy(risposta.messaggio, "utente non trovato o non connesso");
        send(clientfd, &risposta, sizeof(risposta), 0);
    } else {
        /* il server garantisce chi è il mittente */
        strcpy(msg.email, utenti[mio_indice].email);
        send(utenti[indice].socket, &msg, sizeof(msg), 0);
    }
}

pthread_mutex_unlock(&mutex);
}

/* il client si è disconnesso: non è più raggiungibile */
pthread_mutex_lock(&mutex);
if (mio_indice >= 0) utenti[mio_indice].socket = 0;
pthread_mutex_unlock(&mutex);

close(clientfd);
return NULL;
}

int main(void) {
    int sockfd;
    pthread_t t;

    carica_database();
    printf("caricati %d utenti dal database\n", numero_utenti);

    sockfd = create_server(PORTA_SERVER);
    printf("server in ascolto sulla porta %d\n", PORTA_SERVER);
    fflush(stdout);

    while (1) {

```

```

/* il descrittore va in memoria dinamica: ogni thread ha il suo */
int *clientfd = malloc(sizeof(int));
*clientfd = accept(sockfd, NULL, NULL);
if (*clientfd < 0) {
    perror("accept");
    free(clientfd);
    continue;
}
pthread_create(&t, NULL, gestisci_client, clientfd);
pthread_detach(t);
}

close(sockfd);
return 0;
}

```

*Il server della chat*

## 21.3 Il client

Il client ha due flussi: il main legge i comandi da tastiera e li invia, un thread stampa ciò che arriva dal server, sia le risposte ai comandi sia i messaggi degli altri utenti.

```

/* client.c: client della chat (esercizio 4).
comandi:  r <email> <password>   registrazione
          l <email> <password>   login
          m <email> <messaggio>  messaggio a un utente
compilare: gcc client.c -o client -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "chat_protocol.h"

int sockfd;

/* thread che stampa tutto ciò che arriva dal server */
void *ricevitore(void *arg) {
    chat_message msg;
    while (recv(sockfd, &msg, sizeof(msg), MSG_WAITALL) == sizeof(msg)) {
        if (msg.comando == CMD_RISPOSTA)
            printf("[server] %s\n", msg.messaggio);
        else
            printf("[%s] %s\n", msg.email, msg.messaggio);
        fflush(stdout);
    }
    printf("connessione chiusa dal server\n");
    exit(0);
}

int main(void) {
    struct sockaddr_in server_addr;

```

```

chat_message msg;
pthread_t t;
char riga[400], arg1[200], arg2[200];

sockfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORTA_SERVER);
server_addr.sin_addr.s_addr = inet_addr(IP_SERVER);

if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("connect");
    exit(1);
}
pthread_create(&t, NULL, ricevitore, NULL);

while (fgets(riga, sizeof(riga), stdin) != NULL) {
    riga[strcspn(riga, "\n")] = 0;
    if (strlen(riga) < 3) continue;

    /* dopo la lettera del comando: prima parola e resto della riga */
    arg1[0] = 0;
    arg2[0] = 0;
    sscanf(riga + 2, "%199s %199s", arg1, arg2);

    memset(&msg, 0, sizeof(msg));
    if (riga[0] == 'r' || riga[0] == 'l') {
        msg.comando = (riga[0] == 'r') ? CMD_REGISTRA : CMD_LOGIN;
        strncpy(msg.email, arg1, EMAIL_LEN - 1);
        strncpy(msg.password, arg2, PASS_LEN - 1);
    } else if (riga[0] == 'm') {
        msg.comando = CMD_MESSAGGIO;
        strncpy(msg.destinatario, arg1, EMAIL_LEN - 1);
        strncpy(msg.messaggio, arg2, MSG_LEN - 1);
    } else {
        printf("comando non valido\n");
        continue;
    }
    send(sockfd, &msg, sizeof(msg), 0);
}

close(sockfd);
return 0;
}

```

*Il client della chat*

```

gcc server.c -o server -pthread
gcc client.c -o client -pthread
./server
./client                # in due terminali diversi
r anna@x.it pw1         # registrazione
l anna@x.it pw1         # login
m bruno@x.it ciao       # messaggio a bruno, se collegato

```

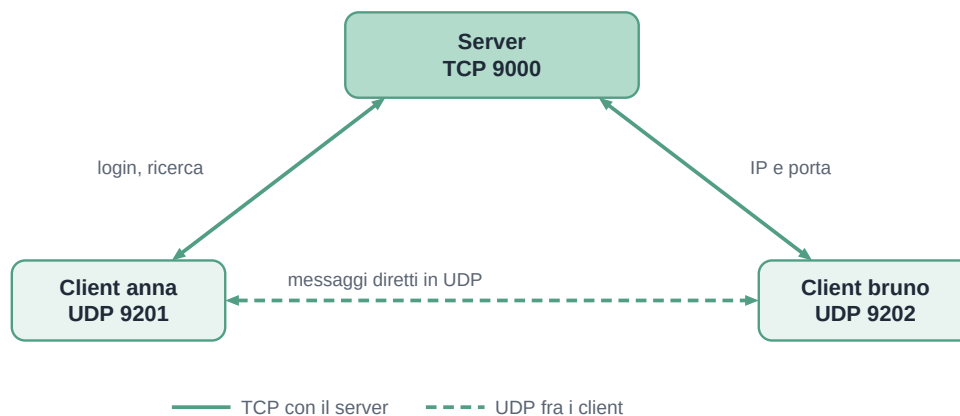
*Compilazione e uso*

# Esercizio svolto: chat peer to peer

Una soluzione completa dell'esercizio 5: il server gestisce utenti e indirizzi, mentre i messaggi viaggiano direttamente fra i client in UDP.

## 22.1 Le scelte di progetto

Rispetto all'esercizio 4 cambia il percorso dei messaggi: il server non li trasporta più, ma si limita a **presentare** i client fra loro. Ogni client apre un socket UDP su una porta scelta all'avvio e, al momento del login, la comunica al server insieme al proprio indirizzo. Quando un utente vuole scrivere a un altro chiede al server dove si trova; il server risponde con IP e porta del destinatario e il client invia il messaggio **direttamente**, in UDP. Fra client e server resta TCP, perché registrazione, login e ricerca devono essere affidabili; fra client e client si usa UDP, come richiesto dalla traccia.



*I client parlano con il server in TCP per registrazione, login e ricerca, e fra loro direttamente in UDP*

```
/* chat_protocol.h: protocollo della chat P2P (esercizio 5).
   TCP fra client e server (registrazione, login, ricerca utenti);
   UDP fra client e client (i messaggi, senza passare dal server). */
#ifndef CHAT_PROTOCOL_H
#define CHAT_PROTOCOL_H

#include <stdint.h>

#define PORTA_SERVER 9000
#define IP_SERVER    "127.0.0.1"

#define MAX_UTENTI   50
#define EMAIL_LEN    50
#define PASS_LEN     50
#define MSG_LEN      200

#define CMD_REGISTRA 1    /* client → server */
#define CMD_LOGIN    2    /* client → server, con la propria porta UDP */
#define CMD_CERCA    3    /* client → server: indirizzo di un utente */
#define CMD_RISPOSTA 4    /* server → client: esito di un comando */
#define CMD_INDIRIZZO 5   /* server → client: IP e porta del cercato */

#pragma pack(push, 1)
```

```

/* viaggia in TCP fra client e server */
typedef struct {
    uint8_t comando;
    char email[EMAIL_LEN];
    char password[PASS_LEN];
    char destinatario[EMAIL_LEN];
    char ip[16]; /* "255.255.255.255" più il terminatore */
    uint16_t porta; /* in ordine di rete (htons) */
    char messaggio[MSG_LEN];
} chat_message;

/* viaggia in UDP direttamente fra due client */
typedef struct {
    char mittente[EMAIL_LEN];
    char testo[MSG_LEN];
} p2p_message;

#pragma pack(pop)

#endif

```

*Il protocollo della chat P2P*

La porta pubblicata viaggia nel campo porta in **ordine di rete**: client e server potrebbero girare su macchine con endianness diverse, e il campo occupa due byte.

## 22.2 Il server

```

/* server.c: server della chat P2P. Non trasporta i messaggi: gestisce
gli utenti e comunica a chi vuole scrivere l'indirizzo UDP del
destinatario, così i due client parlano direttamente.
compilare: gcc server.c -o server -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "chat_protocol.h"

#define FILE_DB "utenti.txt"

typedef struct {
    char email[EMAIL_LEN];
    char password[PASS_LEN];
    char ip[16];
    uint16_t porta; /* porta UDP pubblicata; 0 = offline */
} utente;

utente utenti[MAX_UTENTI];
int numero_utenti = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

```

```

int create_server(uint16_t port) {
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        exit(1);
    }
    listen(sockfd, 5);
    return sockfd;
}

void carica_database(void) {
    FILE *f = fopen(FILE_DB, "r");
    if (f == NULL) return;
    while (numero_utenti < MAX_UTENTI &&
           fscanf(f, "%49s %49s", utenti[numero_utenti].email,
                  utenti[numero_utenti].password) == 2) {
        utenti[numero_utenti].porta = 0;    /* all'avvio nessuno è online */
        numero_utenti++;
    }
    fclose(f);
}

void salva_utente(utente *u) {
    FILE *f = fopen(FILE_DB, "a");
    if (f == NULL) return;
    fprintf(f, "%s %s\n", u->email, u->password);
    fclose(f);
}

int trova_utente(char *email) {
    for (int i = 0; i < numero_utenti; i++) {
        if (strcmp(utenti[i].email, email) == 0) return i;
    }
    return -1;
}

void *gestisci_client(void *arg) {
    int clientfd = *(int *)arg;
    int mio_indice = -1;
    int indice;
    chat_message msg, risposta;

    free(arg);

    while (recv(clientfd, &msg, sizeof(msg), MSG_WAITALL) == sizeof(msg)) {
        memset(&risposta, 0, sizeof(risposta));
        risposta.comando = CMD_RISPOSTA;
        pthread_mutex_lock(&mutex);

        if (msg.comando == CMD_REGISTRA) {
            if (trova_utente(msg.email) >= 0 || numero_utenti == MAX_UTENTI) {
                strcpy(risposta.messaggio, "registrazione non possibile");
            }
        }
    }
}

```

```

    } else {
        strcpy(utenti[numero_utenti].email, msg.email);
        strcpy(utenti[numero_utenti].password, msg.password);
        utenti[numero_utenti].porta = 0;
        salva_utente(&utenti[numero_utenti]);
        numero_utenti++;
        strcpy(risposta.messaggio, "registrazione completata");
    }

} else if (msg.comando == CMD_LOGIN) {
    indice = trova_utente(msg.email);
    if (indice >= 0 && strcmp(utenti[indice].password, msg.password) == 0) {
        /* il client pubblica IP e porta su cui riceve in UDP */
        strcpy(utenti[indice].ip, msg.ip);
        utenti[indice].porta = ntohs(msg.porta);
        mio_indice = indice;
        strcpy(risposta.messaggio, "login effettuato");
        printf("%s online su %s:%d\n", msg.email, msg.ip, utenti[indice].porta);
    } else {
        strcpy(risposta.messaggio, "credenziali errate");
    }

} else if (msg.comando == CMD_CERCA) {
    indice = trova_utente(msg.destinatario);
    if (mio_indice < 0) {
        strcpy(risposta.messaggio, "devi prima fare il login");
    } else if (indice < 0 || utenti[indice].porta == 0) {
        strcpy(risposta.messaggio, "utente non trovato o non connesso");
    } else {
        /* rispondo con l'indirizzo del destinatario e restituisco
           il testo, che il client spedirà da solo in UDP */
        risposta.comando = CMD_INDIRIZZO;
        strcpy(risposta.destinatario, msg.destinatario);
        strcpy(risposta.ip, utenti[indice].ip);
        risposta.porta = htons(utenti[indice].porta);
        strcpy(risposta.messaggio, msg.messaggio);
    }
}

pthread_mutex_unlock(&mutex);
send(clientfd, &risposta, sizeof(risposta), 0);
}

pthread_mutex_lock(&mutex);
if (mio_indice >= 0) utenti[mio_indice].porta = 0; /* torna offline */
pthread_mutex_unlock(&mutex);
close(clientfd);
return NULL;
}

int main(void) {
    int sockfd;
    pthread_t t;

    carica_database();
    sockfd = create_server(PORTA_SERVER);
    printf("server P2P in ascolto sulla porta %d (%d utenti)\n",
        PORTA_SERVER, numero_utenti);

```

```

fflush(stdout);

while (1) {
    int *clientfd = malloc(sizeof(int));
    *clientfd = accept(sockfd, NULL, NULL);
    if (*clientfd < 0) {
        free(clientfd);
        continue;
    }
    pthread_create(&t, NULL, gestisci_client, clientfd);
    pthread_detach(t);
}
return 0;
}

```

*Il server della chat P2P*

## 22.3 Il client

Il client ha tre flussi: il main legge i comandi, un thread riceve le risposte del server e, quando arriva un indirizzo, spedisce il messaggio in UDP al destinatario; un secondo thread riceve i messaggi diretti degli altri client.

```

/* client.c: client della chat P2P.
uso:      ./client <porta_udp>
comandi:  r <email> <password>  l <email> <password>  m <email> <testo>
compilare: gcc client.c -o client -pthread */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "chat_protocol.h"

int      sockfd_tcp;          /* verso il server      */
int      sockfd_udp;          /* verso gli altri client */
uint16_t mia_porta;
char     mia_email[EMAIL_LEN];

/* socket UDP personale, legata alla porta scelta dall'utente */
int create_udp(uint16_t port) {
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in addr;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        exit(1);
    }
    return sockfd;
}

```

```

/* thread 1: riceve i messaggi diretti degli altri client (UDP) */
void *ricevitore_p2p(void *arg) {
    p2p_message msg;
    while (1) {
        if (recvfrom(sockfd_udp, &msg, sizeof(msg), 0, NULL, NULL) == sizeof(msg)) {
            printf("[%s] %s\n", msg.mittente, msg.testo);
            fflush(stdout);
        }
    }
    return NULL;
}

/* thread 2: riceve le risposte del server (TCP); quando arriva un
indirizzo, spedisce il messaggio direttamente al destinatario */
void *ricevitore_server(void *arg) {
    chat_message msg;
    p2p_message p2p;
    struct sockaddr_in dest;

    while (recv(sockfd_tcp, &msg, sizeof(msg), MSG_WAITALL) == sizeof(msg)) {
        if (msg.comando == CMD_RISPOSTA) {
            printf("[server] %s\n", msg.messaggio);
        } else if (msg.comando == CMD_INDIRIZZO) {
            memset(&dest, 0, sizeof(dest));
            dest.sin_family = AF_INET;
            dest.sin_port = msg.porta; /* già in ordine di rete */
            dest.sin_addr.s_addr = inet_addr(msg.ip);

            memset(&p2p, 0, sizeof(p2p));
            strcpy(p2p.mittente, mia_email);
            strcpy(p2p.testo, msg.messaggio);
            sendto(sockfd_udp, &p2p, sizeof(p2p), 0,
                (struct sockaddr *)&dest, sizeof(dest));
            printf("inviato a %s in P2P\n", msg.destinatario);
        }
        fflush(stdout);
    }
    exit(0);
}

int main(int argc, char *argv[]) {
    struct sockaddr_in server_addr;
    chat_message msg;
    pthread_t t1, t2;
    char riga[400], arg1[200], arg2[200];

    if (argc != 2) {
        printf("uso: %s <porta_udp>\n", argv[0]);
        exit(1);
    }
    mia_porta = atoi(argv[1]);
    sockfd_udp = create_udp(mia_porta);

    sockfd_tcp = socket(AF_INET, SOCK_STREAM, 0);
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORTA_SERVER);

```

```

server_addr.sin_addr.s_addr = inet_addr(IP_SERVER);
if (connect(sockfd_tcp, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("connect");
    exit(1);
}

pthread_create(&t1, NULL, ricevitore_server, NULL);
pthread_create(&t2, NULL, ricevitore_p2p, NULL);

while (fgets(riga, sizeof(riga), stdin) != NULL) {
    riga[strcspn(riga, "\n")] = 0;
    if (strlen(riga) < 3) continue;
    arg1[0] = 0;
    arg2[0] = 0;
    sscanf(riga + 2, "%199s %199s", arg1, arg2);

    memset(&msg, 0, sizeof(msg));
    if (riga[0] == 'r') {
        msg.comando = CMD_REGISTRA;
        strncpy(msg.email, arg1, EMAIL_LEN - 1);
        strncpy(msg.password, arg2, PASS_LEN - 1);
    } else if (riga[0] == 'l') {
        msg.comando = CMD_LOGIN;
        strncpy(msg.email, arg1, EMAIL_LEN - 1);
        strncpy(msg.password, arg2, PASS_LEN - 1);
        strcpy(msg.ip, "127.0.0.1"); /* IP su cui sono raggiungibile */
        msg.porta = htons(mia_porta); /* porta UDP pubblicata */
        strncpy(mia_email, arg1, EMAIL_LEN - 1);
    } else if (riga[0] == 'm') {
        msg.comando = CMD_CERCA;
        strncpy(msg.destinatario, arg1, EMAIL_LEN - 1);
        strncpy(msg.messaggio, arg2, MSG_LEN - 1);
    } else {
        printf("comando non valido\n");
        continue;
    }
    send(sockfd_tcp, &msg, sizeof(msg), 0);
}

close(sockfd_tcp);
close(sockfd_udp);
return 0;
}

```

*Il client della chat P2P*

```

gcc server.c -o server -pthread
gcc client.c -o client -pthread
./server
./client 9201          # anna riceve i messaggi sulla porta UDP 9201
./client 9202          # bruno sulla porta 9202

```

*Compilazione ed esecuzione*

# Compilazione, prove ed errori frequenti

Un riepilogo pratico per scrivere, compilare e provare i programmi di rete, con gli errori più comuni e il modo di riconoscerli.

## 23.1 Compilare e provare

I programmi si compilano con gcc: l'opzione `-Wall` attiva gli avvisi, che conviene leggere sempre, e i programmi con i thread richiedono l'opzione `-pthread`. Per provarli su una sola macchina si avvia ogni programma in un terminale diverso, usando come indirizzo `127.0.0.1`, il **loopback**: i pacchetti non lasciano la macchina, ma attraversano lo stesso stack di rete. L'ordine di avvio conta: un client TCP che cerca un server non ancora avviato riceve «Connection refused», mentre con UDP i datagrammi vanno persi in silenzio.

```
gcc programma.c -o programma -Wall           # programma senza thread
gcc programma.c -o programma -Wall -pthread   # programma con i thread
```

Compilazione

## 23.2 Errori frequenti

Quando qualcosa non funziona conviene procedere per livelli, come nello stack: prima la **connettività** (i nodi si raggiungono? le rotte ci sono in entrambe le direzioni?), poi il **trasporto** (il server è in ascolto sulla porta giusta, con lo stesso protocollo?), infine l'**applicazione** (struttura dei messaggi, ordine dei byte, logica).

| SINTOMO                                        | CAUSA                                                                                                                                                  | RIMEDIO                                                                                                                                |
|------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| <b>bind: Address already in use</b>            | Il server è stato appena riavviato e la porta è ancora occupata da connessioni in <code>TIME_WAIT</code> , oppure un'altra istanza è già in esecuzione | Attendere, chiudere l'altra istanza, oppure impostare <code>SO_REUSEADDR</code> con <code>setsockopt</code> prima di <code>bind</code> |
| <b>Numeri sbagliati dall'altra parte</b>       | Mancata conversione nel Network Byte Order                                                                                                             | Usare <code>htons</code> e <code>htonl</code> in invio, <code>ntohs</code> e <code>ntohl</code> in ricezione                           |
| <b>Struttura ricevuta con campi spostati</b>   | Byte di riempimento diversi fra le due parti                                                                                                           | Usare <code>pragma pack(1)</code> in entrambi i programmi                                                                              |
| <b>Messaggio ricevuto a metà con TCP</b>       | TCP è un flusso di byte e può spezzare un messaggio                                                                                                    | Usare <code>MSG_WAITALL</code> o ripetere <code>recv</code> fino alla dimensione attesa                                                |
| <b>Caratteri strani in fondo a una stringa</b> | <code>recv</code> non aggiunge il terminatore                                                                                                          | Scrivere il terminatore dopo i byte ricevuti, o azzerare il buffer con <code>memset</code>                                             |
| <b>Il programma si blocca</b>                  | <code>recv</code> , <code>recvfrom</code> , <code>accept</code> e <code>fgets</code> sono bloccanti                                                    | Spostare le attività parallele in thread separati                                                                                      |
| <b>Comportamenti casuali con più client</b>    | Accesso concorrente a dati condivisi                                                                                                                   | Proteggere le strutture condivise con un mutex                                                                                         |