



NoiStudiamo

DISPENSA DI TEORIA

Reti di Calcolatori

Dal livello di applicazione al livello di collegamento:
protocolli, algoritmi e tecnologie di Internet e delle LAN

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

I capitoli della dispensa, dal livello di applicazione al livello di collegamento.

1. Reti, protocolli e architetture	4
2. Il modello OSI e il modello TCP/IP	6
3. Caratteristiche della comunicazione	8
4. Commutazione di circuito e commutazione di pacchetto	11
5. Dispositivi di rete e primi cenni di sicurezza	14
6. Architettura delle applicazioni di rete	15
7. Processi, socket e indirizzamento	16
8. Servizi di trasporto richiesti dalle applicazioni	18
9. Il protocollo HTTP	19
10. Telnet e FTP	25
11. La posta elettronica	27
12. Il DNS	29
13. Cenni sul protocollo SNMP	34
14. Servizi e protocolli a livello di trasporto	35
15. Multiplexing e demultiplexing	36
16. UDP (User Datagram Protocol)	37
17. Principi del trasferimento dati affidabile	38
18. TCP (Transmission Control Protocol)	44
19. Controllo del flusso TCP	48
20. Apertura e chiusura della connessione TCP	50
21. Cause e costi della congestione	52
22. Controllo di congestione classico in TCP	54
23. Fairness	57
24. QUIC	59
25. Calcolo del throughput	60
26. Routing, forwarding e piani del router	62
27. Modelli di servizio del livello di rete	63

28. Architettura di un router	64
29. Il protocollo IPv4	67
30. Indirizzamento intra LAN e inter LAN	70
31. ARP, RARP e BOOTP	71
32. Routing table e host table	73
33. Subnetting, supernetting e VLSM	74
34. DHCP e NAT	77
35. ICMP	79
36. Il protocollo IPv6	80
37. Firewall e DMZ	83
38. Algoritmi di routing	84
39. Distance Vector	85
40. Link State Routing e algoritmo di Dijkstra	87
41. Internet routing: AS, RIP, OSPF e BGP	90
42. I servizi del Data Link Layer	93
43. Data framing e codifiche di linea	94
44. Rilevazione e correzione degli errori	96
45. Distanza di Hamming e correzione degli errori	99
46. Gestire gli accessi multipli	103
47. Protocolli ad accesso casuale	105
48. Ethernet e IEEE 802.3	109
49. Fast Ethernet e Gigabit Ethernet	113
50. Strutture Ethernet e indirizzamento a livello DL	116
51. VLAN	118

Reti, protocolli e architetture

Il capitolo introduce il concetto di protocollo, la struttura delle unità di dati che i protocolli si scambiano e il principio dell'architettura a livelli su cui si fonda ogni rete di calcolatori.

1.1 Perché studiare le reti

Lo scambio di informazioni è, in senso molto concreto, una forma di potere: chi controlla il flusso dei dati controlla ciò che gli altri possono sapere e fare. È una considerazione che sembra astratta finché non la si applica al mondo reale, e l'esempio più immediato è quello dell'**IoT**, l'*Internet of Things*. Si tratta di dispositivi che semplificano enormemente la vita quotidiana (termostati, telecamere, lampadine, serrature) ma che spesso vengono progettati con scarsa attenzione alla sicurezza. Se non si comprende bene *come* questi oggetti comunicano fra loro, si finisce per introdurre in casa o in azienda dispositivi che espongono l'intera rete a rischi che l'utente nemmeno immagina. Studiare le reti significa quindi anche capire dove stanno le vulnerabilità di ciò che usiamo tutti i giorni.

1.2 Il concetto di protocollo

DEFINIZIONE

Un **protocollo** è un insieme di **regole** che stabiliscono come deve avvenire la comunicazione fra due entità: il formato e l'ordine dei messaggi scambiati e le azioni da compiere quando un messaggio viene trasmesso o ricevuto.

La definizione è breve, ma è il concetto su cui poggia tutto il resto della disciplina.

Pensiamo a un comando banale come «accendi la luce» impartito a un dispositivo domotico. Dietro un'operazione così semplice esiste un insieme di regole che permettono ai due dispositivi di capirsi: chi parla per primo, in che formato viene codificato il comando, come si riconosce che il messaggio è terminato, che cosa deve rispondere il destinatario per confermare di aver eseguito. Senza queste regole condivise i due apparecchi si scambierebbero sequenze di bit prive di significato.

Un protocollo ben progettato non si limita però a descrivere il caso in cui tutto va bene. Deve **gestire errori ed eccezioni**: che cosa accade se il messaggio si perde per strada, se arriva corrotto, se arriva due volte, se il destinatario non risponde. E deve essere il più possibile **ottimale**, cioè ottenere il risultato consumando la minor quantità possibile di risorse: tempo, banda, memoria. Buona parte di ciò che studieremo consiste esattamente nel vedere come protocolli diversi affrontano questo compromesso fra robustezza e costo.

Poiché gestire in un colpo solo tutta la complessità della comunicazione sarebbe impossibile, i protocolli vengono organizzati in **strutture a livelli**. Ogni livello si occupa di risolvere un problema specifico e dialoga idealmente con il livello corrispondente dell'altro host, come se fra i due esistesse un canale diretto.

1.3 Struttura di un protocollo: header, overhead e payload

Quando un protocollo costruisce l'unità di dati che immetterà in rete, questa unità non contiene soltanto le informazioni che l'utente vuole trasmettere. In genere si distinguono tre componenti: le **header information**, cioè l'intestazione con i campi di controllo; il **communication overhead**, cioè tutto ciò che serve al funzionamento del protocollo ma non trasporta contenuto utile; e infine il **payload**, ossia i dati effettivi.

Header e overhead sono complessivamente **informazioni di controllo**: indirizzi, numeri di sequenza, checksum, flag. Sono indispensabili (senza di essi il destinatario non saprebbe a chi è destinato il messaggio né se è arrivato

integro) ma hanno un costo preciso: **occupano banda**. Ogni livello che attraversiamo aggiunge la propria intestazione, e quindi riduce lo spazio effettivamente disponibile per i dati veri e propri. Questo è un punto da tenere sempre presente: l'efficienza di un protocollo si misura anche dal rapporto fra payload e overhead.

1.4 L'architettura a livelli

L'idea di dividere la comunicazione in livelli sovrapposti risponde a un principio di ingegneria molto generale: scomporre un problema difficile in sottoproblemi indipendenti. Ogni livello offre un *servizio* al livello immediatamente superiore e si appoggia a sua volta ai servizi del livello inferiore, senza doversi preoccupare di come questi siano realizzati.

Il vantaggio pratico è enorme. Se domani il cavo di rame venisse sostituito dalla fibra ottica, cambierebbe solo il livello più basso: tutti gli altri continuerebbero a funzionare senza modifiche. Allo stesso modo, un'applicazione web non ha bisogno di sapere se i suoi dati viaggiano su WiFi o su Ethernet.

Un esempio efficace per capire come avviene concretamente la comunicazione è quello della **spedizione di un libro**. Se devo spedire un libro, prima lo scompongo pagina per pagina; poi invio un messaggio che indica al destinatario quante pagine deve aspettarsi; il destinatario riceve le pagine, magari non nell'ordine in cui sono partite, le rimette nell'ordine corretto e infine ricostruisce il libro completo. Questo esempio rappresenta esattamente il modo in cui i dati vengono **suddivisi, trasmessi e ricostruiti** durante la comunicazione fra due sistemi: la segmentazione al mittente, la numerazione che consente il riordino, il riassemblaggio al destinatario.

Il modello OSI e il modello TCP/IP

Il capitolo presenta i due modelli di riferimento per l'organizzazione dei protocolli, il modello OSI a sette livelli e il modello TCP/IP effettivamente impiegato in Internet, e chiarisce la distinzione fra servizio e protocollo.

2.1 Il modello OSI

Osservando che i sistemi di comunicazione potevano essere organizzati in **sette livelli**, si arrivò alla definizione di un modello di riferimento poi standardizzato dall'**ISO** con il nome di **modello OSI** (*Open Systems Interconnection*). L'OSI non è un protocollo realmente implementato, ma un modello concettuale: serve a capire *quali problemi* vanno risolti e *in quale ordine*.

I sette livelli, dal basso verso l'alto, sono i seguenti.

#	LIVELLO	DI CHE COSA SI OCCUPA
1	Fisico	Trasporta i segnali elettrici (o ottici, o radio) sul mezzo trasmissivo. Si occupa di tensioni, connettori, temporizzazione: il singolo bit come fenomeno fisico.
2	Data Link	Prende i bit e li raggruppa in frame , gestisce il collegamento locale fra macchine direttamente connesse e controlla eventuali errori di trasmissione.
3	Rete	Fa arrivare i dati a destinazione attraversando la rete, individuando il percorso migliore con il minor costo fra sorgente e destinatario.
4	Trasporto	Realizza la comunicazione fra i processi applicativi dei due estremi. È il livello di TCP e UDP .
5	Sessione	Gestisce l'apertura, il mantenimento e la chiusura delle sessioni di dialogo fra le applicazioni.
6	Presentazione	Riguarda il modo in cui i dati sono rappresentati: codifiche di caratteri, cifratura, compressione, e differenze di rappresentazione come big endian e little endian .
7	Applicazione	È il livello più vicino all'utente e comprende i servizi applicativi veri e propri (posta, web, trasferimento file).

Il livello di presentazione merita una nota, perché il problema che risolve è meno intuitivo degli altri. Due macchine possono memorizzare un numero intero su più byte in ordine opposto: nel *big endian* il byte più significativo viene per primo, nel *little endian* per ultimo. Se non ci si accorda su una rappresentazione comune, lo stesso numero verrebbe letto in modo completamente diverso dalle due parti.

2.2 Il modello TCP/IP e il confronto con OSI

Anche se il modello OSI resta molto importante dal punto di vista teorico e didattico, nella pratica oggi si utilizza il **modello TCP/IP**. Questo modello non segue rigidamente i sette livelli dell'OSI, ma raggruppa le funzioni in livelli più generali, che spiegano comunque in modo completo il funzionamento della rete.

La differenza sostanziale sta nel fatto che TCP/IP nasce *dall'implementazione*: prima sono esistiti i protocolli, poi si è descritto il modello che li organizza. OSI segue il percorso inverso, dal modello ai protocolli: ed è anche per questo che i suoi protocolli non hanno mai avuto diffusione reale, mentre il modello è sopravvissuto come

strumento concettuale.

MODELLO OSI	MODELLO TCP/IP	PROTOCOLLI TIPICI
7 Applicazione · 6 Presentazione · 5 Sessione	Applicazione	HTTP, FTP, SMTP, DNS, SNMP
4 Trasporto	Trasporto	TCP, UDP
3 Rete	Internet (Rete)	IP, ICMP, ARP
2 Data Link · 1 Fisico	Accesso alla rete	Ethernet, WiFi, PPP

DA RICORDARE

Nel passaggio da OSI a TCP/IP i tre livelli superiori (sessione, presentazione, applicazione) confluiscono nell'unico livello applicativo, mentre i due livelli inferiori confluiscono nel livello di accesso alla rete. Il motivo è che TCP/IP descrive protocolli già esistenti e funzionanti, mentre OSI è un modello progettato a priori, le cui distinzioni più fini non hanno trovato riscontro nelle implementazioni reali.

Un dato utile per dare la misura di quanto questo mondo sia vivo: esistono **numerosi varianti del protocollo TCP**. Non si tratta di protocolli diversi, ma di implementazioni che modificano soprattutto il comportamento in caso di congestione, adattandolo a contesti differenti: reti mobili, collegamenti satellitari, data center.

2.3 Servizi e primitive di servizio

Ogni livello offre a quello superiore un **servizio**, ed è importante distinguere il servizio dal protocollo. Il **servizio** è l'insieme delle operazioni che il livello mette a disposizione: descrive *che cosa* il livello fa. Il **protocollo** è invece l'insieme delle regole con cui due entità dello stesso livello si scambiano messaggi: descrive *come* il servizio viene realizzato. Il livello superiore vede solo il servizio e ignora completamente il protocollo, ed è proprio questa separazione che consente di sostituire un protocollo con un altro senza toccare il resto della pila.

Un servizio viene specificato formalmente attraverso un insieme di **primitive**, cioè di operazioni elementari che il livello superiore può invocare. Per un servizio orientato alla connessione le primitive tipiche sono le seguenti.

- **LISTEN**: il server si mette in attesa di una richiesta di connessione in arrivo.
- **CONNECT**: il client richiede l'apertura di una connessione verso il server.
- **ACCEPT**: il server accetta la connessione richiesta.
- **SEND** e **RECEIVE**: le due parti si scambiano i dati sulla connessione stabilita.
- **DISCONNECT**: la connessione viene chiusa.

In un servizio senza connessione l'insieme si riduce essenzialmente a SEND e RECEIVE, perché non esiste alcuna fase di instaurazione da gestire.

Nella terminologia OSI ogni primitiva può inoltre assumere quattro forme, che descrivono il percorso completo di una richiesta: la **request**, con cui l'entità chiedente avvia l'operazione; la **indication**, con cui l'entità remota viene informata che qualcosa sta accadendo; la **response**, con cui essa reagisce; e la **confirm**, con cui il richiedente riceve conferma dell'esito. Un servizio che prevede tutte e quattro le fasi si dice *confermato*; se si ferma alle prime due si dice *non confermato*.

Caratteristiche della comunicazione

Il capitolo descrive le proprietà fondamentali di un canale di comunicazione: la degradazione del segnale, la direzione della trasmissione, l'affidabilità, la presenza o meno di una connessione, il numero di destinatari e l'estensione geografica della rete.

3.1 Il segnale e la sua degradazione

Il segnale viene inviato nel modo più efficiente possibile, ma nella maggior parte dei casi viaggia **attraverso l'aria** o attraverso un mezzo fisico che non è perfetto. Durante il percorso subisce **attenuazione**, dovuta alla distanza e alla presenza di ostacoli, e può essere disturbato da interferenze.

Idealmente vorremmo un **segnale perfetto, privo di errori**, ma nella realtà questo non è possibile. Molti dei concetti che studieremo sono formulati in condizioni teoriche ideali; nella pratica i protocolli servono proprio a **ridurre al minimo gli errori di comunicazione**, non a eliminarli. È una distinzione da tenere presente: non esiste un protocollo che garantisca l'assenza di errori, esistono protocolli che li rilevano e li correggono con una probabilità di successo molto alta.

3.2 Direzione del canale: simplex, half duplex, full duplex

I canali di comunicazione si distinguono in base alle direzioni in cui l'informazione può viaggiare.

- **Simplex**: il segnale viaggia in una sola direzione. Da solo non permette una vera comunicazione, perché manca completamente il ritorno dell'informazione: il mittente non ha modo di sapere se il messaggio è stato ricevuto.
- **Half duplex**: la comunicazione avviene in entrambe le direzioni, ma **non contemporaneamente**. Prima trasmette un dispositivo, poi l'altro, alternandosi. È il funzionamento tipico di una ricetrasmittente.
- **Full duplex**: entrambi i dispositivi possono trasmettere **simultaneamente**. Un esempio è il **cavo Ethernet**, che dispone di coppie separate per le due direzioni.

Nel corso della comunicazione è inoltre importante **dosare la quantità di dati inviati**, evitando di saturare il canale o il destinatario: questo meccanismo prende il nome di **controllo del flusso** e lo ritroveremo in dettaglio nel livello di trasporto. Un altro requisito fondamentale è che **l'ordine dei messaggi ricevuti corrisponda all'ordine con cui sono stati inviati**, cosa tutt'altro che scontata quando i dati attraversano una rete in cui pacchetti diversi possono seguire percorsi diversi.

3.3 Comunicazioni affidabili e non affidabili

DEFINIZIONE

Una comunicazione si dice **affidabile** quando garantisce che i dati arrivino a destinazione, integri e nell'ordine corretto.

Poiché il mezzo fisico non offre alcuna di queste garanzie, l'affidabilità va costruita: il destinatario conferma ciò che riceve tramite messaggi di **acknowledgment**, il mittente conserva una copia di ciò che ha inviato finché non arriva la conferma e, se questa non arriva entro un tempo prestabilito, ritrasmette.

Una comunicazione **non affidabile** rinuncia a queste garanzie: i dati vengono immessi in rete e non si verifica se siano arrivati. Non è necessariamente una scelta sbagliata, anzi. L'affidabilità costa tempo (bisogna attendere le

conferme) e costa banda, perché le conferme stesse occupano il canale. Per una videochiamata o per uno streaming in diretta è preferibile perdere qualche pacchetto piuttosto che introdurre ritardi: un fotogramma perso è meno fastidioso di mezzo secondo di attesa. La scelta fra i due modelli dipende quindi interamente dal tipo di applicazione.

3.4 Servizi con connessione e senza connessione

Esistono due modalità principali di comunicazione, che si comprendono bene attraverso due esempi concreti.

SISTEMA SENZA CONNESSIONE

L'esempio classico è quello della **cartolina**. Scrivo il messaggio, aggiungo l'indirizzo del destinatario e la spedisco. Non esiste alcuna connessione stabile fra me e chi la riceverà: ogni cartolina è un'entità autonoma, che porta con sé tutte le informazioni necessarie per essere recapitata. Se desidero una risposta, sarà l'altra persona a dover spedire una nuova cartolina. Non ho modo di sapere se la mia sia arrivata, né quando.

SISTEMA ORIENTATO ALLA CONNESSIONE

Un esempio moderno è **WhatsApp**. Prima viene stabilita una connessione fra i due dispositivi, e solo successivamente è possibile scambiarsi messaggi in modo continuo. Poiché il collegamento è mantenuto attivo, possiamo sapere quando il messaggio è stato **consegnato** e quando è stato **letto**: informazioni che nel modello della cartolina sarebbero impensabili.

Prima della comunicazione vera e propria avviene quindi uno **scambio iniziale di informazioni** per instaurare la connessione: le due parti si accordano sui parametri del dialogo, e solo dopo cominciano a trasmettere i dati. Questa fase iniziale ha un costo in termini di tempo, ma abilita tutti i servizi che richiedono di ricordare lo stato della conversazione.

3.5 Punto a punto, broadcast e multicast

Un'ulteriore classificazione riguarda il numero di destinatari raggiunti da una trasmissione.

POINT TO POINT

È una connessione **uno a uno**, spesso realizzata tramite un cavo dedicato che collega due soli dispositivi. Il vantaggio principale è che, se il collegamento è **full duplex**, entrambi possono trasmettere contemporaneamente sfruttando appieno il mezzo. C'è però anche un vantaggio in termini di **sicurezza**: intercettare la comunicazione è difficile, perché occorrerebbe intervenire fisicamente sul cavo. Con il **rame** l'operazione è relativamente più semplice, perché il segnale elettrico genera un campo elettromagnetico che può essere captato; con la **fibra ottica** è invece molto più complessa, dato che bisognerebbe interrompere il fascio luminoso, cosa che rende l'intrusione immediatamente rilevabile.

BROADCAST

Nella modalità **broadcast** la comunicazione avviene su un mezzo condiviso da **più host contemporaneamente**: un pacchetto inviato da un dispositivo può essere ricevuto da tutti gli altri. Questo genera due ordini di problemi. Il primo è che, se molti dispositivi trasmettono nello stesso istante, si verifica una **sovrapposizione dei pacchetti** che rende i dati illeggibili: è il fenomeno della *collisione*, per gestire il quale esistono protocolli dedicati che studieremo nel livello di collegamento. Il secondo è di natura diversa e riguarda la riservatezza: **tutti i dispositivi possono ascoltare i messaggi**, anche quando non ne sono i destinatari.

Le principali topologie broadcast sono lo **shared bus**, il **ring** e la **star**. La differenza più significativa fra bus e star riguarda l'affidabilità: nel **bus** l'interruzione del collegamento condiviso può far cadere l'intera rete, mentre nella **star** lo scollegamento di un singolo computer lascia il resto della rete perfettamente funzionante, perché ogni nodo ha un proprio collegamento verso il centro stella.

MULTICAST

Il **multicast** rappresenta una via di mezzo fra i due estremi: la trasmissione è destinata a un **gruppo specifico** di host, non a uno solo e non a tutti. Gli host interessati aderiscono al gruppo, e il traffico viene consegnato soltanto a loro. Il vantaggio rispetto al broadcast è evidente in termini di efficienza (non si disturbano macchine che non hanno interesse al contenuto) e rispetto all'invio punto a punto ripetuto si risparmia banda, perché la sorgente trasmette una sola volta anziché una volta per destinatario. È il modello tipico della diffusione di contenuti audio e video verso molti utenti simultanei.

3.6 Classificazione per estensione geografica

Le reti possono infine essere classificate in base all'area che coprono.

SIGLA	DENOMINAZIONE	ESTENSIONE ED ESEMPI
PAN	Personal Area Network	Reti molto piccole, nell'ordine del metro attorno alla persona. L'esempio tipico è il Bluetooth .
LAN	Local Area Network	Reti locali, tipicamente all'interno di una stanza, di un piano o di un edificio.
MAN	Metropolitan Area Network	Reti che coprono l'estensione di un'intera città.
WAN	Wide Area Network	Reti che coprono regioni, nazioni o grandi aree geografiche.
Internet	Rete di reti	La rete globale che collega fra loro tutte le altre reti: una <i>rete di reti</i> .

Commutazione di circuito e commutazione di pacchetto

Sulla base del modo in cui le risorse della rete vengono assegnate alle comunicazioni, le reti si dividono in due grandi categorie: quelle a **commutazione di circuito** e quelle a **commutazione di pacchetto**. È una distinzione centrale, perché spiega perché Internet funziona come funziona.

4.1 La commutazione di circuito

In passato il sistema telefonico funzionava interamente a **commutazione di circuito**. Quando si alzava la cornetta si entrava in contatto con una **centralinista**, alla quale si comunicava con chi si desiderava parlare. L'operatrice si occupava quindi di contattare la controparte e di **collegare fisicamente il circuito** fra le due linee, inserendo materialmente uno spinotto in un pannello. Per chiudere la comunicazione il circuito doveva essere **sganciato manualmente**, e per capire quando la conversazione fosse terminata l'operatrice doveva spesso restare in ascolto.

Il vantaggio di questo sistema, va detto, era soprattutto storico: era l'unico che la tecnologia dell'epoca potesse permettersi, e garantiva alla conversazione una qualità costante, dato che le risorse erano interamente riservate.

Gli svantaggi erano però numerosi. Un uso impreciso del **disco combinatore** poteva mettere in comunicazione con la persona sbagliata. Soprattutto, la **linea restava occupata per tutta la durata della comunicazione**, anche nei momenti di silenzio, e non poteva essere utilizzata per altre conversazioni: la risorsa era assegnata in modo esclusivo indipendentemente dal fatto che venisse effettivamente usata. Bastava inoltre non appoggiare bene la cornetta perché la linea rimanesse occupata indefinitamente.

4.2 La nascita della commutazione di pacchetto

La **DARPA** (*Defense Advanced Research Projects Agency*) è l'agenzia del Dipartimento della Difesa statunitense responsabile dello sviluppo di tecnologie emergenti per uso militare, fondata nel **1958** in risposta al lancio dello **Sputnik**. Con un budget dell'ordine dei tre miliardi di dollari e progetti a breve termine, dai due ai quattro anni, il suo obiettivo dichiarato è produrre *sorprese tecnologiche* utili alla sicurezza nazionale.

Una delle motivazioni alla base del progetto era precisa: evitare che la distruzione di alcune centrali telefoniche interrompesse completamente le comunicazioni. La soluzione proposta fu di **suddividere la comunicazione in pacchetti** e farli viaggiare attraverso i nodi della rete, permettendo a ciascun pacchetto di seguire un percorso diverso. In questo modo il sistema sarebbe stato **molto più resistente ai guasti**: venendo meno un nodo, il traffico avrebbe semplicemente aggirato l'ostacolo.

L'idea fu inizialmente scartata, ma i finanziamenti successivi portarono alla realizzazione di **ARPANET**, il cui primo grafo era composto da appena **quattro nodi**. Il primo nodo fu installato nel laboratorio di **Leonard Kleinrock** alla UCLA. Ogni calcolatore era collegato alla rete tramite un dispositivo dedicato, l'**IMP** (*Interface Message Processor*), realizzato dalla società BBN: in sostanza un minicomputer delle dimensioni di un armadio, antenato degli odierni router. Il primo log della comunicazione su ARPANET era cartaceo, e il primo messaggio che si tentò di inviare doveva essere la parola **LOGIN**: arrivò la lettera **L**, poi la **O**, poi il sistema andò in crash. Nonostante l'esito, la prova dimostrò la validità dell'idea.

4.3 Come funziona la commutazione di pacchetto

Nella commutazione di pacchetto il messaggio viene suddiviso in unità autonome che attraversano la rete passando di nodo in nodo. Il meccanismo che rende tutto ciò possibile è la **bufferizzazione**: quando un pacchetto arriva a un router, viene **memorizzato temporaneamente** in una coda e viene inoltrato non appena il collegamento di uscita risulta libero. È il principio noto come *store and forward*.

Questo introduce però un limite fisico inevitabile. Se arrivano troppi pacchetti e il router **non dispone di memoria sufficiente**, i pacchetti eccedenti vengono semplicemente **scartati**. È la radice del fenomeno della congestione, che studieremo diffusamente nel livello di trasporto: la perdita di pacchetti in Internet non è quasi mai dovuta a errori di trasmissione, ma proprio all'esaurimento dei buffer nei nodi intermedi.

	COMMUTAZIONE DI CIRCUITO	COMMUTAZIONE DI PACCHETTO
Risorse	Riservate in modo esclusivo per tutta la durata della comunicazione	Condivise dinamicamente fra tutte le comunicazioni attive
Instaurazione	Necessaria prima di poter comunicare	Non necessaria nel modello datagram
Efficienza	Bassa: la linea resta occupata anche nei silenzi	Alta: la capacità inutilizzata è disponibile per altri
Robustezza	Bassa: il guasto di un nodo interrompe la comunicazione	Alta: i pacchetti possono seguire percorsi alternativi
Qualità	Costante e garantita	Variabile: possibili ritardi, perdite e riordino

4.4 Circuiti virtuali e datagram

All'interno della commutazione di pacchetto si distinguono due modelli di servizio, che rappresentano due filosofie diverse nel gestire il rapporto fra rete e comunicazione.

CIRCUITI VIRTUALI

In questo modello il sistema stabilisce un percorso *prima* che la comunicazione abbia inizio. La macchina che vuole trasmettere invia un pacchetto iniziale in cui indica **quantità e velocità dei dati** che intende immettere in rete; ciascun router lungo il cammino può a quel punto valutare se è in grado di sostenere quella comunicazione. Se la risposta è affermativa, il percorso viene registrato e i pacchetti successivi lo seguiranno, venendo posti **in coda** nei nodi attraversati; se è negativa, si può cercare **un percorso alternativo**.

Il vantaggio di questo approccio è che permette di **gestire meglio le risorse della rete**: sapendo in anticipo quali flussi attraverseranno un nodo, è possibile riservare capacità e offrire garanzie di qualità. Il prezzo da pagare è la necessità di mantenere nei router informazioni di stato su ogni circuito attivo.

DATAGRAM

Nel modello **datagram** i pacchetti viaggiano invece **in modo completamente indipendente** l'uno dall'altro, senza alcuna fase preliminare. Quando un pacchetto arriva a un nodo, il router verifica se dispone di memoria disponibile: se non ne ha, **scarta il pacchetto**; se ne ha, lo elabora e lo inserisce nella coda di uscita corrispondente al percorso scelto in quel momento.

È questo il modello adottato da Internet, e la scelta ha conseguenze profonde. La rete risulta **più flessibile e più robusta**, perché nessun nodo deve ricordare nulla e ogni pacchetto può essere instradato indipendentemente in base allo stato corrente della rete. In compenso ogni pacchetto deve portare con sé l'**indirizzo completo di destinazione**, e la rete da sola non garantisce né l'ordine di consegna né l'affidabilità: i pacchetti possono arrivare fuori ordine, essere persi o essere duplicati. Se l'applicazione ha bisogno di garanzie, queste vanno costruite nei livelli superiori: ed è esattamente ciò che fa TCP.

DA RICORDARE

Il confronto fra *datagram service* e servizio a circuito virtuale si riassume in un compromesso di fondo: il datagram sposta la complessità ai bordi della rete, mantenendo i nodi interni semplici e privi di stato, e ottiene così flessibilità e robustezza al prezzo dell'assenza di garanzie; il circuito virtuale mantiene lo stato nei nodi e può quindi offrire garanzie di qualità, ma diventa più rigido e più fragile.

4.5 Tecniche di multiplexing: FDM e TDM

Poiché il mezzo trasmissivo è una risorsa costosa, esistono tecniche che consentono a più comunicazioni di condividere lo stesso canale fisico. Le due fondamentali sono la **FDM** e la **TDM**.

Con la **FDM** (*Frequency Division Multiplexing*) più comunicazioni viaggiano contemporaneamente sullo stesso canale utilizzando **frequenze diverse**. L'immagine più efficace è quella del prisma ottico che scompone la luce: allo stesso modo il canale viene suddiviso in bande di frequenza distinte, ciascuna assegnata a una comunicazione. Il vantaggio è che le trasmissioni avvengono davvero in parallelo; il limite è che ognuna occupa in modo permanente una porzione fissa dello spettro, anche quando non ha nulla da trasmettere.

Con la **TDM** (*Time Division Multiplexing*) l'intero spettro viene invece utilizzato **a turni nel tempo**: ogni comunicazione dispone del canale per un intervallo temporale limitato, al termine del quale lascia spazio alla successiva. Qui la risorsa non è divisa in frequenza ma in tempo, e la logica di suddivisione è sequenziale anziché parallela.

Ritroveremo entrambe queste tecniche, insieme alla CDMA, quando affronteremo i protocolli di accesso multiplo nel livello di collegamento: sono infatti la base dei cosiddetti *protocolli a suddivisione del canale*.

Dispositivi di rete e primi cenni di sicurezza

Il capitolo introduce i principali dispositivi di interconnessione, che saranno approfonditi nei livelli di collegamento e di rete, e spiega perché la sicurezza è diventata un tema centrale soltanto dopo la diffusione di Internet.

5.1 I dispositivi di interconnessione

Nel tempo sono stati impiegati dispositivi diversi per gestire l'interconnessione nelle reti; qui ci limitiamo a introdurli, poiché saranno trattati in modo approfondito nel livello di collegamento e nel livello di rete.

- L'**hub** è il dispositivo più elementare: ripete su tutte le porte ciò che riceve, senza alcuna forma di selezione. Oggi è praticamente in disuso.
- Lo **switch** opera principalmente a **livello 2**, cioè sugli indirizzi del livello di collegamento, e inoltra il traffico soltanto verso la porta corretta. Esistono anche **switch layer 3**, capaci di operazioni più avanzate di filtraggio e instradamento.
- Il **router** lavora a livello di rete e resta concettualmente distinto anche dagli switch layer 3. Nei modelli più avanzati si impiegano memorie speciali come **CAM** e **TCAM**, progettate per effettuare ricerche nelle tabelle di instradamento a velocità molto elevata: sono memorie *associative*, in cui la ricerca avviene per contenuto anziché per indirizzo, il che consente di trovare la voce corrispondente in un solo accesso.

5.2 Network security: perché il problema si è posto in ritardo

Quando la commutazione di pacchetto venne progettata, l'attenzione era rivolta quasi esclusivamente al **funzionamento** del sistema, non alla sua sicurezza. La ragione è comprensibile: agli inizi le informazioni scambiate avevano un valore modesto, la rete collegava un numero ristretto di centri di ricerca e i partecipanti si conoscevano fra loro. Non esisteva un modello di minaccia perché, di fatto, non esisteva la minaccia.

I problemi emersero non appena la rete crebbe. Il caso più celebre fra i primi è il **Morris Worm**, uno dei primi attacchi informatici della storia, che mostrò quanto rapidamente un programma potesse propagarsi in una rete progettata senza difese. Da allora la sicurezza è diventata una dimensione trasversale a tutti i livelli, e non a caso incontreremo problemi di sicurezza in quasi ogni protocollo che studieremo: dal DNS all'ARP, da Telnet a HTTP.

Fra le tipologie di attacco che incontreremo, due vanno introdotte fin d'ora perché ricorreranno spesso. Il **DoS** (*Denial of Service*) mira a rendere un servizio indisponibile saturandone le risorse, mentre il **packet sniffing** consiste nell'intercettare passivamente il traffico che transita su un mezzo condiviso: ed è proprio la ragione per cui la natura broadcast di molte reti locali costituisce un problema di riservatezza.

Architettura delle applicazioni di rete

Per lo sviluppatore di applicazioni l'architettura della rete è un dato di fatto: la rete è già lì e offre alle applicazioni un insieme prestabilito di servizi. Il compito dello sviluppatore non è progettare la rete, ma progettare l'**architettura dell'applicazione**, cioè stabilire come essa si organizza sui vari sistemi periferici: chi fa che cosa, chi contatta chi, dove risiedono i dati. Le due architetture principali oggi utilizzate sono quella **client/server** e quella **peer to peer**.

6.1 Architettura client/server

Nell'architettura client/server esiste un host **sempre attivo**, chiamato **server**, che risponde alle richieste di servizio di molti altri host, detti **client**. Fra le applicazioni più note costruite su questo modello troviamo HTTP, IMAP, FTP e Telnet.

Il server ha tre caratteristiche che ne definiscono il ruolo: è un host *always on*, cioè deve essere sempre raggiungibile perché non sa quando arriveranno le richieste; possiede un **indirizzo IP permanente e statico**, senza il quale i client non saprebbero dove trovarlo; e si trova spesso in un **data center**, perché un singolo host raramente basta a sostenere tutto il carico: Google, per fare un esempio, distribuisce il servizio su decine di data center sparsi nel mondo.

Il client presenta caratteristiche complementari e in un certo senso opposte. Comunica con il server ma **non comunica direttamente con altri client**; può avere un **indirizzo IP dinamico**, perché è lui a cercare il server e non viceversa; e può essere **connesso in modo intermittente**, con disconnessioni frequenti e problemi di segnale. Tutta l'asimmetria del modello sta qui: il server deve essere stabile e prevedibile, il client può permettersi di non esserlo.

6.2 Architettura peer to peer (P2P)

In un'architettura **P2P** l'infrastruttura di server in data center è minima o del tutto assente. Si sfrutta invece la **comunicazione diretta fra coppie di host**, chiamati **peer**, cioè *pari*, collegati in modo intermittente. Il termine non è casuale: mentre nel client/server i ruoli sono fissi e asimmetrici, qui ogni peer richiede servizi agli altri e contemporaneamente ne fornisce in cambio. I peer inoltre non appartengono a un fornitore di servizi, ma sono computer fissi e portatili controllati direttamente dagli utenti. L'esempio classico è la condivisione di file P2P.

Il vantaggio più interessante di questo modello è l'**autoscalabilità**: ogni nuovo peer porta con sé nuova capacità di servizio oltre a nuove richieste. Nella distribuzione di un file, suddiviso in parti, ogni peer che ha ricevuto una parte può ridistribuirlo, e la capacità complessiva cresce con il numero dei partecipanti; nel client/server, al contrario, la capacità è fissa e coincide con la banda in uscita del server, che diventa il collo di bottiglia. Da qui discendono gli altri vantaggi: non serve un server sempre attivo e l'architettura risulta economicamente conveniente, perché non richiede infrastrutture significative.

Gli svantaggi sono il rovescio della stessa medaglia. I peer sono connessi in modo intermittente e **cambiano continuamente indirizzo IP**, il che rende la gestione complessa: tenere traccia di chi è disponibile e dove non è banale. A questo si aggiungono problemi di sicurezza, prestazioni e affidabilità, dovuti all'assenza di un'entità centrale che garantisca il servizio.

Processi, socket e indirizzamento

Un **processo** è un programma in esecuzione all'interno di un host. Due processi che si trovano sullo stesso host comunicano tramite la **comunicazione interprocesso**, definita dal sistema operativo; due processi su host diversi comunicano invece **scambiando messaggi** attraverso la rete, e questo scambio consente anche la loro sincronizzazione.

All'interno di una comunicazione si distinguono sempre due ruoli: il **processo client** è quello che avvia la comunicazione, il **processo server** è quello che attende di essere contattato. Nel Web, per esempio, il browser è il processo client e il web server è il processo server. Si noti che questa distinzione è indipendente dall'architettura: anche in una comunicazione fra due peer, in ogni singolo scambio uno dei due assume il ruolo di client e l'altro quello di server.

7.1 Le socket

DEFINIZIONE

La **socket** è l'interfaccia software attraverso cui un processo invia e riceve messaggi: è l'**interfaccia fra il livello di applicazione e il livello di trasporto** all'interno di un host, cioè l'**API** (*Application Programming Interface*) fra l'applicazione e la rete.

La socket si può immaginare come una porta.

Il funzionamento è semplice da visualizzare. Il processo mittente spinge il messaggio fuori dalla propria porta e da quel momento non ne ha più il controllo: si affida all'infrastruttura di trasporto che sta dall'altro lato perché lo consegna alla socket del processo ricevente. In ogni comunicazione sono quindi coinvolte **due socket**, una per ciascun lato. Questa è anche la ragione per cui l'applicazione non deve preoccuparsi di come i dati attraverseranno la rete: tutto ciò che accade oltre la socket è responsabilità dei livelli inferiori.

7.2 Identificare un processo: indirizzo IP e porta

Perché un processo possa ricevere messaggi deve possedere un **identificatore**. Ogni host dispone di un indirizzo IP univoco, a 32 bit in IPv4 e a 128 bit in IPv6, ma il solo indirizzo IP **non è sufficiente** a identificare un processo, perché sullo stesso host possono essere in esecuzione molti processi contemporaneamente. L'identificatore completo comprende quindi sia l'**indirizzo IP**, che è l'identificativo logico della macchina, sia il **numero di porta** associato al processo su quell'host.

Un numero di porta è un intero a **16 bit**, il che significa che esistono 65 536 numeri di porta possibili, da 0 a 65 535. Esiste inoltre un file di sistema che indica quali porte sono assegnate ai vari servizi: è per questo che il browser, quando deve accedere a un servizio web, sa già a quale porta rivolgersi senza che l'utente debba specificarla. Le porte sono suddivise in tre categorie.

CATEGORIA	INTERVALLO	CARATTERISTICHE ED ESEMPI
Well Known Ports	da 0 a 1023	Riservate a servizi noti e riconosciuti a livello globale; il loro utilizzo richiede in genere privilegi amministrativi. Esempi: 80 (HTTP), 443 (HTTPS), 22 (SSH), 25 (SMTP), 53 (DNS).
Registered Ports	da 1024 a 49151	Assegnate dall'ICANN a servizi specifici che ne fanno richiesta. Esempi: 3306 (MySQL), 5432 (PostgreSQL).
Dynamic / User Ports	da 49152 a 65535	Liberamente utilizzabili dalle applicazioni utente, tipicamente in modo temporaneo come porta sorgente di una connessione.

7.3 Che cosa definisce un protocollo applicativo

Un protocollo a livello di applicazione stabilisce come i processi di un'applicazione si scambiano messaggi. In particolare ne definisce i **tipi** (di richiesta o di risposta), la **sintassi**, cioè quali campi compongono i messaggi e come questi vengono delimitati, la **semantica**, cioè il significato delle informazioni contenute nei campi, e infine le **regole** che determinano quando e come un processo invia un messaggio e come vi risponde.

I protocolli applicativi si dividono inoltre in due famiglie a seconda della loro accessibilità. Gli **open protocols** sono definiti nelle **RFC** (*Request for Comments*) e sono di dominio pubblico: chiunque può leggerne la specifica e realizzarne un'implementazione: è il caso di HTTP e SMTP. I **proprietary protocols** non sono invece disponibili al pubblico, come nel caso di Skype. La differenza non è puramente formale: solo un protocollo aperto può essere implementato liberamente da più soggetti e dunque diventare uno standard universale.

Servizi di trasporto richiesti dalle applicazioni

Un protocollo a livello di trasporto ha la responsabilità di consegnare i messaggi alla socket del processo ricevente. Nel progettare un'applicazione occorre quindi scegliere quale protocollo di trasporto utilizzare, e questa scelta si fa valutando quattro dimensioni del servizio.

TRASFERIMENTO DATI AFFIDABILE

Alcune applicazioni, come il trasferimento di file o le transazioni web, richiedono che i dati arrivino integri e completi: la perdita anche di un solo byte comprometterebbe il risultato. Altre, come le applicazioni audio e video, possono invece **tollerare una certa perdita** senza che l'utente ne risenta in modo grave.

THROUGHPUT

Il **throughput** è la velocità alla quale il processo mittente può inviare i bit al processo ricevente, misurata in **bps**, cioè bit al secondo. Un servizio di trasporto potrebbe garantire all'applicazione un **throughput di almeno r bps**.

Rispetto a questo requisito le applicazioni si dividono in due categorie. Le **applicazioni multimediali** richiedono una quantità minima di throughput per essere efficaci: sotto una certa soglia la videochiamata si blocca e il video va a scatti. Le **applicazioni elastiche** sfruttano invece qualunque throughput ottengano, andando semplicemente più veloci o più lente: è il caso del trasferimento di file o della posta elettronica.

TEMPORIZZAZIONE

Alcune applicazioni, come la telefonia su Internet e i giochi interattivi, richiedono un **ritardo basso** per risultare utilizzabili. Qui conta la tempestività: un dato che arriva in ritardo equivale a un dato perso.

SICUREZZA

Sotto questa voce ricadono l'integrità dei dati, la crittografia, la confidenzialità e l'autenticazione, cioè tutte le garanzie che riguardano non il *se* e il *quando* i dati arrivano, ma *chi* può leggerli e modificarli lungo il percorso.

8.1 Che cosa offrono davvero TCP e UDP

Internet mette a disposizione delle applicazioni due soli protocolli di trasporto, TCP e UDP, e nessuno dei due offre tutti i servizi appena elencati.

TCP fornisce un trasporto dati **affidabile** fra processo mittente e processo ricevente, il **controllo di flusso**, che impedisce al mittente di sovraccaricare il ricevitore, e il **controllo di congestione**, che limita il mittente quando la rete è sovraccarica. È inoltre **connection oriented**: prima che i messaggi applicativi comincino a fluire, client e server si scambiano informazioni di controllo a livello di trasporto attraverso una procedura detta **handshaking**, che li prepara alla partenza dei pacchetti. Ciò che TCP **non** fornisce è altrettanto importante da ricordare: nessuna garanzia di **tempistica**, nessuna garanzia di **throughput minimo** e nessuna **sicurezza**, che si ottiene aggiungendo protocolli appositi come TLS al di sopra di TCP.

UDP offre un trasferimento dati **inaffidabile** ed è **connectionless**. Non fornisce affidabilità, controllo di flusso, controllo di congestione, tempistica, throughput garantito, sicurezza né instaurazione della connessione. È, in altre parole, poco più di un'estensione minimale del livello di rete che aggiunge soltanto la capacità di indirizzare i processi tramite le porte.

Il protocollo HTTP

HTTP (*HyperText Transfer Protocol*) è il protocollo a livello di applicazione fondamentale del World Wide Web. Nasce alla fine degli **anni Ottanta** per permettere il recupero di documenti pubblici sul web, e per farlo servivano due cose insieme: un linguaggio capace di rappresentare la struttura delle pagine, cioè l'**HTML**, e un protocollo capace di trasferirle, cioè appunto HTTP. Il protocollo definisce sia la struttura dei messaggi sia la modalità con cui client e server se li scambiano.

9.1 Pagine, oggetti e URL

Una pagina web non è un'entità unica: è composta da **oggetti**, ciascuno dei quali può essere memorizzato su server diversi. Un oggetto può essere un file HTML, un'immagine JPEG, un applet Java o un file audio. In genere una pagina consiste di un **file HTML principale** che include diversi oggetti referenziati, ognuno indirizzabile tramite un **URL** (*Uniform Resource Locator*).

Ogni URL ha due componenti essenziali: il **nome dell'host** del server che ospita l'oggetto e il **percorso** dell'oggetto all'interno di quel server. Nella forma completa che si scrive nel browser compare anche il protocollo da utilizzare.

```
http://www.sito.it/percorso/file.html
|           |           |
protocollo  hostname   percorso della risorsa
```

Struttura di un URL

Poiché negli URL non è possibile inserire spazi né altri caratteri speciali, si ricorre a una codifica: la sequenza **%20**, per esempio, rappresenta lo spazio secondo il codice ASCII. È lo stesso motivo per cui, guardando la barra degli indirizzi, si incontrano talvolta sequenze percentuali al posto dei caratteri accentati.

9.2 HTTP e il livello di trasporto

HTTP utilizza il modello **client/server**: il client è il browser, che richiede, riceve e visualizza gli oggetti web; il server è il web server, che invia gli oggetti in risposta alle richieste.

Come protocollo di trasporto HTTP utilizza **TCP**. Il client per prima cosa inizia una connessione TCP con il server; una volta stabilita, i due processi accedono a TCP attraverso le rispettive socket. Il client invia le richieste e riceve le risposte tramite la propria socket, il server fa lo stesso dalla parte opposta. Vale la pena notare un dettaglio concettuale: quando il client ha consegnato un messaggio alla propria socket, quel messaggio **non è più in suo possesso** ma si trova ormai «nelle mani» di TCP, che se ne assume la responsabilità fino alla consegna.

9.3 Stateless e stateful

DEFINIZIONE

Un sistema si dice **stateless**, cioè senza stato, quando non mantiene alcuna informazione sulle richieste precedenti: ogni richiesta è **indipendente dalle altre**. Si dice invece **stateful**, cioè con stato, quando mantiene memoria delle operazioni precedenti e il suo comportamento dipende da esse.

In un sistema stateless ogni richiesta viene trattata come se fosse la prima. L'esempio più immediato è la stampa di un singolo biglietto, un'operazione autoconclusiva che non dipende da nulla di ciò che è accaduto prima.

Per un sistema stateful l'esempio della **prenotazione di un biglietto del treno** rende bene l'idea, perché la sequenza è obbligata: prima si paga, poi viene stampato il biglietto. Il secondo passo ha senso soltanto se il primo è avvenuto, e quindi il sistema deve necessariamente ricordare a che punto ci si trova. Esistono in altre parole **più stati consecutivi** che devono susseguirsi in un ordine preciso. Entrambi i tipi di sistema utilizzano memoria, ma solo il secondo la impiega per conservare la cronologia dell'interazione.

HTTP è stateless: i server non mantengono informazioni sullo stato delle richieste passate dei client. Non esiste alcuna nozione di scambio multifase per completare una transazione, né la necessità per client e server di tracciare lo stato di uno scambio articolato; tutte le richieste HTTP sono indipendenti l'una dall'altra.

La ragione di questa scelta è pratica. I protocolli che mantengono lo stato sono **complessi**, per due motivi precisi: la cronologia passata deve essere conservata, con il costo di memoria che ciò comporta; e soprattutto, se il server o il client si bloccassero, le loro rispettive visioni dello stato potrebbero risultare **incoerenti** e andrebbero riconciliate. Rinunciare allo stato significa rinunciare anche a questo intero ordine di problemi.

Va però chiarito un equivoco frequente. Il fatto che un sito ci mantenga «loggati» durante la navigazione non contraddice la natura stateless di HTTP: quella memoria non appartiene al protocollo, ma viene costruita al di sopra di esso attraverso i **cookie**, che vedremo fra poco. HTTP resta quindi un protocollo **stateless**, anche se nella pratica quotidiana navighiamo su siti che ci riconoscono; FTP, come vedremo, è invece l'esempio tipico di protocollo stateful.

9.4 Connessioni non persistenti e persistenti

Nelle applicazioni Internet client e server comunicano spesso a lungo, con il client che invia richieste e il server che risponde. La scelta progettuale chiave è se utilizzare **connessioni non persistenti**, in cui ogni coppia richiesta/risposta viaggia su una connessione TCP separata, oppure **connessioni persistenti**, in cui molte coppie richiesta/risposta condividono la stessa connessione.

IL CASO NON PERSISTENTE

Conviene seguire un esempio concreto. Supponiamo che una pagina consista di un file HTML principale e di **dieci immagini JPEG**, tutte residenti sullo stesso server. La sequenza è la seguente. Il client inizializza una connessione TCP con il server sulla **porta 80**, che è la porta di default per HTTP, e alla connessione vengono associate due socket, una per lato. Il client invia attraverso la propria socket un messaggio di richiesta HTTP contenente il percorso del file. Il server riceve la richiesta, recupera l'oggetto dalla memoria, lo incapsula in un messaggio di risposta e lo invia. A quel punto il server comunica a TCP di chiudere la connessione, ma TCP la termina soltanto quando il client ha ricevuto integro il messaggio. Il client riceve la risposta, la connessione termina, ed esaminando il file HTML trova i riferimenti ai dieci oggetti JPEG: per ciascuno di essi l'intera procedura viene **ripetuta da capo**.

Il risultato è che in questo esempio vengono generate **undici connessioni TCP** distinte, ciascuna delle quali trasporta soltanto una richiesta e una risposta. Il limite è evidente: per ogni oggetto richiesto occorre stabilire una nuova connessione, con i ritardi e il carico sul server che ne conseguono.

Per quantificare quel ritardo si introduce il concetto di **RTT (Round Trip Time)**, cioè il tempo impiegato da un pacchetto per viaggiare dal client al server e tornare indietro. È importante ricordare che l'RTT non è solo tempo di volo del segnale: **include anche** il ritardo di propagazione, quello di accodamento nei router e quello di elaborazione del pacchetto. Poiché l'apertura di ogni connessione TCP richiede un *handshake a tre vie*, che studieremo in dettaglio nel livello di trasporto, ogni oggetto costa almeno un RTT aggiuntivo prima ancora che il trasferimento cominci.

IL CASO PERSISTENTE

Con le connessioni persistenti il server **lascia la connessione TCP aperta** dopo aver inviato una risposta, cosicché le richieste e le risposte successive fra gli stessi client e server possono viaggiare sulla stessa connessione. Il server può così inviare un'intera pagina web, o addirittura più pagine, su un unico collegamento, e lo chiude solo quando questo rimane inattivo per un certo lasso di tempo.

9.5 Il messaggio di richiesta HTTP

Il messaggio di richiesta è costituito da righe separate da caratteri di ritorno a capo. La **prima riga** è la *riga di richiesta* e contiene tre elementi: il **metodo**, l'**URL** della risorsa e la **versione** di HTTP utilizzata. Le righe successive sono le **righe di intestazione** e forniscono informazioni aggiuntive nella forma *chiave: valore*; possono essercene diverse. Può infine essere presente un **corpo**, che contiene eventuali dati.

```
GET /percorso/file.html HTTP/1.1    ← riga di richiesta
Host: www.sito.it                  ← righe di intestazione
Connection: close
Accept: text/html
                                     ← riga vuota
(corpo del messaggio, eventuale)
```

Struttura di un messaggio di richiesta HTTP

Fra le intestazioni più significative, **Host** specifica l'host dell'oggetto richiesto, informazione indispensabile quando più siti condividono lo stesso indirizzo IP; **Connection** indica al server se chiudere la connessione dopo l'invio dell'oggetto (valore *close*) oppure mantenerla aperta; **Accept** elenca i tipi di contenuto che il client è in grado di gestire. Altre intestazioni comunicano quale browser sta effettuando la richiesta, cosa che permette al server di adattare la risposta, e la lingua preferita dall'utente.

Quanto al corpo, esso è **vuoto** quando il metodo è GET, mentre viene **utilizzato dal metodo POST** quando l'utente invia dati attraverso un form. Va però osservato che i form HTML utilizzano spesso anche il metodo GET, includendo in tal caso i dati immessi direttamente nell'URL richiesto.

I METODI HTTP

Con il metodo **GET** i dati dell'utente vengono inclusi nel campo URL del messaggio di richiesta, dopo il simbolo «?». Questo significa che i parametri sono **visibili nella barra degli indirizzi**, come nell'esempio seguente.

```
GET /login?user=mrossi&password=1234 HTTP/1.1
```

Con il metodo **POST** i dati vengono invece inviati nel **corpo** della richiesta. È il metodo che si utilizza quando i dati sono numerosi, per esempio in un form lungo, o quando non si desidera inserirli nell'URL. Va detto con chiarezza che POST **non cifra** i dati: li rende soltanto meno immediatamente visibili, ma con uno strumento di analisi del traffico come **Wireshark** restano perfettamente intercettabili. La differenza fra GET e POST è di collocazione, non di sicurezza; per la sicurezza serve HTTPS.

Il metodo **HEAD** richiede soltanto le intestazioni che verrebbero restituite se l'URL fosse richiesto con GET, senza il corpo: consente quindi di verificare la correttezza del riferimento senza scaricare l'oggetto completo. Il metodo **PUT** carica un nuovo oggetto sul server oppure sostituisce integralmente il file esistente all'URL specificato, mentre **DELETE** ne richiede la rimozione. Va notato che alcuni di questi metodi vengono spesso limitati o disabilitati sui server pubblici proprio per motivi di sicurezza: consentire a chiunque di eseguire PUT o DELETE significherebbe permettergli di modificare il sito.

9.6 Il messaggio di risposta HTTP

Il messaggio di risposta si divide in tre sezioni. La prima riga è la **riga di stato** e contiene la versione del protocollo, un **codice di stato** e il corrispondente messaggio testuale. Seguono le **righe di intestazione**, e infine il **corpo del messaggio**, che contiene l'oggetto richiesto.

```
HTTP/1.1 200 OK           ← riga di stato
Date: Tue, 21 Jul 2026 10:15:00 GMT ← righe di intestazione
Server: Apache/2.4
Connection: close

<html> ... </html>        ← corpo: l'oggetto richiesto
```

Struttura di un messaggio di risposta HTTP

I codici di stato sono raggruppati per classi e i principali da conoscere sono cinque. **200 OK** indica che la richiesta è avvenuta con successo e le informazioni sono state inviate. **301 Moved Permanently** segnala che l'oggetto è stato trasferito in modo permanente, e il nuovo URL viene specificato nell'intestazione *Location* della risposta. **400 Bad Request** è un errore generico che indica che la richiesta non è stata compresa dal server. **404 Not Found**, il più noto, segnala che il documento richiesto non è disponibile. **505 HTTP Version Not Supported** indica che il server non supporta la versione di protocollo richiesta.

9.7 I cookie

Poiché HTTP è stateless, siti web e browser utilizzano i **cookie** per mantenere uno stato fra le transazioni. I cookie sono piccoli file di testo contenenti codici alfanumerici salvati dal browser, e il meccanismo si articola su **quattro componenti**: una riga di intestazione nel messaggio di risposta HTTP, una riga di intestazione nel messaggio di richiesta successivo, un file dei cookie conservato sull'host dell'utente e gestito dal browser, e infine un database back end presso il sito web.

Il funzionamento, quando un utente accede a un sito per la prima volta, è il seguente. Il server crea un **identificativo univoco** e lo invia al browser attraverso l'intestazione **Set Cookie** del messaggio di risposta (nel protocollo le due parole sono unite da un trattino). Il browser memorizza l'identificativo nel proprio file dei cookie, associandolo all'host del server. Durante le visite successive il browser include automaticamente l'intestazione **Cookie** nei messaggi di richiesta, comunicando quell'ID univoco. Il server, ritrovando l'identificativo nel proprio database, sa con chi sta parlando: e in questo modo uno stato viene ricostruito al di sopra di un protocollo che non ne possiede.

Gli usi dei cookie sono molteplici: mantenere lo stato della sessione e le sessioni di login, realizzare il **carrello della spesa** virtuale, memorizzare le preferenze dell'utente, identificarlo e **tracciarne il comportamento**. Su quest'ultimo punto occorre distinguere: i **cookie proprietari** tracciano il comportamento dell'utente su un determinato sito, mentre i **cookie di terze parti** lo tracciano attraverso **più siti diversi**, senza che l'utente abbia mai scelto di visitare il sito del tracciatore. I principali browser stanno limitando progressivamente i cookie di terze parti, con politiche che variano da un browser all'altro.

Il problema che ne deriva è la **violazione della privacy**: i cookie persistenti di terze parti raccolgono informazioni personali che possono essere cedute ad altri soggetti, rendendo possibile la **profilazione** dell'utente. Quando i cookie permettono di identificare un individuo, essi sono a tutti gli effetti **dati personali** e ricadono sotto la disciplina del **GDPR**.

IL GDPR

Il **GDPR** (*General Data Protection Regulation*) è il regolamento europeo sulla protezione dei dati personali, applicato in tutti gli Stati membri dal **maggio 2018**. Stabilisce regole sulla raccolta, sul trattamento e sulla conservazione dei dati, e si fonda su alcuni principi: trasparenza nel trattamento, **consenso esplicito** dell'utente, diritto alla cancellazione dei dati e tutela della privacy. È per questa ragione che i siti web sono tenuti a informare gli utenti sull'uso dei cookie e a chiederne il consenso. Qui interessa in quanto conseguenza normativa del meccanismo dei cookie.

9.8 L'evoluzione di HTTP

Le successive versioni di HTTP possono essere lette come una lunga lotta contro lo stesso avversario (la **latenza**) combattuta ogni volta su un fronte diverso.

HTTP 1.0

La versione originaria **chiude ogni connessione TCP dopo che l'oggetto è stato trasferito**: è il modello non persistente descritto sopra, con il suo costo di undici connessioni per una pagina con dieci immagini.

HTTP 1.1

Introduce le **connessioni persistenti** e il **pipelining**: le richieste possono essere effettuate una di seguito all'altra senza dover attendere le risposte precedenti. Resta però un vincolo importante, che è la vera chiave per capire il passaggio alla versione successiva: il server risponde in ordine **FCFS** (*first come, first served*), cioè nello stesso ordine in cui le richieste sono arrivate.

Da qui nasce il problema noto come **HOL blocking** (*Head of Line blocking*): con una politica FCFS, un oggetto piccolo può essere costretto ad attendere la trasmissione di uno o più oggetti di grandi dimensioni che lo precedono in coda. L'immagine è quella della fila alla cassa, dove chi ha un solo articolo resta bloccato dietro chi ha il carrello pieno. I browser aggirano in genere il problema aprendo **più connessioni TCP parallele**, il che funziona ma è una soluzione di ripiego. Va aggiunto infine che il recupero delle perdite, cioè la ritrasmissione dei segmenti TCP persi, blocca a sua volta la trasmissione dell'oggetto.

HTTP 2

Definito nella **RFC 7540 del 2015** (oggi aggiornata dalla RFC 9113), ha come obiettivo dichiarato la **riduzione del ritardo** nelle richieste HTTP che coinvolgono più oggetti. È importante capire che HTTP 2 non stravolge il protocollo: metodi, codici di stato, URL e la maggior parte dei campi di intestazione restano **invariati** rispetto a HTTP 1.1. Ciò che cambia è il modo in cui i dati vengono **formattati e trasportati** fra client e server.

Le novità sono quattro. Viene introdotto un **ordine di trasmissione basato sulla priorità** degli oggetti specificata dal client, che non è più necessariamente FCFS. Il server può inviare al client **oggetti non richiesti**, anticipando ciò che prevede gli servirà: è l'operazione di *push*. Gli oggetti vengono **divisi in frame** proprio allo scopo di contrastare il blocco HOL: i frammenti di oggetti diversi possono così essere intrecciati sulla stessa connessione, e un oggetto lento non blocca più quelli che lo seguono. Viene infine introdotta la **compressione delle intestazioni**, che riduce l'overhead dato che gli header sono spesso ripetitivi e voluminosi.

Restano però due limiti. HTTP 2 non aggiunge **alcuna sicurezza** sulla connessione TCP, e soprattutto il recupero delle perdite continua a bloccare la trasmissione: il blocco HOL è stato eliminato a livello HTTP, ma sopravvive a livello TCP, perché è TCP a imporre la consegna ordinata dei byte.

HTTP 3

È il cambiamento più radicale, perché interviene sul livello di trasporto. HTTP 3 abbandona TCP e si basa sul protocollo **QUIC** (nato in Google come acronimo di *Quick UDP Internet Connections*), implementato **su UDP**, allo scopo di ridurre la latenza. Aggiunge inoltre **sicurezza**, **controllo degli errori** e **controllo della congestione** direttamente a livello di trasporto. La logica del passaggio è comprensibile alla luce del limite di HTTP 2: se il vero ostacolo residuo era la consegna ordinata imposta da TCP, l'unico modo di superarlo era cambiare protocollo di trasporto.

Un'ultima osservazione pratica: i protocolli HTTP sono **retrocompatibili**. Se un server non supporta una versione più recente, il client può negoziare automaticamente una versione precedente, cosicché la comunicazione avviene comunque.

VERSIONE	NOVITÀ INTRODOTTA	LIMITE RESIDUO
HTTP 1.0	Modello base richiesta/risposta	Una connessione TCP per ogni oggetto
HTTP 1.1	Connessioni persistenti, pipelining	Risposte in ordine FCFS → HOL blocking
HTTP 2	Priorità, push, frame, compressione header	HOL eliminato a livello HTTP ma non a livello TCP; nessuna sicurezza
HTTP 3	Trasporto QUIC su UDP , sicurezza e controllo congestione integrati	Nessuno di rilievo

9.9 Proxy server

Un **proxy server** è un'entità di rete che soddisfa richieste HTTP **al posto del web server effettivo**. Dispone di una propria memoria su disco, la **cache**, in cui conserva copie degli oggetti richiesti di recente. Il suo impiego ha senso soprattutto quando molti utenti, tipicamente appartenenti alla stessa rete, accedono agli stessi contenuti.

Il funzionamento si articola in quattro passi. Il browser stabilisce una connessione TCP con il proxy e gli invia la richiesta HTTP. Il proxy controlla se possiede una copia dell'oggetto nella cache: se la trova, la inoltra immediatamente al browser dentro un messaggio di risposta. Se invece non ce l'ha, apre una connessione TCP verso il **server di origine** e ne richiede l'oggetto. Ricevutolo, ne salva una copia nella propria memoria locale e ne inoltra un'altra al browser.

Un proxy è quindi **contemporaneamente server e client**: server per il browser che lo interroga, client per il server di origine.

I vantaggi sono la **riduzione dei tempi di risposta** alle richieste dei client e la **riduzione del traffico globale**. Esiste però un problema: la copia conservata in cache potrebbe essere **scaduta**, cioè non corrispondere più al contenuto attuale del server di origine. Per affrontarlo si utilizza il **GET condizionale**, che permette alla cache di verificare se i propri oggetti sono ancora aggiornati chiedendo al server di trasmettere l'oggetto solo se è stato modificato dopo una certa data.

9.10 Pagine dinamiche e CGI

Una pagina **dinamica** è costruita al momento della richiesta anziché conservata già pronta sul server. Il **CGI** (*Common Gateway Interface*) è la tecnologia standard con cui un web server si interfaccia ad applicazioni esterne per generarla: se l'URL richiesto corrisponde a un documento HTML, il server restituisce un **documento statico**; se corrisponde a un programma CGI, il server **lo esegue in tempo reale** e restituisce il risultato. Il termine **DHTML** (*Dynamic HTML*) indica invece pagine modificate direttamente nel browser tramite JavaScript e CSS. Oggi si usano linguaggi come **PHP** lato server e **JavaScript** lato client, con contenuti spesso attinti da un **database**.

Telnet e FTP

Il capitolo presenta due protocolli storici del livello applicativo: Telnet, per il controllo remoto di una macchina tramite riga di comando, e FTP, per il trasferimento di file, con particolare attenzione ai loro limiti di sicurezza.

10.1 Telnet

Telnet (*Teletype Network*) è un protocollo a livello di applicazione che si appoggia a **TCP** e utilizza la **porta 23**. Il suo scopo principale è gestire dispositivi in remoto attraverso una riga di comando: apre di fatto una shell su un'altra macchina, permettendo all'utente di controllarla via rete. Perché funzioni servono un processo server Telnet attivo sulla macchina remota e un client Telnet in esecuzione sul computer dell'utente.

Gli utilizzi comuni sono due: l'**accesso remoto ai dispositivi**, soprattutto apparati di rete come switch e router, e la **diagnostica di rete**, per verificare la reattività di un server e stabilire se determinate porte siano aperte.

Un aspetto che vale la pena capire è che Telnet può essere usato in modo del tutto **generico**: pur essendo associato alla porta 23, può comunicare con qualsiasi porta di un sistema remoto, purché i messaggi vengano inviati nel formato corretto. Possiamo per esempio collegarci con Telnet alla porta 80 di un web server e digitare a mano un comando come *GET / HTTP/1.1*, ricevendo la risposta del server. Questo funziona perché molti protocolli applicativi scambiano messaggi **in chiaro e in formato testuale**, ed è ciò che ha reso Telnet uno strumento prezioso per testare i protocolli: lo rivedremo fra poco con SMTP.

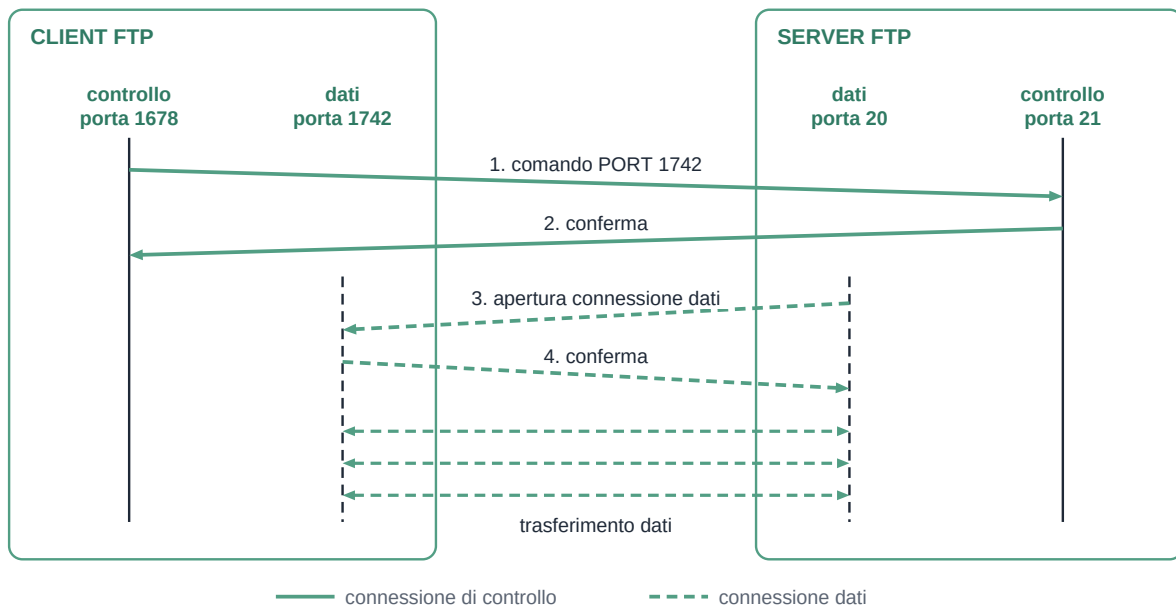
Il problema di Telnet è però esattamente lo stesso che lo rende comodo: **non cifra i dati**. Tutte le informazioni, comprese le password, viaggiano in chiaro attraverso la rete e sono quindi soggette a intercettazione. L'idea originaria era che la cifratura venisse effettuata sul server, ma questa impostazione lascia esposto l'intero traffico a chiunque stia ascoltando. Per questa ragione Telnet è oggi considerato insicuro ed è stato sostituito da **SSH** (*Secure Shell*).

Nonostante sia obsoleto, Telnet non è scomparso: in rete risultano ancora centinaia di migliaia di dispositivi con la porta 23 aperta. Nel 2026 è stata resa pubblica una vulnerabilità critica del server telnetd incluso in GNU InetUtils, che consentiva di ottenere l'accesso come amministratore senza password manipolando una variabile d'ambiente trasmessa durante la negoziazione iniziale. L'episodio mostra in modo concreto quanto sia rischioso mantenere attivi protocolli progettati senza alcuna attenzione alla sicurezza.

10.2 Il protocollo FTP

Il **File Transfer Protocol** è un protocollo a livello di applicazione utilizzato principalmente per il **trasferimento di file** su una rete. Supporta il trasferimento **bidirezionale**: i file possono essere sia inviati sia ricevuti fra client e server. Il suo utilizzo richiede un'**autenticazione** tramite username e password, e stabilisce una **connessione persistente**, che resta aperta finché l'utente non la chiude o finché non scatta un timeout di inattività.

La caratteristica più distintiva di FTP è che utilizza **due connessioni TCP separate su porte diverse**: la **porta 21** per i **comandi** e la **porta 20** per il **trasferimento dei dati**. La separazione ha una logica precisa: la connessione di controllo resta aperta per tutta la sessione e trasporta i comandi dell'utente, mentre la connessione dati viene aperta e chiusa per ogni singolo trasferimento. È proprio il fatto di mantenere aperta la connessione di controllo, ricordando l'utente autenticato e la directory corrente, a rendere **FTP un protocollo stateful**, in netto contrasto con HTTP.



FTP: connessione di controllo (porta 21) e connessione dati (porta 20)

Anche FTP presenta problemi di sicurezza rilevanti. In primo luogo, abilitare l'accesso all'intero *file system* tramite FTP sarebbe rischioso: occorre stabilire con attenzione quali porzioni rendere accessibili. In secondo luogo, e più gravemente, **credenziali e file sono trasmessi in chiaro**. Il problema è affrontato in due modi distinti. **FTPS** aggiunge a FTP la cifratura **TLS**, in modalità esplicita sulla stessa porta 21 oppure in modalità implicita sulla porta 990. **SFTP** è invece un protocollo diverso, che trasferisce file all'interno di una sessione **SSH** e utilizza quindi la **porta 22**.

DA RICORDARE

Nel **confronto fra HTTP e FTP** i punti essenziali sono: entrambi si appoggiano a TCP, ma **HTTP usa una sola connessione** mentre **FTP ne usa due** (controllo sulla 21, dati sulla 20); **HTTP è stateless**, **FTP è stateful**; HTTP non richiede autenticazione per i contenuti pubblici, FTP la richiede sempre; entrambi, nella versione base, trasmettono in chiaro, e vengono resi sicuri rispettivamente da HTTPS e da FTPS o SFTP.

La posta elettronica

Il capitolo descrive il sistema della posta elettronica: il protocollo SMTP per l'invio e il trasferimento dei messaggi fra mail server, e i protocolli POP e IMAP per l'accesso alla casella da parte dell'utente.

11.1 SMTP e i suoi tre componenti

Il protocollo **SMTP** (*Simple Mail Transfer Protocol*), definito nella **RFC 5321**, è utilizzato per **inviare** messaggi di posta elettronica su Internet. Il sistema della posta si articola su tre componenti principali: gli **user agent**, i **mail server** e il protocollo SMTP stesso.

Lo **user agent** è il programma con cui l'utente legge, scrive, inoltra, salva e compone i messaggi: Gmail, Outlook, Thunderbird. Il **mail server** è il server presso cui i messaggi vengono depositati, conservati e da cui vengono inviati. Il protocollo SMTP è ciò che permette ai mail server di dialogare fra loro.

IL PERCORSO COMPLETO DI UN MESSAGGIO

Supponiamo che Alice voglia inviare un messaggio a Bob. Alice invoca il proprio user agent, fornisce l'indirizzo di Bob, compone il messaggio e ne ordina l'invio. Lo user agent consegna il messaggio al **mail server di Alice**, dove esso viene collocato in una **coda di messaggi**. Il lato client di SMTP, in esecuzione sul server di Alice, vede il messaggio in coda e apre una connessione TCP verso il server SMTP in esecuzione sul mail server di Bob. Dopo un handshaking SMTP, il messaggio viene trasferito sulla connessione. Presso il mail server di Bob il lato server di SMTP lo riceve e lo colloca nella **mailbox** di Bob, che potrà leggerlo quando lo riterrà opportuno invocando il proprio user agent.

Un dettaglio importante riguarda il caso in cui il **mail server di Bob sia spento**: in tal caso il messaggio **rimane sul mail server di Alice** e attende un nuovo tentativo. La posta elettronica non fallisce immediatamente, ma riprova nel tempo: ed è questa una delle ragioni per cui i mail server sono necessari e la posta non viene consegnata direttamente da user agent a user agent.

I COMANDI SMTP

I comandi scambiati fra due mail server sono in **testo ASCII** e le risposte consistono in un codice di stato accompagnato da una frase opzionale. I principali sono **HELO** per l'handshaking, **MAIL FROM** per indicare il mittente, **RCPT TO** per il destinatario, **DATA** per iniziare il corpo del messaggio, il carattere «.» su una riga isolata per segnalarne la fine, e **QUIT** per chiudere.

Poiché i comandi sono testuali, è possibile dialogare direttamente con un server SMTP usando Telnet sulla porta 25, digitando a mano i comandi e inviando così un'email senza alcun client di posta. Ciò funziona però soltanto se il server consente connessioni Telnet a quella porta, cosa che sta diventando sempre più rara proprio per ragioni di sicurezza.

Riassumendo le caratteristiche di SMTP: utilizza **TCP sulla porta 25** per trasferire il messaggio in modo affidabile; fa uso di **connessione persistente**; realizza un **trasferimento diretto** dal server di invio, che agisce da client, al server ricevente; articola il trasferimento in **tre fasi**, handshaking, trasferimento dei messaggi e chiusura; e impiega il **formato ASCII a 7 bit**, il che ne limita la capacità di gestire direttamente i file multimediali: limite superato in seguito dalle estensioni MIME.

11.2 POP e IMAP

Se SMTP serve a **inviare** la posta, per **accedere** alla propria casella occorrono protocolli diversi, ed è qui che entrano in gioco POP e IMAP. La distinzione fra i due è concettualmente semplice ma va colta con precisione, perché riguarda *dove risiedono i messaggi*.

POP (*Post Office Protocol*) **scarica** i messaggi dal server alla macchina del client. Una volta scaricati, i messaggi vengono in genere **rimossi dal server** e gestiti localmente. La conseguenza pratica è che, se l'utente accede da più dispositivi, i messaggi **non risultano sincronizzati** fra di essi: ciò che è stato scaricato sul computer non si troverà più sul telefono. Usa TCP sulla **porta 110** in chiaro e sulla **995** cifrata.

IMAP (*Internet Message Access Protocol*) **sincronizza** invece i messaggi fra server e client. I messaggi **restano sul server** e vengono gestiti in modo centralizzato: questo permette di accedere alla stessa casella da più dispositivi mantenendo la piena coerenza fra di essi: se leggo un messaggio dal telefono, risulterà letto anche dal computer. Usa TCP sulla **porta 143** in chiaro e sulla **993** cifrata.

La scelta fra i due dipende dalle esigenze dell'utente. POP è adatto a un uso locale e su un singolo dispositivo, e ha il vantaggio di non occupare spazio sul server; IMAP è di gran lunga più adatto all'uso moderno della posta, in cui la stessa casella viene consultata da computer, telefono e tablet.

II DNS

Il **DNS** (*Domain Name System*) ha lo scopo di **tradurre gli hostname in indirizzi IP**, così da identificare i dispositivi in rete. Ogni pacchetto che viaggia in rete utilizza infatti un indirizzo IP, ma ricordare a memoria sequenze numeriche sarebbe impraticabile per una persona: si utilizzano quindi **nomi simbolici**, che rendono la navigazione intuitiva.

Il DNS è al tempo stesso due cose, e conviene tenerle distinte: è un **database distribuito**, implementato in una gerarchia di server che contengono le associazioni fra nomi e indirizzi, ed è un **protocollo a livello di applicazione** che consente agli host di interrogare quel database. Utilizza principalmente **UDP sulla porta 53**, e TCP sulla stessa porta per i trasferimenti di zona e per le risposte di grandi dimensioni.

Oltre alla traduzione dei nomi, il DNS mette a disposizione altri servizi. L'**host aliasing** consente a un host dal nome complicato di avere uno o più sinonimi più semplici da ricordare; analogamente esiste il **mail server aliasing**. Particolarmente interessante è la **load distribution**: nel caso di server web replicati, a un solo nome corrispondono molti indirizzi IP, e quando un client effettua una query il server risponde con l'intero insieme di indirizzi **variandone però l'ordine a ogni risposta**. Poiché il client contatta di norma il primo indirizzo dell'elenco, il DNS finisce per distribuire il traffico fra i server replicati, svolgendo di fatto una funzione di bilanciamento del carico.

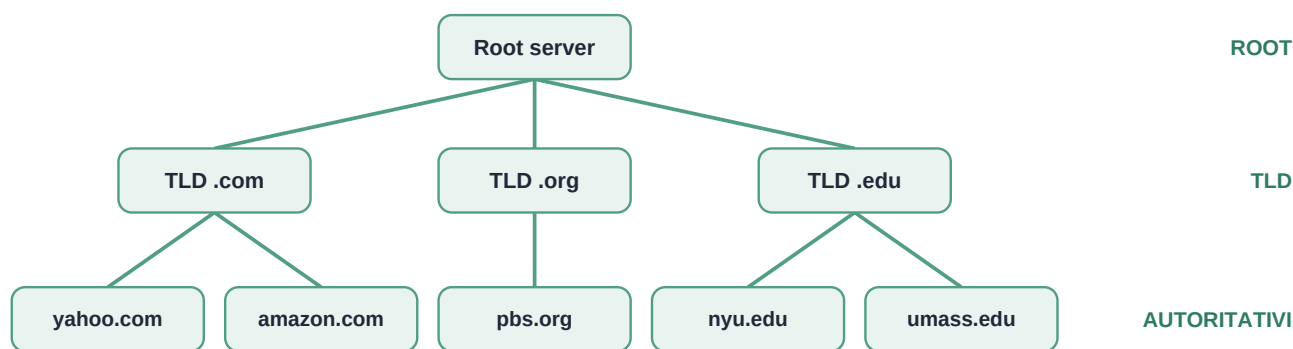
CHE COSA ACCADE PRIMA DI UNA RICHIESTA HTTP

Vale la pena vedere la sequenza completa, perché mostra come DNS e HTTP si incastrano. La macchina dell'utente esegue il lato client dell'applicazione DNS. Il client DNS invia una **query** contenente l'hostname a un server DNS. Prima o poi riceve una risposta contenente l'**indirizzo IP** corrispondente, trasportata all'interno di datagrammi UDP diretti alla porta 53. Solo a quel punto il browser può dare inizio alla connessione TCP verso il processo server HTTP collegato alla porta 80 di quell'indirizzo. In altre parole, **ogni navigazione web comincia con una risoluzione DNS**.

12.1 Perché il DNS è distribuito e gerarchico

Si potrebbe immaginare di realizzare il DNS come un unico grande database centralizzato, ma questa soluzione presenterebbe problemi insormontabili. Vi sarebbe un **single point of failure**: se quel server si guastasse, ne soffrirebbe l'intera Internet. Il **volume di traffico** sarebbe ingestibile, dovendo un solo server rispondere a tutte le richieste del mondo. Il database sarebbe inoltre **distante** dalla maggior parte degli utenti, con tempi di risposta elevati, e la sua **manutenzione** risulterebbe complessa. In una parola, il sistema non sarebbe **scalabile**.

Per affrontare la scalabilità il DNS utilizza un gran numero di server organizzati **gerarchicamente** e distribuiti nel mondo, appartenenti a tre classi: i **root server**, i **TLD server** e i **server autoritativi**.



La gerarchia dei server DNS: root, TLD e server autoritativi

Il meccanismo si comprende meglio seguendo un caso concreto: un client desidera l'indirizzo IP di *www.amazon.com*. Il client interroga anzitutto un **root server** per individuare il server DNS competente per *.com*. Interroga poi uno dei **TLD server** di *.com* per ottenere il server DNS di *amazon.com*. Infine interroga uno dei **server autoritativi** di *amazon.com* per ottenere l'indirizzo IP cercato. Ogni livello, in sostanza, non conosce la risposta finale ma sa **a chi rivolgersi** per avvicinarsi ad essa.

I ROOT SERVER

Esistono più di mille root server, che sono repliche di **13 root server** logici. Forniscono gli indirizzi IP dei TLD server e rappresentano il contatto di ultima istanza per i server che non riescono a risolvere un nome. La loro sicurezza è garantita da **DNSSEC**, che assicura autenticazione e integrità dei messaggi, e il dominio del root DNS è gestito dall'**ICANN**.

I TLD SERVER

Si occupano dei **domini di primo livello**, sia quelli generici come *.com*, *.org*, *.net*, *.edu* e *.gov*, sia quelli nazionali come *.it*, *.uk*, *.fr*, *.ca* e *.jp*. Il loro compito è fornire gli indirizzi dei server autoritativi.

I SERVER AUTORITATIVI

Ogni organizzazione che disponga di host pubblicamente accessibili deve fornire **record DNS pubblicamente accessibili** che associno i nomi di tali host ai rispettivi indirizzi IP. Questi server possono essere gestiti dall'organizzazione stessa oppure dal suo ISP, e per ogni dominio ne sono previsti almeno due.

Il **DNS primario** è il server autoritativo principale: è l'unico su cui gli amministratori possono **modificare i record**. Il **DNS secondario** è una copia del primario e serve a garantire affidabilità e disponibilità; le informazioni memorizzate sui due sono identiche. Il secondario **non può essere modificato manualmente**: si aggiorna automaticamente copiando i dati dal primario attraverso un meccanismo chiamato **zone transfer**. È importante osservare che gli aggiornamenti **non sono istantanei**: prima vengono effettuati sul primario e poi si propagano progressivamente ai secondari.

I SERVER DNS LOCALI

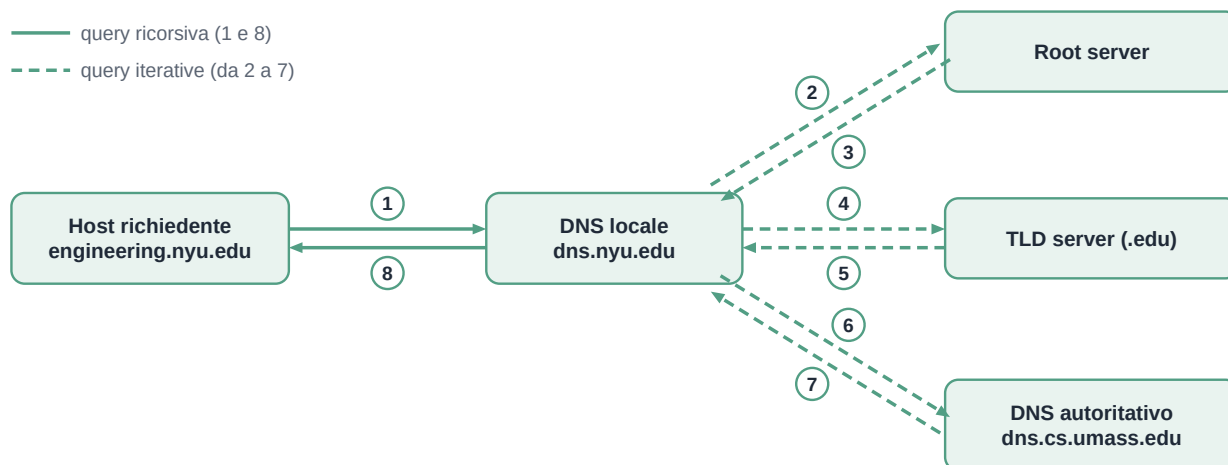
Il **DNS locale** occupa una posizione particolare, perché **non appartiene strettamente alla gerarchia**. Ogni ISP ne possiede uno, chiamato *default name server*, e quando un host effettua una query la invia proprio a lui.

Il suo ruolo si riassume in due funzioni. Da un lato **conosce i nomi locali** e mantiene una **cache** delle traduzioni nome e indirizzo recenti, che gli consente di rispondere immediatamente a molte richieste: con l'avvertenza che tali informazioni potrebbero non essere più aggiornate. Dall'altro, quando non conosce la risposta, sa **a chi chiedere**: agisce come *proxy*, inoltrando la query verso la gerarchia. Poiché il DNS locale può essere molto sollecitato, anche a questo livello si prevedono in genere un server primario e uno o più secondari.

12.2 Risoluzione ricorsiva e iterativa

Le query DNS possono seguire due modalità, ed è utile distinguerle perché nella pratica coesistono nella stessa risoluzione. In una query **ricorsiva** l'host che riceve la domanda si assume l'onere di trovare la risposta completa, interrogando a sua volta altri server e restituendo al richiedente il risultato finale. In una query **iterativa** il server interrogato non svolge il lavoro al posto del richiedente, ma si limita a rispondere indicando **a chi rivolgersi** per il passo successivo.

Tipicamente la richiesta che l'host invia al proprio DNS locale è **ricorsiva** (l'host delega interamente il compito), mentre le successive richieste che il DNS locale rivolge a root, TLD e server autoritativi sono **iterative**: ciascuno risponde indicando il livello successivo, e il DNS locale procede da sé lungo la catena.



Risoluzione di un nome: query ricorsiva verso il DNS locale, iterative lungo la gerarchia

12.3 Il caching DNS

Il DNS introduce inevitabilmente un **ritardo aggiuntivo** prima di ogni comunicazione. Per contenerlo, e per ridurre il numero complessivo di messaggi, si ricorre al **DNS caching**: un server DNS che riceve una risposta può memorizzarne le informazioni, e se in seguito gli giunge un'altra richiesta per lo stesso hostname può fornire direttamente l'indirizzo IP **anche se non è autoritativo** per quel nome.

Nella pratica i server DNS locali memorizzano in cache soprattutto gli indirizzi dei **TLD server**, mentre i root server vengono visitati di rado proprio grazie a questo meccanismo. Vanno però considerati due aspetti: le voci in cache **scadono dopo un certo tempo**, indicato dal valore **TTL** associato a ciascun record, e finché non scadono le informazioni memorizzate **potrebbero essere obsolete**. Esistono a tal fine meccanismi standard di aggiornamento e notifica definiti nella RFC 2136.

12.4 I record DNS

I server che implementano il database distribuito memorizzano i **resource record (RR)**, strutturati secondo il formato **(name, value, type, ttl)**. Il campo *type* determina il significato degli altri due, e i quattro tipi da conoscere sono i seguenti.

TYPE	NAME	VALUE	ESEMPIO
A	L'hostname	L'indirizzo IP	(relay1.bar.foo.com, 145.37.93.126, A)
NS	Il dominio	L'hostname del server autoritativo per quel dominio	(foo.com, dns.foo.com, NS)
CNAME	L'alias	Il nome canonico corrispondente	(foo.com, relay1.bar.foo.com, CNAME)
MX	L'alias di un mail server	Il nome canonico del mail server	(foo.com, mail.bar.foo.com, MX)

La coppia **NS + A** è quella che consente di scendere di un livello nella gerarchia: il record NS dice *quale server* è autoritativo per un dominio, e il record A associato fornisce l'indirizzo IP di quel server. La distinzione fra **CNAME** e **MX** merita attenzione perché entrambi trattano alias, ma il primo riguarda host generici e il secondo specificamente i server di posta: è per questo che l'indirizzo del web server e quello del mail server di una stessa organizzazione possono essere completamente diversi.

12.5 I messaggi DNS

Le query e le risposte DNS condividono lo **stesso formato**, articolato in cinque sezioni. La **sezione di intestazione** occupa i primi **12 byte** e contiene: un campo di **identificazione** a 16 bit, che permette di associare ciascuna risposta alla richiesta corrispondente; un campo **flag** a 16 bit, fra i cui bit uno specifica se il messaggio è una query o una risposta; e quattro contatori a 16 bit che indicano il numero di domande, di record di risposta, di record autoritativi e di record addizionali contenuti nelle sezioni seguenti.

Segue la **sezione delle domande**, che include il nome richiesto e il tipo di domanda; la **sezione delle risposte**, presente nei messaggi di risposta e contenente i record veri e propri, che possono essere anche più di uno; la **sezione autoritativa**, con i record di altri server autoritativi; e infine la **sezione aggiuntiva**, con altri record utili correlati alla richiesta.

12.6 Inserire un nuovo dominio nel DNS

Seguire l'esempio della registrazione di un dominio aiuta a chiarire come i pezzi si compongono. Supponiamo di voler registrare il dominio *networkutopia.com*. Ci si rivolge a un **registrar**, cioè a un'azienda che verifica l'unicità del nome e lo inserisce nel database DNS. Al registrar occorre fornire **nomi e indirizzi IP dei propri server autoritativi**, primario e secondario.

Il registrar inserirà allora nei **TLD server di .com**, per ciascuno dei due server autoritativi, un record **NS** e un record **A**. Per il primario, per esempio, verranno inseriti i record (networkutopia.com, dns1.networkutopia.com, NS) e (dns1.networkutopia.com, 212.212.212.1, A). All'interno dei nostri stessi server autoritativi dovranno poi essere immessi i record **A** per il web server *www.networkutopia.com* e i record **MX** per il server di posta *mail.networkutopia.com*. A questo punto la catena è completa: chiunque nel mondo, partendo dai root server, può raggiungere i nostri servizi.

I DNS DINAMICI

Esistono infine i **DNS dinamici**, pensati per il caso in cui l'indirizzo IP **cambi frequentemente**: situazione tipica delle connessioni domestiche. L'utente si registra a un servizio di DNS dinamico e un'applicazione, spesso in esecuzione sul router stesso, comunica periodicamente con il server DNS, che aggiorna il record con il nuovo indirizzo. Lo svantaggio è che l'aggiornamento **può essere lento**, poiché occorre tempo perché le nuove informazioni si propaghino correttamente attraverso la gerarchia e le cache.

12.7 Sicurezza del DNS

Il DNS è un bersaglio particolarmente appetibile, perché chi ne controlla le risposte controlla di fatto la destinazione del traffico. Per fronteggiare gli attacchi sono state introdotte le **DNS Security Extensions (DNSSEC, RFC 4033)**. Le minacce principali sono tre.

Gli **attacchi DDoS** consistono nell'inondare i server di pacchetti, così che molte richieste legittime rimangano senza risposta. Gli attacchi diretti ai **root server** non hanno finora avuto successo, sia perché molti di essi sono protetti da sistemi di filtraggio configurati per bloccare i messaggi ostili, sia perché il caching presso i DNS locali riduce drasticamente la necessità di interrogarli; un'ulteriore contromisura consiste nel mantenere liste di indirizzi di TLD server notoriamente non malevoli. Potenzialmente **più pericolosi** sono invece gli attacchi DDoS rivolti ai **TLD server**.

L'attacco **man in the middle** consiste nell'intercettare le richieste di un host e fornire risposte proprie. Il **DNS poisoning**, detto anche *DNS spoofing*, consiste invece nell'inviare risposte ingannevoli a un server DNS, così da fargli inserire **record fasulli** nel proprio database: le risposte effettive vengono sostituite da risposte fittizie, ottenendo un reindirizzamento verso destinazioni scelte dall'attaccante. La gravità di quest'ultimo attacco sta nella sua **persistenza**: a causa del caching, i danni si propagano a lungo nel tempo, ben oltre il momento dell'attacco.

Il caso concreto più facile da immaginare è quello di una **rete WiFi pubblica**: collegandosi alla rete gratuita di un locale e accedendo al sito della propria banca, si potrebbe ricevere una risposta DNS falsa che indirizza verso una pagina fraudolenta del tutto simile all'originale, con conseguente furto di credenziali e dati bancari. Le contromisure pratiche consistono nell'impostare manualmente un **DNS affidabile**, nel prestare attenzione alle reti pubbliche e nell'utilizzare connessioni protette.

Cenni sul protocollo SNMP

Il **SNMP** (*Simple Network Management Protocol*) è un protocollo per la **gestione e il monitoraggio** di dispositivi di rete come router e switch. Consente agli amministratori di raccogliere informazioni sullo stato dei dispositivi (il carico del processore, l'utilizzo della memoria, l'attività di una porta, la banda impiegata) e di eseguire operazioni di gestione.

Gli attori coinvolti sono tre. Il **manager** è l'applicazione che comunica con i dispositivi per raccogliere informazioni, inviando richieste agli agenti; esegue essenzialmente due operazioni, **Get** per chiedere informazioni e **Set** per modificare un'impostazione. L'**agente** è il software presente su ogni dispositivo da monitorare: gestisce le variabili che rappresentano gli oggetti di sistema, attende le richieste del manager e può inviare **trap**, cioè notifiche spontanee, in seguito a malfunzionamenti o anomalie. La **MIB** (*Management Information Base*) è infine il database presente in ogni agente, che definisce quali informazioni possono essere raccolte o modificate.

Dal punto di vista architetturale SNMP è basato sul paradigma **client/server**, in cui il manager è il client e l'agente è il server, ed è **connectionless**. Utilizza due porte **UDP**: la **161**, su cui il manager invia le richieste agli agenti, e la **162**, su cui gli agenti inviano le trap al manager.

I messaggi si dividono secondo la direzione. Dal manager all'agente viaggiano **GetRequest**, che richiede il valore di una variabile, **SetRequest**, che ne richiede la modifica, e **GetNextRequest**, che richiede il recupero delle variabili successive e dei loro valori. Dall'agente al manager viaggiano **GetResponse**, che restituisce il valore di una o più variabili, e **Trap**, che notifica in modo **asincrono e non sollecitato** situazioni anomale o eventi significativi. È proprio la trap a distinguere SNMP da un semplice protocollo di interrogazione: consente al dispositivo di segnalare un guasto **senza attendere** di essere interrogato.

Servizi e protocolli a livello di trasporto

Il *Transport Layer* mette a disposizione una **comunicazione logica fra processi applicativi** situati su host differenti. L'aggettivo «logica» è essenziale: dal punto di vista dei due processi è come se essi fossero collegati direttamente, anche se in realtà i dati attraversano una rete complessa di nodi intermedi.

Conviene fissare subito la scala dei tre livelli inferiori, perché è una distinzione che ritorna continuamente. Il **Data Link Layer** realizza una comunicazione **fisica fra host direttamente connessi**, cioè fra due nodi adiacenti collegati dallo stesso link. Il **Network Layer** realizza una comunicazione **logica fra host**, occupandosi di portare i pacchetti dalla sorgente alla destinazione attraverso i router. Il **Transport Layer** realizza infine una comunicazione **logica fra processi**. In termini più operativi si dice che il livello di trasporto **estende la consegna host to host**, fornita da IP, **alla consegna process to process**: IP porta i dati al computer giusto, il livello di trasporto li porta al programma giusto dentro quel computer.

Un punto fondamentale è che i protocolli a livello di trasporto sono implementati **nei sistemi periferici ma non nei router**: per questo si dice che funzionano in modo **end to end**. I router lungo il percorso inoltrano datagrammi IP e sono del tutto ignari del fatto che sopra di essi viaggia una connessione TCP con i suoi numeri di sequenza, i suoi ACK e le sue ritrasmissioni. Tutta l'intelligenza sta ai bordi della rete, e questo è uno dei principi di progetto fondamentali di Internet.

Il lavoro concreto del livello si articola in due direzioni. Il **mittente** divide i messaggi applicativi in **segmenti**, aggiungendo a ciascuno un'intestazione di trasporto, e li passa al livello di rete, dove vengono incapsulati in datagrammi. Il **ricevitore** riceve dal livello di rete i segmenti estratti dai datagrammi, li **riassembla** in messaggi applicativi e li consegna al livello di applicazione.

Una rete TCP/IP fornisce due protocolli di trasporto: **TCP**, servizio affidabile e orientato alla connessione, e **UDP**, servizio non affidabile e non orientato alla connessione.

Multiplexing e demultiplexing

Il problema che questi due meccanismi risolvono è concreto: il livello di rete, tramite l'indirizzo IP, riesce a consegnare un datagramma all'host corretto, ma su quell'host possono essere in esecuzione decine di processi: il browser, il client di posta, un download. L'indirizzo IP da solo non basta a stabilire a quale di essi consegnare i dati. Va inoltre osservato che il livello di trasporto non consegna i dati direttamente al processo, ma a una **socket** che fa da intermediario.

DEFINIZIONE

Il **demultiplexing**, che avviene **al ricevitore**, consiste nell'utilizzare le informazioni contenute nell'intestazione del segmento (in particolare i numeri di porta) per consegnarlo alla **socket corretta**. Il **multiplexing**, che avviene **al mittente**, consiste nel raccogliere i dati provenienti da più socket, incapsularli in segmenti aggiungendo l'intestazione di trasporto e passarli al livello di rete.

L'analogia postale rende bene l'idea: il multiplexing è l'ufficio che raccoglie le lettere di tutti gli inquilini e vi scrive mittente e destinatario; il demultiplexing è lo smistamento nelle singole cassette del condominio.

15.1 Il caso connectionless (UDP)

Seguiamo un esempio. Un processo sull'Host A, associato alla porta UDP 19157, vuole inviare dati a un processo sull'Host B, sulla porta 46428. Il livello di trasporto di A crea un segmento che include i dati applicativi e i numeri di porta di origine e destinazione, e lo passa al livello di rete che lo incapsula in un datagramma IP. Il livello di trasporto di B esamina il **numero di porta di destinazione** e consegna il segmento alla socket identificata da quel numero. Si noti che, se B dovesse rispondere, i numeri di porta di origine e destinazione risulterebbero **invertiti**.

La regola che ne deriva è la seguente: una **socket UDP è identificata da una coppia**, indirizzo IP e numero di porta di **destinazione**. La conseguenza pratica è importante: due segmenti UDP che presentano indirizzi IP e/o porte di **origine diversi**, ma stessa destinazione, vengono diretti **allo stesso processo attraverso la stessa socket**. A UDP non interessa chi ha inviato il pacchetto, ma solo a chi è destinato.

15.2 Il caso connection oriented (TCP)

Una **socket TCP è identificata da quattro parametri**: indirizzo IP di origine, porta di origine, indirizzo IP di destinazione e porta di destinazione. Tutti e quattro concorrono all'identificazione, e questo cambia completamente il comportamento. Tre segmenti TCP che presentino indirizzi IP e/o porte di origine differenti, ma tutti destinati allo stesso indirizzo e alla porta 80, vengono diretti a **socket diverse**.

È esattamente questo meccanismo che permette a un web server di gestire migliaia di client contemporaneamente pur utilizzando tutti la stessa porta 80: la distinzione non avviene sulla porta di destinazione, che è comune, ma sulla **quadrupla nel suo complesso**. Anche due connessioni dallo stesso computer, come due schede del browser aperte sullo stesso sito, restano distinte perché hanno **porte sorgente diverse**. In sintesi, **UDP smista guardando una coppia** e **TCP una quadrupla**: con UDP la separazione fra comunicazioni diverse ricade sull'applicazione, con TCP è garantita dal protocollo attraverso socket distinte.

UDP (User Datagram Protocol)

UDP è un protocollo **minimale**: oltre al multiplexing e demultiplexing **non aggiunge nulla a IP**. Lavora in modalità *best effort*, il che significa che i segmenti possono essere persi e possono arrivare in un ordine diverso da quello di invio; aggiunge soltanto un minimo controllo degli errori. Non è orientato alla connessione: non esiste alcun handshaking fra mittente e destinatario, e **ogni segmento è gestito indipendentemente** dagli altri.

Viene naturale chiedersi perché un protocollo tanto povero continui a esistere, e la risposta sta tutta in ciò che UDP **non** deve fare. Non stabilendo alcuna connessione, **non introduce il ritardo di un RTT** prima di poter trasmettere. Non mantenendo alcuno stato di connessione (che richiederebbe buffer di invio e ricezione, variabili, controllo di congestione) consente a un server di **supportare molti più client attivi** contemporaneamente. La sua intestazione è di dimensioni ridottissime, appena **8 byte**. E l'assenza di controllo di congestione fa sì che UDP possa **trasmettere alla velocità desiderata** e continuare a funzionare anche quando la rete è congestionata.

UDP è utilizzato dalle **applicazioni multimediali in streaming**, che sono tolleranti alle perdite ma sensibili alla velocità, da **DNS**, da **SNMP** e da **HTTP 3**. Quest'ultimo caso è istruttivo: se serve un trasferimento affidabile su UDP, l'affidabilità e il controllo di congestione vengono **aggiunti a livello applicativo**, che è precisamente ciò che fa QUIC.

16.1 Il segmento UDP e il checksum

L'header UDP comprende **quattro campi di due byte ciascuno**: il numero di **porta di origine**, il numero di **porta di destinazione**, la **lunghezza**, che indica il numero di byte del segmento intestazione compresa, e il **checksum**.

Il **checksum** serve a rilevare errori durante il trasferimento e si basa su una **somma in complemento a 1** di tutte le parole a 16 bit del segmento, con l'eventuale riporto in uscita sommato al bit meno significativo; il complemento a 1 si ottiene convertendo gli 0 in 1 e viceversa. In ricezione si somma il checksum ricevuto alla somma delle parole: il risultato dovrà essere composto **da tutti bit a 1**, e se anche un solo bit vale 0 sappiamo che è stato introdotto almeno un errore. Con IPv4 il checksum UDP è facoltativo (il valore zero indica che non è stato calcolato), mentre con IPv6 è obbligatorio; in ogni caso rileva soltanto gli errori, senza correggerli.

Principi del trasferimento dati affidabile

L'astrazione che si vuole offrire ai livelli superiori è quella di un **canale affidabile**: nessun bit corrotto, nessun bit perduto e tutti i bit consegnati **nell'ordine di invio**. È precisamente il modello di servizio offerto da TCP. La difficoltà nel realizzarla nasce dal fatto che il canale sottostante è **inaffidabile**: TCP si appoggia a IP, che non garantisce nulla. A ciò si aggiunge un'ulteriore complicazione: mittente e destinatario **non conoscono lo stato dell'altro** (per esempio se un messaggio sia stato ricevuto) a meno che questo non venga comunicato esplicitamente con un messaggio.

Il compito di un protocollo **RDT** (*Reliable Data Transfer*) è ottenere un canale logico affidabile a partire da una connessione di basso livello inaffidabile. Il percorso didattico che seguiremo procede per raffinamenti successivi: si parte dal caso ideale e a ogni passo si rimuove un'ipotesi semplificatrice, osservando quale nuovo problema emerge e come lo si risolve. Per convenzione si considera il solo trasferimento **unidirezionale** dei dati, anche se le informazioni di controllo fluiscono in entrambe le direzioni, e si utilizzano **macchine a stati finiti** per specificare mittente e destinatario.

17.1 RDT 1.0: il canale perfetto

Si assume un canale **perfettamente affidabile**: nessun errore nei bit, nessuna perdita di pacchetti, ricezione nell'ordine di invio. In questo scenario il destinatario non dovrà mai chiedere al mittente di rallentare, e il protocollo si riduce all'essenziale: il mittente riceve i dati dal livello superiore e li immette nel canale, il destinatario li legge dal canale. È il punto di partenza, privo di interesse pratico ma utile come termine di paragone.

17.2 RDT 2.0: i bit possono corrompersi

Rimuoviamo la prima ipotesi: i pacchetti arrivano ancora tutti e in ordine, ma **i bit possono essere corrotti**. Per gestire questa eventualità servono quattro elementi. Un sistema di **rilevamento degli errori**, cioè il **checksum**. Gli **acknowledgment (ACK)**, con cui il ricevitore comunica esplicitamente che il pacchetto è stato ricevuto correttamente. I **negative acknowledgment (NAK)**, con cui comunica invece che il pacchetto presentava errori. E infine la **ritrasmissione** del pacchetto da parte del mittente alla ricezione di un NAK.

Il protocollo che ne risulta è di tipo **stop and wait**: si invia un pacchetto alla volta e si attende la risposta prima di procedere. Mittente e destinatario devono quindi sincronizzarsi a ogni scambio.

Resta però un problema serio, ed è quello che rende necessario il passo successivo: **anche i pacchetti ACK e NAK possono corrompersi**. Se il mittente riceve una conferma illeggibile, non ha alcun modo di sapere se il destinatario abbia ricevuto correttamente l'ultimo blocco.

17.3 RDT 2.1: i numeri di sequenza

La correzione consiste in due mosse. La prima: il mittente **ritrasmette il pacchetto corrente** se l'ACK o il NAK ricevuto è corrotto. Ma questo introduce immediatamente un nuovo rischio, cioè che il destinatario riceva un **duplicato** senza accorgersene. Da qui la seconda mossa: il mittente aggiunge a ogni pacchetto un **numero di sequenza**, che nel caso più semplice alterna soltanto fra **0 e 1**; il destinatario, controllando quel numero, riconosce e **scarta i duplicati**.

Cambia di conseguenza anche ciò che le due parti devono ricordare. Il mittente deve sapere se il numero di sequenza atteso sia 0 oppure 1, e deve controllare l'integrità degli ACK e dei NAK ricevuti. Il destinatario deve

sapere se attende 0 oppure 1 e verificare se il pacchetto ricevuto sia un duplicato. Resta un'asimmetria irriducibile, che vale la pena notare: il destinatario **non può sapere** se il proprio ultimo ACK o NAK sia arrivato correttamente al mittente.

17.4 RDT 2.2: via i NAK

RDT 2.2 offre le stesse funzionalità di RDT 2.1 ma **utilizza soltanto ACK**, eliminando i NAK. Il meccanismo è elegante: il destinatario invia un ACK per l'ultimo pacchetto ricevuto correttamente, **includendo esplicitamente il numero di sequenza** di quel pacchetto, e si ottengono così ACK della forma $(ACK, 0)$ e $(ACK, 1)$. Se il mittente riceve un **ACK duplicato**, cioè la conferma ripetuta di un pacchetto già confermato, lo interpreta **come se fosse un NAK** e ritrasmette.

Vale la pena sottolineare che **è questo l'approccio adottato da TCP**: il NAK non esiste in TCP, e il ruolo di segnalazione negativa è svolto dagli ACK duplicati: meccanismo che ritroveremo alla base del *Fast Retransmit*.

17.5 RDT 3.0: i pacchetti possono perdersi

Rimuoviamo l'ultima ipotesi: il canale può **smarrire** pacchetti e ACK. In entrambi i casi il sintomo per il mittente è lo stesso (non riceve alcuna risposta) e la soluzione è la **temporizzazione**, cioè l'introduzione di un **timer**.

Il mittente attende un intervallo di tempo «ragionevole» e ritrasmette se entro quel termine non è arrivato alcun ACK. Ma quanto deve essere lungo l'intervallo? Deve valere almeno il minimo ritardo di andata e ritorno più il tempo di elaborazione da parte del destinatario. Attendere invece il **ritardo del caso peggiore non conviene**, perché si tradurrebbe in un'attesa lunghissima a fronte di perdite rare: si stima quindi un tempo oltre il quale la perdita è **probabile ma non certa**.

La conseguenza di questa scelta è accettata consapevolmente: se non vi è stata alcuna perdita ma semplicemente un ritardo particolarmente lungo, la ritrasmissione produrrà un **duplicato**. Il problema però era già stato risolto: sono i **numeri di sequenza** a farsene carico. È un buon esempio di come, nella progettazione dei protocolli, un meccanismo introdotto per un motivo finisca per servirne anche altri.

17.6 Alcune definizioni da tenere distinte

Prima di procedere conviene fissare quattro grandezze che vengono spesso confuse fra loro.

GRANDEZZA	DEFINIZIONE
Larghezza di banda (<i>bandwidth</i>)	La quantità massima di dati che può essere trasmessa attraverso il canale in un dato intervallo di tempo. Si misura in bps e multipli. È una caratteristica del canale.
Latenza	Il ritardo che un pacchetto subisce nel percorso dalla sorgente alla destinazione. Si scompone in latenza di trasmissione , di propagazione (dipendente dalla distanza fisica), di elaborazione (il tempo che un nodo impiega a processare il pacchetto) e di accodamento (l'attesa in coda prima di essere trasmesso).
Throughput	La quantità effettiva di dati trasmessa in un periodo di tempo: la velocità realmente percepita dall'utente. È solitamente inferiore alla larghezza di banda , proprio a causa dei ritardi e delle perdite.
Overhead	La quota di traffico dedicata a dati non informativi ma necessari al funzionamento del protocollo: indirizzi, intestazioni, informazioni di controllo. È banda spesa per gestire la comunicazione anziché per trasportare contenuto.

17.7 Stop and wait e il passaggio alla pipeline

RDT 3.0 è **funzionalmente corretto**, ma ha un problema grave di **prestazioni**, perché è un protocollo **stop and wait**: viene inviato un pacchetto per volta e prima di inviare il successivo il mittente attende l'ACK, mantenendo in buffer un solo pacchetto in attesa di conferma. Per tutto il tempo dell'attesa il canale resta inutilizzato. È il caso emblematico in cui **il protocollo limita il rendimento dell'hardware sottostante**: la banda fisica c'è, ma il protocollo non la sfrutta.

La soluzione è il **pipelining**: consentire al mittente di inviare **più segmenti senza attendere i rispettivi ACK**. Invece di spedire un frame per volta, si stabilisce una **finestra (window) di n frame** dei quali si potranno attendere gli ACK contemporaneamente. Questo riduce enormemente l'influenza dell'RTT sul throughput e abbatta l'overhead delle singole trasmissioni; il vantaggio è tanto maggiore quanto più la rete è ad **alta latenza**. Il concetto è analogo a quello della pipeline della CPU: mentre si completa un'operazione, si preparano già le successive.

In questo schema i frame si trovano in **quattro condizioni** distinte: quelli già trasmessi e confermati, quelli trasmessi ma non ancora confermati, quelli pronti per essere inviati e quelli non ancora utilizzabili perché fuori dalla finestra. Esistono due modi di realizzare il pipelining, il **Go Back N** e il **Selective Repeat**.

17.8 Go Back N

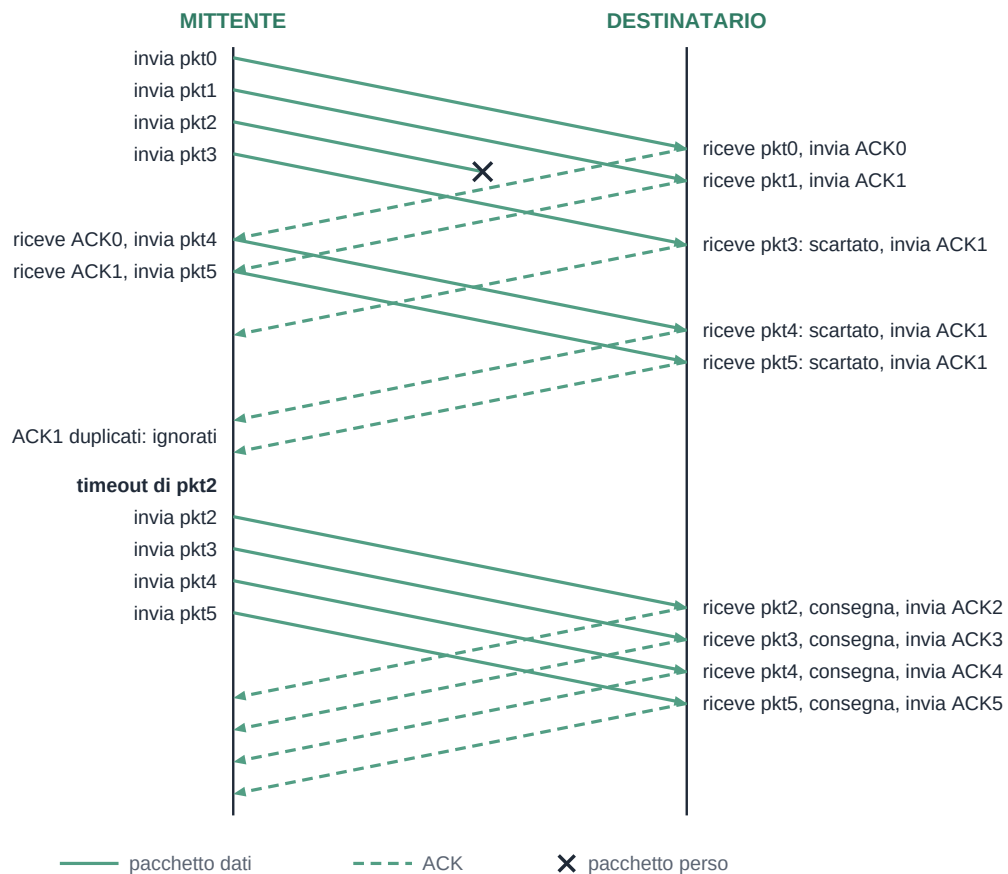
Nel Go Back N il mittente dispone di una **finestra di N pacchetti consecutivi** che può inviare senza attendere gli ACK, e a ogni pacchetto viene aggiunto nell'header un **numero di sequenza di k bit**.

Il meccanismo centrale è l'**ACK cumulativo**: l'ACK del pacchetto con numero di sequenza n indica che **tutti i** pacchetti con numero di sequenza fino a n compreso sono stati ricevuti correttamente. Alla ricezione di un ACK la finestra **scorre in avanti** per ripartire da $n+1$. Non serve quindi una conferma per ogni singolo pacchetto: una sola conferma copre tutto ciò che la precede.

Per gestire perdite e ritardi si utilizza un **unico timer**, associato al **pacchetto in transito più vecchio** che non ha ancora ricevuto ACK; se non vi sono pacchetti in sospeso il timer viene fermato. Alla sua scadenza si verifica il comportamento che dà il nome al protocollo: **Timeout(n) comporta la ritrasmissione del pacchetto n e di tutti i pacchetti con numero di sequenza maggiore di n** . Se l'ACK era soltanto in ritardo, il risultato sono moltissime ritrasmissioni inutili.

Il destinatario è invece volutamente **semplice**. Deve conoscere un solo dato, il numero di sequenza del prossimo pacchetto atteso nell'ordine (rcv_base), e invia sempre l'ACK relativo al numero di sequenza in ordine più alto ricevuto finora, eventualmente ripetendo **ACK duplicati**. Quando riceve un pacchetto **fuori ordine** (attende n ma arriva $n+1$) la scelta più semplice e più conveniente è **scartarlo**, anche se è arrivato integro. In alternativa lo si può memorizzare in buffer e consegnarlo dopo aver ricevuto n , ma questa seconda strada è propria del Selective Repeat.

Da qui il difetto principale del protocollo: **un solo errore può provocare un elevato numero di ritrasmissioni inutili**. Se il pacchetto 2 si perde mentre 3 e 4 arrivano correttamente, il destinatario scarta 3 e 4 e alla scadenza del timer il mittente ritrasmette 2, 3 e 4: due dei quali erano già arrivati integri.



Go Back N: alla perdita di un pacchetto segue la ritrasmissione di tutti i successivi

17.9 Selective Repeat

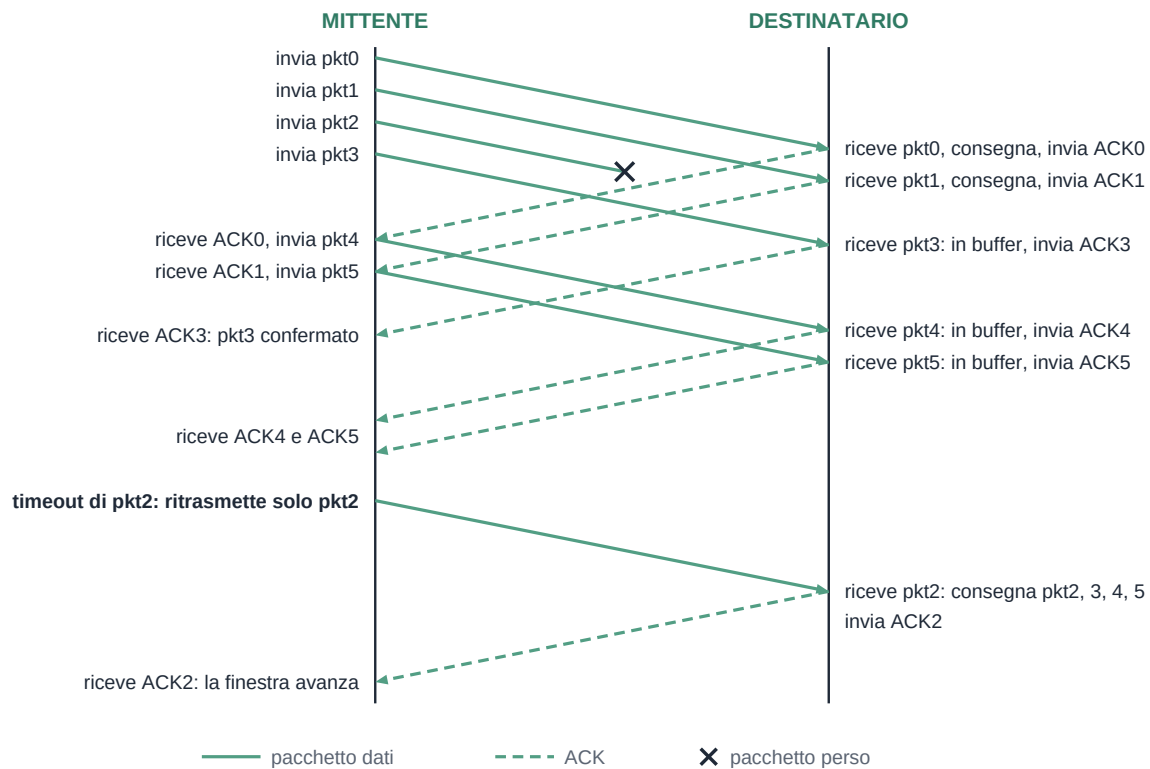
Il Selective Repeat nasce proprio per eliminare quelle ritrasmissioni inutili. Il mittente dispone anche qui di una finestra di N pacchetti consecutivi, ma alla ricezione dell'ACK del pacchetto n lo **marca individualmente** come ricevuto: gli ACK **non sono cumulativi** ma specifici. Di conseguenza il mittente **ritrasmette individualmente solo i pacchetti non confermati**. Il primo frame della finestra è sempre il primo della sequenza non ancora confermato.

Cambia coerentemente anche la gestione dei timer: il Selective Repeat mantiene **un timer per ogni pacchetto non confermato**, e alla scadenza di **Timeout(n)** reinvia **soltanto il pacchetto n** , riavviandone il timer.

Il destinatario è più elaborato. Riconosce **individualmente** tutti i pacchetti ricevuti correttamente, inviando ACK specifici sia per quelli in ordine sia per quelli fuori sequenza, e confermando anche pacchetti con numero di sequenza minore di quello atteso, nel caso in cui un ACK precedente sia andato perso. Soprattutto, i **pacchetti fuori sequenza vengono memorizzati nel buffer** finché non arrivano quelli mancanti, per essere poi consegnati **in ordine** al livello superiore.

Il vantaggio è evidente: **meno ritrasmissioni inutili e maggiore efficienza sui canali con alta perdita**.

Riprendendo l'esempio precedente, se si perde il solo pacchetto 2 mentre 3 e 4 arrivano, questi ultimi vengono bufferizzati e confermati, e il mittente ritrasmette **soltanto il 2**.



Selective Repeat: viene ritrasmesso soltanto il pacchetto effettivamente mancante

IL VINCOLO SUI NUMERI DI SEQUENZA

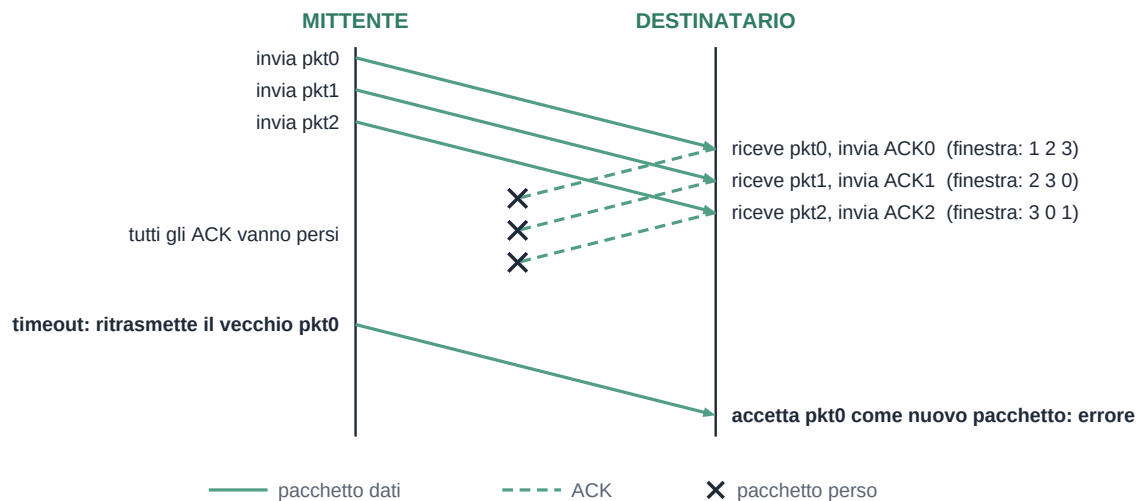
Il Selective Repeat presenta però due criticità. La prima è la **mancanza di sincronizzazione fra la finestra del mittente e quella del destinatario**, che crea problemi quando l'intervallo dei numeri di sequenza è **finito**: e finito lo è sempre, dato che i numeri sono codificati su un numero fisso di bit e quindi si riciclano. La seconda è che il **buffer non ha memoria infinita** e potrebbe saturarsi.

Per non incorrere in questi problemi deve essere rispettata la seguente disequazione:

$$k_{seq} \geq 2 \cdot \text{Window Size}$$

dove k_{seq} indica il **numero massimo di sequenze disponibili**, cioè quante etichette distinte si possono usare per numerare i pacchetti prima di ricominciare da zero. Il significato è quindi: *lo spazio dei numeri di sequenza deve essere grande almeno il doppio della finestra*.

La ragione si comprende con un esempio. Supponiamo di avere 2 bit per i numeri di sequenza, dunque quattro etichette disponibili (0, 1, 2, 3), e di usare erroneamente una finestra di dimensione 3, violando il vincolo. Il mittente invia i pacchetti 0, 1, 2; il destinatario li riceve tutti e invia i relativi ACK, dopodiché attende il pacchetto successivo, che (essendosi i numeri riciclati) sarà nuovamente etichettato 0. Se però **tutti gli ACK si perdono**, alla scadenza del timer il mittente ritrasmette il vecchio pacchetto 0. Il destinatario riceve un pacchetto etichettato 0 e **non ha alcun modo di stabilire** se si tratti della ritrasmissione del vecchio o di un pacchetto nuovo: le due possibilità sono indistinguibili, e il flusso di dati può corrompersi.



Finestra di 3 pacchetti con 4 numeri di sequenza: dopo la perdita degli ACK il vecchio pacchetto 0 viene scambiato per uno nuovo

Il fattore 2 garantisce invece che i numeri usati per i pacchetti «vecchi», potenzialmente ritrasmessi, e quelli per i pacchetti «nuovi» **non coincidano mai**. Operativamente il calcolo si fa in due modi: data la finestra, si moltiplica per 2 per ottenere il numero minimo di etichette necessarie (finestra 5 → almeno 10 etichette); dato il numero di etichette, si divide per 2 per ottenere la finestra massima ammessa (8 etichette → finestra massima 4). Se è noto il numero di **bit** dedicati alla numerazione, le etichette si ottengono elevando 2 a quel numero: 3 bit danno $2^3 = 8$ etichette.

DA RICORDARE

Il confronto fra **Go Back N** e **Selective Repeat** è riassunto nella tabella seguente. Il vincolo $k_{seq} \geq 2 \cdot Window\ Size$ va sempre motivato con l'esempio del riciclo dei numeri di sequenza: è la parte che rivela la comprensione del meccanismo.

ASPETTO	GO BACK N	SELECTIVE REPEAT
Tipo di ACK	Cumulativo	Individuale (specifico per pacchetto)
Timer	Uno solo, sul pacchetto più vecchio	Uno per ogni pacchetto non confermato
Pacchetti fuori ordine	Scartati	Bufferizzati in attesa dei mancanti
Ritrasmissione al timeout	Il pacchetto n e tutti i successivi	Soltanto il pacchetto n
Complessità del ricevitore	Bassa	Elevata
Efficienza con molte perdite	Bassa	Alta
Vincolo sui numeri di sequenza	Finestra $\leq 2^k - 1$	$k_{seq} \geq 2 \cdot Window\ Size$

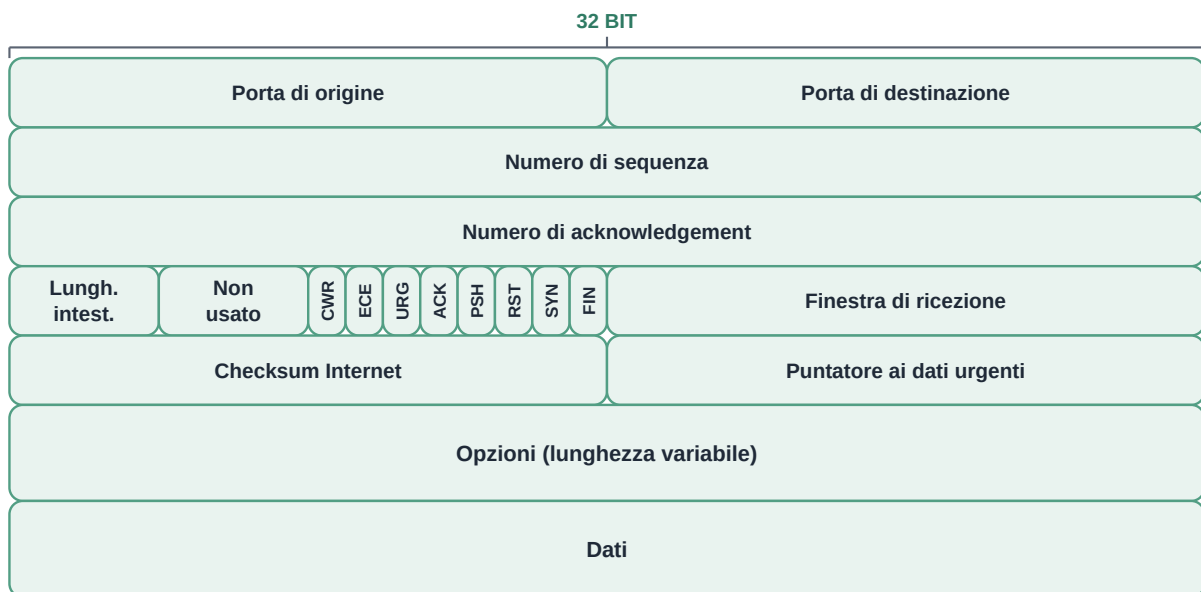
TCP (Transmission Control Protocol)

TCP è un protocollo a livello di trasporto **affidabile** e **connection oriented**: prima di scambiare dati i processi devono effettuare l'handshake. Offre un servizio **full duplex**, cioè i due host inviano e ricevono dati sulla stessa connessione, ed è **point to point**, fra un singolo mittente e un singolo destinatario: di conseguenza **non è possibile il multicast**.

Il ciclo di vita di una connessione TCP si articola così: si instaura la connessione con l'**handshake a tre vie**; i due processi applicativi possono a quel punto scambiarsi dati; TCP dirige i dati al **buffer di invio** della connessione, ne preleva blocchi e li passa al livello successivo; i segmenti vengono passati al Network Layer, dove sono incapsulati in **datagrammi IP**; quando il destinatario riceve un segmento, i dati sono salvati nel **buffer di ricezione**, dal quale l'applicazione leggerà il flusso.

18.1 Il segmento TCP

L'header di un segmento TCP occupa **dai 20 ai 60 byte**, essendo variabile per via del campo opzioni.



Struttura dell'intestazione di un segmento TCP

I campi principali sono i **numeri di porta** di origine e destinazione, a 16 bit ciascuno, usati per il multiplexing; il **numero di sequenza** e il **numero di ACK**, che implementano il trasferimento affidabile; la **lunghezza dell'intestazione**, su 4 bit, necessaria proprio perché il campo opzioni è di lunghezza variabile; uno spazio libero di 4 bit riservato a sviluppi futuri; la **finestra di ricezione**, usata per il controllo di flusso; il **checksum**; il **puntatore a dati urgenti**; e infine il campo **opzioni**, facoltativo e di lunghezza variabile, impiegato per esempio quando mittente e destinatario negoziano l'MSS.

I **flag** meritano un discorso a parte, e conviene raggrupparli per importanza.

Quelli **essenziali** sono quattro. Il flag **ACK** indica che il valore trasportato nel campo *Numero di ACK* è valido. Il flag **SYN**, posto a 1, segnala che il mittente vuole aprire una connessione e specifica nel campo *Sequence number* il valore dell'**ISN** (*Initial Sequence Number*), sincronizzando così i numeri di sequenza dei due host. Il flag **FIN**, posto a 1, indica che il mittente vuole chiudere la connessione, e l'altro lato ne conferma la ricezione con un ACK.

Il flag **RST**, posto a 1, segnala che la connessione **non è valida** ed è usato in caso di errore grave, talvolta insieme al flag **ACK** per chiudere una connessione.

Due flag riguardano la **congestione** e li ritroveremo trattando il controllo *network assisted*: **CWR** (*Congestion Window Reduced*), posto a 1 quando l'host sorgente ha ricevuto un segmento con il flag **ECE** attivo, ed **ECE** (*ECN Echo*), che durante il *three way handshake* indica il supporto di ECN e, a connessione stabilita, segnala al mittente che un router ha marcato un pacchetto come congestionato.

Due flag infine **non vengono usati** nella pratica, ma è bene conoscerli: **URG**, che segnalerebbe la presenza di dati urgenti indicati dal puntatore relativo, e **PSH**, che chiederebbe di non bufferizzare i dati in arrivo ma passarli subito ai livelli superiori.

18.2 Numeri di sequenza e numeri di ACK

Qui sta la differenza più significativa rispetto ai protocolli teorici visti finora, e va colta con precisione: in TCP i numeri di sequenza **si applicano al flusso di byte trasmessi, non alla serie di segmenti**. TCP non conta «pacchetto 1, pacchetto 2», ma conta i **byte**.

Il numero di sequenza di un segmento rappresenta dunque il **numero, nel flusso complessivo di byte, del primo byte contenuto in quel segmento**. Mittente e destinatario scelgono a caso il proprio numero di sequenza iniziale, che viene poi incrementato di n ogni volta che n byte vengono trasmessi: inviato un segmento di 1400 byte, il numero di sequenza successivo sarà il precedente + **1400**.

Il **numero di ACK** rappresenta invece il **numero di sequenza del byte successivo atteso** dall'altro lato. Se l'Host A ha ricevuto correttamente tutti i byte da 0 a 535, attende il byte 536 e porrà quindi **536** nel campo ACK. Non dice «ho ricevuto fino a 535» ma «dammi il 536»: è la stessa informazione, espressa guardando avanti.

Come nel Go Back N, gli **ACK sono cumulativi**: TCP conferma i byte **solo fino al primo byte mancante** nel flusso. L'esempio è illuminante: se A riceve i byte da 0 a 535 e anche quelli da 900 a 1000, lasciando quindi un buco in mezzo, il suo ACK conterrà comunque **536**, perché è quello il primo byte che manca. I byte da 900 a 1000, pur ricevuti, non possono essere confermati finché il buco non viene colmato.

Sui **segmenti fuori ordine** l'RFC non impone alcuna regola, lasciando la decisione a chi implementa TCP. Si può scegliere di **scartarli**, ottenendo una progettazione più semplice, oppure di **mantenerli in attesa dei mancanti**, soluzione più efficiente in termini di banda e in effetti **la più utilizzata**. Nella pratica, quindi, TCP somiglia più al Selective Repeat che al Go Back N.

18.3 MSS e MTU

L'**MSS** (*Maximum Segment Size*) è la **dimensione massima del payload**, cioè dei soli dati che possono essere inseriti in un segmento TCP. Attenzione a non confonderlo con la dimensione del segmento: l'MSS riguarda esclusivamente i dati, senza intestazioni.

Il valore dell'MSS viene calcolato a partire dall'**MTU** (*Maximum Transmission Unit*), che è la lunghezza del frame più grande che può essere inviato al livello di collegamento. La relazione è:

$$\text{MTU} = \text{MSS} + \text{header TCP} + \text{header IP}$$

Il segmento deve infatti poter stare all'interno di un singolo frame, e gli header TCP e IP insieme occupano tipicamente **40 byte** (20 + 20). Poiché i protocolli Ethernet hanno un MTU di **1500 byte**, un valore tipico di MSS è **1460 byte**. TCP richiede inoltre che tutte le reti sappiano trasferire pacchetti IP di almeno **576 byte**, il che corrisponde a un MSS minimo garantito di **536 byte** (576 – 20 – 20).

In sintesi, la differenza fra le due grandezze sta tutta nelle intestazioni: l'MTU **include** gli header dei livelli inferiori, l'MSS considera **solo** i dati trasportati da TCP, ed è quindi sempre più piccolo.

18.4 RTT, stima del timeout e RTO

DA RICORDARE

Il funzionamento del timer di ritrasmissione si riassume in cinque passi: si misura un campione di **RTT**; si aggiorna l'**EstimatedRTT**; si calcola la **DevRTT**; si ricava l'**RTO**; se il timer scade, l'RTO viene **raddoppiato**.

L'**RTT** (*Round Trip Time*) è il tempo trascorso dall'invio di un segmento alla ricezione del relativo ACK, comprensivo dei ritardi di propagazione, di accodamento nei router e di elaborazione. Il problema pratico è: come impostare l'intervallo di timeout? Deve certamente essere **più lungo dell'RTT**, ma l'RTT **varia** continuamente in funzione della congestione. Un valore corretto migliora le prestazioni ed evita di aggravare la congestione, quindi va stimato dinamicamente.

Il primo ingrediente è il **SampleRTT**, cioè il tempo misurato dalla trasmissione di un segmento fino alla ricezione del suo ACK. Un singolo campione però varia molto da segmento a segmento, e serve una stima più «fluida», ottenuta come media di diverse misurazioni recenti anziché del solo campione corrente. Nasce così l'**EstimatedRTT**, una **media ponderata** dei valori di SampleRTT che dà maggiore importanza ai campioni recenti.

Il tipo di media impiegato è una **EWMA** (*Exponential Weighted Moving Average*), in cui il peso dei campioni passati **diminuisce esponenzialmente** man mano che gli aggiornamenti procedono:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT} \quad [\alpha = 0,125]$$

La EWMA è preferita alla media aritmetica per due ragioni concrete: è **computazionalmente semplice**, perché non richiede di conservare lo storico dei campioni, e **si adatta rapidamente** ai cambiamenti delle condizioni di rete.

Oltre alla stima dell'RTT serve però conoscerne anche la **variabilità**, cioè quanto i campioni oscillano attorno alla media. È il ruolo della **DevRTT**, definita come stima di quanto SampleRTT si discosta da EstimatedRTT: in presenza di forti fluttuazioni DevRTT sarà grande, mentre con un RTT stabile sarà piccola.

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}| \quad [\beta = 0,25]$$

I due valori vengono infine combinati nell'**RTO** (*Retransmission Timeout*). La scelta del margine non è arbitraria: un RTO **minore** di EstimatedRTT produrrebbe timeout prematuri e ritrasmissioni inutili; un RTO **molto maggiore** renderebbe la reazione alla perdita troppo lenta, impedendo a TCP di ritrasmettere rapidamente. Il compromesso è dato dalla formula:

$$\text{RTO} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

Il valore iniziale consigliato per l'RTO è **1 secondo** (RFC 6298). Dopo la scadenza del timer il valore di RTO viene **raddoppiato**, operazione realizzata in modo semplicissimo con uno shift a sinistra, per evitare ulteriori timeout prematuri.

18.5 Trasferimento affidabile e scenari di ritrasmissione

Poiché IP non garantisce la consegna, né l'ordine, né l'integrità dei datagrammi, e i segmenti di trasporto vi sono incapsulati, anche questi ultimi ne risentono. TCP costruisce quindi un servizio di trasferimento affidabile **al di sopra** del servizio inaffidabile e *best effort* di IP.

Un aspetto implementativo importante: **TCP utilizza un solo timer alla volta**, anche in presenza di più segmenti in volo. Ricevuto l'ACK del segmento n , viene avviato un nuovo timer per il segmento $n+1$ ancora non confermato. In questo TCP somiglia più al Go Back N che al Selective Repeat, pur adottando di quest'ultimo la bufferizzazione dei fuori ordine.

Tre scenari classici illustrano il funzionamento del meccanismo, e conviene seguirli passo per passo perché mostrano come i numeri di sequenza risolvano da soli situazioni molto diverse.

SCENARIO 1: ACK SMARRITO

L'Host A spedisce un segmento con numero di sequenza **92** e 8 byte di dati, e attende quindi da B un ACK con valore **100** (cioè $92 + 8$). L'ACK inviato da B **si smarrisce**, benché B abbia ricevuto correttamente il pacchetto. Scatta il timeout e A **ritrasmette** lo stesso segmento. B lo riceve, riconosce dal numero di sequenza che si tratta di una **ritrasmissione**, scarta i byte duplicati e ritrasmette l'ACK.

SCENARIO 2: TIMEOUT PREMATURO

A invia due segmenti, il primo con numero di sequenza 92 e 8 byte, il secondo con numero 100 e 20 byte. Entrambi arrivano a B, che invia due ACK separati, 100 e 120. Il **timeout scatta prima** che i due ACK arrivino ad A, il quale ritrasmette il primo segmento e riavvia il timer; il secondo segmento non verrà ritrasmesso, purché il suo ACK arrivi prima della nuova scadenza. A questo punto A riceve i due ACK, mentre B riceve nuovamente il segmento 92, capisce che è una ritrasmissione, scarta i byte e ritrasmette l'ACK con il **valore più alto**, cioè 120.

SCENARIO 3: L'ACK CUMULATIVO EVITA LA RITRASMISSIONE

A invia gli stessi due segmenti. L'ACK del **primo si perde**, ma quello del secondo arriva **prima del timeout**. Poiché gli ACK sono cumulativi, dall'ACK 120 A deduce che B ha ricevuto tutto fino al byte 119, e quindi **non ritrasmette nulla**. È l'esempio più chiaro del vantaggio degli ACK cumulativi: la perdita di una conferma viene automaticamente sanata dalla conferma successiva.

IL RADDOPPIO DELL'INTERVALLO DI TIMEOUT

In tutti i casi di timeout TCP ritrasmette il segmento con il **più basso numero di sequenza non ancora confermato**. Ogni volta che ciò accade, però, TCP imposta il successivo intervallo di timeout al **doppio del valore precedente**, anziché ricalcolarlo dagli ultimi EstimatedRTT e DevRTT: gli intervalli crescono così **esponenzialmente** a ogni ritrasmissione. Quando invece il timer viene avviato in seguito a uno degli altri eventi possibili, l'intervallo viene regolarmente ricavato dai valori più recenti.

Vale la pena capire il senso di questa scelta, perché non è un dettaglio: la scadenza del timer è **probabilmente causata dalla congestione**, e nei periodi di congestione, se le sorgenti continuano a trasmettere, la situazione peggiora. Il raddoppio esponenziale offre quindi una **forma limitata di controllo di congestione**, facendo indietreggiare progressivamente il mittente.

18.6 Fast Retransmit

Il problema delle ritrasmissioni gestite dal solo timer è che **il periodo di timeout può rivelarsi lungo**, e nel frattempo il canale resta sottoutilizzato. Il mittente può però rilevare la perdita **ben prima** che il timer scada, sfruttando un'informazione che già possiede: gli **ACK duplicati** relativi a un segmento il cui ACK era già stato ricevuto.

Il ragionamento è il seguente: se il destinatario continua a confermare sempre lo stesso byte, significa che sta ricevendo segmenti successivi ma **non riesce a colmare un buco**. Se il mittente riceve **tre ACK duplicati**, considera questo evento come indice del fatto che il segmento successivo sia andato perduto e, per evitare lunghe attese, ritrasmette il segmento mancante **prima che scada il timer**. È questa la funzione di **Fast Retransmit**, che ritroveremo come componente di TCP Tahoe e TCP Reno.

Controllo del flusso TCP

Il capitolo descrive il meccanismo con cui TCP impedisce al mittente di saturare il buffer di ricezione del destinatario, e i problemi che nascono quando la finestra disponibile si riduce a pochi byte.

DA RICORDARE

La **differenza fra controllo di flusso e controllo della congestione** si riassume così: il **controllo di flusso protegge il ricevitore**, impedendo che venga sommerso da dati più veloci di quanto riesca a elaborarne; il **controllo di congestione protegge la rete**, impedendo che router e collegamenti vengano saturati. Stesso effetto (rallentare il mittente) ma per due ragioni completamente diverse.

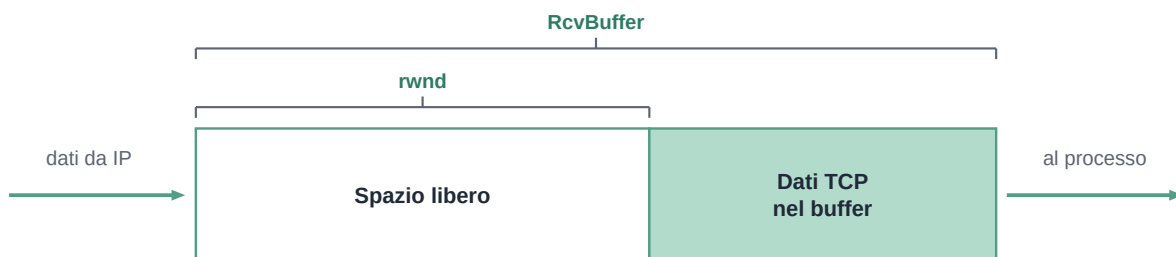
UDP **non offre** alcun controllo di flusso: i segmenti vengono messi in coda in un buffer e, se il processo applicativo non li legge abbastanza rapidamente, si verifica un **overflow** e alcuni segmenti vanno perduti. TCP invece utilizza il controllo di flusso per evitare il sovraccarico del buffer di ricezione, garantendo che il mittente non invii dati più velocemente di quanto l'applicazione destinataria riesca a elaborarne.

Lo strumento è la **finestra di ricezione** (*receive window*), una variabile che indica lo spazio disponibile nel buffer del destinatario. Per definirla formalmente servono tre grandezze: **RcvBuffer**, la dimensione complessiva del buffer; **LastByteRead**, il numero dell'ultimo byte che il processo applicativo ha letto dal buffer; e **LastByteRcvd**, il numero dell'ultimo byte arrivato nel buffer di ricezione. Lo spazio libero è allora:

$$rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)$$

Il destinatario comunica al mittente questo valore scrivendolo nel campo **finestra di ricezione** dell'header TCP. Perché il buffer non venga sovraccaricato deve valere la condizione:

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$



Il buffer di ricezione e lo spazio libero comunicato tramite rwnd

Dal lato del mittente le variabili rilevanti sono due: **LastByteSent**, l'ultimo byte inviato, e **LastByteAcked**, l'ultimo byte per cui ha ricevuto conferma. La loro differenza rappresenta la quantità di dati **inviati ma non ancora confermati**, cioè i dati «in volo». Il mittente si assicura che tale quantità non superi la finestra dichiarata dal destinatario:

$$LastByteSent - LastByteAcked \leq rwnd$$

19.1 Il problema della finestra nulla

Il meccanismo, così com'è, contiene una trappola. Supponiamo che il buffer del destinatario si riempia completamente, cosicché *rwnd* vale 0: il destinatario avvisa il mittente che non c'è più spazio, ma successivamente **non ha più nulla da inviare**. Il mittente, di conseguenza, **non verrà mai informato** di quando lo spazio si sarà liberato, e resterà bloccato indefinitamente in attesa di una notizia che non arriverà.

La soluzione adottata è pragmatica: il mittente **continua a inviare segmenti contenenti un singolo byte** di dati. Il destinatario è così costretto a rispondere, e nel momento in cui il suo buffer si sarà svuotato risponderà con un ACK contenente un valore di *rwnd* non nullo, sbloccando la situazione.

19.2 Silly Window Syndrome, algoritmo di Clark e algoritmo di Nagle

Esiste un difetto noto del protocollo TCP, chiamato **Silly Window Syndrome**, che si manifesta in due forme speculari, entrambe caratterizzate da segmenti minuscoli con overhead sproporzionato.

Nella prima forma il problema è **lato ricevitore**: questi svuota lentamente il buffer di ricezione e invia ACK con una *rwnd* molto piccola; il trasmettitore, adeguandosi, invia segmenti piccolissimi in cui l'intestazione pesa più dei dati. La soluzione è l'**algoritmo di Clark**, secondo cui il ricevitore dichiara una **finestra nulla** finché il buffer non si è svuotato almeno per metà, oppure per una porzione pari a un MSS: meglio far attendere il mittente che indurlo a inviare frammenti inefficienti.

Nella seconda forma il problema è **lato mittente**: l'applicazione genera dati lentamente e TCP finisce per inviare segmenti molto piccoli. La soluzione è l'**algoritmo di Nagle**.

L'ALGORITMO DI NAGLE

L'algoritmo di Nagle riduce l'overhead generato dall'invio di piccoli pacchetti, ed è particolarmente utile per applicazioni che producono molte trasmissioni di dimensioni ridotte: il caso tipico è **Telnet**, che invia un byte alla volta. L'obiettivo è **aggregare i dati** nel buffer di trasmissione per inserirli in pacchetti più grandi, cosa tanto più conveniente quanto più l'RTT della rete è alto.

Il funzionamento è il seguente. Se *rwnd* è maggiore o uguale a un MSS e la quantità di dati da spedire è a sua volta maggiore o uguale a un MSS, si invia subito: si sta infatti sfruttando il pacchetto al massimo. In caso contrario ci si mette in **attesa accumulando dati**: si invia il primo byte e si bufferizza il resto finché non giunge l'ACK del segmento precedente oppure finché il buffer in uscita non contiene almeno MSS byte; a quel punto si inviano **in un unico segmento** tutti i caratteri accumulati.

```
if dati_disponibili > 0 then
  if finestra ≥ MSS and dati_disponibili ≥ MSS then
    invia un segmento di dimensione MSS
  else
    if in_attesa_di_ACK == true then
      accoda i dati      /* fino all'arrivo di un ACK */
    else
      invia i dati
    end if
  end if
end if
```

Pseudocodice dell'algoritmo di Nagle

Apertura e chiusura della connessione TCP

Il capitolo descrive come TCP instaura una connessione con l'handshake a tre vie e come la chiude, spiegando perché un handshake a due vie non basterebbe e perché la chiusura è più complessa dell'apertura.

DA RICORDARE

I due problemi teorici dell'apertura e della chiusura di una connessione TCP sono: **perché non basterebbe un handshake a due vie** e **perché la chiusura è più complicata dell'apertura**.

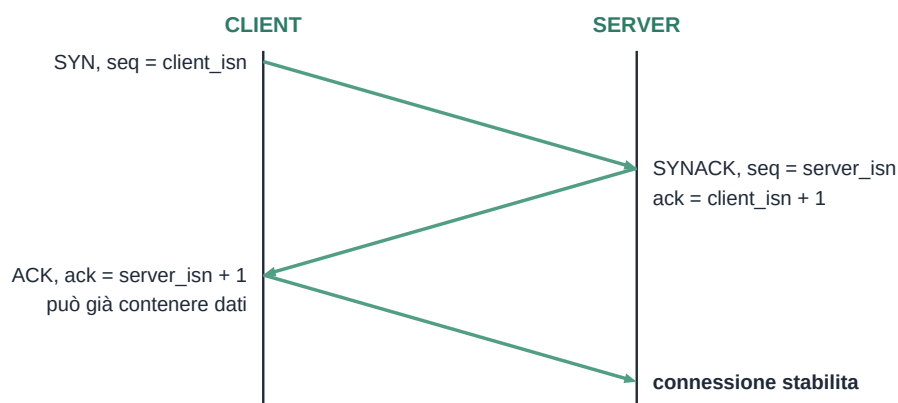
20.1 Perché non basta un handshake a due vie

Prima di arrivare al *three way handshake* era stato considerato un approccio più semplice: A richiede la connessione a B, B risponde con un ACK e apre la connessione, A riceve l'ACK e partecipa alla connessione. Sembra sufficiente, ma non lo è.

Consideriamo infatti che cosa accade se **l'ACK si perde**, oppure se il client A va in crash subito dopo. B, avendo inviato il proprio ACK, entra nello stato **ESTABLISHED** e resta in attesa; A invece non entra mai in quello stato. Si ottiene così una **connessione half open**, anomala e inutilizzabile, con risorse allocate su B per una comunicazione che non avverrà mai.

Il problema di fondo è che, se uno dei messaggi usati per aprire la connessione si perde o arriva molto in ritardo, i due host **non sono più sincronizzati**. Il *three way handshake* serve proprio a garantire che entrambi gli host siano ancora attivi e concordi, e gestisce le situazioni anomale tramite il flag **RST**, che può essere posto a 1 per segnalare una connessione non valida.

20.2 Il three way handshake



L'handshake a tre vie per l'apertura di una connessione TCP

La connessione viene stabilita in tre passi. **Primo:** il TCP lato client invia un segmento speciale detto **SYN**, che non contiene dati applicativi ma ha il bit SYN posto a 1; il client sceglie a caso un numero di sequenza iniziale (*client_isn*) e lo colloca nel campo numero di sequenza. **Secondo:** quando arriva il datagramma contenente il SYN, il server ne estrae il segmento, **alloca buffer e variabili** per la connessione e invia un segmento di

connessione approvata, detto **SYNACK**; anche questo è privo di dati applicativi, ha il bit SYN a 1, pone nel campo ACK il valore $client_isn + 1$ e sceglie il **proprio** numero di sequenza iniziale ($server_isn$). **Terzo**: alla ricezione del SYNACK anche il client alloca buffer e variabili e risponde con un ulteriore segmento, in cui il campo ACK assume il valore $server_isn + 1$ e il bit SYN è posto **a zero**, dato che la connessione è ormai stabilita; questo terzo segmento **può già contenere dati**.

La logica dei tre passi si comprende contando ciò che deve accadere: poiché TCP è full duplex, **entrambi** gli host devono annunciare il proprio numero di sequenza iniziale ed **entrambi** devono ricevere conferma che l'altro lo ha recepito. Servirebbero quattro messaggi, ma il SYN del server e l'ACK verso il client vengono **fusi in un unico segmento**, il SYNACK, riducendoli a tre.

20.3 La chiusura della connessione

La chiusura è **più complicata dell'apertura**, e la ragione è di principio: la chiusura è **unidirezionale**. Un lato può decidere di smettere di inviare dati mentre l'altro potrebbe voler continuare a trasmettere; per chiudere completamente la connessione **entrambi i lati devono essere d'accordo**, e ciascuno lo comunica con un messaggio speciale chiamato **FIN**.

Supponiamo che sia il client a decidere di chiudere. Il client invia un segmento **FIN**, con il bit FIN posto a 1. Il server, ricevutolo, risponde con un **ACK** e **continua a inviare dati** se necessario. Una volta completato l'invio, il server invia a sua volta un segmento **FIN**. Il client, ricevutolo, risponde con un **ACK** finale. A quel punto le risorse allocate sui due host vengono deallocate. Sia il client sia il server possono avviare la terminazione.

Anche qui esiste però un problema teorico da segnalare: se uno dei messaggi FIN o ACK **si perde**, la chiusura può restare bloccata, generando situazioni di **deadlock** in cui un host attende all'infinito una risposta che non arriverà. La soluzione implementativa consiste nell'avviare un **timer**: alla sua scadenza il messaggio viene ritrasmesso, e le ritrasmissioni proseguono per un **numero limitato di tentativi**, dopo i quali l'host conclude che la connessione è persa e libera comunque le risorse.

Va aggiunto un caso particolare: se un host riceve un segmento TCP i cui numeri di porta **non corrispondono a nessuna socket attiva**, risponde al mittente con uno speciale segmento di **reset**, avente il bit RST posto a 1.

20.4 I problemi dell'handshake a tre vie

Il *three way handshake* non è privo di costi. Il primo è che **aggiunge ritardi**: prima di poter trasmettere un solo byte utile occorre attendere circa un RTT, ed è precisamente questo il motivo per cui le connessioni non persistenti di HTTP 1.0 risultavano tanto inefficienti e per cui QUIC cerca di ridurre la latenza di instaurazione.

Il secondo è che costituisce un **punto di vulnerabilità**, sfruttato dall'attacco **SYN flood**. Si tratta di un classico attacco di tipo DoS in cui l'aggressore invia un gran numero di segmenti SYN **senza mai completare il terzo passo** dell'handshake: il server, per ciascuno di essi, alloca buffer e variabili e resta in attesa, finché le risorse si esauriscono e le connessioni legittime non possono più essere servite. È interessante notare che la vulnerabilità nasce esattamente dal punto in cui il protocollo si comporta correttamente, cioè dall'allocazione anticipata delle risorse al secondo passo. La contromisura più diffusa sono i **SYN cookie**, con cui il server non alloca alcuna risorsa finché non riceve il terzo segmento dell'handshake.

Cause e costi della congestione

Prima di studiare come TCP controlli la congestione, conviene capire che cosa la congestione **sia** e quali costi comporti. Le grandezze da tenere presenti sono tre: λ_{in} , la frequenza media di invio; λ_{out} , la frequenza di arrivo dei byte al ricevente; R , la capacità del collegamento condiviso. La congestione comincia a crearsi quando λ_{in} **si avvicina o supera** la capacità disponibile.

21.1 Scenario 1: buffer illimitati

Consideriamo due host, A e B, che condividono un router intermedio, ipotizzando **buffer illimitati** nel router e nessuna necessità di ritrasmissione. La capacità dei collegamenti in ingresso e in uscita è R , e poiché gli host sono due, ciascuno può aspirare al più a $R/2$.

Finché λ_{in} resta minore o uguale a $R/2$, il router riesce a inoltrare i pacchetti senza ritardi significativi e si ha semplicemente $\lambda_{out} = \lambda_{in}$. Quando però il tasso di invio si avvicina o supera $R/2$, accadono tre cose: il **throughput resta comunque limitato a $R/2$** ; compaiono **ritardi crescenti**, anche in presenza di buffer illimitati; e il numero medio di pacchetti nel router **cresce senza limite**.

La lezione di questo primo scenario è che la causa del limite superiore del throughput è semplicemente la **condivisione della capacità del collegamento**, e che quando il tasso di arrivo si avvicina alla capacità si registrano **ritardi di accodamento lunghissimi**. Anche con memoria infinita, dunque, la congestione produce un costo.

21.2 Scenario 2: buffer limitati

Rendiamo il modello più realistico assumendo **buffer limitati**. Ora dobbiamo preoccuparci della ritrasmissione dei pacchetti che, giungendo a un buffer già pieno, vengono scartati. Compare quindi una nuova grandezza: λ'_{in} , il traffico effettivamente immesso al livello di trasporto, che è **maggiore o uguale a λ_{in}** proprio a causa delle ritrasmissioni.

Il comportamento dipende da quanto il mittente sa. Nel caso ideale in cui il **mittente conosca lo stato del buffer** e invii pacchetti solo quando c'è spazio libero, non si hanno perdite, λ'_{in} coincide con λ_{in} e si ottiene $\lambda_{out} = \lambda_{in} \leq R/2$: nessun costo aggiuntivo.

Nel caso realistico in cui il mittente **non conosca lo stato del buffer** e ritrasmetta solo i pacchetti effettivamente persi, emerge il **primo costo della congestione**: sono necessarie **ritrasmissioni per compensare i pacchetti scartati** in caso di overflow. Ne consegue che, anche portando λ'_{in} a $R/2$, il throughput utile λ_{out} resta **inferiore a $R/2$** , perché parte della capacità è spesa per ritrasmettere.

Nel caso peggiore, quello del **timeout prematuro** del mittente, la situazione degenera. Le ritrasmissioni sono **inutili**, il mittente non avanza e continua a inviare copie, la capacità trasmissiva del router viene sprecata e il buffer si riempie di duplicati privi di valore. Emerge così il **secondo costo della congestione**: le ritrasmissioni non necessarie costringono il router a sprecare capacità, e il throughput può scendere fino a circa $R/4$.

21.3 Scenario 3: percorsi multihop

Il terzo scenario considera **quattro host**, con timeout, ritrasmissioni e **percorsi multihop**, cioè che attraversano più router. Qui emerge il costo più insidioso.

Se il traffico di un flusso (chiamiamolo «rosso») aumenta, può accadere che tutti i pacchetti di un altro flusso, il «blu», vengano scartati, riducendone il throughput **a zero**. Il **terzo costo della congestione** è dunque il seguente: la diminuzione del throughput risulta ancora più grave se si considera la **quantità di lavoro sprecato** nella rete, perché ogni volta che un pacchetto viene scartato lungo il percorso, **tutta la capacità trasmissiva impiegata per instradarlo fino a quel punto è stata sprecata**. Un pacchetto perso all'ultimo hop ha consumato inutilmente le risorse di tutti i router precedenti.

21.4 I due approcci al controllo di congestione

Riassumendo le cause: il traffico si avvicina o supera la capacità R della rete; il router lavora al massimo ma le applicazioni restano bloccate; il mittente continua a inviare copie inutili, sottraendo spazio ai pacchetti che servirebbero per avanzare; e soprattutto il mittente **non sa distinguere** se i pacchetti scadano per congestione o per semplice ritardo, e quindi continua a inviare duplicati. Per limitare il fenomeno si opera sulla **finestra di trasmissione cwnd**, aumentando e riducendo opportunamente il numero di pacchetti consentiti contemporaneamente in rete senza aver ricevuto un ACK.

Esistono due filosofie. L'**approccio end to end** non prevede **alcun feedback esplicito dalla rete**: la congestione viene **dedotta** dai sistemi periferici osservando perdite e ritardi. È l'approccio classico di TCP, che rallenta riducendo la propria finestra: con l'inconveniente, però, che qualcun altro potrebbe nel frattempo occupare la coda e prendersi lo spazio lasciato libero.

L'**approccio network assisted** prevede invece che siano i **router a fornire un feedback diretto** agli host i cui flussi li attraversano, indicando il livello di congestione o addirittura imponendo esplicitamente una velocità di invio. Vi rientrano **TCP ECN**, che inserisce due flag nei pacchetti già in transito, **ATM ABR** (*Available Bit Rate*), che informa il mittente sulla frequenza trasmissiva sostenibile, e i **protocolli DECbit**, basati su un bit che segnala il traffico su un collegamento.

Controllo di congestione classico in TCP

Il capitolo descrive l'algoritmo con cui TCP regola la finestra di congestione: le fasi di slow start, congestion avoidance e fast recovery, il principio AIMD e le differenze fra le varianti Tahoe e Reno.

DA RICORDARE

Il nucleo del controllo di congestione di TCP sono le tre fasi (*slow start*, *congestion avoidance*, *fast recovery*), il principio **AIMD** e la differenza fra le varianti **Tahoe** e **Reno**.

L'approccio scelto da TCP consiste nell'imporre a ciascun mittente un limite al tasso di invio **in funzione della congestione percepita** sul proprio percorso: se il mittente rileva scarso traffico incrementa il tasso di trasmissione, se percepisce traffico lo riduce. Lo strumento è la **finestra di congestione (cwnd)**, regolata dinamicamente. Il mittente limita la trasmissione secondo la relazione:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min \{ \text{cwnd}, \text{rwnd} \}$$

Questa formula è il punto in cui i due controlli si incontrano: il mittente rispetta **contemporaneamente** il limite imposto dal ricevitore (*rwnd*) e quello imposto dalla rete (*cwnd*), adeguandosi al più stringente dei due. Il tasso di trasmissione risultante è approssimativamente:

$$\text{TCP rate} \approx \text{cwnd} / \text{RTT bps}$$

L'obiettivo ideale è sfruttare al massimo la banda disponibile senza congestionare la rete. L'algoritmo si articola in tre fasi.

22.1 Slow start

All'inizio della connessione la velocità **cresce in modo esponenziale** fino al primo evento di perdita. Si parte con **cwnd = 1 MSS** e la finestra **raddoppia a ogni RTT**, poiché cwnd viene incrementata di 1 MSS per ogni ACK ricevuto. Il nome è in un certo senso fuorviante: l'avvio è lento in valore assoluto, ma la crescita è rapidissima.

Il problema è stabilire quando questa crescita debba terminare, e la risposta è una soglia chiamata **ssthresh** (*slow start threshold*). Finché **cwnd < ssthresh** si procede in modo esponenziale; quando **cwnd ≥ ssthresh** la fase di slow start termina e si entra in *congestion avoidance*. Se invece vengono rilevati **tre ACK duplicati**, segno che la rete sta rallentando, TCP opera una ritrasmissione rapida ed entra in **Fast Recovery** se si tratta di TCP Reno, oppure **riparte da Slow Start** se si tratta di TCP Tahoe.

In caso di **evento di perdita** rilevato per timeout, il comportamento è drastico: si pone **ssthresh = cwnd/2**, cioè a metà del valore che la finestra aveva al momento della rilevazione, si pone **cwnd = 1** e riparte il processo di slow start.

22.2 Congestion avoidance

Si entra in questa fase quando cwnd raggiunge la soglia ssthresh. Qui la crescita diventa **lineare**: cwnd viene incrementata di **1 MSS ogni RTT**. La logica è prudentiale: avvicinandosi al limite stimato della capacità, conviene procedere per piccoli passi anziché raddoppiare. In caso di perdita il comportamento è lo stesso descritto sopra, con ritorno alla fase di slow start.

22.3 Fast recovery

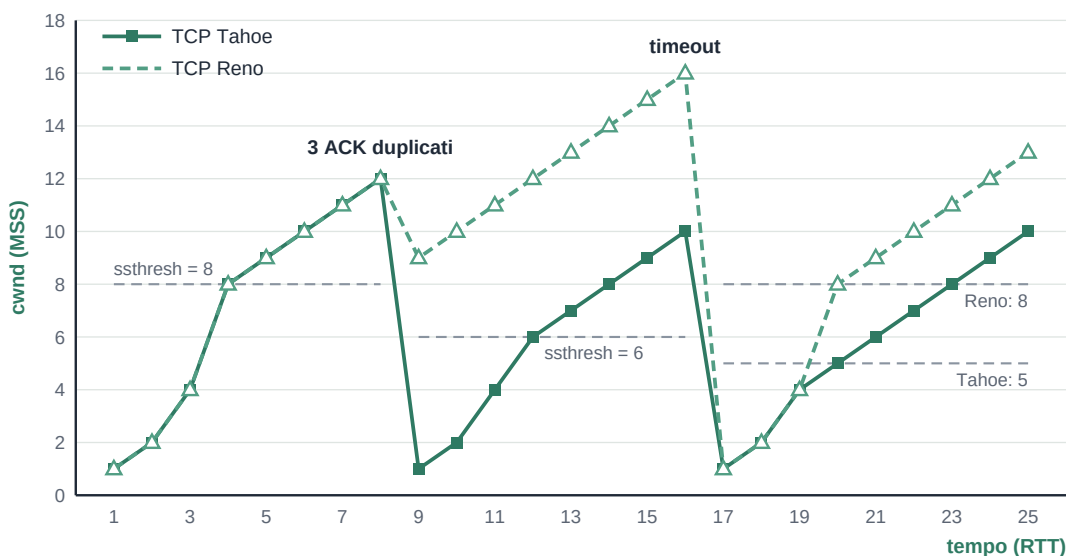
Questa fase è presente **soltanto in TCP Reno**. Durante il *fast recovery* *cwnd* viene incrementata di **1 MSS per ogni ACK duplicato** ricevuto relativo al segmento perso; quando arriva finalmente l'ACK del segmento perduto, TCP entra nello stato di *congestion avoidance*. Se invece si verifica un evento di perdita per timeout, ci si comporta come nei casi precedenti tornando in slow start.

22.4 AIMD

Il controllo di congestione di TCP è indicato come forma di controllo **AIMD**, *Additive Increase, Multiplicative Decrease*. Il principio è che i mittenti possono aumentare la velocità di invio finché non si verifica una perdita, e a quel punto la riducono bruscamente.

L'**Additive Increase** consiste nell'aumentare la velocità di **1 MSS ogni RTT** finché non viene rilevata una perdita. Il **Multiplicative Decrease** consiste nel **dimezzare** la velocità di invio nel caso di TCP Reno, oppure nel ridurla a **1 MSS** quando la perdita viene rilevata tramite timeout o nel caso di TCP Tahoe. L'andamento risultante nel tempo è il caratteristico profilo **a dente di sega**: lunghe salite lineari interrotte da cadute improvvise.

22.5 TCP Tahoe e TCP Reno a confronto



Evoluzione della finestra di congestione: confronto fra TCP Tahoe e TCP Reno

TCP Tahoe implementa *slow start*, *congestion avoidance* e *fast retransmit*. Il suo tratto distintivo è la reazione uniforme e drastica: sia in caso di **tre ACK duplicati** sia in caso di **evento di perdita**, la soglia viene posta a metà della finestra corrente e ***cwnd* riparte dal valore minimo**, cioè da 1 MSS, con una nuova fase di slow start.

TCP Reno implementa le stesse tre componenti più il **fast recovery**, e distingue le due situazioni. In caso di **evento di perdita** rilevato per timeout si rientra in slow start, come in Tahoe. In caso di **tre ACK duplicati**, invece, ***cwnd* riparte dalla soglia** anziché dal valore minimo: la soglia viene posta a metà della finestra corrente, **$ssthresh = cwnd/2$** , e la finestra riparte da **$cwnd = ssthresh + 3 MSS$** . I tre MSS aggiuntivi tengono conto dei tre segmenti che hanno generato gli ACK duplicati e che hanno quindi già lasciato la rete.

Il senso della differenza è importante e vale la pena esplicitarlo: tre ACK duplicati significano che il destinatario **sta comunque ricevendo** segmenti, e quindi la rete non è collassata ma soltanto affaticata; un timeout, al contrario, significa che **non arriva più nulla**, sintomo molto più grave. Reno tratta quindi in modo diverso due segnali di gravità diversa, ottenendo prestazioni migliori di Tahoe che li equipara.

22.6 Controllo di congestione network assisted: ECN

Le distribuzioni TCP implementano spesso anche il controllo *network assisted*, che coinvolge sia **IP**, tramite la marcatura del bit ECN nel proprio header, sia **TCP**, tramite i bit **CWR** ed **ECE** già incontrati nell'intestazione.

A livello di rete vengono utilizzati **due bit nell'intestazione IP**, nel campo *ToS*. Il valore **00** indica che il trasporto non supporta ECN; i valori **10** e **01**, equivalenti, vengono impostati dal **mittente** per dichiarare che i due estremi supportano il meccanismo; il valore **11** viene impostato da un **router** che rileva congestione **prima di dover scartare pacchetti**. La politica che stabilisce quando un router debba considerarsi congestionato è scelta dall'**operatore di rete**.

Il pacchetto contrassegnato prosegue fino al destinatario, che a quel punto deve informare il mittente: imposta il bit **ECE** sul segmento di ACK, notificando la congestione. Il mittente, ricevuta la notifica, **dimezza la cwnd** e imposta il bit **CWR** nell'header del prossimo segmento, comunicando di aver preso atto e ridotto la finestra.

Il vantaggio di questo approccio rispetto a quello puramente end to end è che la congestione viene segnalata **prima che si verifichino perdite**, evitando quindi sia le perdite sia le ritrasmissioni che ne conseguirebbero.

Fairness

Il capitolo dimostra, con un argomento geometrico, che più connessioni TCP che condividono lo stesso collegamento tendono a spartirsi equamente la banda, grazie al principio AIMD.

DA RICORDARE

Della fairness conta la **dimostrazione**, non soltanto l'enunciato: il piano cartesiano con le due rette e il movimento a zigzag verso la bisettrice.

Supponiamo di avere **K connessioni TCP** che condividono lo stesso collegamento di capacità **R bps**, il quale costituisce il collo di bottiglia del sistema.

DEFINIZIONE

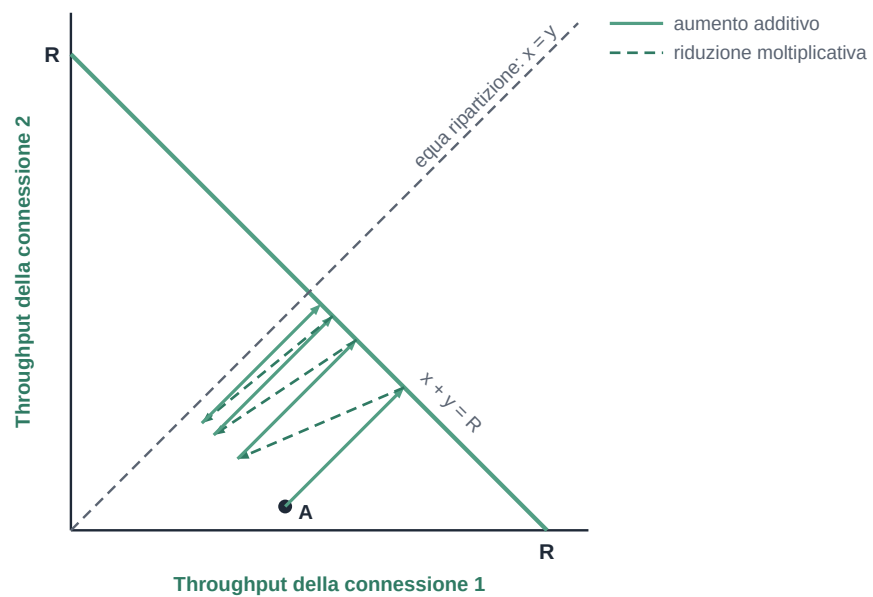
Un meccanismo di controllo della congestione si dice **fair**, cioè equo, se, quando **K** connessioni condividono un collegamento di capacità **R**, la velocità media di ciascuna tende a **R/K**.

Il punto davvero interessante è che questa proprietà **non è il risultato di una collaborazione**. Le connessioni TCP non comunicano fra loro, non si coordinano, e nemmeno il router interviene attivamente per assegnare la banda in modo equo: esso si limita a gestire un buffer di capacità limitata in cui memorizza temporaneamente i pacchetti in ingresso. Come fa allora TCP a raggiungere una suddivisione equa senza alcun coordinamento? La risposta sta interamente nel modo in cui è progettato il controllo di congestione, e in particolare nel meccanismo **AIMD**.

23.1 La dimostrazione grafica

Consideriamo il caso di **due connessioni TCP** che condividono un singolo collegamento di capacità **R bps**, sotto due ipotesi idealizzate: che abbiano gli stessi valori di **RTT e MSS**, e dunque la stessa **cwnd**, e che lavorino soltanto in modalità **congestion avoidance**, con crescita lineare di 1 MSS a ogni RTT.

Rappresentiamo la situazione su un piano cartesiano, ponendo sull'**asse x** il throughput della connessione 1 e sull'**asse y** quello della connessione 2. In questo piano vanno individuate due rette fondamentali. La retta **$x + y = R$** rappresenta il **limite massimo di banda** disponibile: al di sopra di essa il buffer del router si satura, ed è quindi una zona da evitare. La **bisettrice $x = y$** rappresenta invece la **perfetta equità**, poiché su di essa le due connessioni utilizzano esattamente la stessa banda. L'obiettivo è ottenere un throughput situato in prossimità dell'**intersezione** fra le due rette: massimo utilizzo della banda e perfetta equità insieme.



La dimostrazione grafica della fairness: il movimento a zigzag verso la bisettrice

Supponiamo ora che in un dato momento la situazione sia rappresentata dal punto **A(x, y)**, con la connessione 1 a throughput x e la connessione 2 a throughput y , e seguiamo il movimento di quel punto.

Fase di crescita. Finché non si verifica congestione, entrambe le connessioni aumentano di 1 MSS per RTT, e il nuovo punto sarà **A(x+1, y+1)**. Nel piano questo corrisponde a uno spostamento lungo una retta di **coefficiente angolare 1**, cioè **parallela alla bisettrice**. Ne segue una conseguenza importante: le due connessioni crescono alla stessa velocità, quindi anche se la suddivisione iniziale non era equa, essa **non peggiora**: la distanza dalla bisettrice resta invariata.

Fase di riduzione. A un certo punto la somma $x + y$ **supera R**: il buffer del router si riempie e cominciano le perdite. Entrambe le connessioni reagiscono **dimezzando cwnd**, e si passa quindi al punto **A(x/2, y/2)**. Poiché entrambe le coordinate vengono dimezzate, il movimento avviene lungo la retta che **congiunge il punto A all'origine**: geometricamente si tratta di una **compressione verso l'origine (0,0)**, e il fatto decisivo è che questo movimento **avvicina il nuovo punto alla bisettrice**, cioè migliora l'equità.

Il ragionamento si rende rigoroso osservando la differenza fra i due throughput, $d = x - y$. Nella fase di crescita la differenza resta invariata, perché $(x + 1) - (y + 1) = x - y$; nella fase di riduzione si dimezza, perché $x/2 - y/2 = (x - y)/2$. Dopo n eventi di perdita vale quindi:

$$d_n = d_0 / 2^n$$

e la differenza tende a zero.

La conclusione. Ripetendo il processo nel tempo, con periodi di crescita lineare (*Additive Increase*) seguiti da riduzioni improvvise (*Multiplicative Decrease*), i punti si muovono **a zigzag** nel piano ma **tendono sempre più ad avvicinarsi alla bisettrice**, anche partendo da una situazione fortemente squilibrata. L'equità emerge quindi come proprietà spontanea della geometria dell'AIMD: la fase additiva non peggiora la disparità, la fase moltiplicativa la riduce, e il risultato cumulativo è la convergenza verso l'equità.

Va aggiunta una precisazione realistica: in circostanze più vicine alla realtà, le connessioni con **RTT più basso ottengono un throughput maggiore**, semplicemente perché la velocità con cui crescono (1 MSS per RTT) è tanto maggiore quanto più breve è l'RTT. La fairness dimostrata sopra vale dunque **sotto l'ipotesi di RTT uguali**, e viene meno quando questa cade.

QUIC

I principali protocolli a livello di trasporto sono TCP e UDP, ma può accadere che nessuno dei due modelli di servizio sia adatto a una determinata applicazione. In tal caso gli sviluppatori possono realizzare un proprio protocollo **a livello di applicazione**. La situazione si presenta quando un'applicazione necessita di **più servizi di quelli offerti da UDP**, ma non desidera tutte le funzionalità di TCP, oppure richiede servizi **diversi** da quelli che TCP fornisce.

È esattamente l'approccio adottato da **QUIC**, un protocollo a livello di applicazione progettato per migliorare le prestazioni del trasporto per HTTP sicuro. Le sue caratteristiche sono quattro: utilizza **UDP** come livello di trasporto sottostante; è **orientato alla connessione**; fornisce **trasferimento dati affidabile e controllo di congestione**, ricostruiti quindi al di sopra di UDP; ed è il protocollo su cui si basa **HTTP 3**.

Il senso complessivo dell'operazione si coglie bene alla luce di quanto visto finora: QUIC prende da TCP ciò che serve (affidabilità, connessione, controllo di congestione) ma lo ricostruisce sopra UDP per potersi liberare dei vincoli di TCP, in particolare della consegna rigorosamente ordinata dei byte che, come si è visto, manteneva il blocco *head of line* anche in HTTP 2.

Calcolo del throughput

Il capitolo mostra come si calcola il throughput effettivo di un canale, sommando i diversi contributi di tempo, e perché un protocollo stop and wait spreca la capacità dei collegamenti ad alto ritardo.

Il throughput è un indice della capacità di trasmissione di un canale, ed è definito nel modo più semplice possibile:

$$\text{Throughput} = \text{bit (o byte) trasmessi con successo} / \text{tempo impiegato}$$

La difficoltà del calcolo non sta quindi nella formula, ma nel **determinare correttamente il tempo** da mettere al denominatore, che è la somma di più contributi distinti. Svolgiamo il calcolo completo.

PUNTO DI PARTENZA: LA LARGHEZZA DI BANDA

Consideriamo un canale con **BW = 10 Mbps**, il che significa che in un secondo possono essere trasmessi 10^7 bit. Ne segue che un singolo bit viene trasmesso in $1/10^7$ secondi, cioè **0,1 μs**.

TEMPO DI TRASMISSIONE

Vogliamo trasmettere un frame di **1500 byte**, cioè $1500 \cdot 8 = 12\,000$ bit. Il tempo necessario per immettere nel canale l'intero frame è:

$$T_{\text{trasmissione}} = 12\,000 \text{ bit} / 10^7 \text{ bps} = 1200 \mu\text{s} = 1,2 \text{ ms}$$

TEMPO DI PROPAGAZIONE

Dobbiamo poi considerare il tempo che il segnale impiega per arrivare **fisicamente** da A a B. Supponiamo una distanza di **1 km** e una velocità di propagazione $v \approx 200\,000 \text{ km/s}$, valore che dipende dal mezzo e che assumiamo costante. Il tempo di propagazione è il rapporto spazio/velocità:

$$T_{\text{propagazione}} = 1 \text{ km} / 200\,000 \text{ km/s} = 5 \mu\text{s}$$

Da qui si ricava anche un dato interessante, cioè quanti bit possono occupare il canale **contemporaneamente**: in 5 μs, a 10 Mbps, vengono immessi $5 \cdot 10^{-6} \cdot 10^7 = 50$ bit. È fondamentale non confondere questo con la trasmissione del frame: i 5 μs servono soltanto a far arrivare a destinazione il **primo bit**, mentre nel frattempo il mittente ne ha già immessi altri 49.

TEMPO DI TRASFERIMENTO TEORICO

Sommando i due contributi otteniamo il tempo di trasferimento complessivo, nell'ipotesi che non vi siano da considerare ACK o altri ritardi:

$$T_{\text{trasferimento}} = 1200 \mu\text{s} + 5 \mu\text{s} = 1205 \mu\text{s}$$

VERSO IL THROUGHPUT EFFETTIVO

Quello appena calcolato è però un throughput **teorico**. Quello **effettivo** è influenzato da altri elementi di attesa: il *Round Trip Time*, i tempi di elaborazione e il tempo di trasferimento relativo agli **ACK**. Il segnale di ACK ha solitamente dimensione pari a **64 byte**, cioè 512 bit, da cui:

$$T_{\text{trasmissione ACK}} = 512 \text{ bit} / 10^7 \text{ bps} = 51,2 \mu\text{s}$$

Stimiamo inoltre un tempo di **elaborazione dell'ACK** di 3 μs e un **RTT di 10 μs**, ottenuto come $5 \mu\text{s} \cdot 2$, cioè il tempo di propagazione in andata e ritorno. Il tempo complessivo diventa allora:

$$T_{\text{totale}} = 1200 + 51,2 + 3 + 10 = 1264,2 \mu\text{s}$$

IL RISULTATO

Calcoliamo infine il throughput effettivo mettendo a rapporto i bit del frame con il tempo di trasferimento totale:

$$\text{Throughput}_{\text{effettivo}} = 12\,000 \text{ bit} / 1264,2 \mu\text{s} \approx 9,49 \text{ Mbps}$$

Il valore ottenuto è molto vicino alla larghezza di banda di 10 Mbps: i fattori esterni non hanno creato colli di bottiglia significativi e il mezzo risulta **ben sfruttato**. Ma il risultato va letto con attenzione, perché dipende interamente dai numeri scelti. Il throughput reale è infatti fortemente influenzato dal tempo di elaborazione degli ACK e **dall'RTT**: nel caso appena studiato l'RTT era dell'ordine dei **microsecondi**, ma quando esso sale all'ordine dei **millisecondi** si ottengono cali di prestazione tutt'altro che trascurabili: come vedremo subito nell'esempio sulla pipeline.

ATTENZIONE A	REGOLA PRATICA
RTT e three way handshake	L'RTT è stato stimato come due volte il tempo di propagazione, ma in TCP il <i>three way handshake</i> introduce ulteriore ritardo, che può superare di molto quella stima. Se i valori disponibili sono diversi, nel calcolo si considera il maggiore fra i due .
Banda come limite	Se la larghezza di banda effettiva calcolata risultasse superiore a quella teorica , allora è quest'ultima a fungere da collo di bottiglia: la banda effettiva sarà pari a quella teorica, che ne costituisce il limite massimo.
Prefissi	milli = 10^{-3} · micro = 10^{-6} · nano = 10^{-9} · kilo = 10^3 · mega = 10^6 · giga = 10^9
Misura pratica	Il comando ping permette di calcolare l'RTT verso un server e fornisce anche la percentuale di pacchetti persi. Utilizza il protocollo ICMP .

25.1 Modalità pipeline: un esempio numerico

Riprendiamo il concetto di pipeline, questa volta con i numeri alla mano, perché è qui che se ne vede il valore. Vogliamo inviare pacchetti di **1000 byte**, cioè 8000 bit, su un canale con **BW = 1 Gbps** e **RTT = 30 ms**.

Il tempo di trasmissione di un pacchetto è $8000 \text{ bit} / 10^9 \text{ bps} = 8 \mu\text{s}$, a cui però va aggiunto l'RTT di 30 ms, cioè **30 000 μs** . Il throughput effettivo risulta allora:

$$\text{Throughput} = 8000 \text{ bit} / 30\,008 \mu\text{s} \approx 0,267 \text{ Mbps} \approx 267 \text{ kbps}$$

Il confronto è impietoso: su un canale da **1 Gbps** stiamo ottenendo circa **267 kbps**, cioè meno di un millesimo della capacità disponibile. Il collo di bottiglia non è il canale, ma il protocollo *stop and wait*, che lascia la linea inutilizzata per tutta la durata dell'RTT.

Quanti bit potrebbe invece ospitare il canale contemporaneamente? Il prodotto banda per ritardo vale $30\,000 \cdot 10^{-6} \text{ s} \cdot 10^9 \text{ bps} = 3 \cdot 10^7 \text{ bit}$. Dividendo per la dimensione di un frame otteniamo il numero di frame che potrebbero viaggiare insieme:

$$n = 3 \cdot 10^7 \text{ bit} / 8000 \text{ bit} = 3750 \text{ frame}$$

Con una finestra di **n = 3750** frame la banda verrebbe sfruttata pienamente: la finestra ideale coincide con il numero di frame necessari a **tenere il canale sempre pieno** per un intero RTT.

Routing, forwarding e piani del router

Il ruolo principale del livello di rete è trasferire pacchetti da un host a un altro. Per farlo si distinguono due funzioni che vengono spesso confuse ma che operano su scale completamente diverse.

Il **forwarding** (inoltro) è il trasferimento di un pacchetto dal collegamento in ingresso a quello di uscita appropriato: quando un router riceve un pacchetto deve decidere **su quale interfaccia di output inoltrarlo**. Per farlo utilizza le informazioni estratte dall'header per consultare la **forwarding table**. È una funzione **locale**, implementata dal **data plane**, realizzata in **hardware** perché opera nell'ordine dei **nanosecondi**.

Il **routing** (instradamento) riguarda invece la **determinazione del percorso** che i pacchetti devono seguire attraverso la rete, tramite gli algoritmi di instradamento. È una funzione **globale**, implementata dal **control plane**, realizzata via **software** e operante nell'ordine dei **millisecondi**. Il legame fra le due è preciso: si calcola l'instradamento complessivo e da esso si ricavano, per ogni nodo, le informazioni locali di forwarding: in altre parole **l'algoritmo di routing determina i valori della forwarding table**.

Il **data plane** è dunque la funzione locale che stabilisce come i pacchetti in arrivo su una porta di ingresso vengano inoltrati su una porta di uscita; il **control plane** determina come il pacchetto sia instradato fra i router lungo il percorso end to end. Per quest'ultimo esistono due approcci.

Nell'**approccio tradizionale** l'algoritmo di routing è implementato **dentro il router**, che svolge internamente sia il routing sia il forwarding; i router devono comunicare fra loro per determinare i valori delle forwarding table. È una soluzione **più lenta ma più resistente**, perché non esiste un punto singolo il cui guasto comprometta tutto.

Nell'approccio **SDN** (*Software Defined Networking*) il control plane è implementato da un **server remoto**: un controller software calcola e distribuisce le forwarding table, e router e controller interagiscono fra loro. La formulazione corretta di che cosa sia SDN è quindi che **separa control plane e data plane rendendo la rete programmabile via software**, non semplicemente che «crea una rete virtuale». È **più veloce** e più semplice da gestire in modo centralizzato, ma presenta un punto critico: **se il controller si guasta, la rete perde il proprio piano di controllo**.

Modelli di servizio del livello di rete

Quali servizi vorremmo dal livello di rete? L'elenco ideale comprende la **consegna garantita**, la consegna garantita **con ritardo limitato** (per esempio entro 40 ms), la **consegna ordinata**, una **banda minima garantita** e servizi di **sicurezza**.

La realtà è molto più modesta: il livello di rete mette a disposizione **un solo servizio**, noto come **best effort**, che offre il massimo impegno possibile ma **senza alcuna garanzia**. Sarebbe però un errore considerarlo un fallimento, e vale la pena capire perché abbia avuto successo. La **semplicità** del meccanismo ha permesso l'ampia diffusione di Internet; un'adeguata disponibilità di banda rende le prestazioni delle applicazioni in tempo reale «sufficientemente buone» per «la maggior parte del tempo»; i servizi distribuiti e replicati a livello applicativo, collocati in prossimità delle reti dei clienti, consentono l'erogazione da più sedi; e il controllo di congestione dei servizi elastici contribuisce a stabilizzare il sistema. Come si dice efficacemente, **è difficile contestare il successo del modello best effort**.

27.1 Datagram service e Virtual Circuit service

Il **datagram service** è un servizio **senza connessione e best effort**: non viene creato alcun circuito prima della trasmissione, ogni pacchetto è inviato **indipendentemente** e può seguire strade diverse, e i pacchetti possono quindi **arrivare fuori ordine**. Il sistema risulta **robusto ai guasti**: se una linea si guasta si perdono soltanto alcuni pacchetti, mentre i successivi trovano altre strade.

Il prezzo è che i router **eseguono molto lavoro**, dovendo valutare le condizioni della rete in tempo reale e prendere decisioni dinamiche, e che ogni pacchetto deve includere l'**indirizzo completo di destinazione**. Anche la **gestione della congestione è complessa**: i router non hanno informazioni globali sullo stato della rete, e la congestione (che avviene realmente a livello di rete) finisce per essere **dedotta a livello di trasporto**, attraverso i tre ACK duplicati o il timeout di TCP.

Il **virtual circuit service** fornisce invece un percorso **logico e orientato alla connessione** su un'infrastruttura condivisa. Prima della trasmissione viene stabilito un circuito virtuale fra mittente e destinatario, con ciascun router che ne cerca un altro in grado di dare garanzie; una volta stabilita la connessione **tutti i pacchetti seguono lo stesso percorso e arrivano in ordine**. La fragilità è speculare al vantaggio: se un router del circuito si guasta la comunicazione **non è più possibile** finché non se ne crea uno nuovo, e se si guasta una linea occorre ritrasmettere tutto.

La **gestione della congestione è però più semplice**: durante il setup il primo pacchetto, detto di segnalazione, comunica a ciascun router le caratteristiche del flusso; se un router non dispone di risorse sufficienti può **rifiutare la connessione**, si tenta allora un percorso alternativo e si evita così la congestione **prima che si verifichi**, anziché rilevarla dopo.

DA RICORDARE

Il confronto fra **datagram service** e **virtual circuit service** si riassume in un compromesso: il datagram sposta la complessità ai bordi mantenendo i nodi interni semplici e privi di stato, guadagnando flessibilità e robustezza al prezzo dell'assenza di garanzie e di una gestione della congestione indiretta; il circuito virtuale mantiene lo stato nei nodi, può offrire garanzie e prevenire la congestione, ma diventa rigido e fragile ai guasti.

Architettura di un router

Il capitolo descrive la struttura interna di un router: le porte di ingresso con la ricerca a prefisso più lungo, la struttura di commutazione, le code e il dimensionamento dei buffer, le politiche di scarto e di schedulazione.

Un router si compone di **quattro componenti principali**.

- Le **porte di ingresso** (*input port*) svolgono le funzioni di terminazione fisica del collegamento entrante, le funzioni di livello di collegamento e le funzioni di **ricerca** necessarie a far uscire il pacchetto dalla porta corretta. I pacchetti di controllo, come quelli che trasportano informazioni di routing, vengono passati al processore di instradamento.
- La **struttura di commutazione** (*switching fabric*) connette fisicamente le porte di ingresso a quelle di uscita all'interno del router.
- Le **porte di uscita** (*output port*) memorizzano i pacchetti provenienti dalla struttura di commutazione e li trasmettono sui collegamenti in uscita.
- Il **processore di instradamento** (*routing processor*) esegue le funzioni del control plane: nei router tradizionali gestisce le forwarding table, in quelli SDN comunica con il controller remoto.

28.1 Le porte di ingresso e il Longest Prefix Match

Un router **non lavora solo a livello di rete**: la porta di ingresso effettua la **terminazione di linea** a livello fisico, cioè la ricezione a livello di bit, e l'**elaborazione a livello di collegamento**. Utilizzando poi i valori dei campi di intestazione, cerca la porta di output consultando la forwarding table.

Un dettaglio architetturale importante: la forwarding table viene **copiata su ciascuna porta di ingresso** dal processore di instradamento, cosicché la decisione sulla porta di uscita possa essere presa **localmente dalla porta di ingresso** senza dover invocare il processore ogni volta: è questo che rende possibile operare in nanosecondi. Se i pacchetti arrivano più rapidamente di quanto la struttura di commutazione riesca a smaltirli, vengono **accodati sulla porta di input**.

L'inoltro può essere **basato sulla destinazione**, cioè fondato sul solo indirizzo IP di destinazione, oppure **generalizzato**, cioè basato su un qualsiasi insieme di campi dell'intestazione. Nel primo caso, quello classico, si applica il **Longest Prefix Match**.

Il procedimento è il seguente. Il router confronta un prefisso dell'indirizzo di destinazione con ciascuna riga della tabella: esegue un **AND logico** fra l'IP di destinazione e la maschera, ottenendo l'indirizzo di rete, e confronta poi il risultato con il prefisso della riga mediante uno **XOR**. Se lo XOR restituisce **0** c'è corrispondenza, e il pacchetto viene inoltrato sul collegamento associato.

Il caso interessante è quello delle **corrispondenze multiple**, che si verifica quando due o più reti si sovrappongono. In tal caso il router adotta la regola della **corrispondenza a prefisso più lungo**: sceglie cioè la riga la cui maschera di sottorete ha il **maggior numero di bit a 1**, ovvero la rete più specifica. La ricerca avviene in nanosecondi e per questo i router impiegano hardware specializzato come le **TCAM** (*Ternary Content Addressable Memory*); i router Cisco Catalyst arrivano a un milione di voci in TCAM.

28.2 La struttura di commutazione

La struttura di commutazione inoltra i pacchetti dalla porta di ingresso a quella di uscita, e la sua velocità è misurata dallo **switching rate**, spesso espresso come multiplo della velocità di linea: con N ingressi si vorrebbe un tasso di commutazione pari a N volte la velocità di linea. La commutazione può essere realizzata in tre modi.

La **commutazione in memoria** è quella dei computer tradizionali, gestita sotto il controllo diretto della CPU. Le porte funzionano come dispositivi di I/O: quando arriva un pacchetto la porta di ingresso lo segnala tramite **interrupt** e lo copia nella memoria del processore, che individua la porta di uscita e vi copia il pacchetto. Il limite è la **banda della memoria**: se questa consente di scrivere e leggere B pacchetti al secondo, il throughput di inoltro è **inferiore a $B/2$** , perché ogni pacchetto attraversa il bus due volte. Inoltre si può inoltrare **un solo pacchetto per volta**, anche verso porte di uscita diverse.

Nella **commutazione tramite bus** le porte di ingresso trasferiscono il pacchetto direttamente a quelle di uscita attraverso un **bus condiviso**, senza intervento del processore: viene aggiunta un'**etichetta interna** che indica la porta di output, il pacchetto è ricevuto da tutte le porte ma solo quella corrispondente lo accoglie, e infine l'etichetta viene rimossa. Anche qui si trasferisce **un solo pacchetto per volta**, e la velocità è limitata dalla banda del bus.

La **commutazione attraverso rete di interconnessione** supera il limite del bus condiviso impiegando una **matrice di commutazione** (*crossbar switch*), costituita da $2n$ bus che collegano n porte di ingresso a n di uscita. Ogni bus verticale interseca tutti gli orizzontali, e ciascun incrocio può essere aperto o chiuso dal controller: se un pacchetto giunge alla porta A e deve uscire dalla porta Y , il controller chiude l'incrocio fra A e Y , A invia il pacchetto sul proprio bus e solo Y lo riceve.

Il vantaggio decisivo è lo sfruttamento del **parallelismo**: un pacchetto in arrivo su un'altra porta e diretto a una porta di uscita diversa può essere inoltrato **contemporaneamente**, perché usa bus differenti. Se due pacchetti sono però destinati alla **stessa** porta, uno dei due deve attendere in coda. Le reti di interconnessione più sofisticate usano elementi a **più stadi**: il router Cisco CRS impiega una rete a tre stadi con otto strutture di commutazione in parallelo, in cui la porta di ingresso suddivide il pacchetto in K pezzi, li invia attraverso le N strutture e la porta di output li riassume.

28.3 Accodamento e dimensionamento dei buffer

Le code possono formarsi sia alle porte di ingresso sia a quelle di uscita, e la loro lunghezza dipende dal traffico e dalle velocità relative della struttura di commutazione e della linea. Quando crescono troppo la memoria del router può esaurirsi e si verifica **perdita di pacchetti**. Il criterio generale è semplice: se il tasso di commutazione è **N volte più veloce** della velocità di linea, l'accodamento è trascurabile anche nel caso peggiore.

ACCODAMENTO IN INGRESSO E BLOCCO HOL

Se la struttura di commutazione non è abbastanza veloce, si accumulano code alle porte di ingresso. I pacchetti vengono trasferiti con procedura **FCFS** e più pacchetti possono passare in parallelo se destinati a porte di uscita diverse; ma se **due pacchetti in testa a due code sono destinati alla stessa porta di uscita**, uno dei due viene bloccato.

Ne consegue il fenomeno del **blocco HOL** (*Head of Line Blocking*): un pacchetto in coda deve attendere il trasferimento **anche se destinato a una porta libera**, semplicemente perché il pacchetto che lo precede nella stessa coda è bloccato. La coda di ingresso comincia allora a crescere indefinitamente, causando perdite e ritardi. Vale la pena notare che si tratta dello stesso fenomeno già incontrato in HTTP 1.1 con il pipelining: una struttura FIFO in cui il primo elemento bloccato paralizza tutti i successivi.

ACCODAMENTO IN USCITA

Quando il tasso in ingresso supera la velocità di trasmissione della linea, le code si formano alle porte di uscita e, per mancanza di buffer o congestione, si hanno perdite. Occorre allora una politica: con il **drop tail** si scarta il pacchetto in arrivo, mentre con le politiche basate sulla **priorità** si sceglie quale pacchetto eliminare in base alla sua priorità.

Un'osservazione importante: è **vantaggioso eliminare un pacchetto o marcarne l'intestazione prima che il buffer sia pieno**, così da fornire al mittente un segnale anticipato di congestione. A tal fine si può marcare il bit **ECN** nell'header IP (meccanismo già visto nel controllo di congestione network assisted) oppure utilizzare algoritmi di **AQM** (*Active Queue Management*). Quando infine più pacchetti sono accodati in uscita, uno **schedulatore** stabilisce in quale ordine trasmetterli.

QUANTA MEMORIA SERVE

Per molti anni la regola (RFC 3439) prescriveva che la dimensione del buffer B fosse pari al prodotto fra RTT e capacità del link C :

$$B = RTT \cdot C$$

Con $C = 10$ Gbps e $RTT = 250$ ms si otterrebbero 2,5 Gbit di buffer. Studi più recenti suggeriscono però che, in presenza di **N flussi** che attraversano il collegamento, la quantità necessaria sia sensibilmente minore:

$$B = RTT \cdot C / \sqrt{N}$$

Il dimensionamento è un compromesso. Buffer **troppo grandi** riducono il tasso di perdita ma aumentano i ritardi di coda, con RTT e ritardo end to end elevati, prestazioni scadenti per le applicazioni in tempo reale e **risposta lenta di TCP** alla congestione, dato che il segnale di perdita arriva tardi. Buffer **troppo piccoli**, per contro, non riescono a gestire i picchi di traffico.

28.4 Politiche di scarto e schedulazione

Sulle **politiche di scarto** (*packet discarding policy*) si distinguono due approcci. Il **tail drop** scarta a priori i pacchetti in arrivo quando la coda è piena, senza considerare priorità né tipologia di traffico: è semplicissimo da implementare ma può causare **burst di drop**, cioè serie di perdite consecutive. Il **priority queueing** divide invece il buffer in più code con priorità diverse; i pacchetti vengono etichettati con un livello di priorità, oppure questa è stabilita dal router, e i pacchetti ad alta priorità non vengono scartati finché in coda ve ne sono di priorità inferiore.

Sulla **schedulazione**, cioè sull'ordine di trasmissione in uscita, esistono quattro strategie.

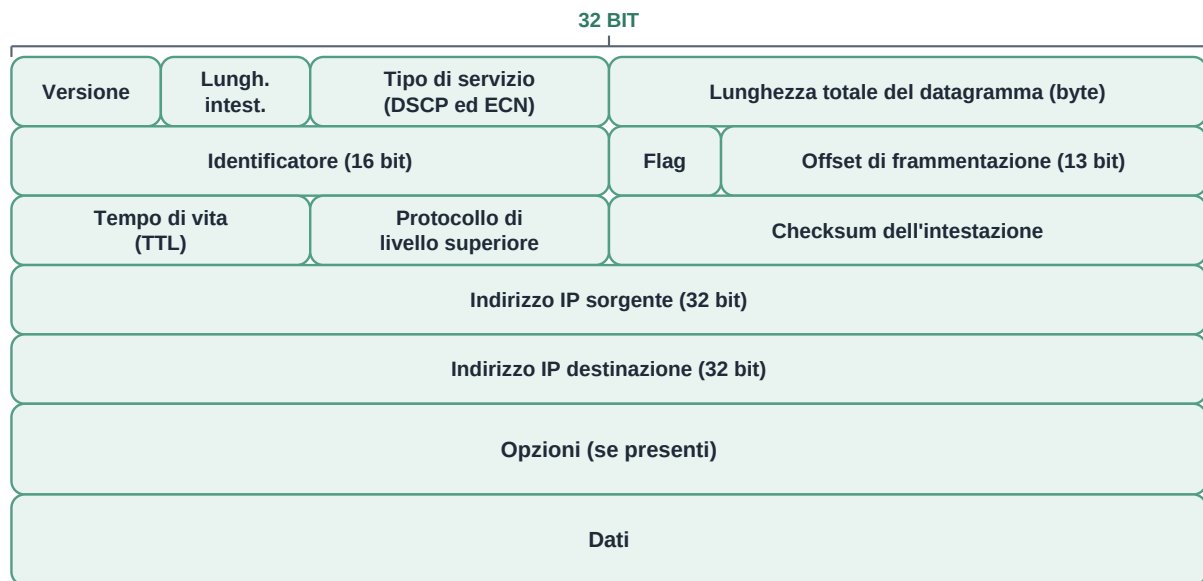
STRATEGIA	FUNZIONAMENTO
FCFS / FIFO	I pacchetti sono trasmessi nell'ordine di arrivo. Semplice ma non conveniente , perché ignora completamente l'importanza del traffico.
Code con priorità	I pacchetti sono classificati in classi di priorità usando i campi dell'header: si può stabilire che il traffico di gestione della rete abbia priorità su quello utente, o che le applicazioni in tempo reale ne abbiano su quelle non in tempo reale. Si trasmette sempre dalla coda non vuota a priorità più alta , e all'interno di ciascuna coda vale il FIFO.
Round Robin	Il traffico è suddiviso in classi e le code vengono alternate : nella forma semplice si invia un pacchetto per classe a rotazione; nella modalità conservativa il collegamento non resta mai inattivo, perché se una coda è vuota si passa subito alla successiva.
Weighted Fair Queuing	Round Robin generalizzato : a ogni classe i è assegnato un peso w_i , e si garantisce che la coda i riceva una frazione di servizio pari a $w_i / \sum w_j$. Fornisce così una garanzia di banda minima per classe di priorità.

Il protocollo IPv4

Il capitolo descrive il protocollo IPv4: il formato del datagramma, il meccanismo di frammentazione e la struttura degli indirizzi, dal vecchio schema a classi al CIDR.

29.1 Il formato del datagramma

Un pacchetto a livello di rete è detto **datagramma**. I suoi campi sono i seguenti.



Struttura del datagramma IPv4

La **versione** occupa 4 bit e specifica la versione del protocollo. La **lunghezza dell'intestazione** è necessaria perché l'header ha lunghezza variabile a causa del campo opzioni, e indica dove iniziano i dati; poiché di norma le opzioni non sono presenti, l'header occupa tipicamente **20 byte**. Il **Type of Service**, nato per distinguere i tipi di datagramma e assegnare priorità, è stato in seguito ridefinito: sei dei suoi bit (**DSCP**) classificano il traffico ai fini della qualità del servizio, e due sono usati dal meccanismo **ECN** per la notifica della congestione. La **lunghezza del datagramma** indica la lunghezza totale in byte, header più dati.

Tre campi servono alla frammentazione: **ID**, che identifica il datagramma e i suoi frammenti; i **Flag**, tre bit di cui uno inutilizzato e gli altri due **DF** (*Don't Fragment*) e **MF** (*More Fragments*); e l'**Offset**, che indica la posizione del frammento.

Il **TTL** (*Time to Live*) impedisce ai datagrammi di restare in circolazione indefinitamente. È importante non lasciarsi ingannare dal nome: **non indica un tempo** ma è un **contatore di salti**, decrementato ogni volta che il datagramma giunge a un router; quando raggiunge 0 il datagramma viene scartato e il router invia al mittente un messaggio ICMP di tempo scaduto. Su questo comportamento si basa il comando **traceroute**, che invia datagrammi con TTL crescente per scoprire uno a uno i router lungo il percorso.

Il campo **Protocollo** specifica il protocollo di trasporto trasportato: **6** per TCP, **17** per UDP. L'**Header Checksum** consente ai router di rilevare errori: ogni router lo ricalcola per verificare la correttezza dell'intestazione. Seguono gli **indirizzi IP sorgente e destinazione**, il campo **opzioni** a lunghezza variabile e infine i **dati** (payload), che contengono di norma il segmento di trasporto. Se il datagramma trasporta un segmento TCP, l'intestazione complessiva sarà di **40 byte**, sommando i 20 di IP e i 20 di TCP.

29.2 La frammentazione

La frammentazione è il processo di suddivisione dei datagrammi in pezzi più piccoli quando risultano troppo grandi per essere trasmessi su un collegamento con **MTU** limitata. Si realizza con i tre campi già citati.

Tutti i frammenti condividono lo **stesso ID**, che è ciò che permette di riconoscerli come appartenenti allo stesso datagramma originario. Il flag **DF** è impostato dal mittente per **impedire** la frammentazione: se il router non riesce a far passare il datagramma, o cambia strada o invia al mittente un messaggio chiedendo pacchetti più piccoli. Il flag **MF** indica se vi siano altri frammenti: **MF = 0** segnala che si tratta dell'ultimo, mentre **MF = 1** indica che ne seguono altri. L'**Offset** infine indica la posizione del frammento all'interno del datagramma originale, espressa in unità di 8 byte, e **Offset = 0** identifica il primo.

Due osservazioni pratiche. La prima: **se si perde un frammento si perde tutto**, perché IP non prevede la ritrasmissione dei singoli frammenti e la destinazione, non potendo completare il riassemblaggio, scarta l'intero datagramma: ed è precisamente questa fragilità che ha portato a sconsigliare la frammentazione in IPv6. La seconda riguarda i ruoli: in IPv4 **frammentano sia il mittente sia i router**, mentre il **riassemblaggio** viene effettuato soltanto **dalla destinazione**.

29.3 Indirizzamento IPv4

Gli indirizzi IP sono lunghi **32 bit**, cioè 4 byte, il che dà $2^{32} \approx 4$ miliardi di indirizzi, scritti in **notazione decimale puntata**. Ogni indirizzo è associato a un'**interfaccia**, e ogni interfaccia ha un indirizzo globalmente univoco, fatta eccezione per gli indirizzi privati usati dietro un NAT.

Vale la pena soffermarsi sulla scelta di fondo dello schema di indirizzamento, perché è un tema che ritornerà anche a livello di collegamento. Uno **schema flat** funziona come il numero di cellulare: identifica univocamente ma **non dice nulla sulla posizione**. Uno **schema gerarchico** funziona come il telefono fisso: il numero è legato al luogo, il che rende l'instradamento efficiente ma risulta **poco flessibile**, perché non si può spostare l'indirizzo da una zona all'altra. Gli indirizzi IP adottano un compromesso: hanno **una parte gerarchica e una parte flat**.

DAL CLASSFUL ADDRESSING AL CIDR

In passato l'indirizzamento seguiva uno schema a classi, il **classful addressing**: la **classe A** prevedeva 128 reti (2^7) da circa 16 milioni di indirizzi ciascuna, la **classe B** circa 16 mila reti (2^{14}) da 65 536 indirizzi, la **classe C** circa 2 milioni di reti (2^{21}) da 256 indirizzi.

Lo schema si rivelò **inefficiente** per organizzazioni con esigenze variabili, generando grandi blocchi inutilizzati. Il problema si coglie con un esempio: una sottorete di classe C offre $2^8 - 2 = 254$ **host**, troppo pochi per molte organizzazioni; ma la successiva, di classe B, ne offre **65 534**, un numero eccessivo. Non esisteva alcuna via di mezzo.

Per affrontare il problema è stato introdotto il **CIDR** (*Classless Inter Domain Routing*). L'indirizzo viene diviso in una parte che identifica la **sottorete** e una che identifica gli **host**, nella forma **a.b.c.d/x**, dove gli **x** bit, detti **prefisso**, sono quelli destinati alla sottorete. La notazione **/24** indica per esempio che i primi 24 bit sono dedicati alla rete e i restanti 8 agli host, dei quali se ne possono usare $2^8 - 2$, perché l'indirizzo che termina con tutti zeri identifica la **rete** e quello con tutti uno il **broadcast**. Invece di classi rigide, il CIDR consente maschere di **lunghezza variabile**, offrendo flessibilità e riducendo lo spreco.

INDIRIZZI DI RETE NOTI

INDIRIZZO	SIGNIFICATO
0.0.0.0	Indica <i>questo host</i> : è usato come indirizzo sorgente da un host che non ha ancora un indirizzo, per esempio durante la richiesta DHCP, e da un server per indicare l'ascolto su tutte le proprie interfacce. Non è instradabile.
127.0.0.0/8	Indirizzo di loopback : permette a un dispositivo di comunicare con sé stesso. Il traffico scende la pila fino al livello di rete e risale, senza coinvolgere i livelli bassi; è usato per testare applicativi. 127.0.0.1 è l'indirizzo di <i>localhost</i> .
10.0.0.0/8	Blocco di indirizzi privati , riservati alle reti interne: questi indirizzi non vengono instradati su Internet .
192.168.0.0/16 e 172.16.0.0/12	Altri blocchi di indirizzi privati, non instradabili su Internet.
169.254.0.0/16	Indirizzo che l'host si assegna automaticamente quando non riesce a ottenerne uno da un server DHCP. È l' APIPA (<i>Automatic Private IP Addressing</i>), valido soltanto sul collegamento locale.

Quanto all'**assegnazione** degli indirizzi, per ottenere un blocco occorre rivolgersi al proprio **ISP**, che a sua volta si rivolge a un ente superiore. Il compito era inizialmente affidato alla **IANA**; nel **1992** nacquero i **RIR** (*Registri Internet Regionali*), che gestiscono gli indirizzi IP e i numeri di sistema autonomo per ciascuna area geografica, e nel **1998** fu fondata l'**ICANN** per coordinare tutti i RIR. Oggi ne esistono cinque (APNIC, ARIN, LACNIC, RIPE NCC e AFRINIC) e ciascuno è responsabile dei **LIR** (*Local Internet Registries*) che gestiscono gli indirizzi a livello locale.

Indirizzamento intra LAN e inter LAN

Il capitolo segue un pacchetto dal mittente al destinatario, distinguendo il caso in cui i due host si trovano nella stessa rete locale da quello in cui si trovano su reti diverse.

DA RICORDARE

L'indirizzamento **intra LAN** e **inter LAN** richiede di seguire un pacchetto attraverso tre passaggi: l'incapsulamento, la decisione di instradamento e la risoluzione ARP.

Seguiamo che cosa accade quando un host di una rete vuole inviare un messaggio a un host di un'altra rete. Il messaggio deve essere **incapsulato** attraversando la pila.

Anzitutto il messaggio viene **segmentato** dal livello di trasporto, che aggiunge l'header TCP o UDP con le porte di mittente e destinatario. Il segmento viene poi **incapsulato in un pacchetto**, con l'aggiunta dell'header IP contenente gli indirizzi IP di mittente e destinatario. Infine, per essere trasmesso, il pacchetto viaggia dentro un **frame**, al quale viene aggiunto un header contenente gli **indirizzi MAC** di mittente e destinazione.

Qui si colloca la decisione cruciale. L'host mittente capisce **dalla parte iniziale dell'indirizzo IP di destinazione**, cioè dalla parte di rete, se il destinatario si trovi nella **stessa sottorete**. Il confronto si esegue applicando l'AND logico fra i due indirizzi e la maschera e verificando se gli indirizzi di rete risultanti coincidano.

Se il destinatario è nella stessa LAN (caso *intra LAN*), il mittente inserisce nel frame direttamente il **MAC address del destinatario**, ottenuto se necessario tramite ARP, e il frame viaggia localmente senza coinvolgere alcun router.

Se il destinatario è su un'altra rete (caso *inter LAN*), il mittente deve passare il frame al **router (gateway)**, e quindi inserisce nel frame il **MAC address del gateway**: non quello del destinatario finale, che non conosce e che sarebbe comunque inutile. Il router dovrà a quel punto **decapsulare il frame** per leggere l'indirizzo IP di destinazione e, consultando la propria forwarding table, inviare il pacchetto sull'interfaccia corretta.

È il punto concettuale da fissare: **l'indirizzo IP di destinazione resta invariato lungo tutto il percorso, mentre l'indirizzo MAC cambia a ogni salto**. L'IP identifica il destinatario finale, il MAC identifica soltanto il prossimo dispositivo sul collegamento locale.

A livello di collegamento si utilizza infatti il **MAC address** (*Media Access Control*), che è un **indirizzo flat**: l'indirizzo della scheda di rete, univoco, rappresentato in esadecimale su **6 byte** e scritto a livello hardware. È la **IEEE** che si occupa di distribuire i blocchi di indirizzi MAC alle aziende produttrici. All'interno di una LAN i dispositivi comunicano esclusivamente tramite MAC address.

ARP, RARP e BOOTP

Il capitolo descrive il protocollo ARP, che associa gli indirizzi IP agli indirizzi MAC all'interno di una rete locale, le sue vulnerabilità e i protocolli storici RARP e BOOTP.

31.1 Il protocollo ARP

Il protocollo **ARP** (*Address Resolution Protocol*) è un componente fondamentale delle reti Ethernet e svolge il compito di **mappare gli indirizzi IP sui corrispondenti indirizzi MAC** all'interno di una rete locale. Risolve cioè il problema che si è appena visto: conosco l'IP, ma per costruire il frame mi serve il MAC.

Prima ancora di avviare ARP, il mittente verifica (grazie alla parte di rete dell'indirizzo IP) se il destinatario si trovi nella propria LAN. Se si trova altrove, l'indirizzo MAC che dovrà cercare **non è quello del destinatario ma quello del router**.

Il protocollo si articola in due messaggi. Con l'**ARP Request** un dispositivo che non conosce il MAC corrispondente a un IP invia un frame **in broadcast all'indirizzo FF:FF:FF:FF:FF:FF**, raggiungendo tutti i dispositivi della LAN; il pacchetto contiene sia l'indirizzo IP e MAC del mittente sia l'indirizzo IP cercato, e chiede in sostanza «chi ha questo indirizzo IP? comunicami il tuo MAC». Con l'**ARP Reply** il dispositivo che riconosce come proprio quell'indirizzo IP risponde (questa volta **non in broadcast** ma direttamente al richiedente) comunicando il proprio indirizzo MAC. Da notare che anche il **destinatario aggiorna la propria ARP table** memorizzando la corrispondenza IP e MAC del mittente, che ha ricevuto nella richiesta.

LA ARP TABLE

La **tabella ARP** memorizza temporaneamente le associazioni fra indirizzi IP e MAC della rete locale. Ogni nodo ne possiede una, i cui record contengono **tre valori**: indirizzo IP, indirizzo MAC e **Time To Live**. Si tratta di una **cache temporanea**: i contenuti si aggiornano quando arrivano risposte ARP e **scadono nel tempo**, con un valore tipico di **20 minuti**. La sua funzione è evitare di dover effettuare una richiesta ARP per ogni singolo pacchetto inviato.

I DUE CASI D'USO

Invio nella stessa sottorete. Se A deve inviare un datagramma a B nella stessa sottorete, deve conoscerne il MAC. Se questo è già nella propria tabella ARP, invia direttamente. Altrimenti emette una ARP Request in broadcast; la risposta arriva in modalità non broadcast dal dispositivo il cui IP corrisponde, contenendo il suo MAC, e la tabella viene aggiornata.

Invio su sottorete diversa. Alice vuole scrivere a Bob, che si trova su un'altra sottorete. Alice deve indicare nel frame il **MAC dell'interfaccia del router** a cui è collegata: se lo ha in tabella lo usa, altrimenti lo ottiene con una ARP Request contenente l'IP del router, al quale il router risponde con una ARP Reply. Alice memorizza il MAC e invia il frame. A questo punto il **router** confronta l'IP di destinazione con la propria tabella di inoltri e determina che il pacchetto deve uscire da una certa interfaccia: ma **non conosce il MAC di Bob**. Emette quindi a sua volta una ARP Request, che lo switch diffonde in broadcast; Bob riconosce il proprio IP e risponde con una ARP Reply, e il router memorizza la corrispondenza nella propria ARP table. Solo ora dispone di tutto il necessario per inoltrare il pacchetto a Bob.

Si osservi come, una volta popolate le tabelle ARP e per tutta la durata delle relative *entry*, l'inoltro risulti **molto più veloce**, perché l'intera fase di risoluzione viene saltata.

31.2 Le vulnerabilità di ARP

ARP presenta vulnerabilità significative, che vale la pena elencare perché sono alla base di attacchi molto comuni.

Il messaggio di richiesta è inviato **in broadcast**, e quindi visibile a tutti. Alcuni router adottano meccanismi di sicurezza basati sul MAC address, per esempio per bloccare l'accesso a determinati siti, ma **il MAC di un dispositivo può essere cambiato**, il che rende tali meccanismi aggirabili. Soprattutto, **ARP non richiede alcuna autenticazione** dei dispositivi che inviano richieste o risposte: un qualsiasi dispositivo può affermare di essere il router pur non essendolo, e dirottare così il traffico verso di sé: traffico che, se non cifrato, potrà anche leggere.

A ciò si aggiunge il fatto che **ARP è stateless**: non tiene traccia degli stati precedenti, e quindi un dispositivo può inviare una risposta ARP **senza che vi sia stata una richiesta corrispondente**. Infine, un host che riceve un pacchetto ARP è tenuto ad **aggiornare la propria tabella** con l'associazione IP e MAC fornita, **anche se questa informazione è falsa**.

L'insieme di queste debolezze rende possibile l'attacco noto come **ARP spoofing** o **ARP poisoning**, con cui un attaccante convince gli altri dispositivi a salvare associazioni sbagliate: tipicamente facendosi passare per il router. È il meccanismo alla base degli attacchi **man in the middle** in rete locale, con cui l'attaccante può intercettare, modificare o bloccare il traffico.

31.3 RARP e BOOTP

Il **RARP** (*Reverse Address Resolution Protocol*) consente a una macchina di **richiedere e ottenere il proprio indirizzo IP** da un server, basandosi sul proprio indirizzo MAC: è quindi esattamente l'opposto di ARP. Il dispositivo invia una richiesta RARP e il server, che conosce la mappatura fra MAC e IP, risponde con l'indirizzo corretto. Anche RARP **non è sicuro**, perché richieste e risposte non sono autenticate, ed è stato in gran parte sostituito da DHCP.

Il **BOOTP** è stato sviluppato per fornire una **configurazione iniziale e un avvio remoto** dei dispositivi: consente di ottenere un indirizzo IP, l'indirizzo del server di avvio e altre informazioni di configurazione. Funziona su **UDP** e utilizza il MAC address per identificarsi al server. Anch'esso è stato in gran parte sostituito da DHCP, il cui supporto tuttavia lo include spesso.

Routing table e host table

Il capitolo descrive la struttura delle tabelle con cui host e router decidono dove inviare i pacchetti, e i tre modi in cui queste tabelle vengono popolate.

Le **host table** contengono tre campi: la **rete di destinazione**, che specifica quali reti sono raggiungibili da quell'host; il **router successivo**, campo **vuoto** se la destinazione è nella stessa LAN e contenente altrimenti l'indirizzo IP del gateway; e il **numero di salti** necessari a raggiungere la destinazione.

Le **routing table** ne contengono quattro: la **rete di destinazione**, il **router successivo** (vuoto quando la rete di destinazione è direttamente collegata al router), il **numero di salti** e l'**interfaccia**, cioè il punto di connessione fisica o logica attraverso cui il router invia i pacchetti.

I router devono mantenere una mappa di tutte le reti di cui sono a conoscenza, e in essa vengono definite le **rotte**: una rotta è un'istruzione su come raggiungere una rete specifica attraverso una specifica interfaccia. Esistono **tre modi** per popolare le tabelle: le rotte **directly connected**, relative a reti direttamente collegate al router; le **static route**, fornite manualmente dall'amministratore; e le **dynamic route**, apprese scambiando tabelle con altri router mediante protocolli di routing dinamico come RIP.

Subnetting, supernetting e VLSM

Il capitolo mostra come suddividere un blocco di indirizzi in sottoreti, anche di dimensioni diverse, e come aggregare più reti contigue in un'unica supernet, con un esercizio svolto completo.

33.1 La subnet mask

Le **subnet mask** servono a definire la dimensione della rete, separando la parte dell'indirizzo IP che identifica la rete da quella che identifica gli host. Una netmask è una sequenza di **32 bit**, in cui i **bit a 1** indicano la parte di rete e i **bit a 0** la parte host.

Più precisamente, un indirizzo IP si può suddividere in **tre parti**: la **net**, che identifica la rete ed è determinata dalla netmask o dal prefisso CIDR; la **subnet**, costituita da bit «presi in prestito» dalla parte host per creare le sottoreti, e che serve a suddividere una rete grande in reti più piccole; e l'**host**, che identifica il singolo dispositivo e deve essere univoco all'interno della sua sottorete.

Per determinare l'**indirizzo di rete** di una subnet si procede in tre passi: si trasformano indirizzo IP e subnet mask in **binario**; si esegue l'**AND logico bit a bit** fra i due; il risultato è l'indirizzo di rete.

IP	192.168.1.34	→	11000000.10101000.00000001.00100010	
Mask	255.255.255.0	→	11111111.11111111.11111111.00000000	
				AND
Rete	192.168.1.0	→	11000000.10101000.00000001.00000000	
Broadcast	192.168.1.255		(parte host tutta a 1)	
Host utili da	192.168.1.1 a 192.168.1.254			

Calcolo dell'indirizzo di rete tramite AND logico

Lo stesso procedimento serve a stabilire **se due indirizzi appartengano alla stessa LAN**: si trasformano entrambi in binario, si esegue l'AND con la rispettiva subnet mask e si confrontano gli indirizzi di rete ottenuti. Se sono **identici** i due host sono sulla stessa LAN e possono comunicare direttamente; se sono **diversi**, il frame va inviato al **gateway** per uscire dalla rete.

Due casi particolari meritano una nota. La maschera **/31** è spesso usata per un'interfaccia che corrisponde all'estremità di una rete punto a punto. La maschera **/32** definisce invece una rete che contiene un solo indirizzo, cioè l'indirizzo di rete stesso.

33.2 Supernetting

Il **supernetting** è la tecnica di **aggregazione di più reti contigue in un'unica supernet**. Richiede l'uso del CIDR e impone tre condizioni: le reti devono essere **contigue**, devono avere la **stessa subnet mask** e il loro numero deve essere una potenza di 2, con il primo indirizzo allineato al nuovo prefisso (192.168.1.0/24 e 192.168.2.0/24, pur contigue, non formano una /23). Si utilizza poi una maschera di **dimensione minore**, cioè un prefisso più corto.

L'utilità è concreta: **riduce il numero di rotte** nella routing table, semplificando la gestione e migliorando le prestazioni del routing dinamico. Un ISP può per esempio aggregare le sottoreti dei propri clienti in un'unica voce, evitando di annunciare separatamente 10.0.1.0/24, 10.0.2.0/24 e così via.

Vediamo l'esempio classico. Supponiamo di avere quattro reti di classe C: **192.168.0.0/24**, **192.168.1.0/24**, **192.168.2.0/24** e **192.168.3.0/24**. Sono contigue e hanno la stessa maschera, quindi possiamo unirle. Il procedimento consiste nello scrivere gli indirizzi in binario e **confrontare i bit da sinistra verso destra** per individuare quanti ne hanno in comune.

Rete IP	Binario
192.168.0.0	11000000.10101000.00000000 00.00000000
192.168.1.0	11000000.10101000.00000000 01.00000000
192.168.2.0	11000000.10101000.00000000 10.00000000
192.168.3.0	11000000.10101000.00000000 11.00000000
	← 22 bit in comune →

Confronto binario dei quattro indirizzi: i primi 22 bit coincidono

I bit in comune risultano **22**, e questa è quindi la lunghezza del prefisso CIDR della supernet. La **supernet mask** è dunque **/22**, l'indirizzo della supernet è **192.168.0.0/22**, l'intervallo coperto va da **192.168.0.0** a **192.168.3.255** e gli indirizzi utilizzabili sono $2^{10} - 2$.

33.3 VLSM

Il **VLSM** (*Variable Length Subnet Masking*) è la tecnica di subnetting che permette di suddividere una rete in **più sottoreti di dimensioni diverse**, usando maschere di lunghezza variabile. Si basa sul CIDR, consente di **adattare la dimensione delle sottoreti al numero effettivo di host** necessari ed evita così sprechi di indirizzi.

Il vantaggio si apprezza solo confrontandolo con l'alternativa. Supponiamo un'azienda con due reparti, **marketing con 50 host** e **vendite con 10 host**, e a disposizione la rete **192.168.1.0/24** con 254 indirizzi utilizzabili.

Con il **subnetting a lunghezza fissa** si deve dimensionare tutto sulla sottorete più grande: per 50 host servono 6 bit di host, perché $2^6 - 2 = 62 \geq 50$, e quindi una maschera **/26** ($32 - 6$). Applicando la stessa maschera a entrambe le sottoreti, però, al reparto vendite verrebbero assegnati 62 indirizzi a fronte di 10 necessari: **si sprecano 52 indirizzi**.

Con il **VLSM** si dimensiona invece ogni sottorete sul proprio fabbisogno. Il marketing riceve una **/26** (62 host disponibili per 50 necessari); le vendite ricevono una **/28**, perché per 10 host bastano 4 bit di host ($2^4 - 2 = 14 \geq 10$) e quindi $32 - 4 = 28$. Il risultato è che il VLSM **adatta la dimensione delle subnet alle reali necessità**, riducendo drasticamente lo spreco.

33.4 Esercizio svolto

Vediamo il procedimento completo su un caso con più sottoreti. Supponiamo di disporre del blocco **192.168.10.0/24** e di dover configurare quattro LAN con, rispettivamente, **100**, **50**, **25** e **2** host.

Passo 1: ordinare le sottoreti dalla più grande alla più piccola. È il passaggio che rende possibile il VLSM: assegnando prima i blocchi grandi si evita di frammentare lo spazio in modo da non poterli più collocare.

Passo 2: per ciascuna, calcolare i bit di host necessari risolvendo $2^n - 2 \geq \text{host richiesti}$, e da questi la maschera come $32 - n$.

LAN	host	bit host (n)	$2^n - 2$	maschera	blocco
LAN A	100	7	126	/25	128 indirizzi
LAN B	50	6	62	/26	64 indirizzi
LAN C	25	5	30	/27	32 indirizzi
LAN D	2	2	2	/30	4 indirizzi

Passo 2: dimensionamento delle sottoreti

Passo 3: assegnare i blocchi in sequenza, partendo dall'inizio dello spazio disponibile e facendo iniziare ogni sottorete subito dopo la fine della precedente.

LAN A	192.168.10.0/25	rete .0	broadcast .127	host da .1 a .126
LAN B	192.168.10.128/26	rete .128	broadcast .191	host da .129 a .190
LAN C	192.168.10.192/27	rete .192	broadcast .223	host da .193 a .222
LAN D	192.168.10.224/30	rete .224	broadcast .227	host da .225 a .226

Spazio residuo: da 192.168.10.228 a 192.168.10.255 (28 indirizzi liberi)

Passo 3: assegnazione dei blocchi

Passo 4: verificare. Per ogni sottorete l'indirizzo di rete deve essere **multiplo della dimensione del blocco** (0, 128, 192, 224 lo sono rispettivamente di 128, 64, 32 e 4), e i blocchi non devono sovrapporsi. Il collegamento punto a punto fra due router, come la LAN D, richiede sempre e soltanto una **/30**: due indirizzi utili, uno per interfaccia.

33.5 Quando la maschera assegnata è sbagliata

Un caso particolare merita attenzione, perché rovescia il problema: può accadere che il blocco assegnato **non sia sufficiente** a ospitare le sottoreti richieste, e la risposta corretta consiste nell'accorgersene e nel dirlo, non nel forzare una soluzione impossibile.

La **verifica preliminare** da fare sempre, prima di iniziare ad assegnare, è la seguente: si arrotonda il fabbisogno di ciascuna sottorete alla **potenza di 2 immediatamente superiore** (ricordando i due indirizzi riservati per rete e broadcast), si sommano i blocchi così ottenuti e si confronta il totale con la dimensione del blocco assegnato.

Consideriamo il caso seguente: viene assegnato il blocco **4.20.64.0/20** e si chiede di configurare cinque LAN con **2340, 501, 128, 127 e 5** indirizzi. Un prefisso /20 lascia 12 bit di host, quindi il blocco contiene $2^{12} = 4096$ indirizzi. Procediamo con la verifica.

LAN	richiesti	blocco necessario (potenza di 2)	maschera
Lan10	2340	4096	/20
Lan12	501	512	/23
Lan18	128	256	/24
Lan14	127	256	/24
Lan20	5	8	/29
<hr/>			
TOTALE		5128 indirizzi	> 4096 disponibili

Verifica preliminare: il blocco assegnato non basta

Il totale necessario, **5128 indirizzi**, supera i **4096** disponibili: il blocco /20 **non è sufficiente**. Si noti che già la sola Lan10, con 2340 indirizzi, richiede da sola un blocco da 4096 e dunque l'intero /20, senza lasciare spazio alle altre quattro LAN.

La conclusione corretta è quindi che **la maschera assegnata è sbagliata** e occorre richiedere un prefisso più corto: il blocco immediatamente superiore, **/19**, offre $2^{13} = 8192$ indirizzi ed è ampiamente sufficiente a ospitare i 5128 necessari.

Un'ultima osservazione di metodo: l'arrotondamento alla potenza di 2 superiore è **obbligatorio** e non una precauzione. Una sottorete non può avere una dimensione arbitraria, perché la maschera lavora sui bit: le uniche dimensioni possibili sono 2, 4, 8, 16, 32 e così via. Chiedere 501 indirizzi significa quindi consumarne inevitabilmente 512.

DHCP e NAT

Il capitolo descrive due meccanismi presenti in quasi ogni rete locale: DHCP, che assegna automaticamente la configurazione di rete agli host, e NAT, che permette a un'intera rete privata di uscire su Internet con un solo indirizzo pubblico.

34.1 Il protocollo DHCPv4

Il **DHCP** (*Dynamic Host Configuration Protocol*) consente a un host di ottenere automaticamente le configurazioni di rete: un **indirizzo IP**, persistente o temporaneo, la **maschera di sottorete**, l'indirizzo del **gateway** e quello del **DNS server locale**.

Proprio quest'ultimo punto apre un problema di sicurezza che vale la pena segnalare: **chiunque potrebbe fingersi un server DHCP** e distribuire l'indirizzo di un DNS malevolo, indirizzando così tutte le comunicazioni a proprio piacimento. Ci si può proteggere configurando manualmente sul proprio dispositivo indirizzi di server DNS noti, come 8.8.8.8 e 8.8.4.4.

DHCP è un protocollo **client/server** che utilizza **UDP**: il **client** usa la **porta 68** ed è l'host appena connesso che necessita della configurazione; il **server** usa la **porta 67** e può essere un server DHCP vero e proprio oppure un **agente di relay**, generalmente implementato in un router, che conosca l'indirizzo di un server DHCP.

Il protocollo si articola in **quattro fasi**, che vale la pena seguire osservando gli indirizzi impiegati.

- | | |
|---------------------|------------------------------|
| 1. DHCP DISCOVER | client → broadcast |
| SRC: 0.0.0.0 | :68 DST: 255.255.255.255 :67 |
| 2. DHCP OFFER | server → broadcast |
| SRC: dhcp_server_ip | :67 DST: 255.255.255.255 :68 |
| 3. DHCP REQUEST | client → broadcast |
| SRC: 0.0.0.0 | :68 DST: 255.255.255.255 :67 |
| 4. DHCP ACK | server → broadcast |
| SRC: dhcp_server_ip | :67 DST: 255.255.255.255 :68 |

Le quattro fasi del protocollo DHCPv4

Nella prima fase l'host, appena collegato, invia un **DHCP Discover** per individuare il server. Nella seconda il server risponde con un **DHCP Offer**, inviato in broadcast a tutti i nodi della sottorete perché in una sottorete possono esservi **più server DHCP**; il messaggio contiene un indirizzo IP proposto, la maschera e la durata della concessione (*lease time*). Nella terza l'host **sceglie fra le varie offerte** e risponde con un **DHCP Request** che conferma la configurazione richiesta. Nella quarta il server risponde con un **DHCP ACK** che conferma i parametri, e da quel momento l'host può usare l'indirizzo assegnato.

Si noti che il client invia con indirizzo sorgente **0.0.0.0** perché non ne possiede ancora uno, e in **broadcast** perché non sa dove si trovi il server. Il **Transaction ID (XID)** serve a identificare univocamente una singola transazione, permettendo al client di associare le risposte ai propri messaggi ed evitando confusione quando più server DHCP sono presenti in rete o quando il client effettua più richieste in momenti diversi.

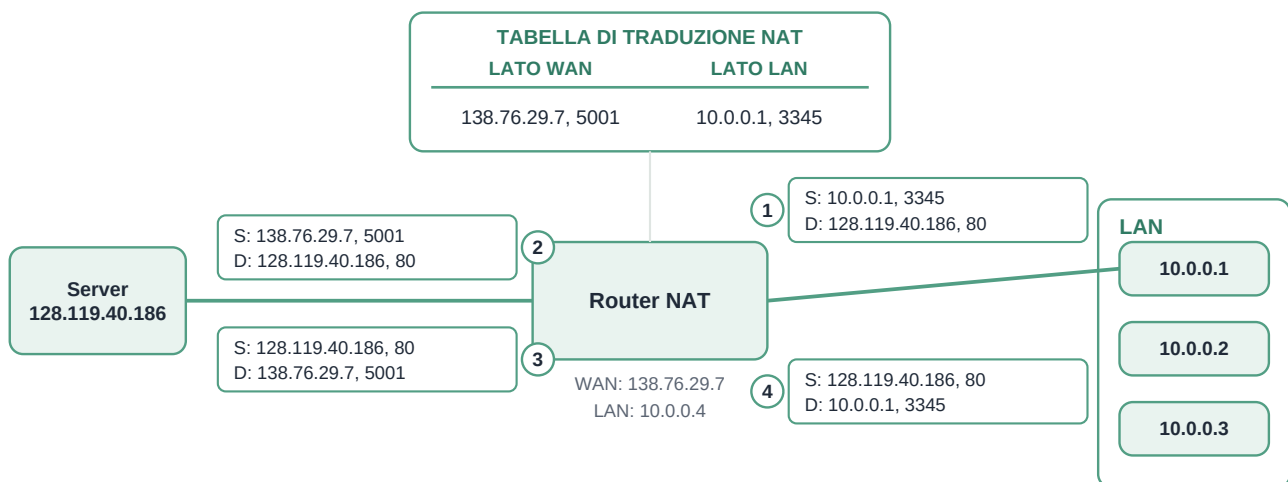
DHCP è adatto a situazioni con **molti host che vanno e vengono**, in cui gli indirizzi servono solo per un tempo limitato: reti wireless e LAN. Presenta però problemi di **mobilità**: quando un nodo si sposta fra sottoreti, DHCP gli rilascia un nuovo indirizzo IP, **interrompendo le connessioni TCP** in corso, che erano identificate anche dal

vecchio indirizzo.

34.2 NAT

Tutti i dispositivi di una rete locale hanno indirizzi in uno **spazio privato**, utilizzabile solo localmente: esistono infatti moltissime reti private che usano lo stesso spazio di indirizzamento, mentre su Internet gli indirizzi devono essere **univoci**. Con il **NAT** (*Network Address Translation*) tutti i dispositivi di una LAN si interfacciano a Internet con **un unico indirizzo IP**.

I vantaggi sono tre: serve **un solo indirizzo IP** dall'ISP per collegare tutti i dispositivi; si possono **cambiare gli indirizzi interni senza avvisare il mondo esterno**; e si ottiene un beneficio di **sicurezza**, poiché i dettagli della rete interna non sono visibili dall'esterno.



La tabella di traduzione NAT

Il funzionamento si basa su una **tabella di traduzione**. Quando il router riceve un pacchetto diretto all'esterno, il NAT controlla se la coppia IP/porta sia già presente in tabella; se non lo è, inserisce un nuovo record che associa il **lato LAN**, cioè IP e porta dell'interfaccia effettiva, al **lato WAN**, cioè l'IP dell'interfaccia del router verso Internet e una **porta generata casualmente** fra quelle non ancora usate. Grazie a questo meccanismo il NAT può gestire **ben oltre 60 000 connessioni simultanee** con un solo indirizzo IP lato WAN.

Operativamente, un router abilitato al NAT deve sostituire, nei **datagrammi in uscita**, la coppia (indirizzo IP di origine, porta di origine) con (indirizzo IP NAT, nuova porta); i server remoti risponderanno quindi verso (indirizzo IP NAT, nuova porta), e nei **datagrammi in entrata** il router eseguirà la sostituzione inversa, ripristinando la coppia originale memorizzata in tabella.

Di default il NAT consente **solo connessioni in uscita**, dal client verso il server. Se un server esterno volesse iniziare una connessione verso un dispositivo interno, il router **non saprebbe a chi indirizzarla** e la bloccherebbe, a meno che non sia stata registrata in precedenza una comunicazione in uscita. Le due soluzioni sono il **Port Forwarding**, cioè una configurazione manuale nel router, e l'**UPnP** (*Universal Plug and Play*), un sistema automatico che permette ai dispositivi interni di chiedere dinamicamente al router di aprire temporaneamente una porta.

Va infine detto che il NAT **viola alcuni principi architetturali**: i router dovrebbero elaborare solo fino al livello 3, mentre il NAT manipola le porte, che appartengono al livello 4; la carenza di indirizzi avrebbe dovuto essere risolta da IPv6 anziché da un espediente; e viene violato il **principio end to end**, poiché gli host non sono più direttamente raggiungibili. Ciò nondimeno il NAT è **ampiamente utilizzato** nelle reti domestiche, istituzionali e nelle reti cellulari 4G e 5G.

ICMP

Il protocollo **ICMP** (*Internet Control Message Protocol*) permette ai dispositivi di **comunicare problemi di connessione** o altri malfunzionamenti che si verifichino durante lo scambio di dati. Funziona **sopra il protocollo IP** ed esiste sia per IPv4 sia per IPv6; in quest'ultimo trova un'applicazione particolarmente importante nei messaggi del *Neighbour Discovery Protocol*.

Le sue funzioni sono tre. La **segnalazione di errori**: invia messaggi al mittente originale quando un pacchetto non può essere consegnato, per esempio perché la destinazione è irraggiungibile o perché il TTL è scaduto. La **diagnostica di rete**: **ping** invia messaggi ICMP di tipo 8 per testare la connettività fra due host, mentre **traceroute** determina il percorso fra due host identificando nomi e indirizzi dei router attraversati. E infine le **informazioni sullo stato della rete**, come la notifica di congestione o l'indisponibilità di servizi.

Quando un dispositivo incontra un problema con un pacchetto IP, genera un messaggio ICMP che viene a sua volta **incapsulato in un pacchetto IP** e inviato al mittente originale. I messaggi ICMP contengono un campo **tipo** di 8 bit, che indica la categoria del messaggio, un campo **codice** di 8 bit con una descrizione ulteriore, un **checksum** di 16 bit e (dettaglio importante) **l'header e i primi 8 byte del datagramma che ha generato il messaggio**, così da consentire al mittente di identificare esattamente quale pacchetto abbia causato l'errore.

Il protocollo IPv6

IPv6 nasce principalmente perché lo **spazio di indirizzamento IPv4 a 32 bit stava esaurendosi**, a causa della crescente connessione di nuove sottoreti e nodi: nel **2011** la IANA allocò l'ultimo blocco di indirizzi IPv4 rimasto. Oltre a questo, IPv6 introduce nuovi servizi: **sicurezza**, **configurazione automatica** (*plug & play*), gestione della **qualità del servizio**, indirizzamento **multicast** e indirizzamento per **host mobili**.

36.1 Il datagramma IPv6

Il formato presenta quattro modifiche significative rispetto a IPv4. L'**indirizzamento esteso** porta gli indirizzi da 32 a **128 bit**, fornendo una quantità praticamente illimitata di indirizzi. Sono supportati indirizzi **unicast**, **multicast** e **anycast**, questi ultimi non presenti in IPv4. L'**intestazione è ottimizzata**: fissa a **40 byte** e contenente solo i campi essenziali, con le opzioni rimosse dall'header standard e trasformate in intestazioni successive. Infine viene introdotta l'**etichettatura dei flussi**, che consente di dare priorità e offrire garanzie di servizio migliori.

I campi principali sono: **Versione** (4 bit); **Classe di Traffico** (8 bit), analogo al ToS di IPv4 e usato per attribuire priorità; **Etichetta di Flusso** (20 bit), per identificare un flusso di datagrammi; **Lunghezza del Payload** (16 bit), che limita i dati trasportabili a $2^{16} - 1 = 65\,535$ byte; **Intestazione Successiva** (*Next Header*); **Limite di Hop** (8 bit), decrementato da ogni router e che provoca l'eliminazione del datagramma quando raggiunge 0; gli **indirizzi sorgente e destinazione**; e i **dati**.

CONFRONTO CON IPV4

Sapere **che cosa è stato rimosso e perché** è il modo migliore per ricordare le differenze.

CAMPO IPV4	SORTE IN IPV6	MOTIVO
ID, Flags, Offset	Rimossi	Servivano alla frammentazione, non più effettuata dai router.
ToS	Sostituito	Ne prende il posto la Classe di Traffico , con la stessa funzione.
HLEN	Rimosso	L'header ha dimensione fissa di 40 byte, quindi indicarla è superfluo.
Checksum	Rimosso	TCP e UDP al livello di trasporto ed Ethernet al livello di collegamento implementano già un controllo: era una funzionalità ridondante .
Options	Trasformato	Non del tutto rimosso: diventa una delle possibili intestazioni successive .
Total length	Modificato	Diventa Lunghezza del payload .
Protocol	Modificato	Diventa Header successivo .
TTL	Modificato	Diventa Hop Limit , nome che ne descrive correttamente la natura di contatore di salti.

36.2 Gli header opzionali

DA RICORDARE

Sugli **header opzionali in IPv6** i punti essenziali sono tre: **dove** si collocano, **come** sono concatenati e **a che cosa** servono concretamente: in particolare frammentazione e sicurezza.

Gli **header opzionali** (o *extension header*) sono parti aggiuntive dell'intestazione di un pacchetto IPv6 che forniscono informazioni supplementari o facoltative. Vengono inseriti **fra l'header principale IPv6 e il payload**.

Il meccanismo con cui sono organizzati è elegante e va compreso: **funzionano come una lista concatenata**. Il campo **Next Header** dell'intestazione principale specifica il tipo del prossimo header, che può essere l'header del protocollo di trasporto oppure un header aggiuntivo; ognuno degli header aggiuntivi contiene a sua volta un proprio campo *Next Header*, cosicché su un datagramma è possibile costruire una vera e propria **catena di header**. In questa catena l'**ultimo** header aggiuntivo contiene il tipo dell'header del livello di trasporto, chiudendo la sequenza.

Il vantaggio rispetto al campo *opzioni* di IPv4 è sostanziale: l'header principale resta **fisso e compatto** (40 byte sempre), così i router lo elaborano rapidamente, e le funzionalità aggiuntive vengono trasportate solo quando servono e possono essere ignorate dai nodi intermedi che non le riguardano.

Fra gli usi concreti, due meritano di essere citati. Per la **frammentazione** si utilizza un **Fragment Header**. Per la **sicurezza** si impiegano l'**Authentication Header**, che garantisce autenticità e correttezza del pacchetto, e l'**Encapsulating Security Payload** (ESP), che cifra il payload. Sono gli header della suite **IPsec**, progettata insieme a IPv6 e integrata nel protocollo attraverso il meccanismo delle intestazioni successive.

36.3 Indirizzi IPv6

Un indirizzo IPv6 è un valore a **128 bit**, rappresentato da **otto gruppi di quattro cifre esadecimali** separati dai due punti.

Gli indirizzi **Global Unicast** identificano una singola interfaccia e si compongono di quattro parti: i primi 3 bit (001), che indicano il tipo di indirizzo; il **Global Routing Prefix**, che identifica il sito; il **Subnet ID**, gestito all'interno del sito; e l'**Interface ID**, di 64 bit, che identifica l'interfaccia. Gli indirizzi **link local** appartengono al blocco **FE80::/10**, sono utilizzabili solo fra nodi della stessa LAN e non sono instradabili. Gli indirizzi **site local** (FEC0::/10), pensati per l'uso interno a un'organizzazione, sono stati deprecati e sostituiti dagli indirizzi **unique local** del blocco **FC00::/7**, anch'essi non instradabili su Internet.

Per distinguere rapidamente le tre modalità di indirizzamento è utile ricordare i prefissi: gli indirizzi **global unicast** iniziano per **2xxx** o **3xxx** e i **multicast** per **FFxx**. Gli **anycast** invece non hanno un prefisso proprio: sono indirizzi unicast assegnati a più interfacce, e un caso particolare è l'indirizzo anycast del router di sottorete, che ha l'identificatore di interfaccia tutto a zero.

EUI 64

L'**EUI 64** (*Extended Unique Identifier 64*) è il metodo con cui IPv6 genera **automaticamente** l'identificatore di interfaccia, derivandolo dall'indirizzo MAC a 48 bit della scheda di rete. Il procedimento è in tre passi: l'indirizzo MAC viene esteso a 64 bit **inserendo il valore 0xFFFE al centro**; il **settimo bit** dell'indirizzo MAC viene **invertito**; i 64 bit così ottenuti diventano l'*interface ID* dell'indirizzo IPv6. Combinato con il prefisso FE80 forma l'indirizzo link local, valido soltanto sulla LAN; combinato con un prefisso globale annunciato dal router forma un indirizzo instradabile su Internet.

Va segnalato che **Windows non utilizza EUI 64**: calcola l'indirizzo IPv6 a partire da un **numero casuale**, e infatti nei suoi indirizzi il gruppo FFFE non compare nella parte centrale. La ragione è di privacy: derivare l'indirizzo dal MAC renderebbe il dispositivo tracciabile fra reti diverse.

36.4 Frammentazione in IPv6 e transizione da IPv4

Mentre in IPv4 **qualsiasi nodo** può frammentare un pacchetto, in IPv6 la frammentazione è **sconsigliata**. L'MTU minimo è di **1280 byte**, con 1500 consigliati. Quando un router riceve un datagramma IPv6 troppo grande **lo cestina** e rimanda al mittente un errore ICMP. Se la frammentazione è comunque necessaria, può effettuarla **soltanto il mittente**, utilizzando il *Fragment Header*, mentre il riassemblaggio spetta al destinatario.

Quanto alla **transizione**, tutti i sistemi operativi moderni implementano il **doppio stack**, IPv4 e IPv6. I nuovi sistemi IPv6 sono retrocompatibili, ma i sistemi IPv4 esistenti **non sono in grado di gestire datagrammi IPv6**. Per ovviare si utilizza il **tunneling**: se i due estremi della comunicazione supportano IPv6 ma una porzione della rete intermedia no, **l'intero datagramma IPv6 viene incapsulato in un datagramma IPv4** prima di attraversare quella porzione.

Il meccanismo funziona così: il nodo all'ingresso del tunnel prende il datagramma IPv6 e lo colloca nel **campo dati** di un datagramma IPv4; i router intermedi lo trattano come un qualsiasi datagramma IPv4, **ignari** che il payload contenga un datagramma IPv6; il nodo all'uscita del tunnel riceve il datagramma IPv4, determina che ne contiene uno IPv6 e lo **estrae**, proseguendo normalmente.

36.5 NDP e DHCPv6

L'**NDP** (*Neighbour Discovery Protocol*) è il protocollo con cui IPv6 svolge le funzioni che in IPv4 erano distribuite fra ARP e altri meccanismi. Serve ad **autoconfigurare** gli indirizzi dei nodi, **individuare altri nodi** sulla rete (i vicini), **individuare un router**, **rilevare l'irraggiungibilità** di un vicino, **rilevare indirizzi duplicati** e trovare DNS operativi.

Il rapporto con ARP merita una precisazione: ARP era stato inventato per collegare IPv4 con Ethernet ma **non faceva parte di ICMPv4**, mentre **IPv6 incorpora ARP dentro ICMPv6**. NDP definisce infatti **cinque tipi di messaggi ICMPv6**: **133 (Router Solicitation)**, con cui l'host chiede quali router siano presenti; **134 (Router Advertisement)**, con cui i router annunciano la propria presenza, periodicamente o su richiesta; **135 (Neighbor Solicitation)**, inviato in multicast per scoprire l'indirizzo di un altro host o in unicast per verificarne la raggiungibilità; **136 (Neighbor Advertisement)**, risposta al precedente; **137 (Redirect)**, con cui un router informa dell'esistenza di un instradamento migliore.

Anche **DHCPv6** presenta differenze rispetto a DHCPv4, dovute alla natura stessa di IPv6. La sequenza è di quattro messaggi: **SOLICIT**, con cui il client cerca i server utilizzando il **multicast** anziché il broadcast, soluzione più efficiente nelle grandi reti; **ADVERTISE**, con cui il server segnala la propria disponibilità; **REQUEST**, con cui il client richiede indirizzo e informazioni; **REPLY**, con cui il server risponde. Conviene ricordare la distinzione: il **broadcast** invia a **tutti** i dispositivi della rete, il **multicast** soltanto a un **gruppo specifico** di destinatari.

Firewall e DMZ

Un **firewall** è un sistema di sicurezza progettato per **monitorare e controllare il traffico** in entrata e in uscita, sulla base di regole predefinite. Il suo compito è bloccare o filtrare il traffico malevolo in base ai valori dei campi degli header, oppure effettuare un reindirizzamento per ulteriore elaborazione. Lo scopo principale è **creare una barriera** fra una rete interna sicura e reti esterne non sicure, ma è bene ricordare che esistono casi in cui l'attacco proviene **dall'interno**, da un computer già compromesso.

I firewall possono operare a livello di **rete, di trasporto e applicativo**, a seconda delle esigenze. Si distinguono in **software**, presenti in tutti i sistemi operativi e basati su un insieme di regole (per i quali vale il principio fondamentale di **negare tutto e consentire solo ciò che serve**) e **hardware**, dispositivi dedicati al filtraggio, più difficili da violare ma più costosi.

Le funzioni principali sono tre: **bloccare gli accessi non autorizzati dall'esterno, consentire solo il traffico legittimo** verso e dalla rete interna e **impedire che malware o utenti compromessi all'interno comunichino verso l'esterno**. Le regole di filtraggio si basano su indirizzi IP e porte di origine e destinazione, sui protocolli (TCP, UDP, ICMP), sullo **stato della connessione** e, nei firewall a ispezione profonda, sul **contenuto del pacchetto**.

La **DMZ (Demilitarized Zone)** è una zona della rete che si trova, dal punto di vista della sicurezza, in **posizione intermedia** fra rete interna ed esterna. Vi si collocano i server che devono essere accessibili dall'esterno (server web, server di posta) tenendoli però **separati dalla rete interna**: in caso di attacco a un server della DMZ, l'attaccante **non avrebbe accesso diretto** alla rete interna.

È in questo contesto che si distinguono le tre azioni possibili di un firewall: **Allow**, far passare il pacchetto; **Deny**, bloccarlo notificando il rifiuto al mittente; **Drop**, bloccarlo **senza notificare** il mittente, il che permette anche di risparmiare banda. Un firewall può inoltre implementare il **NAT**, mascherando gli indirizzi interni verso la rete esterna.

Algoritmi di routing

Per formulare i problemi di instradamento si utilizza un **grafo** $G = (N, E)$ in cui i nodi rappresentano i router e gli archi i collegamenti, con un **costo** associato a ciascun arco. L'obiettivo di un algoritmo di routing è trovare il **percorso di costo minimo** fra sorgente e destinazione.

Gli algoritmi si classificano secondo **tre criteri**, e conviene tenerli distinti perché rispondono a domande diverse.

Il **primo criterio** riguarda *dove* avviene il calcolo. Gli algoritmi **centralizzati** calcolano i percorsi minimi con una **conoscenza globale** della rete, ricevendo in ingresso tutti i collegamenti e i loro costi: sono detti **link state (LS)**. Gli algoritmi **decentralizzati** calcolano il percorso in modo **distribuito e iterativo**, con ogni nodo che elabora informazioni solo sui propri collegamenti diretti; nessun nodo ha conoscenza globale, e i router si scambiano informazioni con i vicini. È il caso del **distance vector**.

Il **secondo criterio** riguarda *quando* i percorsi cambiano. Gli algoritmi **statici** cambiano percorso raramente e richiedono un intervento umano; quelli **dinamici** determinano gli instradamenti al variare del traffico o della topologia.

Il **terzo criterio** riguarda *che cosa* determina i costi. Gli algoritmi **sensibili al carico** hanno costi che variano con la congestione attuale: erano i primi a essere impiegati, rispondono meglio ai cambiamenti ma sono molto soggetti a **instradamenti in loop e oscillazione dei percorsi**. Gli algoritmi **insensibili al carico** hanno costi che non riflettono la congestione, ed è l'approccio adottato oggi, con una motivazione precisa: la congestione potrebbe **non durare abbastanza a lungo** perché la scelta di un cammino alternativo resti valida.

38.1 Flooding

Il **flooding** («inondazione») è una tecnica che inonda la rete di duplicati: appena un nodo riceve un pacchetto lo **duplica su tutti i collegamenti eccetto quello da cui l'ha ricevuto**. Viene usato per studiare la topologia della rete e condividere fra i nodi le informazioni sui costi.

I **vantaggi** sono due. Trova un percorso verso la destinazione **se questo esiste**, con certezza. E trova il **percorso migliore**: benché il messaggio venga trasmesso a tutti, è possibile che uno dei primi nodi a riceverlo sia proprio il destinatario, e quindi il percorso più breve può essere scoperto rapidamente.

Lo **svantaggio** è però decisivo: genera un **numero elevatissimo di pacchetti duplicati**, inviati anche a nodi che li hanno già ricevuti, e questo può **saturare la rete molto rapidamente**. Occorre dunque controllare il flooding, e a tal fine esistono quattro tecniche.

- **Usare un ID per i pacchetti**: ogni pacchetto porta un identificatore univoco, così i nodi riconoscono e scartano i duplicati. Il costo è che ogni nodo deve mantenere una **lista** e verificarvi ogni pacchetto in arrivo.
- **Usare un contatore di salti**: si blocca il pacchetto quando il contatore arriva a zero. Presenta due problemi: non è detto che si raggiunga la destinazione, e se il numero di salti è troppo grande non si risolve nulla.
- **Eliminare i cicli dal grafo**, che sono la causa per cui i pacchetti tornano indietro.
- **Trasformare il grafo in un albero**: in un albero esiste **un'unica strada** fra due nodi qualsiasi, e quindi i duplicati sono impossibili per costruzione. Si utilizza a tal fine lo **Spanning Tree Protocol (STP)**.

Distance Vector

Il capitolo descrive l'algoritmo Distance Vector, basato sull'equazione di Bellman Ford, in cui ogni router calcola i cammini minimi scambiando soltanto informazioni con i propri vicini, e i problemi di convergenza che ne derivano.

DA RICORDARE

Oltre al meccanismo di base dell'algoritmo, del Distance Vector vanno compresi il **count to infinity** e la **poisoned reverse**.

Il Distance Vector è un approccio con cinque caratteristiche. È **decentralizzato**; è **iterativo**, perché il calcolo procede per scambi successivi finché le stime non smettono di cambiare; è **distribuito**, perché ciascun nodo riceve informazioni dai vicini direttamente connessi; è **asincrono**, perché non è necessario che i nodi operino in modo sincronizzato e ciascuno può aggiornarsi indipendentemente; è **autoterminante**, perché non richiede alcun segnale per interrompere i calcoli ma si arresta quando non avvengono più scambi fra i vicini. Si basa sull'algoritmo di **Bellman Ford** ed è utilizzato come fondamento di protocolli reali quali **RIP** e **BGP**.

Il funzionamento generale è il seguente: si parte da un grafo pesato; ogni nodo crea una **tabella** in cui le colonne sono le proprie uscite e le righe le destinazioni, e ciascuna cella indica il costo noto per raggiungere quella destinazione da quell'uscita. Inizialmente ogni nodo conosce solo i costi verso i **vicini diretti** e mantiene un **vettore delle distanze** per tutte le destinazioni; i nodi si scambiano poi i propri vettori e li aggiornano con la formula di Bellman Ford.

39.1 L'algoritmo

Introduciamo la notazione: D_x è il vettore delle distanze del nodo x , e $D_x[v]$ è la distanza stimata da x a v . Su ciascun nodo x del grafo gira il seguente algoritmo.

INIZIALIZZAZIONE

1. $D_x[x] = 0$ il nodo x dista 0 da se stesso
2. per ogni v : $D_x[v] = w(x,v)$ costo dell'arco diretto
 se v non è adiacente a x : $D_x[v] = +\infty$
3. inoltra D_x a tutti i nodi adiacenti

ITERAZIONE

1. attendi finché non ricevi un vettore delle distanze, oppure finché non cambia il costo verso un nodo adiacente
2. ricevuto D_y da un vicino y , per ogni v calcola:

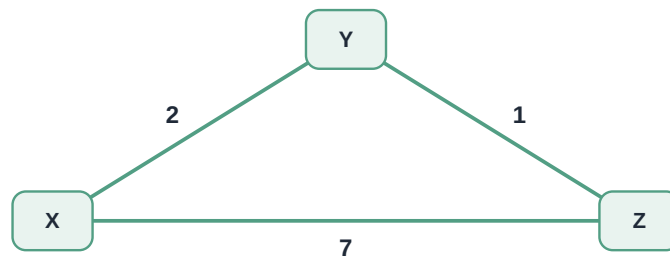
$$D_x[v] = \min \{ D_x[v], D_y[v] + w(x,y) \}$$

se qualche $D_x[v]$ cambia, inoltra D_x a tutti i nodi adiacenti

3. torna al passo 1

L'algoritmo Distance Vector

Il cuore è la riga centrale, ed è utile leggerla a parole: *la mia stima verso v è il minimo fra quella che avevo già e il costo per arrivare al vicino y sommato alla stima che y dichiara di avere verso v* . Se questa condizione migliora la stima, il vettore aggiornato viene inoltrato ai vicini, che a loro volta ricalcoleranno il proprio.



Esempio di calcolo del vettore delle distanze

Nella pratica, dalle matrici di ciascun nodo si ricava il vettore delle distanze **prendendo sempre il valore minimo** di ogni riga. Quando un nodo rileva un cambiamento nel costo di un collegamento con un vicino aggiorna il proprio vettore, e la modifica si propaga: i nodi che se ne accorgono ricalcolano il proprio vettore e lo inviano agli adiacenti.

39.2 Count to infinity e poisoned reverse

C'è un'asimmetria fondamentale nel comportamento del Distance Vector, ed è quella che genera il suo problema più noto: **i miglioramenti si propagano rapidamente, i peggioramenti molto lentamente**. Quando un collegamento migliora, la buona notizia si diffonde in poche iterazioni; quando invece un collegamento si guasta, i nodi continuano a ingannarsi a vicenda incrementando le stime di uno alla volta, in un processo lentissimo chiamato **count to infinity** (*infinity counting*).

Il motivo di fondo è che i valori nel vettore delle distanze **possono ingannare**: sono valori esatti, ma un nodo non sa **attraverso quale percorso** il vicino ottenga quella distanza, e non si accorge quindi che il percorso dichiarato passa proprio per sé stesso. Le soluzioni possibili sono resettare tutto oppure utilizzare la tecnica della **poisoned reverse**.

La **poisoned reverse** permette a un nodo di «mentire» deliberatamente per rompere un loop. La circostanza problematica si presenta così: y vuole inoltrare un pacchetto a z per raggiungere x, perché così costa meno; ma contemporaneamente z vuole inoltrarlo a y per lo stesso motivo. Si arriva a questa situazione quando varia il peso di un arco non adiacente a z ma adiacente a y. In tal caso z osserva che y vuole raggiungere x passando per lui, mentre lui stesso farebbe l'opposto: allora **mente a y dichiarando un costo fittizio $D_z[x] = +\infty$** , interrompendo così il loop.

La regola generale è semplice: se z in strada verso x passando per y, dichiara a y una distanza infinita verso x. In questo modo y non sceglierà mai di raggiungere x passando per z, e il ciclo fra i due nodi non può formarsi. La tecnica elimina i cicli fra due nodi adiacenti, ma non quelli che coinvolgono tre o più nodi.

Link State Routing e algoritmo di Dijkstra

Il capitolo descrive il Link State Routing, in cui ogni router conosce l'intera topologia della rete e calcola i cammini minimi con l'algoritmo di Dijkstra, svolto passo per passo su un esempio.

DA RICORDARE

Del Link State Routing va saputo svolgere a mano l'**algoritmo di Dijkstra** su una topologia data: il paragrafo 40.2 ne mostra un esempio completo.

La prima versione di ARPANET utilizzava il Distance Vector, sostituito **dopo il 1979** con il Link State Routing, perché il primo **non teneva conto della capacità delle linee** ma soltanto dei ritardi, e convergeva troppo lentamente a causa del count to infinity.

Con questo algoritmo ogni router **indaga sui propri vicini** e ne determina i costi di comunicazione, quindi invia pacchetti sullo stato dei propri collegamenti affinché ogni router possa calcolare i percorsi ottimali. Il risultato è che **la topologia di rete è nota a tutti e i costi dei collegamenti sono noti a tutti**. La diffusione avviene tramite il **link state broadcast**, cioè l'invio di messaggi detti **link state advertisement**, distribuiti nella rete tramite **flooding**; per evitare di inondarla si utilizza un **ID diverso per ogni pacchetto**, e l'operazione viene ripetuta periodicamente.

Un vantaggio che vale la pena sottolineare, perché distingue nettamente LS da DV: **qui si scopre se uno dei vicini sta mentendo**. Se il nodo D manda un pacchetto ad A e A risponde, entrambi comunicano il peso del collegamento; se i due valori dichiarati sono diversi, allora qualcuno sta mentendo. Nel Distance Vector, dove ciascuno riceve soltanto stime aggregate, un controllo del genere è impossibile.

Il calcolo dei percorsi a costo minimo si basa sull'**algoritmo di Dijkstra**.

40.1 L'algoritmo di Dijkstra

Fissiamo prima le convenzioni. **G** è il grafo, **G.V** l'insieme dei nodi e **G.E** quello degli archi; **w** è la funzione che stabilisce il costo di un arco fra due nodi adiacenti; **d.v** indica il costo minimo **stimato** dal nodo sorgente nell'iterazione corrente, che parte da $+\infty$ tranne per la sorgente per cui $d.s = 0$; **c(s,v)** indica l'**effettivo** costo del cammino minimo da s a v, e l'obiettivo è ottenere $d.v = c(s,v)$ per ogni nodo; **p.v** è il **predecessore** di v lungo il cammino di costo minimo.

INIZIALIZZAZIONE

1. $d.v = +\infty$ per ogni nodo, tranne $d.s = 0$ per la sorgente
2. $p.v = \text{null}$ per ogni nodo
3. $S = \text{insieme vuoto}$ (nodi di cui si conosce il cammino minimo)
4. $Q = \text{insieme vuoto}$ (coda con priorità)
5. inserisci tutti i nodi in Q

ITERAZIONE

1. estrai da Q il nodo u con $d.u$ minimo
(alla prima iterazione sarà la sorgente s , poiché $d.s = 0$)
2. inserisci u nell'insieme S
3. per ogni nodo v adiacente a u :

$$d.v = \min \{ d.v, d.u + c(u,v) \}$$

se $d.v$ si riduce passando da u , aggiorna Q e poni $p.v = u$

4. se $S = G.V$ termina, altrimenti torna al passo 1

L'algoritmo di Dijkstra

Quando l'algoritmo termina, per ogni nodo si conosce il **predecessore** lungo il percorso di costo minimo dalla sorgente; risalendo la catena dei predecessori si ricostruisce l'**intero percorso** dall'origine a tutte le destinazioni.

Un'osservazione sulla complessità, che dipende da come si implementa la struttura **Q**: su cui si eseguono inserimenti, estrazioni del minimo e aggiornamenti delle chiavi. Con Q implementata come **array semplice** si ottiene una complessità $O(|V|^2)$; con Q implementata come **min heap** binario si ottiene $O((|V| + |E|) \log |V|)$.

40.2 Dijkstra: esempio svolto

Consideriamo un grafo con cinque nodi e i seguenti archi con i relativi costi: $(A, B) = 2$, $(A, C) = 5$, $(B, C) = 1$, $(B, D) = 4$, $(C, D) = 1$, $(C, E) = 8$, $(D, E) = 3$. Calcoliamo i cammini minimi a partire da **A**.

Inizializzazione: $d.A = 0$ $d.B = \text{inf}$ $d.C = \text{inf}$ $d.D = \text{inf}$ $d.E = \text{inf}$

Iter. 1 estraggo A ($d=0$) $S = \{A\}$
B: $\min(\text{inf}, 0+2) = 2$ $p.B = A$
C: $\min(\text{inf}, 0+5) = 5$ $p.C = A$
→ $d.A=0$ $d.B=2$ $d.C=5$ $d.D=\text{inf}$ $d.E=\text{inf}$

Iter. 2 estraggo B ($d=2$) $S = \{A,B\}$
C: $\min(5, 2+1) = 3$ $p.C = B$ (migliorato!)
D: $\min(\text{inf}, 2+4) = 6$ $p.D = B$
→ $d.A=0$ $d.B=2$ $d.C=3$ $d.D=6$ $d.E=\text{inf}$

Iter. 3 estraggo C ($d=3$) $S = \{A,B,C\}$
D: $\min(6, 3+1) = 4$ $p.D = C$ (migliorato!)
E: $\min(\text{inf}, 3+8) = 11$ $p.E = C$
→ $d.A=0$ $d.B=2$ $d.C=3$ $d.D=4$ $d.E=11$

Iter. 4 estraggo D ($d=4$) $S = \{A,B,C,D\}$
E: $\min(11, 4+3) = 7$ $p.E = D$ (migliorato!)
→ $d.A=0$ $d.B=2$ $d.C=3$ $d.D=4$ $d.E=7$

Iter. 5 estraggo E ($d=7$) $S = \{A,B,C,D,E\} = G.V \rightarrow \text{FINE}$

Svolgimento di Dijkstra a partire dal nodo A

Risalendo i predecessori si ricostruiscono i cammini minimi: verso **B** il percorso è $A \rightarrow B$ con costo 2; verso **C** è $A \rightarrow B \rightarrow C$ con costo 3, migliore del collegamento diretto che costava 5; verso **D** è $A \rightarrow B \rightarrow C \rightarrow D$ con costo 4; verso **E** è $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ con costo 7, molto migliore dell'arco diretto (C, E) che avrebbe portato a 11.

Vale la pena notare due cose nello svolgimento. La prima: le stime **possono migliorare più volte** prima che il nodo venga estratto: d.C è passata da ∞ a 5 a 3; ma **non cambiano più dopo l'estrazione**, ed è questa la proprietà che rende corretto l'algoritmo. La seconda: il cammino minimo verso E **non usa affatto** l'arco diretto (C, E), benché esista; è un buon promemoria del fatto che il costo di un arco singolo non dice nulla sulla convenienza del percorso complessivo.

40.3 Il problema delle oscillazioni

Supponiamo di avere un grafo in cui i costi dei collegamenti sono **uguali al carico trasportato**. Il problema nasce dal fatto che ogni router aggiorna periodicamente i vicini sullo stato dei link, basandosi sulle condizioni correnti.

Quando il traffico viene instradato su un percorso congestionato, il costo di quel link **aumenta**, spingendo l'algoritmo a scegliere un percorso alternativo di costo inferiore; ma lo spostamento del traffico **congestiona il nuovo percorso**, facendone aumentare il costo e inducendo l'algoritmo a **tornare al percorso originale**. Il risultato è che a ogni esecuzione dell'algoritmo il percorso viene invertito, in un'oscillazione perpetua che non si stabilizza mai.

Le soluzioni sono due, di valore molto diverso. La prima consiste nel fare in modo che **i costi non rispecchino la quantità di traffico trasportato**, ma è considerata **inaccettabile** perché rinuncia proprio all'informazione che si voleva sfruttare. La seconda, praticabile, consiste nell'assicurarsi che **non tutti i router lancino l'algoritmo nello stesso istante**, sfasandone l'esecuzione così da smorzare l'oscillazione.

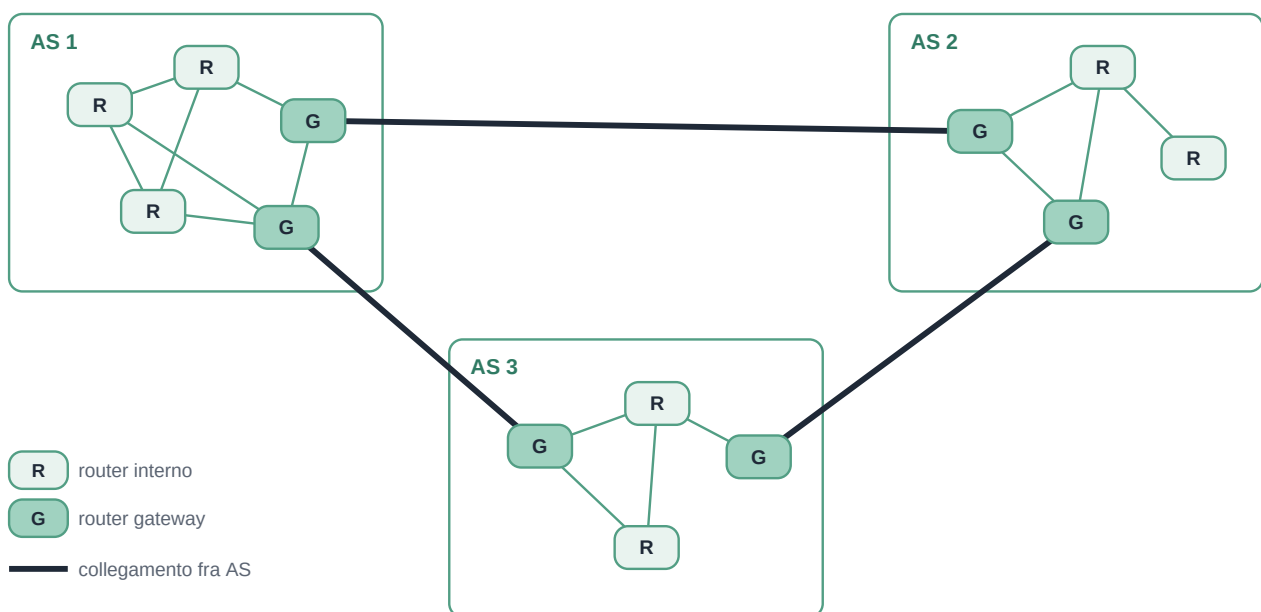
40.4 Distance Vector e Link State a confronto

ASPETTO	DISTANCE VECTOR	LINK STATE
Conoscenza della rete	Solo i vicini diretti; nessuna visione globale	Topologia e costi noti a tutti i nodi
Che cosa si scambia	Il proprio vettore delle distanze , ai soli vicini	Link state advertisement , diffusi in flooding a tutti
Algoritmo di base	Bellman Ford	Dijkstra
Tipo	Decentralizzato, distribuito, asincrono	Centralizzato nel calcolo (ogni nodo lo esegue su dati globali)
Convergenza	Rapida sui miglioramenti, lenta sui peggioramenti (count to infinity)	Più rapida e prevedibile
Rilevamento di errori/menzogne	Impossibile: si ricevono solo stime aggregate	Possibile: costi dichiarati da due nodi sullo stesso arco sono confrontabili
Traffico generato	Contenuto, solo verso i vicini	Elevato, per via del flooding
Problemi tipici	Count to infinity, loop di instradamento	Oscillazioni con costi sensibili al carico
Protocolli reali	RIP , BGP	OSPF

Internet routing: AS, RIP, OSPF e BGP

Gli algoritmi visti finora funzionano bene **in locale**, ma a livello nazionale e globale le cose vanno diversamente. Quando si passa da un provider a un altro, il routing interno di ciascuna delle due reti **interessa ben poco all'altra**: ciò che conta sono i **punti di aggancio** verso le altre reti.

Per questo Internet è organizzata in **sistemi autonomi (AS, Autonomous System)**, definiti come unità di politica di routing: una singola rete o un gruppo di reti controllato da un amministratore comune per conto di una singola entità amministrativa: un'università, un'impresa, una divisione aziendale. Ogni AS è identificato da un numero, l'**ASN**.



Organizzazione di Internet in sistemi autonomi

La regola di fondo è duplice. **All'interno di un AS** tutti i router eseguono lo stesso algoritmo, chiamato **IGP (Interior Gateway Protocol)**: per esempio **RIP**, che usa Distance Vector, oppure **OSPF**, che usa Link State. **Fra AS diversi** si utilizza invece un **EGP (Exterior Gateway Protocol)**, come **BGP**.

Una nota che vale la pena ricordare perché sorprende sempre: i protocolli inter AS **spesso non tengono conto del percorso migliore**, ma di altre informazioni, anche **di natura politica**. Si può decidere di far transitare le informazioni per un paese «amico» anziché per uno che potrebbe intercettarle. Questa organizzazione garantisce **autonomia amministrativa e migliore scalabilità**.

41.1 RIP (intra AS)

Il **RIP (Routing Information Protocol)** si utilizza per il routing interno ai sistemi autonomi, si basa sull'algoritmo **Distance Vector** e utilizza come metrica gli **hop**, cioè il numero di collegamenti attraversati: due nodi adiacenti sono a distanza 1, e così via. È **semplice da implementare** ma **non tiene conto dello stato dei link**, il che ne costituisce il limite principale.

RIPv1 supporta al più **15 hop**: una destinazione a 16 hop è considerata **irraggiungibile**. Il vincolo non è arbitrario: con reti troppo estese la convergenza risulterebbe troppo lenta, e il valore 16 funge da «infinito» pratico per limitare il count to infinity. I router si scambiano le tabelle **ogni 30 secondi**; se un percorso non viene aggiornato

entro **180 secondi** la sua distanza è posta a infinito e viene marcato irraggiungibile; trascorsi altri **120 secondi** (*garbage collection timer*) la rotta viene eliminata dalla tabella. In totale, dopo **300 secondi**, cioè cinque minuti, non è più considerato parte della rete.

RIP sfrutta due tipi di messaggi: **REQUEST**, per richiedere informazioni di routing ai router adiacenti, e **RESPONSE**, per fornire aggiornamenti sulla propria tabella. Una tabella RIP contiene l'**indirizzo di destinazione**, la **distanza in hop**, il **next hop**, il **timeout** e il *garbage collection timer*. Il datagramma RIPv1 ha lunghezza variabile fino a **512 byte**, può contenere al massimo **25 reti** di destinazione e **non trasporta informazioni sulla maschera**, essendo *classful*. Le sue limitazioni si riassumono così: supporta solo indirizzamento classful senza subnet, il campo next hop non è esplicito e non prevede autenticazione.

RIPv2 corregge tutti e tre i punti: supporta indirizzamento **CIDR e VLSM**, supporta l'**autenticazione** dei messaggi e rende **esplicito il next hop**. Entrambe le versioni adottano lo **split horizon** con **poisoned reverse** per contrastare i loop. Esiste infine **RIPng**, basata su RIPv2 ed esclusiva per IPv6, poco utilizzata e priva di autenticazione.

41.2 OSPF (intra AS)

L'**OSPF** (*Open Shortest Path First*) si utilizza anch'esso per il routing interno ai sistemi autonomi, ma adotta l'instradamento **Link State**. È stato **progettato per sostituire RIP** e funziona bene anche su reti di dimensioni notevoli e nei backbone degli ISP. Rispetto a RIP offre tre miglioramenti concreti: i **messaggi sono autenticati**, consente l'uso di **diverse metriche** e non soffre del limite dei 15 hop.

Nello schema generale del Link State ogni nodo misura il costo verso i vicini, per esempio inviando un messaggio e misurando il tempo di risposta. In OSPF i costi dei collegamenti sono invece **configurati dall'amministratore**, tipicamente in modo inversamente proporzionale alla **capacità** del link. Ogni nodo prepara poi una tabella con i propri vicini e i relativi costi, e la diffonde in **flooding** a tutti, usando un **ID** per evitare che la stessa tabella ritorni sugli stessi nodi intasando la rete.

Ogni router OSPF **costruisce una mappa topologica completa** del sistema autonomo, cioè un grafo della rete; **esegue localmente Dijkstra** determinando l'albero dei cammini minimi da sé stesso (di cui è la radice) verso tutte le sottoreti; e mantiene un **Link State Database** costituito da *link state record*. Il database viene aggiornato tramite **Link State Advertisement** in quattro circostanze: quando un router incontra un nuovo vicino, quando un router o un collegamento si guasta, quando il costo di un collegamento cambia e **periodicamente**, in genere ogni 30 minuti.

Il protocollo si compone di **tre parti**: **HELLO**, per il rilevamento e la verifica dei nodi adiacenti; **EXCHANGE**, per la sincronizzazione iniziale del database mediante scambio e confronto delle informazioni; **FLOODING**, per l'aggiornamento del database, con cui i cambiamenti dei cammini minimi vengono propagati. I datagrammi corrispondenti sono **HELLO** per il *neighbor discovery*, **Database Description** per condividere le informazioni link state, **LS Request** per richiedere parti specifiche del database di un vicino, **LS Update** per effettuare il flooding e **LS ACK** per confermare gli aggiornamenti.

Resta un problema: i nodi potrebbero **non avere informazioni aggiornate** quando calcolano i cammini minimi. In assenza di errori gravi ciò causerebbe soltanto qualche salto in più; in casi eccezionali un messaggio potrebbe girare «all'infinito», ma il **TTL** garantisce che venga scartato prima di creare problemi seri.

41.3 BGP (inter AS)

Il **BGP** (*Border Gateway Protocol*) è il protocollo di routing **fra sistemi autonomi**. È fondamentale per il funzionamento di Internet **tanto quanto IP**, in quanto costituisce il tramite fra tutti i vari AS. Una differenza sostanziale rispetto ai protocolli intra AS: i pacchetti non vengono instradati verso uno specifico indirizzo, ma verso **prefissi CIDR**.

BGP mette a disposizione di ogni router due funzioni. La prima è **ottenere informazioni sulla raggiungibilità dei prefissi**: consente a ciascuna sottorete di comunicare la propria esistenza al resto di Internet, ed è tramite BGP che tutti i router conoscono le sottoreti vicine: **senza BGP ogni sottorete sarebbe isolata**. La seconda è **determinare i percorsi ottimali**: un router può apprendere più percorsi verso lo stesso prefisso, e la selezione del migliore avviene in base alle **politiche del sistema autonomo** oltre che alle informazioni di raggiungibilità, **non basandosi soltanto sul costo minore**.

Lo scambio di informazioni avviene tramite messaggi scambiati, per ogni coppia di router comunicante, all'interno di una **sessione TCP**. Fra router dello stesso sistema autonomo si parla di **sessioni iBGP**, fra router di AS diversi di **sessioni eBGP**.

Supponiamo di voler dichiarare la raggiungibilità del prefisso x. L'annuncio viene passato ai nodi adiacenti, che costruiscono una **tabella BGP** contenente alcuni attributi: l'**AS_PATH**, che indica i sistemi autonomi attraverso cui l'annuncio è passato; il **NEXT_HOP**, che indica l'IP dell'interfaccia attraverso cui «entrare» nel primo sistema autonomo dell'**AS_PATH**; e naturalmente il **prefisso x** da raggiungere. L'insieme del prefisso e dei suoi attributi si chiama **route** (rotta), e valutando ogni **NEXT_HOP** secondo una metrica stabilita i router determinano il percorso migliore.

L'**AS_PATH** svolge peraltro una funzione ulteriore che vale la pena cogliere: elencando tutti gli AS attraversati, consente a un AS di **riconoscere sé stesso** nell'elenco e scartare l'annuncio, prevenendo così i loop di instradamento su scala globale.

Tre termini per completare il quadro: l'**AS Border Router (ASBR)** è un router connesso ad altri sistemi autonomi; il **BGP Speaker** è un router che supporta BGP, e non necessariamente coincide con un ASBR; i **BGP Neighbours** sono una coppia di speaker che si scambiano informazioni di instradamento inter AS, e si dicono **interni** se appartengono allo stesso AS, **esterni** se appartengono ad AS differenti.

I servizi del Data Link Layer

Il Data Link Layer è il livello che si trova **a cavallo fra il mondo logico e quello fisico**.

Fornisce al livello di rete un servizio di trasmissione per flussi di bit, e i suoi compiti principali sono cinque.

Il primo è il **data framing**, che opera in due direzioni: **dal fisico al logico** raggruppa i simboli e i bit del livello fisico in **frame**, trasformando il flusso grezzo di bit in informazioni strutturate; **dal logico al fisico** incapsula i datagrammi del livello di rete in frame composti da un campo dati e dai campi di intestazione.

Il secondo riguarda il **sottolivello MAC**, che gestisce l'accesso al mezzo fisico nel caso di collegamenti condivisi. Qui i dispositivi sono identificati tramite **indirizzi MAC**, veri e propri indirizzi fisici, e nei collegamenti broadcast l'accesso al canale deve essere regolato da **protocolli di accesso multiplo** per evitare collisioni. Nelle connessioni punto a punto questo sottolivello è semplificato, se non del tutto assente.

Il terzo è **fornire una consegna affidabile, se necessario**: e la clausola condizionale è il punto interessante. Se il mezzo fisico è sufficientemente affidabile, come la fibra o il doppino di rame, implementare meccanismi di affidabilità **non conviene**: sarebbero ridondanti rispetto a quelli già presenti ai livelli di trasporto e di applicazione, e si perderebbe tempo anziché sfruttare al massimo la velocità. Nel caso di mezzi meno affidabili, come le reti wireless, il trasporto affidabile viene invece ottenuto con meccanismi di ACK/NACK e ritrasmissioni.

Il quarto è **gestire gli errori** dovuti al canale, rilevandoli o correggendoli. Per «errore» si intende banalmente che una trasmissione è arrivata così degradata da non permettere di interpretare correttamente i bit: un bit trasmesso come 1 viene letto come 0. Le cause sono l'attenuazione del segnale, il rumore, le interferenze fra campi elettromagnetici o l'usura del mezzo fisico. I tassi di errore variano enormemente: nella **fibra ottica** si va da circa 1 bit errato su 10^{14} a 1 su 10^{16} , mentre in un collegamento **WiFi** il tasso è nell'ordine di 1 bit su 10 000. La **Gigabit Ethernet** su rame lavora con margini di rumore così ridotti che, senza una codifica correttiva, il tasso di errore sarebbe inaccettabile: per questo impiega una codifica a traliccio (*trellis*) con decodifica di Viterbi, descritta nel capitolo 49. Gli strumenti impiegati sono checksum, CRC, controllo di parità e codici di Hamming.

Il quinto è **regolare il flusso di dati** fra sorgente e destinazione. I collegamenti possono essere **full duplex** o **half duplex**, permettendo o meno la trasmissione contemporanea da entrambi i nodi, e il *pacing* di trasmissione deve essere proporzionato a quello di ricezione.

DOVE È IMPLEMENTATO IL DLL

In ogni host di una rete, dispositivo o router che sia, il Data Link Layer è implementato **nella NIC** (*Network Interface Card*) tramite una combinazione di hardware, software e firmware. La NIC implementa sia il DLL sia il livello fisico e si collega ai bus di sistema dell'host. Buona parte del livello è realizzata **in hardware**, nel **link layer controller**, il chip dedicato che costituisce il cuore della scheda; la parte software fa da ponte fra il livello di rete e quello fisico.

La comunicazione fra due interfacce procede così. **Lato mittente**: il datagramma viene incapsulato in un frame, vengono aggiunti i bit di controllo degli errori e le informazioni per affidabilità e controllo di flusso, e il frame viene trasferito. **Lato ricevente**: la NIC riceve l'intero frame, effettua le verifiche sui campi di rilevazione e correzione degli errori, estrae il datagramma e lo passa al livello di rete.

Data framing e codifiche di linea

Nella trasmissione dei dati è fondamentale trovare una codifica efficace che permetta di interpretare in modo **univoco** il segnale elettrico come stringhe di 0 e 1. Il modo più semplice sarebbe un **sistema a due livelli**: segnale alto uguale a 1, segnale basso uguale a 0, con **trasmissione sincrona** in cui mittente e ricevente condividono un clock, cosa che consente velocità maggiori.

Questo schema presenta però tre problemi che lo rendono inutilizzabile. Non consente di **distinguere una sequenza di 0 dall'assenza di comunicazione**, cioè dal silenzio. Non consente di capire **quando inizia o termina** una comunicazione. E in presenza di **lunghe sequenze** uguali si può perdere la frequenza di clock, cioè la sincronia, e non capire più quanto sia lunga quella sequenza.

Si ricorre allora a un **sistema a tre livelli**, in cui un livello è dedicato a segnalare l'assenza di comunicazione. Il prezzo è la **ridondanza**: si sta sprecando un livello che avrebbe potuto trasportare dati.

Esistono tecniche per limitare questa ridondanza, e si basano sul non associare un bit a un livello preciso ma alle **transizioni**. In una **trasmissione asincrona** si specificano bit di inizio e fine, ottenendo una velocità inferiore ma una minore complessità di gestione. La regola diventa: se il prossimo bit è **0** non c'è alcuna transizione di segnale; se è **1** e il livello è diverso da 0, il segnale passa a 0; se è **1** e il livello è 0, il segnale passa a un livello opposto rispetto al precedente e diverso da 0.

Si osservi che **le transizioni avvengono solo per trasmettere bit uguali a 1**. Trattandosi di trasmissione asincrona, una sequenza di quattro bit a 1 (cioè un ciclo completo) fornisce al ricevitore un vero e proprio **clock recovery**, stabilendo una frequenza di lettura adeguata ai dati in arrivo. Questo meccanismo permette poi al ricevitore di dedurre quanti bit a 0 leggere quando, in un quanto di tempo, il segnale resta a un certo livello. Restano comunque problematiche le **lunghe stringhe di zeri**, che possono ancora desincronizzare il ricevitore: è per questo che servono meccanismi ulteriori.

43.1 Codifiche a tre e a cinque livelli

La **Fast Ethernet** (100BaseTX) utilizza una codifica a **tre livelli** di segnale (positivo, nullo e negativo, nell'ordine di ± 1 V), valori peraltro non statici ma con un intervallo di tolleranza. Specificando **più di tre livelli** diventa possibile trasferire 2 bit per simbolo: la **Gigabit Ethernet** (1000BaseT) utilizza infatti una codifica a **cinque livelli** (± 1 V, $\pm 0,5$ V e 0 V) che permette di trasmettere a ogni simbolo una delle coppie 00, 01, 10, 11; il quinto livello fornisce la ridondanza sfruttata dalla codifica trellis.

La codifica a tre livelli è la base della trasmissione dati, ma presenta due criticità. Non è **DC balanced**: il livello di tensione medio non è necessariamente mantenuto tendente a zero, il che può portare complicazioni di vario tipo. E in presenza di lunghe sequenze di zeri si ripresenta il problema della **perdita di sincronizzazione**.

La soluzione è la **codifica a blocchi**, che sostituisce blocchi di bit di una data dimensione con blocchi leggermente più grandi (da 4 a 5, da 8 a 10) traducendo le sequenze in modo da **garantire un numero minimo di transizioni**.

43.2 Codifica 4B5B

Nella codifica **4B5B** blocchi da 4 bit vengono associati a blocchi da 5 bit: si trasmettono 4 bit di dati con 5 bit di segnale, introducendo quindi **1 bit di ridondanza ogni 4**.

Le proprietà garantite sono due, ed entrambe risolvono i problemi visti sopra. Non vengono **mai trasmesse sequenze di 4 o più zeri consecutivi**, così che una lunga sequenza di zeri possa essere interpretata

inequivocabilmente come assenza di comunicazione. E poiché le transizioni del segnale sono legate ai bit a 1, limitare le sequenze di zeri basta a garantire transizioni frequenti e quindi a **mantenere la sincronizzazione**.

L'aritmetica del codice è istruttiva: con 5 bit si hanno $2^5 = 32$ **combinazioni**, di cui **16** (cioè 2^4) servono a rappresentare i dati e le **altre 16 restano disponibili per informazioni di controllo**, per esempio per indicare inizio e fine comunicazione. È qui che sta il valore aggiunto rispetto a un semplice bit di ridondanza: lo spazio in eccesso non è sprecato ma riutilizzato.

Il costo è che la codifica richiede una **banda del 25% più capiente**. È utilizzata nella Fast Ethernet, ma **non garantisce il DC balance**.

43.3 Codifica 8B10B

La **8B10B** si basa sullo stesso principio della 4B5B ma trasmette 8 bit di dati con 10 bit di segnale, associando il bit 1 a un livello positivo e il bit 0 a un livello negativo.

La sua caratteristica distintiva è di essere **eletticamente neutra**, cioè **DC balanced**: il numero di bit a 1 e quello di bit a 0 risultano equilibrati. La motivazione è fisica: molti collegamenti sono **accoppiati in corrente alternata**, tramite trasformatori o condensatori che non lasciano passare la componente continua. Un segnale sbilanciato, con una componente continua non nulla, provocherebbe una lenta deriva del livello di riferimento al ricevitore e quindi errori di lettura.

Il meccanismo divide gli 8 bit in due gruppi: i **5 bit meno significativi** vengono associati a 6 bit, i **3 bit più significativi** a 4 bit. Alcune codifiche risultano differenti perché, per garantire l'equilibrio fra 1 e 0, si introduce un eccesso di bit dell'uno o dell'altro tipo per riequilibrare; la ridondanza complessiva però non cambia.

La 8B10B è impiegata in numerosi standard: **PCI Express** (fino alla 2.0), **IEEE 1394b** (Firewire), **Serial ATA**, **Fibre Channel**, alcune versioni di **Gigabit Ethernet**, **DisplayPort** e **USB 3.0**; **DVI** e **HDMI** usano una codifica a 10 bit analoga, detta TMDS.

43.4 Quantità di informazione e limite teorico di bitrate

Vale la pena formalizzare quanta informazione si possa trasportare con un dato numero di livelli. Con un sistema a 3 livelli (alto, medio, basso) guardando 2 simboli per volta si hanno 3^2 combinazioni, con 3 simboli 3^3 , e così via. Ma si possono utilizzare **solo le combinazioni date dalla potenza di 2 immediatamente più piccola**: con 4 passi si hanno 81 combinazioni, e la potenza di 2 inferiore è 64, cioè 2^6 .

In generale, dati **n simboli distinti** (nel nostro caso livelli di tensione), il numero di informazioni binarie trasportabili è:

$$x = \log_2 n ; \text{poiché non si possono esprimere bit parziali, } x = \lfloor \log_2 n \rfloor$$

Moltiplicando questo valore per il numero di passi al secondo si ottiene il **limite teorico di bitrate**:

$$\text{Limite teorico di bit rate} = \text{simboli al secondo} \cdot \log_2 n$$

Un'osservazione che segue da questi conti: nei casi in cui la potenza di 2 non coincide con il numero di combinazioni disponibili, esistono **sequenze di bit che non significano nulla**, e serve dunque un controllo degli errori.

Rilevazione e correzione degli errori

Il capitolo distingue la rilevazione dalla correzione degli errori e descrive due tecniche di rilevazione: il controllo di parità e il CRC, con un esempio di calcolo svolto.

DA RICORDARE

I due obiettivi vanno distinti con nettezza: *rilevare* e *correggere*. Il **CRC** fa solo il primo, i **codici di Hamming** fanno entrambi.

L'**error detection** consiste nell'individuare errori di trasmissione in un frame, **senza effettuare correzione**. La **correzione** consiste invece nell'individuare gli errori ed effettuare correzioni, necessariamente **di dimensioni limitate**.

Entrambi i metodi si basano sull'analisi di **informazioni ridondanti**. L'analogia più efficace è la seguente: se in un pacchetto trasmettiamo dei dati anagrafici e vi aggiungiamo il **codice fiscale**, abbiamo aggiunto informazione ridondante (il codice fiscale è ricavabile dai dati anagrafici) ma proprio questa ridondanza ci permette di individuare a posteriori eventuali errori di trasmissione.

Un datagramma inoltrato attraverso un mezzo che ammette errori contiene quindi una sequenza di bit **D** e un campo **EDC** (*Error Detection and Correction bits*), che può essere un CRC, un checksum o un controllo di parità. Va detto con chiarezza che **il rilevamento degli errori non è affidabile al 100%**: un campo EDC più ampio produce un rilevamento e una correzione migliori, ma non la certezza.

44.1 Controllo di parità

Il **parity check** consiste in un bit che specifica se il numero di bit a 1 sia pari o dispari. Si calcola con un'operazione di **XOR** sui bit a 1 e si aggiunge un bit a fine sequenza in modo da rendere pari il numero complessivo di uni.

Il limite è evidente: si rileva soltanto un **numero dispari di errori**, perché con due errori, o con un qualsiasi numero pari, la parità tornerebbe corretta e la sequenza sembrerebbe integra. Funziona quindi solo se la frequenza di errori è bassa (è **inutilizzabile nel WiFi**) e più la sequenza è lunga più è improbabile che il sistema funzioni. Al ricevitore la verifica è semplice: si calcola la parità dei bit ricevuti e la si confronta con il bit di parità ricevuto; se differiscono, si è rilevato un errore.

Un'applicazione concreta interessante sono le **memorie RAM con controllo di parità**: hanno un costo maggiore e sono usate nei server, impiegano **9 bit per byte** di cui uno di parità, e funzionano bene proprio perché nella memoria i bit errati sono rari ma potrebbero causare problemi gravissimi, come il reset della macchina.

Si distinguono infine due varianti. Il controllo di parità **monodimensionale** consente esclusivamente di **individuare** un singolo errore. Quello **bidimensionale**, che dispone i bit in una matrice calcolando la parità per righe e per colonne, consente di individuare e **correggere** un singolo errore: l'intersezione fra la riga e la colonna che risultano sbagliate identifica esattamente il bit errato.

44.2 CRC: Cyclic Redundancy Check

Il **CRC** è un metodo di *error detection*, **non di correzione**, detto anche controllo dei codici polinomiali. Sfrutta l'**aritmetica modulo 2**, in cui addizione e sottrazione diventano entrambe operazioni di **XOR**: il che rende tutto l'algoritmo implementabile in modo estremamente efficiente.

L'idea di base è rappresentare la sequenza di bit **D** come un **polinomio** di grado $|D| - 1$, con coefficienti pari al valore dei bit: il bit più a sinistra è il coefficiente di grado massimo, quello più a destra il termine noto. Alla sequenza 10010111 corrisponde per esempio il polinomio $x^7 + x^4 + x^2 + x + 1$.

Dal punto di vista matematico i coefficienti appartengono al campo $\mathbb{Z}_2$, in cui esistono solo 0 e 1: la **somma** coincide con lo **XOR** e il **prodotto** con l'**AND**.

IL PROCEDIMENTO

Siano **D(x)** il polinomio associato al messaggio, **G(x)** il **polinomio generatore** stabilito dallo standard di rete (la cui cifra più significativa è sempre 1) e **r** il grado di G(x). Si definisce $R(x) = x^r \cdot D(x) \bmod G(x)$.

Operativamente si prendono i **d** bit di dati e si aggiungono **r bit di zeri** in coda, che equivale a uno shift a sinistra, ottenendo una sequenza di **d + r** bit; si divide questo polinomio per G(x) e se ne prende il **resto**. La proprietà fondamentale è che, **sommando il resto R(x) al polinomio $x^r \cdot D(x)$** , si ottiene un polinomio **perfettamente divisibile** per G(x).

MITTENTE

1. D sono i dati da trasmettere
2. calcola $D * 2^r$ (nella pratica: $D \ll r$)
3. esegue la divisione mod 2: $(D \ll r) / G$
4. il resto della divisione è R
5. trasmette D seguito da R (concatenazione)
6. questo valore è sempre divisibile per G con resto nullo

DESTINATARIO

1. calcola $\text{StringaRicevuta} / G$
2. se il resto è diverso da 0 → ci sono errori

Il procedimento del CRC

ESEMPIO NUMERICO SVOLTO

Prendiamo **D = 101110** e come generatore **G = 1001**, di grado **r = 3**. Aggiungiamo tre zeri a D e dividiamo modulo 2, cioè eseguendo XOR a ogni passo in cui il bit guida vale 1.

```
101110000      ← D spostato a sinistra di r = 3 posizioni
1001             ← XOR (bit guida = 1)
-----
001010000
 1001           ← XOR
-----
000011000
 1001           ← XOR
-----
000001010
 1001           ← XOR
-----
000000011
```

resto R = 011
trasmissione: D seguito da R = 101110 011 = 101110011

VERIFICA AL RICEVITORE

101110011 / 1001 → resto 000 → nessun errore rilevato
se un bit si altera (per esempio 100110011):
100110011 / 1001 → resto 001 → errore rilevato

CRC con D = 101110 e G = 1001

Nella pratica la divisione modulo 2 si realizza con un **semplice registro a scorrimento** e alcune porte XOR, e la dimensione del registro dipende dal grado del polinomio generatore. È questo che rende il CRC realizzabile con pochissime risorse.

CHE COSA IL CRC RILEVA E CHE COSA NO

C'è un'asimmetria fondamentale da tenere a mente: **se il CRC dice che un frame è rovinato, lo è sicuramente; ma se dice che non ha errori, esiste una probabilità che invece ne abbia**. Più ridondanza si aggiunge, più questa probabilità si riduce.

Nel dettaglio, il CRC rileva **tutti gli errori a bit singolo**, perché la modifica di un solo bit altera in modo significativo il polinomio producendo un resto diverso. Con un generatore scelto opportunamente rileva **tutti gli errori a doppio bit** entro la lunghezza massima del frame prevista dallo standard. Rileva **qualsiasi numero dispari di errori** quando il generatore contiene il fattore $(x + 1)$. Ed è **particolarmente efficace sugli errori a raffica (burst)**: un CRC a n bit rileva tutte le raffiche lunghe al più n bit.

Quanto all'efficienza: per un CRC a n bit la probabilità di errore non rilevato è circa 2^{-n} ; un CRC a 32 bit ha quindi una probabilità di circa $2,3 \cdot 10^{-10}$, cioè **circa un errore non rilevato ogni quattro miliardi** di frame danneggiati in modo casuale. La copertura è quasi totale, ma non infallibile: potrebbero sfuggire errori che producono lo stesso resto quando divisi per il generatore, casi però statisticamente rarissimi.

I **polinomi generatori** non si scelgono a caso: devono contenere il termine x^0 , cioè **terminare con +1**, così da rilevare tutti gli errori singoli, e sono fissati dagli standard dopo un'attenta analisi delle loro proprietà. Il CRC è largamente utilizzato perché **facile da implementare**, sia in software sia, soprattutto, in hardware con registri a scorrimento: **WiFi ed Ethernet usano il CRC a 32 bit**, e codici CRC si trovano anche nei dischi e in molti altri sistemi di memorizzazione e trasmissione.

Distanza di Hamming e correzione degli errori

Dalla distanza fra codeword alle formule che legano distanza minima, errori rilevabili ed errori correggibili, fino alla costruzione dei codici di Hamming e alla tecnica dell'interleaving.

Date due **codeword**, si definisce **distanza di Hamming** il numero di bit che differiscono fra di esse, cioè il numero di bit che occorre modificare per passare dall'una all'altra. Si calcola semplicemente con uno **XOR** contando gli uni del risultato.

```

10001100  XOR
11000100  =
-----
01001000  →  due bit a 1  →  d = 2

```

Trattandosi di una distanza in senso matematico, gode delle proprietà attese: è un numero **positivo o nullo**, vale zero se e solo se gli elementi sono uguali, e **si somma** rispettando la disuguaglianza triangolare: la distanza fra A e B più quella fra B e C è **maggiore o uguale** alla distanza fra A e C.

Si definisce **vocabolario** un insieme di codeword, e un vocabolario si dice **completo** se con n bit contiene tutte le 2^n combinazioni possibili. Il punto centrale di tutta la teoria è che **i vocabolari utili non possono essere completi**. Consideriamo un vocabolario con 9 bit, di cui uno di parità: le combinazioni possibili sarebbero 512, ma solo 256 sono quelle effettivamente valide. Il canale permette di trasmettere tutte e 512 le combinazioni, ma sarà poi il **ricevitore a gestire** ciò che gli arriva: se riceve una delle 256 combinazioni non definite, scarta il frame; altrimenti il messaggio potrebbe essere corretto.

45.1 L'idea della correzione

Consideriamo un vocabolario con sole quattro codeword: **000000**, **000111**, **111000**, **111111**. Abbiamo 2 bit di dati (sufficienti a generare quattro codeword) e 4 bit di ridondanza. La **distanza minima** di questo vocabolario è **d = 3**. Le combinazioni possibili su 6 bit sarebbero 64, ma ne consideriamo valide soltanto 4: il mittente può trasmettere **solo le 4 codeword definite**, mentre il ricevente potrebbe ricevere una qualsiasi delle 64 combinazioni.

Limitando un vocabolario a stringhe con una **distanza minima di Hamming fissata** si limita il numero di stringhe valide, e proprio questo permette di introdurre meccanismi di riconoscimento e correzione. Il ragionamento si fonda su un'**ipotesi probabilistica** che va sempre esplicitata: la probabilità che vi sia un solo errore è molto maggiore di quella che ve ne siano due, e così via: formalmente $P(e=1) \gg P(e=2) \gg P(e=3)$ e oltre.

Sotto questa ipotesi, se riceviamo una sequenza non valida, la codeword corretta è **quella a distanza minima** dalla sequenza ricevuta: essendo più probabile un errore singolo, è più probabile che il messaggio originale fosse quello più vicino. La correzione avviene quindi **solo su base probabilistica**, e questo va detto con onestà: non è una certezza matematica.

45.2 Le due formule fondamentali

Generalizzando il ragionamento si ottengono le due relazioni che vanno assolutamente ricordate.

Per correggere e bit errati serve una distanza minima $d = 2e + 1$

Per rilevare e bit errati serve una distanza minima $d = e + 1$

L'intuizione geometrica dietro la prima è che, immaginando attorno a ogni codeword valida una «sfera» di raggio e contenente tutte le sequenze ottenibili con al più e errori, occorre che **le sfere non si sovrappongano**: se due codeword fossero troppo vicine, una sequenza errata potrebbe cadere nell'intersezione e non si saprebbe da quale delle due provenga. La distanza $2e + 1$ garantisce esattamente questa separazione.

Riassumendo, per procedere alla correzione degli errori singoli servono tre ingredienti: l'**ipotesi probabilistica** di base, la definizione di un **vocabolario** con un numero ridotto di simboli leciti, e una **distanza minima $d = 3$** .

45.3 Quanta ridondanza serve

Restano due domande operative: dati m bit di dati, di quanti bit di ridondanza abbiamo bisogno per correggere e errori? E come costruire concretamente il vocabolario?

Affrontiamo la prima. Sia m il numero di bit di dati, r quello dei bit di controllo, e $n = m + r$ la lunghezza totale; avremo 2^n combinazioni possibili di cui 2^m valide. Data una codeword valida, modificandone un singolo bit si ottengono n **codeword non valide**: attorno a ogni codeword valida ce ne sono dunque n errate, e considerando anche la codeword stessa il «cerchio» ne contiene $n + 1$.

Estendendo il ragionamento a tutte le codeword valide, i cerchi sono 2^m e ciascuno contiene $n + 1$ elementi. Poiché con $d = 3$ i cerchi sono necessariamente **tutti separati** (nessuna sequenza può trovarsi a distanza 1 da due codeword valide diverse) possiamo scrivere con certezza:

$$(n + 1) \cdot 2^m \leq 2^n$$

Sostituendo $n = m + r$ e semplificando si arriva alla forma operativa:

$$\begin{array}{rcl} (n + 1) \cdot 2^m & \leq & 2^n \\ (m + r + 1) \cdot 2^m & \leq & 2^{(m+r)} \\ (m + r + 1) & \leq & 2^r \\ m + 1 & \leq & 2^r - r \end{array}$$

Derivazione del numero minimo di bit di ridondanza (valida per $d = 3$)

L'ultima espressione è valida **soltanto per $d = 3$** , cioè per la correzione di un singolo errore. Un esempio: volendo trasmettere un messaggio di $m = 5$ bit correggendo un errore, cerchiamo il più piccolo r che soddisfi $5 + 1 \leq 2^r - r$. Con $r = 3$ si ha $8 - 3 = 5 < 6$, insufficiente; con $r = 4$ si ha $16 - 4 = 12 \geq 6$, quindi bastano **4 bit di ridondanza** e la codeword sarà lunga 9 bit.

IL CASO $d = 5$

Ripetiamo il ragionamento per $d = 5$. Anzitutto, quanti errori si possono correggere? Da $d = 2e + 1$ si ricava $5 = 2e + 1$ e quindi $e = 2$. Il cerchio attorno a ciascuna codeword valida deve ora contenere tutte le sequenze a distanza 1 e a distanza 2. Le sequenze a distanza 1 sono n ; quelle a distanza 2 sono le combinazioni di 2 bit su n , cioè $n(n - 1)/2$. Il totale, inclusa la codeword stessa, è:

$$1 + n + n(n - 1)/2$$

e la condizione diventa quindi $[1 + n + n(n - 1)/2] \cdot 2^m \leq 2^n$. È lo stesso schema di prima, esteso di un termine: **il metodo generale consiste nel contare quante sequenze cadono nella sfera di raggio e e imporre che le sfere non si sovrappongano**.

45.4 I codici di Hamming

Passiamo alla seconda domanda: come costruire operativamente il vocabolario. La costruzione proposta da Hamming sfrutta le **potenze di 2**.

Si prende una codeword e, numerando le posizioni dei bit in ordine crescente, si collocano **nelle posizioni che sono potenze di 2** (1, 2, 4, 8, 16, ...) i **bit di controllo**, mentre nelle altre posizioni si mettono i **bit di dati**. Equivalentemente, la posizione di ciascun bit di dati è data da una **combinazione di potenze di 2**. Il pregio di questo sistema è l'automatismo: aumentando i bit di dati, i bit di controllo si inseriscono da soli nelle posizioni giuste.

Il valore del bit di controllo in posizione 2^k è dato dallo **XOR** (cioè è il bit di parità) di tutti i bit di dati la cui posizione, scritta come somma di potenze di 2, **include quella potenza**.

ESEMPIO NUMERICO SVOLTO

Costruiamo il codice di Hamming per i **4 bit di dati 1011**. Verifichiamo prima quanti bit di controllo servono: con $m = 4$ cerchiamo r tale che $m + r + 1 \leq 2^r$, e con $r = 3$ si ha $4 + 3 + 1 = 8 \leq 8$, che è soddisfatta. La codeword sarà quindi lunga **$n = 7$** bit.

DISPOSIZIONE (p = controllo nelle potenze di 2, d = dati altrove)

posizione	1	2	3	4	5	6	7
ruolo	p1	p2	d1	p4	d2	d3	d4
dati			1		0	1	1

CALCOLO DEI BIT DI CONTROLLO

p1 copre le posizioni 3,5,7 (contengono la potenza 1)

$$p1 = 1 \text{ XOR } 0 \text{ XOR } 1 = 0$$

p2 copre le posizioni 3,6,7 (contengono la potenza 2)

$$p2 = 1 \text{ XOR } 1 \text{ XOR } 1 = 1$$

p4 copre le posizioni 5,6,7 (contengono la potenza 4)

$$p4 = 0 \text{ XOR } 1 \text{ XOR } 1 = 0$$

CODEWORD TRASMESSA

posizione	1	2	3	4	5	6	7
bit	0	1	1	0	0	1	1

→ 0110011

Costruzione del codice di Hamming per i dati 1011

Vediamo ora come il ricevitore individua un errore. Supponiamo che durante la trasmissione si alteri il bit in **posizione 5**, cosicché arrivi **0110111** anziché 0110011. Il ricevitore ricalcola le stesse parità e le confronta con quelle ricevute.

RICEVUTO 0110111

posizione	1	2	3	4	5	6	7
bit	0	1	1	0	1	1	1

VERIFICA (ogni controllo include ora anche il proprio bit di parità)

$$c1 = \text{bit } 1,3,5,7 = 0 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 = 1 \rightarrow \text{INCOERENTE}$$

$$c2 = \text{bit } 2,3,6,7 = 1 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 = 0 \rightarrow \text{coerente}$$

$$c4 = \text{bit } 4,5,6,7 = 0 \text{ XOR } 1 \text{ XOR } 1 \text{ XOR } 1 = 1 \rightarrow \text{INCOERENTE}$$

SINDROME $c4 \ c2 \ c1 = 1 \ 0 \ 1 = 4 + 1 = 5$

→ il bit errato è quello in POSIZIONE 5: basta invertirlo

Individuazione e correzione dell'errore tramite la sindrome

Il meccanismo è elegante: i controlli incoerenti sono quelli che coprono la posizione errata, e **sommando le potenze di 2 corrispondenti si ottiene direttamente il numero della posizione sbagliata**. Nell'esempio risultano incoerenti p1 e p4, e $1 + 4 = 5$.

Resta però un limite fondamentale: tutto questo è **valido solo se vi è al più un bit errato**, e nessuno può garantirlo. Per questo nella pratica si combinano le due tecniche: **Hamming corregge**, e il **CRC verifica** che la correzione abbia effettivamente prodotto un frame valido.

45.5 Interleaving: gestire le raffiche di errori

I codici di Hamming correggono bene gli errori **isolati**, ma sono impotenti di fronte alle **raffiche**, cioè a sequenze di bit consecutivi danneggiati: se in una codeword da 7 bit ne sbagliano tre, non c'è correzione possibile. Esiste però una tecnica per trasformare una raffica in un insieme di errori isolati.

L'idea è la seguente. Si prende la sequenza di bit da trasmettere e la si separa in codeword più piccole, disponendole come **righe di una matrice**. Si trasmette poi la matrice **colonna per colonna** anziché riga per riga. Se durante la trasmissione si verifica una raffica di errori, questa colpirà bit consecutivi **nel flusso trasmesso**, ma una volta riposizionati i bit in verticale al ricevitore, quegli errori risulteranno **distribuiti su codeword differenti**: al più uno per codeword. A quel punto i codici di Hamming possono correggerli tutti.

Il vantaggio è che la tecnica **si adatta ai diversi tipi di canale**. Nella **fibra ottica** è raro che vi siano bit errati, non essendovi interferenze elettromagnetiche; nel **wireless** si possono invece avere frame con cinque o sei bit consecutivi errati, e distribuendoli su più codeword si rende Hamming di nuovo utilizzabile.

Lo svantaggio è che occorre **bufferizzare** prima di trasmettere, trasmettere colonna per colonna e compiere l'operazione inversa a destinazione: tutto ciò comporta un **ritardo di trasmissione**.

Gestire gli accessi multipli

Quando più nodi condividono lo stesso canale occorre regolare l'accesso. Il capitolo presenta le famiglie di protocolli di accesso multiplo, le topologie broadcast e le tecniche a suddivisione del canale.

In una rete si individuano due tipi di collegamento. I **collegamenti punto a punto** sono diretti fra trasmittente e ricevente e non pongono problemi. I **collegamenti broadcast** vedono invece più nodi trasmittenti e riceventi che sfruttano **lo stesso canale**, e la loro gestione non è affatto un problema intuitivo da risolvere.

Il fenomeno da evitare è la **collisione**, che si verifica quando due nodi trasmettono sullo stesso canale nello stesso istante: le collisioni causano perdite di frame e, se troppo frequenti, un generale **spreco di banda**. I **protocolli di accesso multiplo** fissano le modalità con cui i nodi regolano le trasmissioni sul canale condiviso, e si dividono in tre famiglie.

- **Protocolli a suddivisione del canale:** il canale viene diviso in porzioni accessibili esclusivamente a un nodo.
- **Protocolli ad accesso casuale:** l'accesso avviene in modo casuale, **ammettendo collisioni** che verranno poi gestite opportunamente.
- **Protocolli a rotazione, senza collisioni:** coordinando opportunamente l'accesso, le collisioni vengono evitate del tutto.

LE TOPOLOGIE BROADCAST

La **topologia a stella** necessita di un hub centrale che riceve e trasmette i segnali da e verso tutte le macchine. I punti di forza sono la semplicità di installazione e gestione, il basso costo dei componenti e l'elevata scalabilità; i punti di debolezza sono la **vulnerabilità al guasto del nodo centrale** e le prestazioni influenzate dal numero di dispositivi connessi.

La **topologia ad anello** si basa su collegamenti punto a punto, e ogni macchina ha **due schede di rete**. I punti di forza sono che ogni dispositivo ha le stesse possibilità di accesso al mezzo e che non serve un dispositivo centrale; con un doppio anello si ottiene inoltre ridondanza in caso di guasti. I punti di debolezza sono la velocità limitata dal numero di dispositivi, poiché **ciascuno introduce ritardo**, e la sensibilità ai guasti dei nodi.

La **topologia a bus** vede il segnale viaggiare su tutto il canale, con possibilità di collisioni se più macchine parlano contemporaneamente. I punti di forza sono la struttura semplice ed economica e la facilità di installazione ed espansione; i punti di debolezza sono la **scarsa affidabilità**, dipendendo tutto da un unico cavo, e la velocità limitata dal numero di dispositivi.

46.1 Protocolli a suddivisione del canale

Prima di entrare nel dettaglio, un'osservazione importante che vale per tutte le tipologie di accesso multiplo: **il volume complessivo è sempre lo stesso**, perché la capacità del canale non cambia. Ciò che cambia è **come** viene ripartita.

Il **TDMA** (*Time Division Multiple Access*) suddivide il tempo in intervalli e ogni intervallo in **N slot temporali**, uno per nodo. Si parla «a turno» in uno slot predefinito, il che limita la banda di ciascun nodo a **R/N bps**; si utilizza tutto lo spettro delle frequenze, ma in momenti differenti. Il difetto è che se ci si trova nello slot di un nodo che **non deve trasmettere**, si verifica un'attesa non necessaria e gli slot inutilizzati diventano **inattivi**: in una LAN con 6 stazioni, se le stazioni 2, 5 e 6 non trasmettono, i loro slot restano sprecati.

L'**FDMA** (*Frequency Division Multiple Access*) suddivide invece lo **spettro** del canale in bande di frequenza, associandone una fissa a ogni nodo. Limita anch'esso la banda a **R/N bps**, ma **non blocca la trasmissione** dei

nodi, che possono trasmettere in qualsiasi momento all'interno della propria banda. Il difetto è speculare a quello del TDMA: il tempo di trasmissione non utilizzato in una banda diventa inattivo, e in una LAN con 6 stazioni le bande 2, 5 e 6 resterebbero inutilizzate.

Il **CDMA** (*Code Division Multiple Access*) adotta un approccio radicalmente diverso: associa un **codice univoco** a ogni nodo, i dati inviati vengono codificati secondo quel codice, e **si può trasmettere simultaneamente**. Il ricevitore deve conoscere il codice del mittente che vuole ascoltare per estrarne correttamente i dati.

Il funzionamento merita una spiegazione, perché è controintuitivo. Quando avviene una collisione i segnali si sovrappongono e normalmente non si riesce più a distinguere i dati. L'idea del CDMA è **modificare la sequenza di trasmissione moltiplicandola per un codice opportuno** prima di immetterla nel canale; il ricevente riapplica lo stesso codice per ottenere la sequenza originale. Considerando due trasmettitori con codifiche differenti, si riesce ad avere **due trasmettenti che trasmettono contemporaneamente sulle stesse frequenze**, sovrapponendo i segnali **senza rovinare il risultato finale**, perché il segnale originale è ricostruibile. La condizione matematica è che i codici siano **ortogonali** fra loro, cioè che il prodotto scalare fra due codici diversi sia nullo: in questo modo, moltiplicando il segnale ricevuto per il codice di un mittente, i contributi degli altri si annullano.

Protocolli ad accesso casuale

Dai protocolli ALOHA al CSMA con rilevazione delle collisioni, fino ai protocolli senza collisioni e a rotazione: come si condivide un canale accettando, o evitando, le collisioni.

47.1 Pure ALOHA

Il protocollo **ALOHA** fu sviluppato all'Università delle Hawaii fra la fine degli anni Sessanta e il 1971 per realizzare una rete di comunicazione wireless fra i vari campus sparsi nelle isole adiacenti. La rete era composta da nodi **non sincronizzati** fra loro e utilizzava **due canali differenti**, uno per trasmettere e uno per ascoltare.

Le regole del protocollo sono due, di disarmante semplicità: ogni nodo ha un frame di **dimensione fissa**; e **si parla quando si vuole**: se devi comunicare trasmetti, altrimenti taci. Naturalmente si prova a parlare, ma non è detto che funzioni.

La struttura fisica prevedeva stazioni periferiche nelle varie isolette e un'**antenna centrale** che fungeva da ripetitore: ascoltava i messaggi delle stazioni e li ritrasmetteva su un **canale differente** da quello di ricezione, così da tenere quest'ultimo sempre libero per l'ascolto. Nelle comunicazioni provenienti dal ripetitore nessuno parlava e tutti ascoltavano; poteva però capitare che due nodi parlassero nello stesso momento, provocando una **collisione**. Le probabilità erano alte proprio per l'assenza di sincronizzazione, ed entrambi i frame venivano danneggiati, con i segnali che si mescolavano rendendo i messaggi incomprensibili.

I **periodi critici** per cui un frame può essere rovinato sono tre: due stazioni trasmettono contemporaneamente; si trasmette un frame quando uno trasmesso poco prima deve ancora terminare; si trasmette un frame e subito dopo ne viene trasmesso un altro. È da questa considerazione che deriva il modello matematico.

Detti **G** i tentativi per tempo di pacchetto e **S** il throughput per tempo di frame, il **throughput massimo si ha per $G = 0,5$ con $S = 0,184$** : il canale è utilizzato per un massimo del **18,4%** del suo potenziale. È un valore molto basso, e giustifica da solo tutte le evoluzioni successive.

Un dettaglio interessante: una macchina poteva **rilevare subito** se il proprio frame era stato rovinato, semplicemente ascoltando il canale di ritorno: se vi avesse sentito rumore anziché il proprio frame, avrebbe compreso l'avvenuta collisione e ritrasmesso. Quindi, benché non fosse definito esplicitamente, **di fatto esisteva un controllo di collisione**, reso possibile proprio dalla separazione fra canale di andata e di ritorno.

47.2 Slotted ALOHA

È una versione migliorata che introduce una **sincronizzazione a slot temporali**: il tempo è suddiviso in slot di uguale durata e i nodi possono trasmettere **solo all'inizio** di uno slot, il che evita le sovrapposizioni parziali all'inizio e alla fine dei frame. Se due nodi tentano di trasmettere nello stesso slot la collisione avviene comunque, ma il sistema è più efficiente perché la **probabilità di collisione è ridotta**.

I vantaggi sono che il protocollo resta **semplice**, che un **singolo nodo attivo può lavorare al massimo delle prestazioni** senza limitazioni di banda, che è **scalabile** e **altamente decentralizzato**: solo gli slot devono essere sincronizzati. Gli svantaggi sono che **le collisioni sprecano slot interi** e che possono esservi **slot inutilizzati**.

Entrambe le versioni usano un **tempo di attesa casuale semplice** (*backoff*) prima di ritrasmettere dopo una collisione, che **non dipende dal numero di collisioni** rilevate. Il confronto si riassume così: **Slotted ALOHA ha prestazioni all'incirca doppie rispetto a Pure ALOHA** (dal 18,4% si passa a circa il 37%), ed entrambi funzionano bene soltanto con traffico basso.

47.3 CSMA

ALOHA aveva due difetti evidenti: non prevedeva **alcun controllo del canale** prima della trasmissione, e i nodi trasmettevano quando volevano anche se qualcun altro stava già trasmettendo. Da qui nasce il **CSMA** (*Carrier Sense Multiple Access*), fondato su un principio semplice ma efficace: **ascolta prima di trasmettere**. Se il canale è inattivo si trasmette l'intera frame; se è occupato si rinvia la trasmissione.

La domanda successiva è: **che cosa fare nel momento in cui il canale si libera?** Le strategie sono tre.

STRATEGIA	COMPORTAMENTO	VALUTAZIONE
1 persistent	Appena il canale è libero il nodo trasmette subito . È «persistente» perché continua ad ascoltare finché il canale non si libera.	Semplice e reattivo, ma se più nodi stanno aspettando si scontrano tutti appena il canale si libera.
p persistent	Quando il canale si libera il nodo trasmette con probabilità p , altrimenti attende un poco e ritenta.	Riduce la probabilità che tutti trasmettano insieme e quindi le collisioni, ma introduce ritardo .
non persistent	Il nodo non controlla continuamente il canale: se lo trova occupato attende un tempo casuale prima di ricontrollare.	Riduce ulteriormente le collisioni ma è meno reattivo e introduce più attesa.

Un caso limite illuminante è il **CSMA 0,01 persistent**, che equivale a lanciare un dado a cento facce e trasmettere solo se esce 1. Le collisioni diventano rarissime e il grafico del throughput può sembrare buono, ma **inganna**: il canale resta quasi sempre libero mentre le stazioni attendono, e i ritardi diventano enormi. Un valore di p troppo basso spreca il canale tanto quanto uno troppo alto lo congestiona.

47.4 CSMA/CD

CSMA non è comunque in grado di **evitare** le collisioni: può sempre accadere che due host trovino la rete libera nello stesso momento e inizino entrambi a trasmettere. Il **CSMA/CD** (*with Collision Detection*) aggiunge allora una seconda regola alla prima: **«se qualcun altro inizia a parlare insieme a voi, smettete di parlare»**.

Concretamente il nodo che sta trasmettendo **rimane in ascolto sul canale** mentre trasmette; se rileva una collisione **interrompe immediatamente la propria trasmissione**, riducendo così lo spreco del canale: è questo il guadagno essenziale rispetto a CSMA, dove entrambi i nodi avrebbero continuato a trasmettere un frame ormai irrimediabilmente rovinato. Successivamente entrambi i nodi attendono un intervallo di tempo casuale, detto **backoff**, e ritentano. Il backoff **non può essere un tempo fisso**, altrimenti i due nodi ritenterebbero simultaneamente all'infinito rendendo il protocollo inutilizzabile.

C'è però un problema ulteriore: quando un nodo interrompe la trasmissione il canale si libera, e altri nodi potrebbero approfittarne per trasmettere, innescando nuove collisioni. Per evitarlo, il nodo che si interrompe invia un **segnale di jamming**, avvertendo tutti i nodi in ascolto dell'avvenuta collisione.

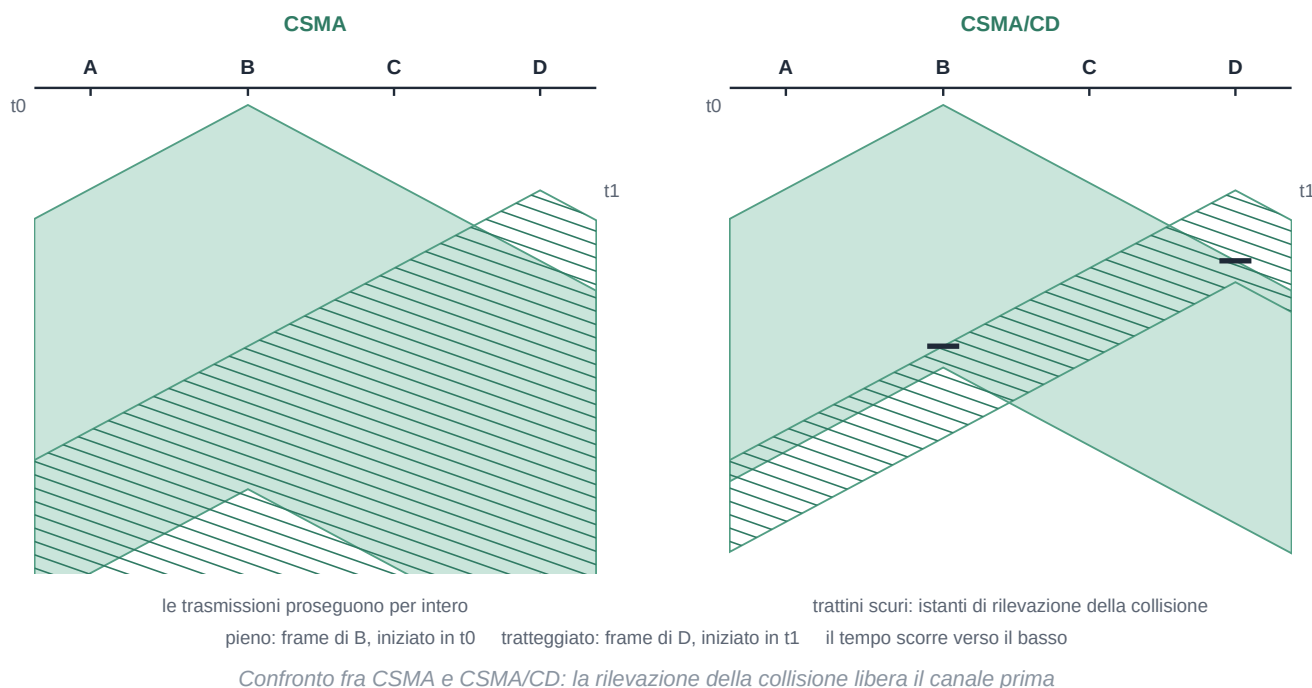
IL BINARY EXPONENTIAL BACKOFF

Se il tempo di backoff fosse troppo breve si provocherebbero altre collisioni. La soluzione è l'algoritmo di **binary exponential backoff**: quando si verifica la collisione numero **m** si estrae un valore casuale **k** dall'insieme **{0, 1, ..., $2^m - 1$ }**, e si attende un tempo **$T = k \cdot t_{\text{slot}}$** . Il valore massimo di **m** è **10**, quindi **k** arriva al più a **1023**; dall'undicesima collisione in poi **m non cresce più**, e dopo **16 collisioni** si dichiara il fallimento definitivo di quel frame. In Ethernet **t_{slot} coincide con il tempo richiesto per trasmettere 512 bit**.

La logica è che **più collisioni si verificano, più lungo diventa l'intervallo di attesa**: il protocollo si adatta automaticamente al livello di congestione del canale, esattamente come faceva il raddoppio del timeout in TCP.

PERCHÉ SERVE TEMPO PER ACCORGERSENE

Consideriamo lo scenario peggiore: A sta trasmettendo verso B, e B comincia a trasmettere **un istante prima** che il pacchetto di A gli arrivi. La collisione avviene **in prossimità di B**, che avverte per primo il rumore, interrompe la propria trasmissione e lancia il segnale di jamming per avvertire A. Ma quel rumore, per raggiungere A, deve **percorrere tutto il canale in senso opposto**, e questo richiede tempo. Nel caso peggiore occorre quindi attendere **due volte il tempo di propagazione** da un estremo all'altro del canale: trascorso tale periodo senza rilevare rumore, A può considerarsi sicuro che non vi sia stata collisione. Il fenomeno è interamente dovuto ai **ritardi di propagazione** sul canale, ed è la ragione per cui esiste una dimensione minima del frame Ethernet.



Il confronto grafico chiarisce la differenza. Con **CSMA** il frame viene rovinato quasi del tutto, le due macchine **continuano a trasmettere** nonostante la collisione, e con la variante p persistent poteva accadere che due trasmettenti tentassero ripetutamente nello stesso periodo finché uno dei due non rinunciava. Con **CSMA/CD** il canale **si libera prima**, perché rilevata la collisione entrambi interrompono la trasmissione, e il nuovo tentativo non avviene in base a una probabilità ma **in base a un tempo** calcolato.

DOVE CSMA/CD NON FUNZIONA

CSMA/CD è un ottimo protocollo, ma richiede che la *collision detection* avvenga **rapidamente**, e questo dipende dal mezzo.

Nei **sistemi cablati** il rilevamento è facile: nelle schede Ethernet si può ascoltare mentre si parla, e in un cavo condiviso il segnale elettrico è lo stesso per tutti, cosicché se due nodi trasmettono i segnali **si sommano** e si rileva subito che il livello è diverso o corrotto.

Nelle **reti wireless** è invece difficile da implementare, per due ragioni. La prima è che il segnale trasmesso è **enormemente più potente** di quello ricevuto, di molti ordini di grandezza, e lo copre completamente: mentre si parla, semplicemente, non si sente nulla. La seconda è il **problema dei terminali nascosti**: A e C potrebbero essere entrambi collegati a B ma **fuori portata reciproca**, cosicché nessuno dei due può rilevare se l'altro stia trasmettendo, e la collisione avviene inevitabilmente presso B. È per questo che il WiFi adotta CSMA/CA, con *collision avoidance* anziché *detection*.

47.5 Protocolli senza collisioni e a rotazione

Il **bit map protocol** si basa su due fasi distinte. Nella **fase di contesa**, con N nodi il tempo viene suddiviso in N **minislot** che consentono la trasmissione di un solo bit; ogni nodo ha il proprio slot identificativo e vi inserisce **1** se deve trasmettere, **0** altrimenti, costruendo così una mappa di bit. Al termine della fase **tutti i nodi sanno quali stazioni vogliono parlare e in quale ordine**. Nella **fase di trasmissione** le stazioni trasmettono una per volta seguendo l'ordine determinato dalla mappa, il che riduce drasticamente le collisioni.

I problemi sono tre. La **scarsa scalabilità**: aggiungere una stazione richiede di estendere la mappa e comunicare il nuovo slot a tutti. L'**inefficienza con molti nodi poco attivi**: se le stazioni sono tante ma poche vogliono trasmettere, la fase di contesa spreca troppo tempo. E il fatto che l'assenza di collisioni **presuppone che tutti i nodi rispettino il protocollo**.

Il **binary backward counting** (noto anche come **CSMA/BA (with Bitwise Arbitration)** e usato nel **CAN bus**) assegna a ogni nodo un **ID binario**, eventualmente calcolato a partire dal proprio MAC address. Il meccanismo è a eliminazione progressiva: tutti i nodi che vogliono trasmettere condividono nel proprio minislot la **cifra più significativa** del proprio ID; escono dalla competizione quelli che hanno trasmesso 0 mentre qualcun altro ha trasmesso 1; si passa alla cifra successiva e si ripete, finché resta un solo vincitore, che trasmetterà. Nel CAN bus la logica è rovesciata: il bit dominante è lo 0, e vince quindi l'identificativo più basso.

I vantaggi sono che lo **spreco di banda è minimo**, perché l'ID della stazione potrebbe essere il suo stesso indirizzo e quindi la contesa coincide con l'inizio della scrittura dell'indirizzo mittente, e che **non occorre sapere a priori quante macchine ci siano**. Lo svantaggio è che il protocollo può risultare **unfair**: i dispositivi con l'identificativo vincente hanno sistematicamente priorità maggiore.

TOKEN BUS

Fra i **taking turns protocols**, il **Token Bus** prevede una topologia **fisica** in cui i nodi condividono un collegamento, tipicamente un cavo coassiale, e una topologia **logica** in cui i nodi che vogliono trasmettere si inseriscono in un **anello logico**. Nell'anello circola un **token**, e **può trasmettere solo chi lo possiede**; dopo aver trasmesso, il token viene passato al nodo successivo.

È però un sistema **difficile da gestire**, per una serie di ragioni concrete. Il token è a sua volta un frame e **potrebbe collidere** con altri frame. Potrebbe **perdersi**: per gestire questa eventualità tutte le macchine sorvegliano il canale, e se dopo un certo tempo notano che il token non sta circolando, tutte sono autorizzate a generarne uno nuovo: nella speranza che ne venga generato uno solo. Se invece i token risultassero **due**, occorre una politica per rimuovere quello superfluo. Quando un nodo passa il token al successivo deve inoltre **verificare che quest'ultimo stia effettivamente parlando**. E infine va gestita l'entrata e l'uscita dei nodi dall'anello. Il bilancio è che il Token Bus **offre molte garanzie ma non viene utilizzato**, perché richiede un software troppo complesso.

Ethernet e IEEE 802.3

Lo standard di rete locale più diffuso: la famiglia IEEE 802, l'algoritmo CSMA/CD di Ethernet, le tecnologie standardizzate, i cavi in rame, la struttura del frame e la codifica di Manchester.

DA RICORDARE

Una trattazione completa di **IEEE 802.3** tocca cinque punti: che cosa standardizza (Ethernet e CSMA/CD), la **struttura del frame** campo per campo, l'**algoritmo CSMA/CD con binary exponential backoff**, le **tecnologie standardizzate** con la loro nomenclatura, e la collocazione di 802.3 all'interno della famiglia IEEE 802.

Ethernet è lo standard di reti cablate più diffuso al mondo per le reti locali. Non offre garanzie particolari, ma è **facile da implementare**; il suo nome ufficiale è **IEEE 802.3** e implementa il protocollo di accesso multiplo **CSMA/CD**. È riuscito a restare al passo con i tempi tramite versioni via via più veloci, mantenendo al contempo **costi hardware contenuti**: ed è probabilmente questa combinazione, più che meriti tecnici assoluti, a spiegarne il dominio.

LA FAMIGLIA IEEE 802

La **IEEE** (*Institute of Electrical and Electronics Engineers*) si occupa dei protocolli per l'accesso al mezzo fisico. I gruppi principali da conoscere sono i seguenti.

STANDARD	OGGETTO	STATO
802.1	High Level Interface (HILI): comprende anche 802.1Q per le VLAN	Attivo
802.2	Logical Link Control (LLC)	Non più attivo
802.3	CSMA/CD (Ethernet) : tutti i nodi hanno la stessa priorità, non offre garanzie	Attivo
802.4	Token Bus: permette di gestire priorità, offre garanzie	Non più attivo
802.5	Token Ring: permette di gestire priorità, offre garanzie	Non più attivo
802.11	Wireless LAN (WLAN)	Attivo
802.15	Wireless Personal Area Network (WPAN)	Attivo
802.16	Broadband Wireless Access (BBWA)	Non più attivo

Il contrasto fra **802.3** da un lato e **802.4** e **802.5** dall'altro è concettualmente significativo: i protocolli a token offrivano **priorità e garanzie**, cioè esattamente ciò che 802.3 non dà, eppure i loro gruppi di lavoro non sono più attivi mentre Ethernet ha vinto. La ragione è la stessa già vista per il Token Bus: la complessità di gestione non ha compensato i vantaggi.

48.1 L'algoritmo CSMA/CD di Ethernet

1. Ethernet riceve il datagramma dal livello di rete e crea il frame
2. Ethernet ascolta il canale:
 - inattivo : avvia la trasmissione e continua ad ascoltare
 - occupato : attende finché il canale non diventa inattivo
3. Se viene rilevata un'altra trasmissione:
 - interrompe la trasmissione
 - invia un segnale di JAMMING per comunicare la collisione
4. Attende un tempo casuale (BACKOFF) con binary exponential backoff:
 - dopo m collisioni estrae k a caso in $\{0, 1, \dots, 2^m - 1\}$
 - attende $T = k \cdot t_{\text{slot}}$ (t_{slot} = tempo per 512 bit)
 - torna al punto 2
 - m cresce fino a un massimo di 10 → k fino a 1023
 - dopo 16 collisioni: fallimento definitivo del frame

L'algoritmo CSMA/CD in Ethernet

48.2 Le tecnologie standardizzate

Le tecnologie Ethernet hanno una **denominazione fissa** composta da tre parti: un **numero** che indica la velocità dello standard; la sigla **BASE**, che indica la trasmissione in **banda base** (*baseband*), cioè di un singolo segnale sul mezzo; e un'**etichetta** che specifica il mezzo fisico. La prima definizione di 802.3 prevedeva velocità di 1 e 2 Mbps che non ebbero mai implementazione; la famiglia più diffusa è stata la **10Base[X]**.

STANDARD	MEZZO FISICO	PORTATA E NODI	NOTE
10Base5	Cavo coassiale spesso (<i>thick coax</i>), diametro 1 cm	500 m per segmento, max 100 nodi ; fino a 5 segmenti con ripetitori → 2,5 km	I computer non si collegano direttamente al cavo ma tramite un transceiver (MAU) fissato con presa a vampiro , che perfora la guaina fino al conduttore; dal transceiver parte un cavo AUI verso la scheda di rete. Sulla guaina erano stampati segni ogni 2,5 m a indicare dove installare i transceiver, così da limitare le riflessioni del segnale. Pro : schermatura elevata, cavo robusto adatto ad ambienti industriali. Contro : installazione complicata, poco flessibile: la rottura del cavo bloccava tutta la rete.
10Base2	Cavo coassiale sottile (<i>thin coax</i>), diametro 0,5 cm	185 m per segmento, max 30 nodi ; fino a 5 segmenti → 1 km	Molto più flessibile; per estenderlo bastava tagliarlo e inserire un connettore, e si collegava direttamente alla scheda di rete. Contro : si rompeva facilmente ed era sensibile ai disturbi.
10BaseT	Doppino intrecciato UTP (<i>Unshielded Twisted Pair</i>), Cat 3, connettore RJ45	100 m per segmento, max 1024 nodi	Il cavo contiene quattro coppie di fili di rame intrecciati per ridurre i disturbi, senza schermatura esterna; 10BaseT ne usa due. Prevedeva un hub centrale in topologia a stella. Pro : la rottura di un collegamento non guasta l'intera rete, e il sistema è scalabile aggiungendo porte. Contro : sicurezza, perché l'hub invia il segnale a tutti.
10BaseF	Fibra ottica	2000 m per segmento, max 1024 nodi	Collegamenti punto a punto, immune a tutti i segnali elettromagnetici.

48.3 I cavi in rame

La trasmissione è influenzata da come il cavo è realizzato, e vale la pena conoscere i fenomeni in gioco. Il **passaggio di corrente induce un campo magnetico** che genera a sua volta corrente indotta, la quale scorre in verso opposto al flusso di partenza opponendovisi. Anche i **cavi adiacenti** creano campi magnetici e quindi flussi opposti: questo fenomeno si chiama **diafonia** (*crosstalk*) e produce aumento del rumore di fondo, possibili errori e degradazione della qualità del segnale. Infine la **lunghezza del cavo è direttamente proporzionale alla sua resistenza**, ragione per cui su grandi distanze servono ripetitori.

Un cavo di rame si compone di **conduttore**, **jacket** (il materiale isolante esterno), **shield** (la schermatura complessiva su tutti i cavi) e **foil** (la schermatura su una singola coppia). Da qui le sigle: **UTP** senza schermatura, **STP** schermato, **FTP** con foil, **SFTP** con entrambi.

Le **categorie** si distinguono in base a quanto è regolare l'intreccio: **Cat5** supporta fino a 100 Mbps a 100 MHz; **Cat5e** arriva a **1 Gbps** sempre a 100 MHz; **Cat6** arriva a **10 Gbps** a 250 MHz, ma solo su distanze ridotte (circa 55 m); **Cat6a** mantiene i 10 Gbps ma su distanze maggiori, a 500 MHz.

Il connettore **RJ45** è la terminazione dei cavi Ethernet, di tipo **8P8C** (otto posizioni, otto contatti). Esistono due schemi di cablaggio, **T568A** e **T568B**, che differiscono per un'inversione delle coppie 2 e 3; lo standard ANSI/TIA li ammette entrambi, e **T568B** è molto diffuso nelle installazioni commerciali. È la **polarità opposta** dei segnali nei doppietti intrecciati a permettere la riduzione del rumore. Si distingue infine fra **cavo incrociato**, in cui la disposizione dei doppietti differisce fra le due estremità e che serve a collegare dispositivi **dello stesso tipo**, e **cavo diretto**, in cui la disposizione è uguale e che serve a collegare dispositivi **di tipo diverso**.

48.4 Il frame Ethernet

Preambolo	SFD	Destin.	Sorgente	Tag 802.1Q	Tipo o lunghezza	Dati e padding	FCS
7 byte	1 byte	6 byte	6 byte	4 byte	2 byte	da 46 a 1500 byte	4 byte

Il tag 802.1Q è presente solo nei frame VLAN. Dagli indirizzi alla FCS il frame misura da 64 a 1518 byte (1522 con il tag).

Struttura del frame Ethernet

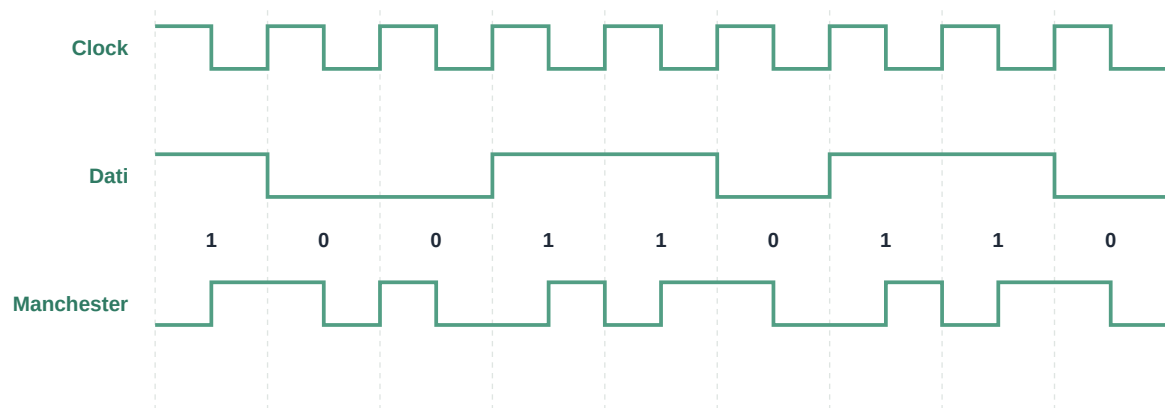
I campi del frame sono i seguenti.

- **Preambolo**: 7 byte con il pattern 10101010, per **sincronizzare i clock** delle schede di rete.
- **Start Frame Delimiter**: un byte a 10101011, che interrompe il pattern del preambolo e indica **l'inizio effettivo** del frame. Si noti il dettaglio: cambia solo l'ultimo bit, ed è proprio quella rottura del ritmo a segnalare l'inizio.
- **MAC address** di destinazione e di mittente.
- **Q Tag / VLAN Tag**: presente **esclusivamente** nei frame che supportano le VLAN, e quindi **non sempre**, dato che non tutti i frame Ethernet sono *tagged*.
- **Type/Length**: 2 byte che, se il valore è almeno 1536 (0x0600 in esadecimale), identificano il **tipo** di protocollo trasportato (*EtherType*: IPv4, IPv6, ARP...); se il valore non supera 1500 indicano invece la **lunghezza** del campo dati. Oggi viene utilizzato quasi sempre per indicare il tipo.
- **Data field**: minimo **46 byte**, massimo **1500 byte**, così che il frame, dagli indirizzi alla FCS, misuri almeno **64 byte**. Se i dati non raggiungono la dimensione minima si applica un **padding**; Ethernet invece non frammenta: datagrammi più lunghi devono essere frammentati dal livello di rete.
- **Frame Check Sequence**: 4 byte dedicati al controllo **CRC** per rilevare gli errori; un frame errato viene scartato, non corretto.

Questo stesso frame è stato utilizzato anche per Fast Ethernet e Gigabit Ethernet; molti apparati supportano inoltre, al di fuori dello standard, un campo dati fino a circa **9000 byte** con i cosiddetti *jumbo frame*.

48.5 La codifica di Manchester

La **codifica di Manchester** rappresenta i bit 0 e 1 in un segnale elettrico ed è utilizzata nel passaggio fra livello DLL e livello fisico. Nasce per risolvere due delle criticità già incontrate. Contro l'**ambiguità delle lunghe sequenze**, forza **transizioni obbligatorie** a ogni bit, rendendo più facile riconoscere malfunzionamenti e interruzioni. Contro la **sincronizzazione precaria**, fa sì che **ogni bit contenga al suo interno una transizione**, cosicché la sincronizzazione non possa perdersi.



La codifica di Manchester: ogni bit contiene una transizione al centro

La codifica richiede che ogni bit abbia una **transizione al centro del periodo**, realizzata tramite un'operazione di **XOR fra il segnale di clock e il segnale dati**. La trasmissione viene identificata in base ai passaggi: secondo la convenzione di IEEE 802.3, un **passaggio da alto a basso** rappresenta il bit **0**, un **passaggio da basso ad alto** il bit **1** (la convenzione originale, attribuita a G. E. Thomas, è opposta). Il principio di fondo è che **è più semplice individuare una variazione che un segnale statico**.

Resta da sincronizzare mittente e destinatario per scandire i passaggi da un bit all'altro: il destinatario deve allineare il proprio clock entro un tempo massimo, operazione svolta dal **preambolo**, costituito da una sequenza nota a entrambi. Quegli 8 byte **sprecano tempo sul canale**, tanto che il WiFi prevede un preambolo corto e uno lungo. Grazie allo XOR fra dati e clock, però, il recupero del clock lato ricevente resta possibile **anche se la sincronizzazione iniziale non è avvenuta** entro il tempo massimo. In pratica Manchester **aiuta a risolvere il problema del framing**, perché semplifica la sincronizzazione e facilita il riconoscimento del preambolo.

Esiste anche la **codifica di Manchester differenziale**, usata in Token Ring: la transizione al centro del bit serve solo al clock, mentre il valore del bit è indicato dalla **presenza o assenza di una transizione all'inizio del periodo**. Basandosi sulle variazioni anziché sui livelli, non risente di un'eventuale inversione dei due fili. La codifica Manchester era usata in tutte le versioni di Ethernet a **10 Mbps**, compresa 10BaseT, ed è oggi sostituita da codifiche più efficienti.

Fast Ethernet e Gigabit Ethernet

Come Ethernet è passata da 10 Mbps a 100 Mbps e poi a 1 Gbps, con le codifiche impiegate, i tipi di fibra ottica e la codifica trellis con decodifica di Viterbi.

49.1 Fast Ethernet (100 Mbps)

Volendo aumentare la velocità, i cavi coassiali vengono abbandonati. Le tre varianti sono le seguenti.

100BaseT4 rappresenta un caso interessante di ingegneria vincolata: invece di passare direttamente al Cat5, sfruttava i **cavi Cat3 già presenti negli edifici**, che però non erano nati per supportare 100 Mbps. La soluzione fu utilizzare **tutte e quattro le coppie** contemporaneamente, a differenza del 10BaseT che ne usava due. Il traffico veniva diviso su **tre coppie parallele** da circa 33,3 Mbps ciascuna ($3 \times 33,3 \approx 100$ Mbps): delle quattro coppie una trasmette sempre, una riceve sempre e serve a rilevare le collisioni, le altre due sono bidirezionali; il collegamento è **half duplex**.

La codifica impiegata era la **8B6T**: 8 bit venivano trasformati in **6 simboli ternari**, ciascuno dei quali può valere -1 , 0 oppure $+1$. Con 6 simboli ternari si ottengono $3^6 = 729$ **combinazioni**, mentre ne bastano $2^8 = 256$. Poiché ogni simbolo ternario rappresenta più informazione di un simbolo binario, a parità di velocità si trasmettono più dati, e la tecnica permise di **ridurre la frequenza a circa 25 MHz per coppia**, rientrando nei limiti del Cat3. I problemi erano la complessità realizzativa e l'**incompatibilità con il full duplex**.

100BaseTX passa ai **Cat5** e utilizza solo due coppie, una per trasmettere e una per ricevere, lasciando le altre due inutilizzate. Impiega la codifica **4B5B**, cosicché i 100 Mbps effettivi diventano **125 Mbaud**, e la modulazione **MLT3** (*Multi Level Transmit 3*), tecnica a tre livelli (-1 , 0 , $+1$) in cui un **bit a 1 provoca il passaggio al livello successivo** del ciclo, mentre un bit a 0 lascia il livello invariato. Essendo full duplex, offre 100 Mbps in entrambe le direzioni, cioè 200 Mbps teorici.

100BaseFX utilizza la **fibra ottica** con collegamenti punto a punto, è immune alle interferenze e impiega anch'essa la codifica 4B5B.

49.2 Gigabit Ethernet (1 Gbps)

Il problema, nel passaggio al Gigabit, era il **cablaggio**. Con Fast Ethernet il cavo coassiale era già stato rimosso sia per problemi di affidabilità sia per una ragione meno ovvia legata alla **dimensione minima del frame**: per Ethernet è di 64 byte, e a parità di estensione della rete decuplicare la velocità l'avrebbe portata a **640 byte**. Fast Ethernet ha invece mantenuto i 64 byte riducendo di circa dieci volte il diametro massimo del dominio di collisione, cosa compatibile solo con un cablaggio a stella.

Il percorso per arrivare a 1 Gbps (1000BaseT, su cavo Cat5) si compone di cinque passi, che vale la pena seguire perché mostrano come si accumulano i guadagni.

1. Si parte dalla velocità di simbolo di Fast Ethernet:
125 Mbaud per coppia, senza però l'overhead della 4B5B
2. Si usano tutte e 4 le coppie simultaneamente
3. Ogni coppia trasmette e riceve contemporaneamente
(full duplex sullo stesso doppino, con cancellazione d'eco)
4. Si aumentano i livelli per simbolo: PAM5, cinque livelli
(-2, -1, 0, +1, +2); ogni simbolo trasporta 2 bit di dati
 $125 \text{ Mbaud} \times 4 \text{ coppie} \times 2 \text{ bit} = 1 \text{ Gbps}$ per direzione
5. Correzione degli errori (Forward Error Correction)
livelli più vicini significano più sensibilità al rumore:
il quinto livello fornisce la ridondanza per una codifica
trellis, decodificata al ricevitore con l'algoritmo di Viterbi

Come si arriva a 1 Gbps

Conviene richiamare la definizione di **framing**: è il processo con cui il ricevitore capisce **dove inizia e dove finisce un messaggio** all'interno di un flusso continuo di bit; servono quindi la sincronizzazione del clock, per stabilire dove inizia e finisce un bit, e un modo per delimitare il frame.

Va inoltre chiarito il termine **baud**: è l'unità della **velocità di simbolo**, cioè del numero di simboli (stati del segnale) trasmessi in un secondo. A ogni baud si può associare **un solo bit**, come in MLT3, oppure **due bit**, come nella codifica a cinque livelli PAM5.

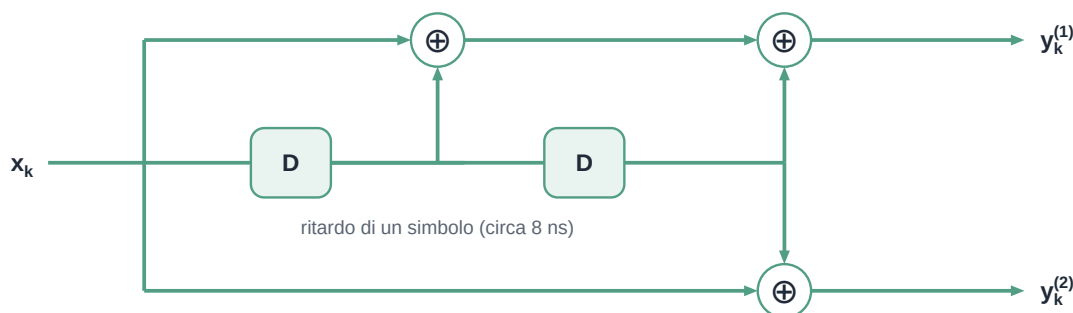
FIBRA MONOMODALE E MULTIMODALE

La **fibra monomodale** trasmette un **singolo raggio di luce** su lunghe distanze, oltre i 40 km, con alta larghezza di banda, utilizzando un nucleo sottile di circa $9 \mu\text{m}$ e sorgenti **laser**. È ideale per telecomunicazioni a lungo raggio, WAN, reti dorsali di campus e grandi data center, ed è **più costosa**.

La **fibra multimodale** trasmette **più raggi di luce** attraverso un nucleo più ampio, da 50 a $62,5 \mu\text{m}$, ed è adatta a distanze brevi e applicazioni meno esigenti, con sorgenti **LED o laser**. È perfetta per LAN, campus aziendali e data center, ed è **più semplice da installare e più economica**.

49.3 Trellis e decodifica di Viterbi

Il codificatore è un piccolo circuito con due **elementi di ritardo** (*delay box*), ciascuno dei quali trattiene un bit per la durata di un simbolo (circa 8 ns a 125 Mbaud), e due **sommatori modulo 2** (XOR). Per ogni bit in ingresso x_k produce **due bit** in uscita, calcolati dal bit corrente e dai due precedenti secondo le relazioni seguenti.



Il codificatore convoluzionale: due elementi di ritardo D e due sommatore modulo 2

$$y_k^{(1)} = x_k \oplus x_{k-1} \oplus x_{k-2} \quad y_k^{(2)} = x_k \oplus x_{k-2}$$

Il funzionamento si comprende meglio attraverso una **macchina a stati finiti** con quattro stati (0, 1, 2, 3): lo stato è il contenuto dei due ritardi, cioè vale $2 \cdot x_{k-1} + x_{k-2}$, e da ciascuno stato escono due frecce, una per ogni possibile bit in ingresso.

STATO	INPUT 0: USCITA	INPUT 0: NUOVO STATO	INPUT 1: USCITA	INPUT 1: NUOVO STATO
0	00	0	11	2
1	11	0	00	2
2	10	1	01	3
3	01	1	10	3

Prendiamo lo stato 0 come esempio: se riceve in input il bit **0** resta nello stesso stato ed emette in uscita la coppia **00**; se riceve il bit **1** passa allo stato 2 ed emette la coppia **11**. La struttura presenta regolarità precise: dagli stati 0 e 1 escono solo **coppie uguali** (00 oppure 11), dagli stati 2 e 3 solo **coppie diverse** (01 oppure 10); le frecce entranti negli stati 0 e 1 hanno tutte input 0, quelle entranti in 2 e 3 hanno input 1.

La proprietà decisiva è la seguente: da ogni stato si possono generare solo **due delle quattro coppie** possibili, e quindi non tutte le sequenze ricevute sono valide. A **due salti** di distanza si può arrivare in tutti gli stati, ma ogni cammino lascia una traccia ben precisa: è questa **ridondanza** a rendere possibile la correzione degli errori.

COME FUNZIONA LA CORREZIONE

Seguiamo l'esempio. In input riceviamo la sequenza **011001**, partendo dallo stato 0. La sequenza generata dall'automa è **00 11 01 01 11 11**.

Inseriamo ora degli errori e vediamo se riusciamo a riconoscerli: supponiamo di ricevere **01 11 01 11 11 11**. Già dalla prima coppia sappiamo che c'è un errore, perché **lo stato 0 genera solo coppie di bit uguali**, cioè 00 oppure 11, mentre abbiamo ricevuto 01. Non sappiamo però se il bit sbagliato sia il primo o il secondo. Tenendo conto dell'incertezza, si percorrono **entrambi i cammini possibili**, annotando per ciascuno la **distanza di Hamming** rispetto a ciò che l'automa avrebbe potuto generare.

Si procede quindi esaminando le coppie successive, mantenendo per ogni cammino la distanza cumulata e conservando, per ogni stato, solo il cammino migliore che vi arriva: è l'**algoritmo di Viterbi**. Sotto la consueta ipotesi probabilistica, per cui **un errore è più probabile di due e due più probabili di tre**, la strada corretta è quella a **distanza minima**. Proseguendo fino in fondo resta un solo cammino con la distanza più bassa, e quello è il cammino corretto: nell'esempio è il cammino che corrisponde all'input **011001**, a distanza 2 dalla sequenza ricevuta, e i due errori risultano entrambi corretti.

Il codice di questo esempio ha **distanza libera 5**: due sequenze codificate valide differiscono in almeno cinque bit, e quindi si correggono con certezza fino a due errori, purché non siano troppo ravvicinati rispetto alla memoria del codificatore. Raddoppiando il numero di bit trasmessi per ogni bit di informazione, gli errori pesano meno sull'informazione complessiva; ma è soprattutto grazie alle **transizioni limitate dell'automa** che si capisce quando è avvenuto un errore, e la distanza di Hamming fra le sequenze possibili e quella ricevuta indica come correggerlo. Nella Gigabit Ethernet lo stesso principio è applicato direttamente ai livelli del segnale PAM5 (*trellis coded modulation*): la ridondanza non raddoppia i bit trasmessi, ma sfrutta il quinto livello.

Poiché anche la correzione può fallire, si aggiunge il **CRC** sopra la codifica. Al mittente: si calcola il CRC sui dati e poi si codifica il tutto in modo ridondante con il trellis. Al ricevente: si usa **Viterbi per correggere** gli errori e poi si **controlla con il CRC**: se il controllo passa i dati sono corretti, se fallisce significa che gli errori erano troppi perché Viterbi potesse correggerli.

Strutture Ethernet e indirizzamento a livello DL

I dispositivi che compongono una LAN Ethernet e i domini di collisione e di broadcast che determinano, i bridge trasparenti e il confronto fra indirizzamento piatto e gerarchico.

La definizione originale di LAN riguardava tutto il sistema di bus condiviso, cioè il **dominio di collisione**, e quindi coincideva con il collegamento fisico: in 10Base2 e 10Base5 c'era un unico mezzo fisico ben identificabile, e quello era la LAN. L'evoluzione verso 10BaseT, con un dispositivo centrale, ha progressivamente separato i due concetti.

DISPOSITIVO	LIVELLO	COMPORTAMENTO	DOMINI
Repeater	Fisico	Dispositivo con due porte : rigenera il segnale ricevuto su una porta e lo ritrasmette sull'altra, senza interpretare i frame. Usato in 10Base2 e 10Base5 per collegare al massimo 5 segmenti di cavo coassiale. Obsoleto .	Collisione: unico; broadcast: unico
Hub	Fisico	Simile al repeater ma con più di due porte : ritrasmette il segnale ricevuto su tutte le altre porte , senza interpretare i frame. Topologia a stella. Obsoleto : collisioni frequentissime e traffico inutile.	Collisione: tutta la struttura ; broadcast: unico
Bridge	Data Link	Collega due reti LAN filtrando e inoltrando in base all' indirizzo MAC . Grazie alla tabella MAC sa su quale porta si trova un determinato dispositivo. Può effettuare anche la conversione fra tecnologie diverse , per esempio da frame Ethernet a frame WiFi: operazione problematica, dato che il frame WiFi ha fino a 4 indirizzi e fino a 2304 byte di dati contro i 2 indirizzi e i 1500 byte di Ethernet.	Collisione: una per porta ; broadcast: unico
Switch	Data Link (alcuni anche Rete)	Riceve un frame e lo inoltra solo alla porta giusta in base al MAC di destinazione, effettuando inoltro e filtraggio . Contiene processore, memoria e firmware per interpretare le intestazioni. Mantiene una tabella MAC i cui record contengono MAC, interfaccia e tempo . Standard attuale nelle LAN.	Collisione: una per porta ; broadcast: unico

Sul campo **tempo** nella tabella MAC dello switch vale la pena soffermarsi: serve a stabilire quando un dispositivo non è più presente in rete, cosa che viene **dedotta** dal fatto che per molto tempo non sono più stati inviati pacchetti verso quel MAC. È un meccanismo di scadenza analogo a quello della cache ARP.

Lo switch **elimina le collisioni**: con collegamenti full duplex non possono verificarsi, e in ogni caso una collisione su una tratta punto a punto **non può propagarsi** all'intera LAN. Offre maggiore velocità ed efficienza rispetto a hub e bridge, e il numero di porte è espandibile.

Di conseguenza, **il concetto di LAN non coincide più con il dominio di collisione**. Oggi si parla di LAN, inclusi gli switch intermedi, laddove **si parla lo stesso linguaggio**, cioè Ethernet. Se un dispositivo ha anche un'interfaccia wireless si parla di LAN cablata e LAN wireless collegate fra loro, tenendo presente che i frame WiFi possono avere 4 indirizzi e che le reti wireless necessitano di un **access point** che faccia da coordinatore.

50.1 Bridge trasparenti

Un **bridge trasparente** è un bridge che opera **senza che gli host se ne accorgano**: non richiede alcuna configurazione né modifica ai dispositivi collegati, e per questo si dice appunto «trasparente». La sua caratteristica interessante è che **costruisce da solo** la propria tabella di inoltra, mediante un meccanismo chiamato **backward learning**.

Il funzionamento è il seguente. Il bridge parte con la tabella **vuota** e osserva tutti i frame che transitano. Per ogni frame ricevuto su una porta, legge l'**indirizzo MAC sorgente** e registra in tabella che quel MAC è raggiungibile attraverso **quella porta**: è un'inferenza corretta, perché se il frame è arrivato da lì, evidentemente da lì si può rispondere. Quanto all'inoltra, il bridge cerca il **MAC di destinazione** in tabella: se lo trova e corrisponde a una porta **diversa** da quella di arrivo, inoltra il frame **solo su quella porta**; se corrisponde alla **stessa** porta di arrivo, **scarta** il frame, perché mittente e destinatario sono sullo stesso segmento e la consegna è già avvenuta; se **non lo trova**, effettua il **flooding**, inoltrandolo su tutte le porte tranne quella di provenienza.

Anche qui le voci hanno una **scadenza temporale**, così da adattarsi allo spostamento dei dispositivi. Poiché il flooding e la presenza di cammini ridondanti potrebbero generare cicli, i bridge trasparenti eseguono lo **Spanning Tree Protocol**, che (come già visto a proposito del controllo del flooding) riduce la topologia a un albero, in cui fra due nodi qualsiasi esiste **un solo cammino**.

50.2 Indirizzamento piatto e gerarchico

A livello di collegamento i dispositivi sono identificati dall'**indirizzo MAC**, che è un indirizzo **piatto** (*flat*): 6 byte rappresentati in esadecimale, univoco, **scritto a livello hardware** nella scheda di rete, con i blocchi distribuiti dalla **IEEE** alle aziende produttrici. A livello di rete si usa invece l'**indirizzo IP**, che ha una struttura **gerarchica**: o più precisamente, come si è visto, una parte gerarchica e una piatta.

La differenza si coglie bene con l'analogia già incontrata: l'indirizzo piatto è come il **numero di cellulare**, che identifica univocamente una persona ma **non dice nulla su dove si trovi**; l'indirizzo gerarchico è come il **numero di telefono fisso**, il cui prefisso indica la zona geografica.

	INDIRIZZAMENTO PIATTO (MAC)	INDIRIZZAMENTO GERARCHICO (IP)
Struttura	Nessuna: bit senza significato posizionale	Suddiviso in parte di rete e parte host tramite maschera
Assegnazione	Permanente, scritta in hardware dal produttore	Configurabile via software, staticamente o via DHCP
Mobilità	Vantaggio : il dispositivo mantiene il proprio indirizzo ovunque, senza riconfigurazione	Svantaggio : spostandosi in un'altra rete l'indirizzo deve cambiare
Aggregazione	Svantaggio : nessuna aggregazione possibile: servirebbe una voce per ogni singolo dispositivo	Vantaggio : indirizzi contigui si aggregano in un solo prefisso (supernetting), riducendo le tabelle
Scalabilità	Svantaggio : non scala oltre la LAN, una tabella globale dei MAC sarebbe ingestibile	Vantaggio : scala a livello mondiale, i router ragionano su prefissi
Instradamento	Apprendimento e flooding (bridge, STP)	Algoritmi di routing e <i>longest prefix match</i>
Ambito d'uso	All'interno di una singola LAN	Fra reti diverse, su scala globale

La conclusione da trarre è che i due schemi **non sono in competizione ma complementari**: il MAC è ottimale **dentro** la LAN, dove i dispositivi sono pochi e il flooding è sostenibile; l'IP è indispensabile **fra** le reti, dove senza aggregazione le tabelle di instradamento diventerebbero ingestibili. Per questo un pacchetto porta **entrambi** gli indirizzi: l'IP, invariato fino alla destinazione finale, e il MAC, che cambia a ogni salto.

VLAN

Le reti locali virtuali permettono di separare logicamente più LAN sulla stessa infrastruttura fisica. Il capitolo descrive le VLAN port based, il tag 802.1Q e il concetto di scope link.

Le **VLAN (Virtual LAN)** sono **LAN logiche separate che risiedono sopra la stessa struttura fisica**. La connessione fra VLAN diverse avviene attraverso il **routing**, cioè al livello 3, e questo è il punto concettuale chiave: due VLAN sullo stesso switch sono, a tutti gli effetti, due reti distinte. Lo standard **IEEE 802.1Q** ne definisce le specifiche.

Gli scopi principali sono quattro. Il **riutilizzo dell'architettura di partenza**, con conseguente risparmio, poiché non occorre cablare nuove reti fisiche. La **flessibilità**, essendo molto più semplice spostare gli utenti fra VLAN. Le **prestazioni migliori**, perché il traffico di broadcast resta **isolato** all'interno di ciascuna VLAN. E la **sicurezza**, dato che gli utenti di VLAN differenti non vedono i frame degli altri.

51.1 VLAN untagged (port based)

Nelle **port based VLAN** lo switch viene partizionato **logicamente** in più insiemi di porte, e le porte vengono associate alle VLAN; naturalmente lo switch deve supportare le VLAN. Per estendere una VLAN port based su più switch occorre un collegamento dedicato **per ciascuna VLAN**, fra porte che appartengono alla stessa VLAN.

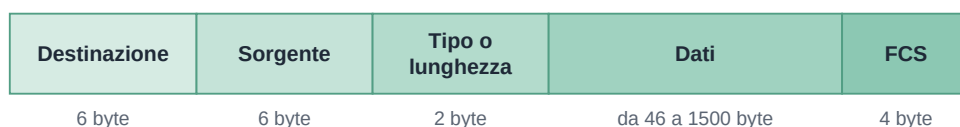
Questa VLAN si dice **untagged** perché è sprovvista di un tag identificativo: **la VLAN non compare nel frame**, ed è lo switch a deciderla **in base alla porta** da cui il frame proviene. Ne consegue che **una porta può trasportare il traffico di una sola VLAN**, e l'utilizzo tipico è il collegamento dei dispositivi finali, come PC e stampanti (porte di accesso).

Il vantaggio è che l'implementazione avviene **via software**: per spostare un dispositivo da una VLAN a un'altra non occorre toccare il cablaggio, ma basta riconfigurare la porta.

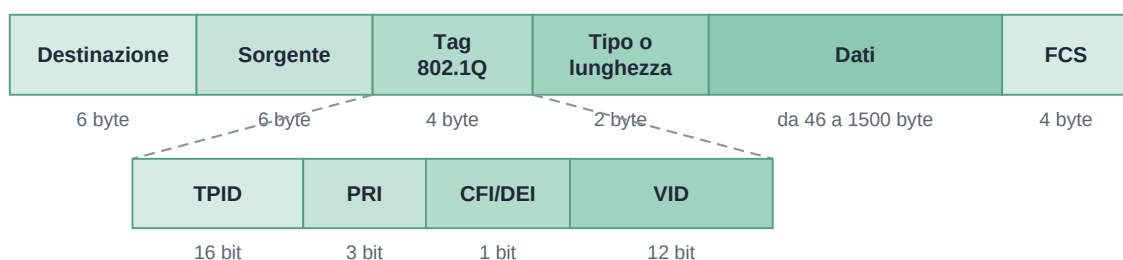
51.2 VLAN tagged e lo standard 802.1Q

Lo standard **802.1Q** inserisce nel frame Ethernet **4 byte** aggiuntivi per associare esplicitamente il frame a una VLAN.

FRAME SENZA TAG



FRAME CON TAG 802.1Q



I campi del tag 802.1Q inserito nel frame Ethernet

I campi sono quattro: il **Tag Protocol Identifier** (16 bit, valore 0x8100), che occupa la posizione del campo Type/Length e segnala la presenza del tag; il campo **Priority** (3 bit), che identifica il livello di priorità; il **Canonical Format Identifier** (1 bit), nato per indicare il formato degli indirizzi MAC nelle reti Token Ring, vale 0 in Ethernet ed è stato poi ridefinito come **Drop Eligible Indicator**, che segnala i frame da scartare per primi in caso di congestione; e il **Virtual LAN Identifier** (12 bit), che identifica la VLAN vera e propria. Dai 12 bit dell'identificatore discendono 4096 valori, di cui due riservati: le VLAN utilizzabili sono quindi **4094**.

Un aspetto pratico importante è la **retrocompatibilità**: il tag viene aggiunto e rimosso dagli switch, e sulle porte di accesso i frame viaggiano senza tag. I dispositivi *legacy* che non supportano le VLAN non vedono quindi mai frame *tagged*, e l'introduzione delle VLAN in una rete esistente non ne compromette il funzionamento.

A differenza del caso untagged, qui **una porta può trasportare il traffico di più VLAN contemporaneamente**, poiché l'appartenenza è scritta nel frame stesso anziché dedotta dalla porta. L'utilizzo tipico sono i collegamenti **fra due switch o fra switch e router**, detti collegamenti **trunk**.

51.3 Scope link

Lo **scope link** indica un ambito di validità **limitato a un singolo collegamento di rete**, fisico o logico, cioè a una singola interfaccia come *eth0*. In pratica significa che un indirizzo IP, o un pacchetto, con scope link può essere usato **soltanto per comunicare con dispositivi collegati direttamente sulla stessa interfaccia**, cioè sullo stesso segmento di rete. È il concetto che si è già incontrato negli indirizzi IPv6 link local **FE80::/10**, utilizzabili solo fra nodi della stessa LAN e non instradabili su Internet.